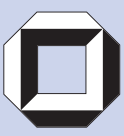
Olaf Seng

Suchbasierte Strukturverbesserung objektorientierter Systeme



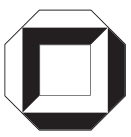
universitätsverlag karlsruhe

Olaf Seng

Suchbasierte Strukturverbesserung objektorientierter Systeme

Suchbasierte Strukturverbesserung objektorientierter Systeme

von
Olaf Seng



universitätsverlag karlsruhe

Dissertation, Universität Karlsruhe (TH)
Fakultät für Informatik, 2007

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Universitätsverlag Karlsruhe 2008
Print on Demand

ISBN: 978-3-86644-213-9

Danksagung

An erster Stelle möchte ich mich bei meinem Doktorvater Professor Goos bedanken, der mich in einer unserer Gruppenrunden auf die Idee brachte, mich näher mit evolutionären Algorithmen zu befassen und somit maßgeblich zur Entstehung dieser Arbeit beigetragen hat. Herr Goos hat mich in außergewöhnlichem Maße unterstützt, indem er regelmäßig meinen Fortschritt überprüft, zielsicher Unklarheiten gefunden und Anregungen zu weiteren Fragestellungen gegeben hat.

Bei Herrn Professor Reussner bedanke ich mich für die Übernahme des Korreferats und für seine Hinweise und Anmerkungen.

Meinen Kollegen am Forschungszentrum Informatik, Christoph Andriessens, Holger Bär, Markus Bauer, Thomas Genssler, Volker Kuttruff, Marco Mevius, Benedikt Schulz, Peter Szulman, Adrian Trifu, Mircea Trifu und Jan Wiesenberger danke ich für die schöne Zeit und zahlreiche Diskussionen und Tipps während meiner Promotion. Insbesondere die Kollegen, die nach mir ihre Laufbahn am FZI begonnen haben, verschafften mir in meinem letzten Jahr den nötigen Freiraum, um meine Promotion beenden zu können.

Bei Mircea Trifu bedanke ich mich für seine Geduld beim Lesen meiner Veröffentlichungen und für seine Anregungen und Tipps. Volker Kuttruff danke ich für die zahlreichen Diskussionen über Restrukturierungen und für die intensive Durchsicht meiner Arbeit, bei der ihm Dank seiner legendären Adleraugen nichts entging. Markus Bauer danke ich für die zahlreichen Gespräche über Softwarequalität und insbesondere für seine Hinweise, die mir halfen die Botschaft meiner Arbeit klarer zu vermitteln.

Meinen Kollegen am Lehrstuhl von Professor Goos danke ich für ihre Unterstützung und ihre Diskussionsbereitschaft.

Auch meine Studenten Gert Pache, David Burkhart und Johannes Stammel haben maßgeblich zur vorliegenden Arbeit beigetragen. Im Rahmen ihrer Studien- und Diplomarbeiten haben sie einen großen Teil meiner Konzepte in Software umgesetzt und dafür gesorgt, dass ich meine Ideen in der Praxis evaluieren konnte.

Meiner Frau Neda danke ich für die vielen Korrekturlesungen meiner Arbeit. Sie hat mich in den Jahren meiner Promotion tatkräftig unterstützt und mir den nötigen Rückhalt gegeben.

Einen herzlichen Dank auch an meine Geschwister Daniela und Markus, die mir mit gutem Beispiel vorangegangen sind und immer ein offenes Ohr für meine Sorgen hatten.

Den größten Dank verdienen jedoch meinen Eltern, die meine ersten Lehrer waren und ohne deren Unterstützung diese Arbeit niemals entstanden wäre.

Karlsruhe im Januar 2008,
Olaf Seng

Zusammenfassung

Die Struktur von Softwaresystemen zerfällt aufgrund von nötigen Anpassungen an eine sich verändernde Umwelt. Als Konsequenz der sich verschlechternden Struktur werden weitere Änderungen des Systems immer aufwändiger, es sei denn die Struktur wird vor den Änderungen immer wieder verbessert. Mit Metriken können Entwurfsheuristiken überprüft und somit die Qualität der Struktur bewertet werden. Mit Restrukturierungen kann die Struktur von Softwaresystemen verhaltensbewahrend verändert werden. Jedoch ist es sehr schwierig, die richtige Restrukturierung an der richtigen Stelle auszuführen, da Restrukturierungen unbeabsichtigte Auswirkungen haben können.

In dieser Arbeit stellen wir ein Verfahren vor, das Restrukturierungen bestimmt, die die Qualität der Struktur eines gegebenen Systems verbessern, um Anpassungen des Systems an eine sich verändernde Umwelt zu vereinfachen. Die Bestimmung von Restrukturierungen zur Verbesserung der Qualität der Struktur eines Systems ist ein Suchproblem. Aus der Menge der Strukturen von Systemen mit gleichem Ein- /Ausgabeverhalten suchen wir die Systemstruktur mit der höchsten Qualität. Die Struktur können wir verändern, indem wir die Zuordnung von Klassen zu Teilsystemen und von Methoden und Attributen zu Klassen modifizieren. Dieses Problem können wir auf das NP-vollständige Problem, eine bezüglich einer Zielfunktion optimale Zerlegung von Elementen zu bestimmen, zurückführen. Um eine Näherungslösung zu bestimmen, verwenden wir einen evolutionären Algorithmus. Das Grundprinzip besteht darin, dass wir Restrukturierungen auf einem Modell des Systems zufallsgesteuert simulieren und anhand einer Zielfunktion entscheiden, ob eine Restrukturierung gut oder schlecht ist.

Das Verfahren liegt in einer in Java geschriebenen Implementierung vor. Für mehrere Open-Source-Softwaresysteme und ein am FZI entwickeltes System konnten Restrukturierungen bestimmt werden, die die Qualität der Struktur verbesserten und zukünftige Änderungen vereinfachten. Die vorliegende Arbeit stellt eine Verallgemeinerung des existierenden Stands der Technik dar. Zum einen berücksichtigt unser Verfahren im Vergleich zu existierenden Verfahren mehr Entwurfsheuristiken und mehr Restrukturierungen gleichzeitig. Zum anderen gelingt es uns mit der vorliegenden Arbeit zu zeigen, dass ein evolutionärer Algorithmus auch zur Verbesserung von Klassenstrukturen realer Systeme eingesetzt werden kann.

Inhaltsverzeichnis

Kapitel 1.

Einleitung

1.1. Kontext

Die meisten Softwaresysteme modellieren einen Teil der wirklichen Welt. Da sich die Welt verändert, müssen diese Softwaresysteme ständig an die sich verändernde Umwelt angepasst werden. Bleiben Anpassungen aus, sinkt der Nutzen eines Systems im Laufe der Zeit [LB85].

Die Qualität der Struktur eines Systems bestimmt maßgeblich die Kosten für diese Anpassungen. Untersuchungen konnten zeigen, dass die Qualität der Struktur mit zunehmendem Alter des Systems abnimmt [LB85] und ein struktureller Zerfall einsetzt [BBC+99]. Der strukturelle Zerfall manifestiert sich in Form von Strukturproblemen, also Fragmenten einer Softwarestruktur, die die Eigenschaft haben, sich nachteilig auf den Aufwand von Entwicklungsaktivitäten auszuwirken [Ciu01]. Als Konsequenz des strukturellen Zerfalls steigen die zur Wartung und Weiterentwicklug anfallenden Kosten kontinuierlich an.

Es gibt drei Hauptgründe für den strukturellen Zerfall. Erstens ist es nahezu unmöglich, alle zukünftigen Anforderungen im ursprünglichen Entwurf der Software zu berücksichtigen. Der Entwurf muss folglich bei unvorhergesehenen Änderungen nachträglich angepasst werden. Zweitens erfolgen die Anpassungen oft unter hohem Zeit- und Kostendruck in Form von unsauberen, schnellen Lösungen. Drittens verfügen die beteiligten Entwickler nicht immer in ausreichendem Maße über das Wissen über gute Strukturen und verschlechtern deshalb die Qualität der Struktur unbeabsichtigt.

1.2. Problemstellung und Ziel

Zwei grundsätzliche Arten von Methoden und Werkzeugen, um dem oben beschriebenen strukturellen Zerfall entgegenzuwirken, existieren bereits: Erstens können Strukturen, die Änderungen erschweren, als Verletzungen von Entwurfsheuristiken mit Metriken identifiziert werden. Eine Metrik misst entweder eine strukturelle Eigenschaft, wie z.B. Kohäsion, oder die Anzahl der Verletzungen von Regeln, wie z.B. Anzahl der Klassen, die mehr als 50 Methoden besitzen. Zweitens kann die Struktur mit Restrukturierungen verändert werden, ohne dass sich das von außen beobachtbare Verhalten verändert, Änderungen aber anschließend wieder leichter durchgeführt werden können.

Zwischen diesen beiden Arten von Verfahren existiert jedoch eine konzeptionelle Lücke: Eine Restrukturierung kann zusätzlich zu den gewünschten Auswirkungen Nebenwirkungen haben, die nicht oder nur sehr schwer abzuschätzen sind. So könnte eine Restrukturierung die Qualität der Struktur an einer Stelle bzw. bezüglich einer Entwurfsheuristik verbessern, als Nebenwirkung könnte sich jedoch die Struktur an einer anderen Stelle bzw. bezüglich einer anderen Entwurfsheuristik verschlechtern. Beispielsweise resultiert eine sehr geringe Kopplung zwischen den einzelnen Teilen eines Systems oft in einer verschlechterten Kohäsion der Teilsysteme, da fast alle Abhängigkeiten innerhalb der Teilsysteme verlaufen. Umgekehrt gilt, dass sich bei sehr hoher Kohäsion innerhalb der Teilsysteme die Kopplung zwischen den Teilsystemen meist stark erhöht. Ein Softwareingenieur muss diese Lücke manuell überbrücken, indem er alle Auswirkungen der Restrukturierungen abschätzt, bevor er sich für bestimmte Restrukturierungen entscheidet.

Das *Ziel der vorliegenden Arbeit* ist ein Verfahren, das Restrukturierungen bestimmt, die die Qualität der Struktur eines gegebenen Systems verbessern, um Anpassungen des Systems an eine sich verändernde Umwelt zu vereinfachen.

Unter der Struktur eines Systems verstehen wir die in einem System implizit und explizit vorkommenden Einheiten (z.B. Teilsysteme, Klassen und Methoden) und die Beziehungen zwischen diesen Einheiten (z.B. Vererbungsbeziehungen und Methodenaufrufe). Wir betrachten und verändern die beim Programmieren-im-Großen [DK75] relevanten Elemente, d.h. Teilsysteme und Klassen. Dazu gehört insbesondere auch die Verteilung der Methoden auf die Klassen. Allerdings gehen wir davon aus, dass die Ausformulierung der Methoden unveränderlich ist, obwohl diese in seltenen Fällen einer besseren Klassen- und Teilsystemstruktur im Wege stehen kann.

Damit die vorliegende Arbeit nutzbringend ist, werden folgende Anforderungen an das Verfahren gestellt. Das Verfahren muss

- *automatisch ablaufen.* Das Verfahren muss automatisch Restrukturierungen zur Verbesserung der Qualität der Struktur bestimmen können und garantieren, dass die Restrukturierungen auf dem vorliegenden System ausführbar sind. Eine Umsetzung der Restrukturierungen kann jedoch manuell oder automatisiert mit gängigen Werkzeugen erfolgen. Eine anfängliche Parametrisierung des Verfahrens ist erforderlich. Ein Softwareingenieur muss abhängig von seinen Anforderungen spezifizieren, was für ihn Qualität der Struktur bedeutet. Hierzu legt er fest, wie relevant einzelne Entwurfsheuristiken für ihn sind.

- *multikriteriell sein.* Bei der Bestimmung der Restrukturierungen muss das Verfahren alle, von einem Softwareingenieur ausgewählten, Entwurfsheuristiken als Kriterium berücksichtigen und sicherstellen, dass sich die Qualität der Struktur insgesamt verbessert.

- *das extern beobachtbare Verhalten des Systems unverändert lassen.* Das beobachtbare Verhalten definieren wir als das Ein- und Ausgabeverhalten des Systems. Führen Eingangsdaten E auf dem ursprünglichen System zu Ausgaben A, so müssen diese Eingangsdaten E auf dem veränderten System ebenfalls zu Ausgaben A führen.

- *Strukturen bewahren, die Entwurfsheuristiken bewusst verletzen.* Manche Teilstrukturen, wie z.B. Entwurfsmuster verletzen übliche Entwurfsheuristiken. Trotzdem sollen diese besonderen Strukturen erhalten bleiben.

- *auf reale objektorientierte Systeme anwendbar sein.* Das Verfahren muss mit existierenden Programmen umgehen können, die in einer objektorientierten Programmiersprache implementiert sind.

- *die wesentlichen Strukturierungseinheiten berücksichtigen.* Auf unterster Ebene tragen Klassen zur Struktur eines Systems bei. Somit muss das Verfahren insbesondere die Baupläne von Klassen berücksichtigen, d.h. die Zuordnung von Methoden zu Klassen verändern können. Auf höherer Ebene muss das Verfahren den Aufbau von Teilsystemen berücksichtigen, d.h. es muss die Zuordnung von Klassen zu Teilsystemen verändern können. Idealerweise sollte das Verfahren diese beiden Ebenen verzahnt betrachten, da eine ungeschickte Verteilung der Methoden auf Klassen eine gute Aufteilung in Teilsysteme unmöglich machen kann.

1.3. Verwandte Arbeiten

Es existiert bereits eine Reihe von Ansätzen, deren Ziel es ist, die Qualität der Struktur eines Systems zu verbessern. Allerdings ist keines der existierenden Verfahren in der Lage unsere Anforderungen zu erfüllen.

Verfahren zur allgemeinen Abschätzung der Auswirkung von Restrukturierungen empfehlen unabhängig vom konkret vorliegenden System Restrukturierungen und können somit nicht gewährleisten, dass sich insgesamt die Qualität der Struktur verbessert. Sie sind somit nicht multikriteriell.

Ansätze aus der Kategorie *manuelle analytische Verfahren* verbessern die Qualität der Struktur, indem sie vorhandene Strukturprobleme nacheinander entfernen. Diese Verfahren sind nicht multikriteriell, da sie nur ein oder wenige Entwurfsheuristiken gleichzeitig betrachten. Auswirkungen der Restrukturierungen auf andere Systemteile werden nicht betrachtet, was bedeutet, dass die Verfahren nicht garantieren können, dass sich die Qualität der Struktur insgesamt verbessert.

Existierende *automatisierte Spezialverfahren* verbessern die Qualität der Struktur automatisch bezüglich einer Entwurfsheuristik, beispielsweise indem sie ein objektorientiertes System in Demeter-Normal-Form überführen [Lie95]. Diese Verfahren berücksichtigen jedoch nicht alle wesentlichen Strukturierungseinheiten und können nur eine Entwurfsheuristik gleichzeitig betrachten, d.h. sie sind nicht multikriteriell.

Gleiches gilt für vorhandene Ansätze zur *Analyse bzw. Verbesserung von Teilsystemzerlegungen.* Diese Ansätze berechnen automatisch eine Zerlegung in Teilsysteme. Mit Hilfe dieser berechneten Zerlegung können Restrukturierungen abgeleitet werden, die die Teilsystemstruktur eines Systems und somit die Qualität der Struktur verbessern. Allerdings lassen diese Verfahren die Klassenstruktur unberücksichtigt und betrachten bei der Berechnung der Qualität der Systemstruktur nur wenige Entwurfsheuristiken.

Existierende *Suchbasierte Verfahren* sind nur in der Lage, die Qualität der Klassenstruktur zu verbessern und können nicht auf reale objektorientierte Systeme angewendet werden.

1.4. Lösungsansatz

Die Bestimmung von Restrukturierungen zur Verbesserung der Qualität der Struktur eines Systems ist ein Suchproblem. Aus der Menge der Strukturen von Systemen mit gleichem Ein- /Ausgabeverhalten suchen wir die Systemstruktur mit der höchsten Qualität. Die Qualität einer Systemstruktur können wir mit Metriken messen. Die Struktur können wir verändern, indem wir die Zuordnung von Klassen zu Teilsystemen und von Methoden und Attributen zu Klassen modifizieren. Dieses Problem können wir auf das NP-vollständige Problem, eine bezüglich einer Zielfunktion optimale Zerlegung von Elementen zu bestimmen, zurückführen.

Um eine Näherungslösung zu bestimmen, verwenden wir einen evolutionären Algorithmus. Evolutionäre Algorithmen haben gegenüber anderen Suchverfahren den Vorteil, dass sie eine Population von Lösungen betrachten und den Suchraum somit besser durchqueren können. Das Grundprinzip besteht darin, dass wir Restrukturierungen auf einem Modell des Systems zufallsgesteuert simulieren und anhand einer Zielfunktion entscheiden, ob eine Restrukturierung gut oder schlecht ist. Die Zielfunktion besteht aus einer Reihe von Metriken, die Entwurfsheuristiken überprüfen.

Unser Verfahren läuft folgendermaßen ab. Zunächst kann ein Softwareingenieur durch Parameter in der Zielfunktion festlegen, welche Metriken und damit welche Entwurfsheuristiken für ihn wichtig sind und welche nicht. Weiterhin kann er Systemteile vom Verfahren ausschließen, da er z.B. weiß, dass in diesen Teilen in naher Zukunft keine Änderungen zu erwarten sind.

Als nächstes überführen wir den Quelltext eines Systems in ein Modell. Diesen Schritt bezeichnen wir als Faktenextraktion. Im nächsten Schritt - der Klassifikation - suchen wir nach Strukturen in unserem Modell, die Entwurfsheuristiken bewusst verletzen. Diese Strukturen markieren wir und schließen sie von unserem Verfahren aus.

Der nächste Schritt unseres Verfahrens ist die Ausführung des evolutionären Algorithmus. Als erstes erzeugen wir die Startpopulation: Das in der Faktenextraktion erstellte Modell kopieren wir n-fach, da wir während der Optimierung mit einer Population von Lösungen arbeiten. Diese Modelle verändern wir mit Modellrestrukturierungen. Modellrestrukturierungen können auf das ursprüngliche System übertragen werden und garantieren, dass sie das beobachtbare Ein-/Ausgabeverhalten des Systems nicht verändern. Entweder wenden wir nur eine Modellrestrukturierung an, d.h. wir führen eine Mutation aus, oder wir kombinieren bereits durchgeführte Modellrestrukturierungen zweier Modelle, d.h. wir führen eine Rekombination aus. Auf diese Weise entstehen neue Modelle. Um die Population wieder auf die ursprüngliche Größe zu reduzieren, wählen wir mit Hilfe der Zielfunktion die Modelle aus, deren Qualität der Struktur am besten ist. Das Erzeugen neuer Modelle und die Auswahl der besten Modelle fassen wir unter dem Begriff Evolutionsschritt zusammen. Nach einer vorgegebenen Anzahl von Evolutionsschritten

terminiert der evolutionäre Algorithmus. Das bezüglich der Zielfunktion beste Modell, inklusive der Modellrestrukturierungen, die das ursprüngliche Modell in das aktuelle Modell überführen, wird dem Softwareingenieur präsentiert.

Ein Softwareingenieur kann die von unserem Verfahren berechneten Restrukturierungen automatisch mit einem Werkzeug ausführen. Die Liste der Restrukturierungen enthält alle Informationen, die er bei gängigen Werkzeugen selbst zur Verfügung stellen muss. Gegebenenfalls kann er aussagekräftigere Namen vergeben. Da unser Verfahren nicht den Quelltext eines Systems verändert, hat der Softwareingenieur die Gelegenheit, die Restrukturierungen zu prüfen. Stimmt er nicht allen Restrukturierungen zu, so kann er die Parameter des Verfahrens verändern und das Verfahren bei Bedarf erneut ausführen lassen.

Das Verfahren liegt in einer in Java geschriebenen Implementierung vor. Für die Sprache *Java* können wir unser Modell vollständig aufbauen. Unser Faktenextraktor für *C++* liefert nicht alle nötigen Details, so dass wir für in *C++* geschriebene Systeme nur die Teilsystemstruktur verbessern können. Für mehrere Open-Source-Softwaresysteme und ein am FZI entwickeltes System konnten Restrukturierungen bestimmt werden, die die Qualität der Struktur verbesserten und zukünftige Änderungen vereinfachten.

Die vorliegende Arbeit stellt eine Verallgemeinerung des existierenden Stands der Technik dar. Zum einen berücksichtigt unser Verfahren im Vergleich zu existierenden Verfahren mehr Entwurfsheuristiken und mehr Restrukturierungen gleichzeitig. Zum anderen gelingt es uns mit der vorliegenden Arbeit zu zeigen, dass ein evolutionärer Algorithmus auch zur Verbesserung von Klassenstrukturen realer Systeme eingesetzt werden kann.

1.5. Aufbau der Arbeit

Die Arbeit gliedert sich wie folgt: In Kapitel 2 gehen wir genauer auf die Struktur von Softwaresystemen ein und stellen grundlegende Verfahren zur Bewertung der Qualität von Systemstrukturen vor. Wir beschreiben ebenfalls die Grundzüge evolutionärer Algorithmen, die wir in unserer Arbeit als Basistechnik verwenden. In Kapitel 3 prüfen wir den Stand der Technik und zeigen, dass kein existierendes Verfahren unser Ziel erreicht. In Kapitel 4 stellen wir unseren Lösungsansatz im Überblick vor, bevor wir die Bestandteile des Ansatzes in den Folgekapiteln genauer beschreiben. In Kapitel 5 stellen wir zuerst unser entwickeltes Metamodell vor. Am Ende dieses Kapitels beschreiben wir, wie wir Teilstrukturen klassifizieren. In Kapitel 6 beschreiben wir die Ausgestaltung des von uns verwendeten evolutionären Algorithmus. Wir erläutern die von uns gewählte Repräsentation, die Mutations- und Rekombinationsoperatoren, unsere Zielfunktion und unsere Selektionsstrategie. Kapitel 7 enthält die Evaluation unseres Verfahrens anhand von fünf Fallstudien. In Kapitel 8 geben wir eine kurze Zusammenfassung und einen Ausblick auf weiterführende Fragestellungen.

Kapitel 2.

Grundlagen

In diesem Kapitel stellen wir Grundlagen und Begriffe vor, auf die wir im weiteren Verlauf dieser Arbeit immer wieder zurückgreifen werden, da sie das Fundament unserer Arbeit bilden.

Zuerst erklären wir, wie die Struktur objektorientierter Systeme definiert ist. Wir geben Prinzipien an, nach denen die Struktur solcher Systeme entworfen werden soll und beschreiben im Anschluss daran, wie die Qualität der Struktur im Sinne eines Einhaltens dieser Prinzipien werkzeuggestützt bestimmt werden kann. Im nächsten Abschnitt beschreiben wir Strukturen, die Entwurfsheuristiken bewusst verletzen und gehen anschließend auf die Rolle der Struktur während der Wartung von Softwaresystemen ein.

Als nächstes stellen wir Restrukturierungen als grundlegende Technik zur Veränderung und somit zur Verbesserung der Systemstruktur vor. Zuletzt beschreiben wir die Grundlagen evolutionärer Algorithmen, die wir in unserem Ansatz als Basistechnik verwenden.

2.1. Die Struktur objektorientierter Systeme

Ein *System* stellt einen Ausschnitt der realen oder gedachten Welt - der Anwendungsdomäne - dar. Es besteht aus mehreren Komponenten, die in Beziehung zueinander stehen [Goo96]. Die Komponenten des Systems nennen wir *Teilsysteme*, elementare Teilsysteme heißen *Objekte*. Der Systembegriff ist rekursiv, d.h die Teilsysteme des Systems können wiederum aus mehreren Teilsystemen bestehen.

Die Beziehungen zwischen Teilsystemen sind in objektorientierten Systemen sowohl statischer als auch dynamischer Natur. Die dynamischen Beziehungen erfassen wir mit dem Begriff Kooperation im System, sie werden im Rahmen dieser Arbeit allerdings nicht näher betrachtet. In dieser Arbeit sind ausschließlich die statischen Beziehungen von Interesse. Wir konzentrieren uns auf ausdrucksbasierte, stark typisierte Sprachen. In diesen Sprachen können dynamische Beziehungen nur dann entstehen, wenn bereits statische Beziehungen vorliegen. Aussagen über den statischen Aufbau eines Systems aus Teilsystemen und Objekten nennen wir zusammenfassend ein Objektmodell.

Ein Objekt ist ein elementares Teilsystem, das einen beliebigen Gegenstand repräsentiert und messbare, durch Werte erfassbare Eigenschaften besitzt. Diese Eigenschaften sind die Attribute des Objekts. Ferner kann ein Objekt Dienste ausführen, zu deren

Ausführung es mehrere Handlungsvorschriften enthält: die Methoden. Zusammen mit seinen Attributen bilden die ausführbaren Methoden die Merkmale eines Objekts.

Objekte mit gleichen Merkmalen und gleichem Verhalten gehören zur gleichen *Klasse*. Objekte einer Klasse bezeichnen wir auch als Instanzen oder Ausprägungen der Klasse. Technisch gesehen gibt eine Klasse das Baumuster ihrer Objekte wieder. Eine Klasse repräsentiert also das Konzept oder den abstrakten Begriff, den wir uns von ihren Objekten machen.

Da wir in dieser Arbeit nur statische Beziehungen betrachten, sind die Objekte für uns nicht von Interesse. Wir betrachten und verändern nicht die Objekte, sondern die Klassen, die die Baupläne für die Objekte bereitstellen.

In dieser Arbeit definieren wir die Struktur eines objektorientierten Systems als die Menge aller Teilsysteme und ihrer statisch beobachtbaren Beziehungen untereinander. Diese Definition von Struktur entspricht dem *Programmieren-im-Großen* [DK75]. *Programmieren-im-Kleinen* beschäftigt sich im Gegensatz dazu mit der Ausformulierung einzelner Methoden.

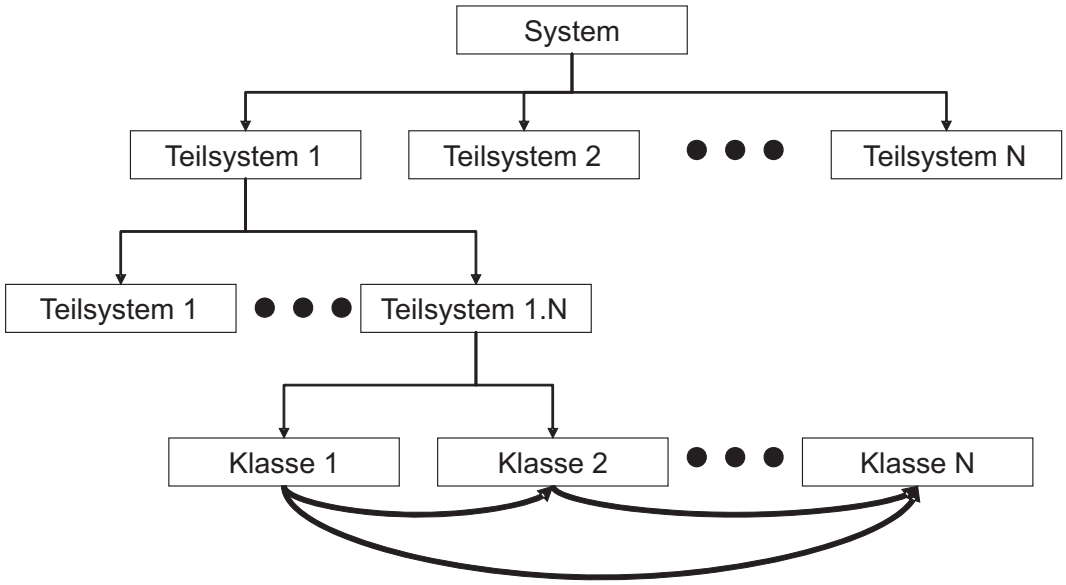


Abbildung 2.1.: Grundsätzliche Struktur objektorientierter Systeme

Abbildung 2.1 veranschaulicht diese Struktur. Auf oberster Ebene befindet sich das System selbst, das aus einer Menge von Teilsystemen besteht. Diese Teilsysteme bestehen wiederum aus mehreren Teilsystemen, bis auf unterster Ebene die Klassen die Objekte als elementare Teilsysteme bestimmen. Zwischen den Klassen existieren statisch beobachtbare Abhängigkeiten wie z.B. Vererbungsbeziehungen.

2.2. Qualität der Systemstruktur

Es gibt mehrere Prinzipien, die die Qualität der Struktur eines objektorientierten Systems bestimmen. Ein Architekt einer solchen Struktur sollte diese Prinzipien berück-

sichtigen, um ein wartbares System zu konstruieren. Diese Prinzipien existieren schon seit vielen Jahren, z.B. das in den Arbeiten von Parnas [Par72] in den 70er Jahren geprägte *Geheimnisprinzip*. Trotzdem kennen viele Architekten diese Prinzipien nicht, oder befolgen sie absichtlich oder unbeabsichtigt nicht. Die drei nachfolgend beschriebenen Prinzipien gelten für den Entwurf von allen Arten von Teilsystemen, also sowohl für Klassen als auch für Teilsysteme, die aus mehreren Klassen bestehen.

2.2.1. Geheimnisprinzip und Schnittstellen

Das Geheimnisprinzip [Par72] besagt, dass jedes Teilsystem genau eine bestimmte Verantwortlichkeit besitzen soll. Ein Teilsystem soll dadurch charakterisiert sein, dass es eine Entwurfsentscheidung enthält, die es vor den anderen Teilsystemen geheim hält. Es soll dieses Geheimnis hinter einer wohldefinierten Schnittstelle verstecken, die das Teilsystem von allen anderen Teilsystemen abgrenzt. Die Schnittstelle besteht aus zwei Dingen: Zum einen aus dem Verzeichnis der Signaturen der Funktionen, die das Teilsystem nach außen hin anbietet. Zum anderen aus einer Menge von Verträgen. Für jede Funktion beschreibt ein Vertrag Vor- und Nachbedingungen. Vorbedingungen legen fest, welche Zusicherungen ein Aufrufer einhalten muss, Nachbedingungen beschreiben, welche Zusicherungen der Aufgerufene einhalten wird [Mey88]. Ein Softwareingenieur kann hinter der Schnittstelle Änderungen vornehmen, die keinen Einfluss auf die Umgebung des Teilsystems haben. Dies setzt allerdings voraus, dass die existierenden funktionalen und nichtfunktionalen Verträge der Schnittstelle dabei nicht verletzt werden.

Parnas spricht in seinem Papier nicht von Teilsystemen, sondern von Modulen. Gemäß [Goo96] heißt in imperativen und objektorientierten Sprachen ein Programmstück Modul, wenn es aus logisch zusammengehörigen Vereinbarungen besteht und von außen nur über eine genau festgelegte Schnittstelle angesprochen werden kann. Das Programmstück soll eine syntaktische Einheit im Sinne der Programmiersprache oder sogar eine getrennt übersetzbare Programmeinheit sein. Ein Modul ist somit nichts anderes als ein Teilsystem mit einer klaren Grenze, der Schnittstelle zu seiner Umgebung.

Dies bedeutet für objektorientierte Systeme, dass jedes Teilsystem und somit auch jede Klasse das Geheimnisprinzip befolgen sollte. Die Schnittstelle einer Klasse besteht aus den öffentlich sichtbaren Signaturen von Funktionen. Die Schnittstelle eines Teilsystems bestehend aus mehreren Klassen liegt häufig in Form einer Fassadenklasse vor [GHJV94].

2.2.2. Kopplung und Kohäsion

Die Kohäsion misst die Abgeschlossenheit der Beziehungen innerhalb eines Teilsystems [Som96]. Ein Teilsystem sollte eine einzelne, logisch zusammengehörige Funktion oder eine logisch zusammengehörige Einheit implementieren. Alle Bestandteile des Teilsystems sollten zu dieser Implementierung beitragen. Die Kohäsion eines Teilsystems ist niedrig, falls es Bestandteile enthält, die nicht direkt zu dieser Implementierung beitragen. Hohe Kohäsion ist ein wichtiges Kriterium für den Aufbau eines Teilsystems, da ein Teilsystem mit hoher Kohäsion einen einzelnen Bestandteil einer Problemlösung darstellt. Falls dieser Bestandteil verändert werden muss, muss nur das zugehörige Teilsystem angepasst

werden. Die Änderungen erfordern keine Anpassungen weiterer Teilsysteme.

Die Kopplung ist mit der Kohäsion verwandt, sie ist ein Indikator für die Stärke der Beziehungen zwischen den einzelnen Teilsystemen einer Struktur [Som96]. Stark gekoppelte Systeme besitzen starke Beziehungen zwischen den Teilsystemen, d.h. die Teilsysteme hängen stark voneinander ab. Schwach gekoppelte Systeme bestehen aus Teilsystemen, die nahezu unabhängig voneinander sind. Im allgemeinen sind Teilsysteme stark aneinander gekoppelt, falls sie gemeinsame Variablen benutzen, oder Kontrollflussinformation austauschen.

2.2.3. Komplexität

Die einzelnen Teile eines Systems müssen möglichst leicht zu verstehen sein, wenn eine Zerlegung in Teilsysteme das Verständnis des Systems erleichtern soll. Ein Teilsystem ist zum einen leicht zu verstehen, wenn es nur eine schwache Kopplung zu anderen Teilsystemen aufweist, und zum anderen wenn die Komplexität innerhalb des Teilsystems nicht zu hoch ist. Komplex ist ein Teilsystem dann, wenn es viele weitere Teilsysteme mit vielen Beziehungen untereinander enthält.

Die Komplexität des Gesamtsystems kann nicht reduziert werden, sie kann nur eventuell besser verteilt werden. Innerhalb eines Teilsystems kann die Komplexität also nur dadurch verringert werden, dass manche Bausteine des Teilsystems zu einem weiteren Teilsystem zusammengefasst werden, und mit einer Abstraktion in Form einer Schnittstelle versehen werden.

Der hier verwendete Komplexitätsbegriff stammt aus der Erkenntnis, dass das menschliche Gehirn mit Strukturen, die zahlreiche miteinander verknüpfte Elemente enthalten, schlecht zurecht kommt.

2.3. Werkzeuggestützte Bewertung der Qualität der Systemstruktur

Es existiert bereits eine Reihe von Verfahren und Werkzeuge zur Bewertung der Qualität von Systemstrukturen [Ciu01][Mar01]. Diese Verfahren überprüfen die Einhaltung der Entwurfsprinzipien aus Abschnitt 2.2, indem sie die Prinzipien auf Entwurfsheuristiken abbilden und sie mit Metriken messen.

Zur Überprüfung der Entwurfsheuristiken beschreiben die Verfahren meist Strukturen, die die Entwurfsprinzipien verletzen und nicht Strukturen, die sich an diese Prinzipien halten. Somit wird eigentlich nicht die Qualität der Struktur sondern ihre Abwesenheit ermittelt. Diese Vorgehensweise hat den Vorteil, dass ein Softwareingenieur sehr einfach die Stellen im System identifizieren kann, an denen er die Qualität der Struktur verbessern kann. Um die Qualität der Struktur zu messen, können mit Hilfe dieser Verletzungen ebenfalls Metriken formuliert werden. Die Qualität ist umso besser, je weniger Verletzungen darin enthalten sind.

Es kann Strukturen geben, die gegen Entwurfsheuristiken verstoßen, aber kein Problem für die Wartbarkeit des Systems darstellen. In diesem Fall sprechen wir von **Struk-**

turanomalien. Strukturen, die gegen Entwurfsheuristiken verstoßen und einen negativen Einfluss auf die Wartbarkeit eines Systems haben, nennen wir analog zu [Ciu01] **Strukturprobleme**.

Beispiel 1: *Lange Parameterliste*
Um Strukturen zu identifizieren, die gegen das Entwurfsprinzip *ausgeglichene Komplexität* verstoßen, kann die Entwurfsheuristik *Lange Parameterliste* verwendet werden. Die Entwurfsheuristik besagt, dass eine Methode nicht zu viele Parameter besitzen darf. Eine hohe Anzahl von Parametern ist ein Anzeichen für eine zu hohe Komplexität der Methode.

Alle vorhandenen Werkzeuge, die Strukturprobleme identifizieren können, arbeiten nach folgendem Schema: Zunächst extrahieren sie aus dem Quelltext eines Systems die benötigten Fakten. Hierzu verwenden sie die aus üblichen Übersetzungsvorgängen bekannten Schritte lexikalische, syntaktische und semantische Analyse. Das Ergebnis der Faktenextraktion ist ein Systemmodell des Programms. Ein Systemmodell enthält Informationen über die im Quelltext existierenden Elemente und die statisch beobachtbaren Beziehungen zwischen diesen Elementen. Es enthält also zum Beispiel die Methoden des Systems und die Aufrufbeziehungen zwischen diesen Methoden. Der Inhalt eines Systemmodells wird durch ein Metamodell festgelegt. Dieses definiert, wie detailliert ein Systemmodell sein soll, also z.B. ob Kontrollflusselemente innerhalb von Methoden modelliert werden.
Auf Basis des Systemmodells führen die Werkzeuge zwei grundlegende Arten von Analysen aus:

1. Maße: Beispiele für Maße sind Lines of Code (LOC) und Tight Class Cohesion (TCC) [Mar01].

2. Graphmustersuchen: Das Systemmodell wird als Graph interpretiert, in dem nach bestimmten Teilgraphen gesucht wird. Ein klassisches Beispiel für solche Teilgraphen sind direkte zyklische Abhängigkeiten zwischen Methoden, die durch das Graphmuster Methode A ruft Methode B und umgekehrt identifiziert werden können.

Mit Hilfe von Maßen und Graphmustersuchen können komplexere Regeln erstellt werden, die Entwurfsheuristiken überprüfen. Eine Regel enthält dabei eine Interpretationsvorgabe für die verwendeten Maße in Form von Schwellwerten. Diese Regeln identifizieren Strukturanomalien, die gegebenenfalls von einem Softwareingenieur als Strukturproblem bestätigt werden müssen.

Beispiel 2: *Oberklasse kennt Unterklasse*
Die Entwurfsregel *Oberklasse kennt Unterklasse* besagt, dass eine Oberklasse keine ihrer Unterklassen auf irgendeine Art und Weise verwenden darf. Diese Regel kann mit Hilfe einer Graphmustersuche überprüft werden. Dabei werden die Klassen eines Systems als Knoten modelliert, und die Abhängigkeiten zwischen den Klassen als Kanten.

Gesucht wird nach Teilgraphen, bei denen erstens zwischen einem Klassenknoten B und einem Klassenknoten A eine Vererbungskante existiert und zweitens zwischen dem Klassenknoten A und dem Klassenknoten B eine weitere Abhängigkeitskante beliebigen Typs existiert. Abbildung 2.2 zeigt ein Beispiel für ein solches Graphmuster.

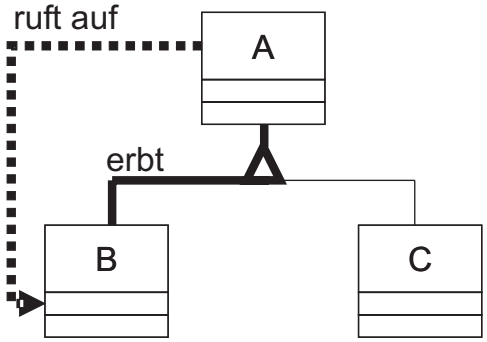


Abbildung 2.2.: Entwurfsregel *Oberklasse kennt Unterklasse*

2.4. Strukturen, die bewusst Entwurfsheuristiken verletzen

Wie wir im vorherigen Abschnitt beschrieben haben, liefern existierende Verfahren zur Erkennung von Strukturproblemen oft nur Strukturanomalien, die als tatsächliches Problem bestätigt werden müssen. Viele Entwurfsmuster werden als Strukturanomalien erkannt, da der Aufbau ihrer Struktur vorhandene Entwurfsheuristiken bewusst verletzt. Entwurfsmuster beschreiben eine allgemeine Lösung eines immer wiederkehrenden Entwurfsproblems, sie stellen somit prinzipiell sicher kein Strukturproblem dar. Entwurfsmuster können allerdings falsch angewendet werden, diese falschen Verwendungen aufzudecken liegt allerdings nicht im Fokus dieser Arbeit. Entwurfsmuster müssen bei einer werkzeuggestützten Qualitätsbewertung möglichst automatisch ausgeblendet werden. Wir stellen im folgenden zwei Beispiele für Entwurfsmuster vor, um diese Tatsache zu verdeutlichen.

Beispiel 3: *Fassade*

Das Entwurfsmuster Fassade ist ein Beispiel für eine Teilstruktur, die von Werkzeugen als Strukturanomalie erkannt wird. Eine Fassade hat das Ziel, eine einheitliche Schnittstelle für eine Menge von Schnittstellen eines Teilsystems zur Verfügung zu stellen. Sie bietet eine höherwertigere Schnittstelle an, die die Nutzung eines Teilsystems vereinfacht.

Eine Fassade wird deshalb von vielen Klienten benutzt, die auf die Funktionalität des Teilsystems angewiesen sind. Sie selbst verwendet viele Klassen des Teilsystems, die die Wünsche ihrer Nutzer erfüllen können. Eine Fassade stellt somit einen Flaschenhals [Ciu01] für Änderungen dar. Dieser Flaschenhals lässt sich gut an der Struktur der Abhängigkeiten ablesen, wie sie in Abbildung 2.3 beispielhaft dargestellt ist.

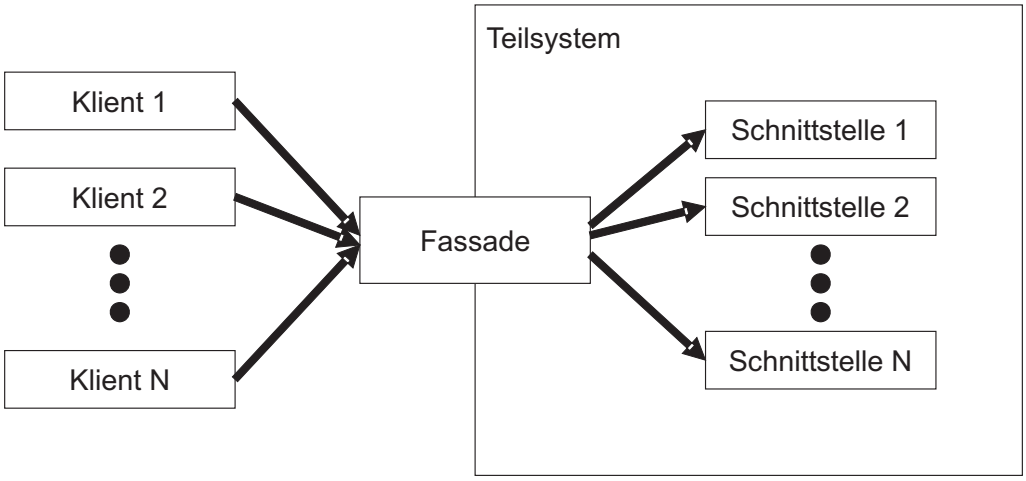


Abbildung 2.3.: Abhängigkeitsstruktur einer Fassade

Änderungen an der Fassade sind wahrscheinlich, da diese von vielen Schnittstellen des Teilsystems abhängt. Die Änderungen an der Fassade werden wiederum an viele Klienten weitergegeben, die von der Fassade abhängen. Die Entwickler des Teilsystems nehmen dies jedoch in Kauf, da durch die Fassade ein Teilsystem vom Rest des Systems gekapselt werden kann. Eine gute Kapselung ist für die Entwickler wichtiger, als einen Flaschenhals zu vermeiden.

Beispiel 4: *Besucher*

Das Entwurfsmuster Besucher ist ein weiteres Beispiel für eine Strukturanomalie, die gewollt ist und deshalb kein Strukturproblem darstellt. Ein Besucher repräsentiert eine Operation, die auf den Elementen einer Objektstruktur ausgeführt werden soll. Insbesondere können mit einem Besucher neue Operationen definiert werden, ohne die Klassen der Elemente, auf denen er arbeitet, zu verändern. Die allgemeine statische Struktur eines Besuchers ist in Abbildung 2.4 zu sehen.

Deutlich erkennen wir die zyklischen Abhängigkeiten zwischen den Klassen *Besucher* und *ElementA* und den Klassen *Besucher* und *ElementB*. Allgemein sollten zyklische Abhängigkeiten vermieden werden, im Falle des Besuchermusters sind sie allerdings unvermeidlich.

2.5. Die Rolle der Struktur während der Wartung von Softwaresystemen

Die IEEE definiert den Begriff Softwarewartung folgendermaßen [iee93]: Die Änderung eines Softwareprodukts nach seiner Auslieferung mit dem Ziel, Fehler zu beseitigen, die Leistung oder andere Eigenschaften zu verbessern oder das Produkt an eine geänderte Umgebung anzupassen.

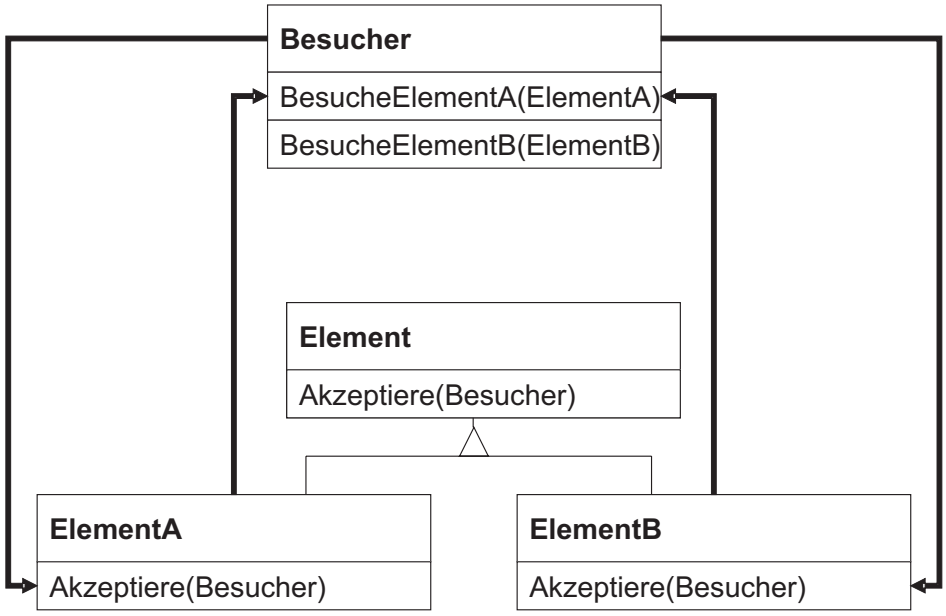


Abbildung 2.4.: Abhängigkeitsstruktur eines Besuchers

In einer anderen Quelle wird der Begriff Softwarewartung in folgende vier Teilbereiche aufgeschlüsselt [LS80]:

- Korrektive Wartung: Im Rahmen der korrektiven Wartung werden Defekte beseitigt und Anforderungen eingearbeitet, die sich erst während des Systemeinsatzes ergeben.
- Adaptive Wartung: Alle Tätigkeiten, die aufgrund geänderter Umgebungsparameter erfolgen, werden unter dem Begriff adaptive Wartung zusammengefasst. Hierzu zählen z.B. Betriebssystemwechsel oder geänderte Geschäftsprozesse.
- Perfektive Wartung: Unter perfektiver Wartung werden Tätigkeiten verstanden, die die für ein Softwaresystem geforderten Eigenschaften verbessern, wie z.B. einfachere Bedienung.
- Vorbeugende Wartung: Unter dem Begriff vorbeugende Wartung werden alle Tätigkeiten zusammengefasst, die zukünftige Probleme verhindern sollen.

In beiden Fällen geht das Verständnis von Wartung über das sonst übliche Verständnis hinaus, in dem man unter dem Begriff Wartung ausschließlich funktionserhaltende Maßnahmen zusammenfasst. Viele Forscher verwenden deshalb synonym zu dem Begriff Softwarewartung den Begriff Softwareevolution, da Softwaresysteme auch funktional weiterentwickelt werden.

Heutzutage sind die meisten Softwareprojekte keine Neuentwicklungen, sondern Wartungsprojekte. Abbildung 2.5 zeigt diesen Sachverhalt anhand der Priorität der strategischen Ausrichtung von Softwareprojekten [IDC01].

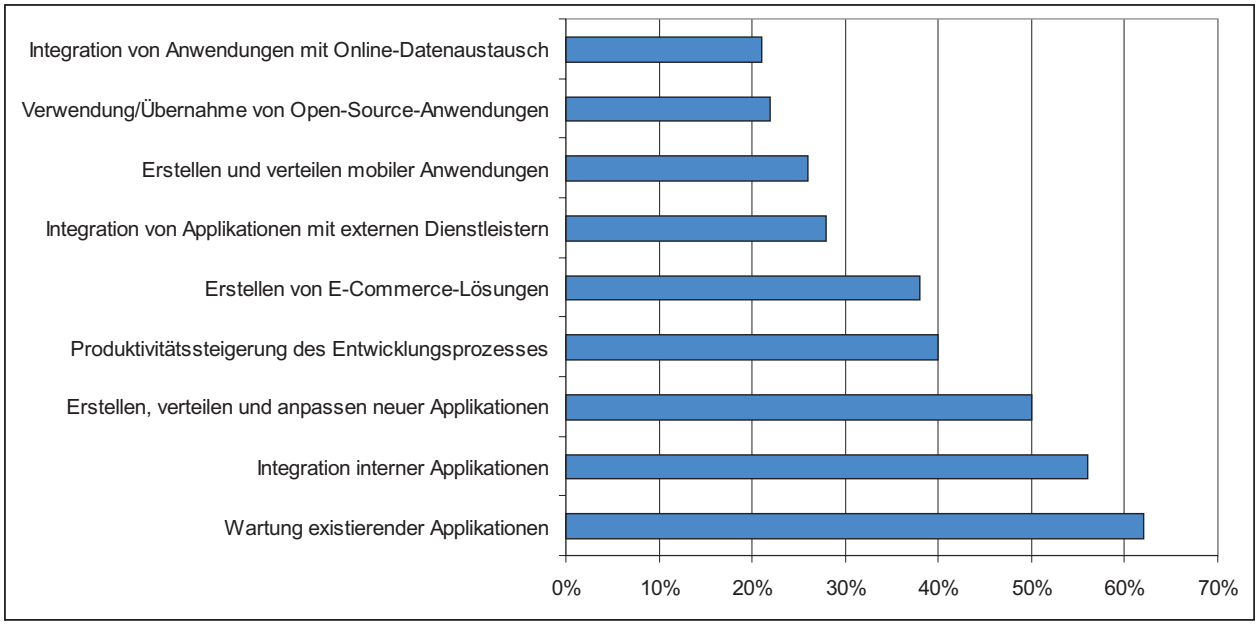


Abbildung 2.5.: Priorität der strategischen Ausrichtung von Softwareprojekten

Softwaresysteme können nicht immer von Grund auf neu entwickelt werden, da in den bereits existierenden Softwaresystemen sehr viel geschäftskritisches Wissen enthalten ist, das außerhalb des Quelltextes nicht in vollem Umfang dokumentiert ist.

Allerdings beobachteten schon Lehman und Belady, dass ein Softwaresystem zunehmend nutzlos wird, wenn es einen Teil der realen Welt widerspiegelt und nicht ständig angepasst wird [LB85].

Da Softwaresysteme also ständig angepasst werden müssen, ist es nicht erstaunlich, dass bezogen auf die Gesamtkosten eines Softwaresystems die Wartungskosten den größten Anteil haben. Diesen Sachverhalt beobachten wir in Abbildung 2.6[1]. Pro Jahr ist für eine Reihe von Softwareprojekten das Projekt mit dem minimalen und das Projekt mit dem maximalen Anteil an Wartungskosten dargestellt. Wir können erkennen, das selbst im besten Fall über 50 % der Kosten während der Wartung verursacht wurden.

Fenton hat festgestellt [FP97], dass zwischen den Wartungsaufwänden und der inneren Qualität eines Softwareystems ein Zusammenhang besteht. Die innere Qualität eines Softwaresystems ist dabei die Qualität, die ein Softwareingenieur wahrnimmt. Im Gegensatz dazu steht die äußere Softwarequalität als Qualität, die ein Nutzer eines Softwaresystems wahrnimmt. Innere Qualität äußert sich in Qualitätseigenschaften wie z.B. Analysierbarkeit oder Modifizierbarkeit, während äußere Qualität in Qualitätseigenschaften wie Benutzbarkeit und Korrektheit verfeinert werden kann.

Die innere Qualität eines Softwaresystems kann mit Hilfe der in Abschnitt 2.3 beschriebenen Verfahren bestimmt werden, da die Struktur eines Softwaresystems dessen

[1]Quelle: University of Jyväskylä, Finnland; http://www.cs.jyu.fi/ koskinen/smcosts.htm

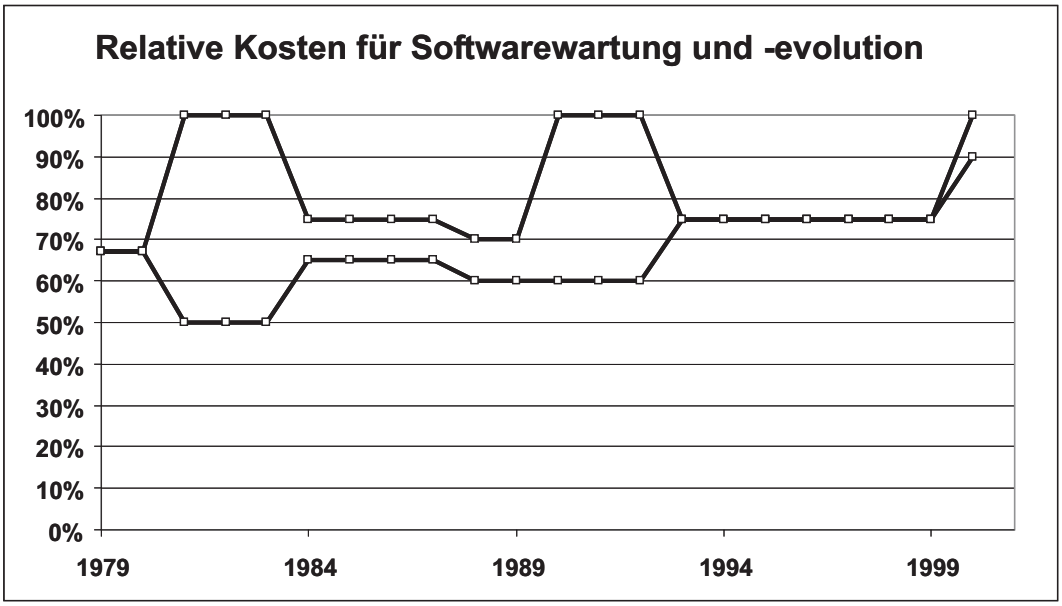


Abbildung 2.6.: Relative Kosten für Softwarewartung und Evolution

innere Qualität maßgeblich bestimmt. Wie Lehman in seinem zweiten Gesetz aber schon festgestellt hat, nimmt die Unordnung in einem System und damit in seiner Struktur im Laufe der Zeit zu, wenn nicht bestimmte Tätigkeiten ausgeführt werden, um dieser Unordnung entgegen zu wirken [LB85].

Gründe für den strukturellen Verfall eines Softwaresystems gibt es wie in Kapitel 1 bereits erwähnt viele. Erstens müssen Softwaresysteme meist unter hohem Zeit- und Kostendruck angepasst werden, so dass die Struktur immer weniger auf die neuen Anpassungen ausgerichtet ist. Zweitens ist das Wissen um gute Strukturen nicht bei allen Entwicklern verbreitet, so dass schlechte Strukturen unbewusst entstehen können. Drittens machen Menschen einfach Fehler, so dass schlechte Strukturen unbeabsichtigt entstehen.

Softwaresysteme müssen also immer wieder auf Strukturprobleme hin untersucht werden, und es müssen Maßnahmen ergriffen werden, deren Ziel die Entfernung dieser Strukturprobleme ist. Bleiben diese Maßnahmen aus, steigen die zur Wartung eines Softwaresystems nötigen Aufwände unaufhaltsam an. Thema des nächsten Abschnitts ist, wie die Struktur von Softwaresystemen verändert werden kann.

2.6. Veränderung der Struktur von Softwaresystemen

Die Struktur von Softwaresystemen kann mit Hilfe von Restrukturierungen verändert werden, ohne das beobachtbare Verhalten zu verändern. Der Begriff *beobachtbares Verhalten* ist dabei als das beobachtbare Ein-/Ausgabeverhalten definiert. Eine gleiche Eingabe muss auf dem Orginalsystem und dem restrukturierten System zur gleichen Ausgabe führen.

Viele Softwareentwicklungsprozesse verwenden Restrukturierungen als festen Prozessbestandteil. Eine nachträgliche Änderung der Systemstruktur wird als Notwendigkeit und nicht als Fehler angesehen, da nicht alle Anforderungen von Beginn an berücksichtigt werden können. Extreme Programming [Bec99] ist der wohl prominenteste Softwareentwicklungsprozess, der Restrukturierungen als festen Prozessschritt einsetzt.

Die Ursprünge des Begriffs Restrukturierung gehen auf die Arbeiten von Opdyke [Opd92] und Fowler [Fow99] zurück. Restrukturierungen sind atomare Quelltexttransformationen, die zu komplexeren Quelltexttransformationen zusammengesetzt werden können. Zusammengesetzte Restrukturierungen werden z.B. benötigt, um nachträglich Entwurfsmuster in existierende Systeme einzubauen.

Ein Softwareingenieur kann Restrukturierungen einsetzen, um die Qualität einer Systemstruktur zu verbessern. In Zusammenhang mit dem Begriff Restrukturierung ist in der Fachliteratur häufig von *schlechten Gerüchen*[2] die Rede. Unter einem schlechten Geruch wird ein Hinweis auf ein Strukturproblem verstanden. In der ursprünglichen Literatur zu Restrukturierungen gehen die Autoren davon aus, dass schlechte Gerüche manuell bei der Durchsicht des Quelltextes identifiziert werden müssen. Strukturprobleme und somit auch schlechte Gerüche können aber wie in Abschnitt 2.3 beschrieben auch werkzeuggestützt identifiziert werden.

Moderne Entwicklungsumgebungen wie z.B. Eclipse[3] bieten werkzeuggestützte, semiautomatische Restrukturierungen in ihrem Funktionsumfang an. Ein klassisches Beispiel ist das automatische Umbenennen von Programmelementen. Hier passt die Entwicklungsumgebung automatisch alle Verwendungsstellen des Programmelements an. Benennt ein Softwareingenieur also z.B. eine Methode um, so passt die Entwicklungsumgebung automatisch die Verwendungsstellen der Methode an den neuen Namen an. Nicht alle Restrukturierungen können vollautomatisch umgesetzt werden, da ein Nutzer fehlende Zusatzinformationen angeben muss.

Beispiel 5: *Methode verschieben*
Wird z.B. eine Methode *m()* von einer Klasse *A* in eine Klasse *B* verschoben, so muss an den Aufrufstellen von *m()* ein Objekt der Klasse *B* vorhanden sein. Nur der Softwareingenieur kann entscheiden, auf welchem Objekt *m()* aufgerufen werden soll, und wie dieses Objekt an den Verwendungsstellen verfügbar gemacht wird.

2.7. Evolutionäre Algorithmen

Seit Tausenden von Jahren inspirierte die Natur Ingenieure bei der Lösung ihrer Probleme. Dabei stellen sich die Ingenieure immer wieder zwei Fragen:

- Können wir technische Geräte wie z.B. Flugzeugtragflächen optimieren, indem wir die biologische Evolution nachahmen?

[2] englisch: bad smells
[3] http://www.eclipse.org

- Verschafft uns die Nachahmung der Evolution die Ideen, neue und komplexe Lösungen für bekannte, schwierige Probleme zu finden?

Evolutionäre Algorithmen (EA) sind darauf ausgerichtet, diese Fragestellungen zu beantworten. Sie basieren auf sehr einfachen Modellen einer biologischen Evolution und umfassen die Hauptbestandteile eines natürlichen Evolutionsprozesses. In der Informatik werden EAs auf viele Probleme angewendet, die mit Hilfe konventioneller Methoden nicht einfach zu lösen sind, wie z.B. kombinatorische Optimierungsprobleme [BNKF98].

2.7.1. Bestandteile evolutionärer Algorithmen

Die Hauptbestandteile eines evolutionären Algorithmus sind die folgenden:

2.7.1.1. Repräsentation

Die Repräsentation legt fest, wie eine mögliche Lösung eines Problems modelliert wird. Dabei wird oft zwischen Genotyp und Phänotyp einer möglichen Lösung unterschieden. Der *Genotyp* trägt sozusagen die Erbmasse einer möglichen Lösung. Er ermöglicht die Variation einer Lösung, indem er durch die Operatoren Mutation und Rekombination verändert wird. Der *Phänotyp* leitet sich aus der Form des Genotyps ab und repräsentiert die Menge der beobachtbaren Eigenschaften einer möglichen Lösung. Die Qualität einer Lösung kann bestimmt werden, indem der Phänotyp bewertet wird. Auf Menschen bezogen ist der Genotyp die DNA, und der Phänotyp besteht aus den beobachtbaren Eigenschaften des Menschen wie z.B. Größe oder Haarfarbe.

2.7.1.2. Population möglicher Lösungen

EAs arbeiten mit einer Population möglicher Lösungen, um parallel suchen zu können. Die Größe der Population ist ein wichtiger Parameter eines EAs.

2.7.1.3. Erneuernde Operatoren

Erneuernde Operatoren versuchen neue Aspekte des Problems zu berücksichtigen, indem Elemente der Population zufällig verändert werden. Drei Parameter beeinflussen die Wirkung eines solchen Operators. Zunächst bestimmt die *Stärke* des Operators, an wie vielen Stellen ein Operator ein vorhandenes Element verändert. Die *Ausdehnung* des Operators legt fest, wie viele Stellen des Elements gleichzeitig verändert werden und die *Häufigkeit* seiner Anwendung bestimmt, wie oft der Operator im Algorithmus angewandt wird. Erneuernde Operatoren werden oft Mutationen genannt.

2.7.1.4. Erhaltende Operatoren

Erhaltende Operatoren versuchen die bisherigen Ergebnisse eines EAs zu bewahren, indem sie aus zwei existierenden Elementen der Population durch Kombination ihrer Bestandteile ein neues Element erzeugen. Im Idealfall beschleunigt diese Vorgehensweise

den Suchprozess, da die bereits guten Originalelemente zu einem noch besseren Element kombiniert werden. Ein erhaltender Operator wird von zwei Parametern beeinflusst: erstens legt der *Typ* der Rekombination z.B. fest, wie viele Elemente gleichzeitig kombiniert werden. Zweitens legt die *Häufigkeit* analog zu den erneuernden Operatoren fest, wie oft der Operator im Algorithmus angewandt wird. Erhaltende Operatoren werden meist Rekombinationen genannt.

2.7.1.5. Zielfunktion

Die Fitness eines Elements der Population gibt an, wie gut das Element die zu lösende Problemstellung erfüllt. Mit diesem Wert kann der Algorithmus steuern, welche Elemente sich mit höherer und welche Elemente sich mit niedriger Wahrscheinlichkeit fortpflanzen sollen. Die Funktion, die die Fitness berechnet, nennen wir Ziel- oder Fitnessfunktion. Eine standardisierte Zielfunktion ist eine Zielfunktion, die der besten Lösung einen Wert von 0 zuweist. Eine normalisierte Zielfunktion liefert Fitness-Werte, die immer zwischen 0 und 1 liegen.

2.7.1.6. Selektion

Nicht alle durch die Operatoren erzeugten neuen Elemente können beibehalten werden, da die Größe der Population von möglichen Lösungen beschränkt ist. Der Algorithmus muss festlegen, welche Elemente Nachkommen erzeugen dürfen, welche erzeugten Elemente übernommen werden, und welche Elemente dafür verworfen werden. Diesen Schritt nennen wir Selektion, manchmal wird er auch in Anlehnung an die Darwinsche Evolutionstheorie Auslese genannt. Die Selektion kann mit verschiedenen Strategien durchgeführt werden:

2.7.1.6.1. Fitness-proportionale Selektion: Jedes Element der Population erhält eine proportionale Chance, mit der es sich fortpflanzen darf. Die Chance ergibt sich aus der Fitness eines Elements dividiert durch die Summe der Fitness aller Elemente.

2.7.1.6.2. Abschneide-Selektion oder (μ, γ)-Selektion: μ Eltern dürfen γ Nachkommen erzeugen. Aus diesen Nachkommen werden die μ besten ausgewählt. Die $(\mu + \gamma)$-Selektion ist eine Variation der (μ, γ)-Selektion, bei der die Eltern in die Auswahl der überlebenden Elemente einbezogen werden.

2.7.1.6.3. Rangbasierte Selektion: Die rangbasierte Selektion macht sich die durch die Fitness der Elemente mögliche Sortierung der Elemente zu nutze. Jedes Element erhält somit einen Rang in der Population. Je höher der Rang ist, desto wahrscheinlicher ist es, dass das Element überlebt.

2.7.1.6.4. Turnierselektion: Die Selektion ist hier auf eine Teilmenge der Population beschränkt. Innerhalb einer Teilmenge einer bestimmten Größe werden die Elemente mit der besten Fitness ausgewählt. Die Teilmengen werden dabei zufällig bestimmt. Das

kleinste mögliche Turnier besteht also aus zwei Elementen. Je größer die Anzahl der Elemente ist, desto größer ist der Selektionsdruck.

Heutzutage ist die Turnierselektion die am weitesten verbreitetste Selektionsstrategie. Da die Selektion nur innerhalb einer Teilmenge der Population erfolgt, kann sie erstens gut parallelisiert werden und zweitens verhindern, dass ausschließlich die besten Elemente überleben. Somit kann sich die Fitness von manchen Elementen während der Evolution wieder verschlechtern. Diese Eigenschaft ist wünschenswert, da so bei der Suche nach Lösungen lokale Extrema besser umgangen werden können. Wenn die Eltern bei der Selektion berücksichtigt werden, ist die Turnierselektion die einzige Möglichkeit, um Verschlechterungen tolerieren zu können. Setzt sich die neue Population nur aus erzeugten Elementen zusammen, sorgt Turnierselektion mit einer höheren Wahrscheinlichkeit als z.B. rangbasierte Selektion dafür, dass schlechtere Elemente überleben.

2.7.2. Der evolutionäre Grundalgorithmus

In Abbildung 2.7 ist die Grundform eines evolutionären Algorithmus dargestellt. Zunächst erzeugt der Algorithmus zufällig eine initiale Population von Lösungen. Diese Lösungen werden dann evaluiert, d.h. mit Hilfe einer Zielfunktion wird ihnen ein Wert zugewiesen, der ihre Qualität quantifiziert. Anschließend können während der Selektion die Elemente ausgewählt werden, die beibehalten werden sollen. Der Algorithmus terminiert nun, falls die Population von Lösungen eine ausreichend gute Lösung enthält. Anderenfalls erzeugen die Mutations- und die Rekombinationsoperatoren Varianten der existierenden Lösungen und der Algorithmus führt anschließend erneut die Evaluation durch.

2.7.3. Ausprägungen evolutionärer Algorithmen

Evolutionäre Algorithmen existieren in vielen verschiedenen Ausprägungen. Die vier bekanntesten Ausprägungen werden im Folgenden kurz vorgestellt.

2.7.3.1. Genetische Algorithmen

Goldberg [Hol92] verwendete als erster den Begriff *Genetischer Algorithmus*. Ein genetischer Algorithmus ist eine Spielart eines evolutionären Algorithmus, der zwischen Geno- und Phänotyp stark unterscheidet. Traditionell ist der Genotyp eine binäre Kodierung, auf die universelle Operatoren angewendet werden. Diese Operatoren mutieren und rekombinieren die Bits der Kodierung ohne deren Semantik zu berücksichtigen. Ein weiteres Merkmal eines genetischen Algorithmus ist die Dominanz des Rekombinationsoperators gegenüber dem Mutationsoperator.

2.7.3.2. Evolutionsstrategien

Schwefel und Rechenberg prägten den Begriff *Evolutionsstrategie* [Rec93] [Sch95]. Diese Variante evolutionärer Algorithmen zeichnet sich dadurch aus, dass die Elemente der

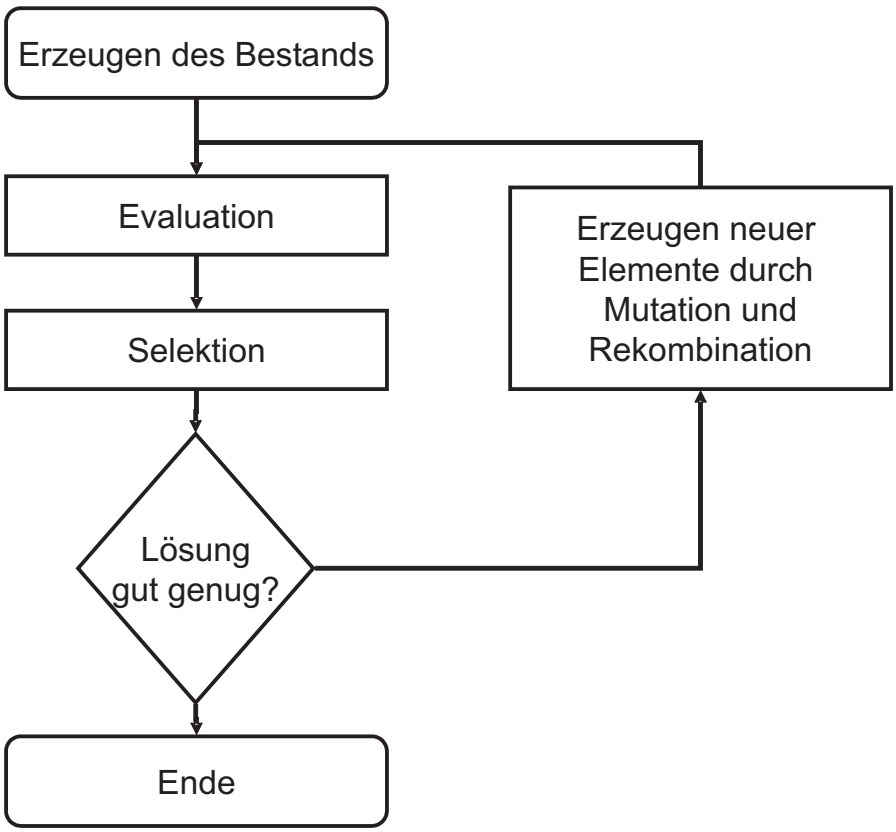


Abbildung 2.7.: Einfacher evolutionärer Algorithmus

Population aus Vektoren reellwertiger Zahlen bestehen. Weiterhin enthält jedes Individuum eine Strategievariable, die festlegt, wie stark eine Mutation im Erwartungswert den Genotyp des Individuums verändern soll. Die Strategievariable steuert also die Lokalität der Suche und wird im Rahmen des Verfahrens angepasst. Evolutionsstrategien setzen fast ausschließlich Mutationsoperatoren ein, die Rekombinationsoperatoren spielen nur eine untergeordnete Rolle.

2.7.3.3. Genetisches Programmieren

Genetisches Programmieren [Koz89] ist eine evolutionsgesteuerte Variante der Metaprogrammierung. Ein Individuum repräsentiert ein ausführbares Programm, das meist als Baum dargestellt wird. Eine Mutation verändert den Baum durch Austauschen eines Programmoperators bzw. eines Terminals oder durch Verschiebung eines Teilbaums. Eine Rekombination selektiert in zwei Elternindividuen je einen Teilbaum und vertauscht diese anschließend. Die Programme müssen meist ausgeführt werden, um den Wert der Zielfunktion bestimmen zu können.

2.7.3.4. Evolutionäre Programmierung

Evolutionäre Programmierung [FOW66] ist ein Vorläufer des genetischen Programmierens. Die Mutationsoperatoren der evolutionären Programmierung arbeiten auf endlichen Automaten, die einfache Computerprogramme modellieren. Evolutionäre Programmierung unterscheidet nicht zwischen Genotyp und Phänotyp und erlaubt deshalb eine sehr problemnahe Repräsentation der Individuen.

2.8. Zusammenfassung

Die Struktur objektorientierter Softwaresysteme ist durch die Aufteilung des Systems in Teilsysteme und deren Beziehungsgeflecht definiert, wobei die kleinsten Teilsysteme Klassen bzw. Objekte sind. Diese Struktur hat maßgeblichen Einfluss auf die erforderlichen Wartungsaufwände. Zur Bewertung der Struktur existieren Entwurfsprinzipien, die werkzeuggestützt überprüft werden können. Zur Veränderung der Struktur können verhaltensbewahrende Restrukturierungen eingesetzt werden.

Evolutionäre Algorithmen werden auf viele Probleme angewendet, die mit Hilfe konventioneller Methoden nicht einfach zu lösen sind, und werden in dieser Arbeit zur Verbesserung der Qualität der Struktur von Softwaresystemen eingesetzt. Im folgenden Kapitel widmen wir uns dem existierenden Stand der Technik.

Kapitel 3.

Stand der Technik

Es existiert bereits eine Reihe von Verfahren, deren Ziel es ist, die Qualität der Struktur eines Systems zu verbessern. In diesem Kapitel untersuchen wir, ob diese vorhandenen Ansätze unsere in Kapitel 1 aufgestellten Anforderungen erfüllen. Dazu beschreiben wir zunächst ihre Funktionsweise und untersuchen anschließend, ob sie in der Lage sind, die in einer von uns vorgegebenen Referenzfallstudie vorhandenen Strukturprobleme vollständig zu entfernen.

Wir teilen die vorhandenen Ansätze mit einer zu unserer ähnlichen Zielsetzung in fünf Kategorien ein, da viele der Ansätze sich nur geringfügig unterscheiden und wir ihre Stärken und Schwächen somit übersichtlicher diskutieren können.

- *Verfahren zur allgemeinen Abschätzung der Auswirkung von Restrukturierungen:* Diese Verfahren schätzen allgemein ab, wie sich bestimmte Restrukturierungstypen auf verschiedene Metriken auswirken.

- *Manuelle analytische Verfahren:* Ansätze aus dieser Kategorie verbessern die Qualität der Struktur, indem sie ein Strukturproblem nach dem anderen entfernen.

- *Automatisierte Spezialverfahren:* Diese Verfahren verbessern die Qualität der Struktur eines Softwaresystems bezüglich genau einer Entwurfsheuristik.

- *Automatisierte Verfahren zur Analyse von Teilsystemzerlegungen:* Verfahren aus dieser Kategorie haben entweder das Ziel, für ein existierendes System eine nicht mehr existente Teilsystemzerlegung wiederzugewinnen, oder eine optimale Teilsystemzerlegung zu berechnen. Die entstandene Teilsystemzerlegung kann ein Softwareingenieur verwenden, um die existierende Teilsystemstruktur zu verbessern und damit die Qualität der Systemstruktur zu erhöhen.

- *Suchbasierte Verfahren zur Verbesserung der Klassenstruktur:* Diese Ansätze verändern die Klassenstruktur mit Restrukturierungen und bewerten mit einer Zielfunktion, ob die Restrukturierungen die Qualität der Struktur verbessert haben. Sie ähneln somit dem in dieser Arbeit vorgestellten Verfahren.

Bevor wir nun die einzelnen Kategorien im Detail diskutieren, stellen wir unsere Referenzfallstudie vor.

3.1. Referenzfallstudie

Unsere Referenzfallstudie ist das Softwaresystem *Rheingold*. Dieses System dient zur Simulation von Schiffsbewegungen und wurde am FZI für die Bundesanstalt für Wasserbau entwickelt. In dieser Fallstudie gibt es eine Reihe von Strukturproblemen. Wir haben aus Gründen der Einfachheit ein Teilsystem ausgewählt, das gemessen an seiner Größe besonders viele Strukturprobleme enthält: das Teilsystem `XML`

Eine Anforderung des Auftraggebers war, dass die im Programm vorgenommenen Einstellungen, wie z.B. Fließgeschwindigkeiten in einer XML-Datei persistent gemacht werden können. Der Funktionsumfang von Rheingold nimmt ständig zu, zum Beispiel bedingt durch eine neue zweidimensionale Kartendarstellung. Somit gibt es ständig neue Einstellungen, die in das vorhandene XML-Format aufgenommen, gespeichert und wieder gelesen werden müssen.

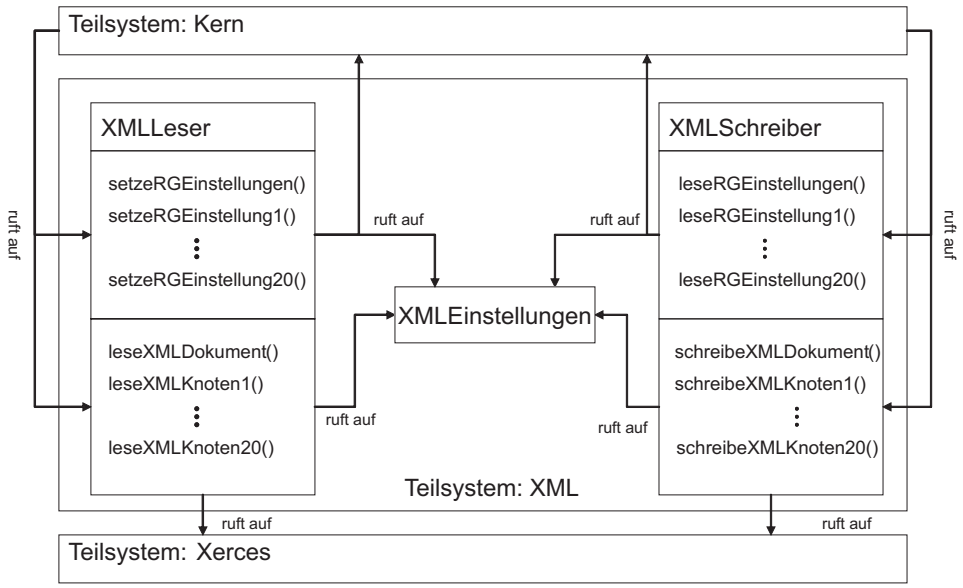


Abbildung 3.1.: Referenzfallstudie

Abbildung 3.1 zeigt diesen Ausschnitt der Systemstruktur. Im Detail ist nur das Teilsystem `XML` zu sehen, das die Einstellungen mit Hilfe der Bibliothek Xerces [1] speichern und lesen kann. Die anderen Klassen von Rheingold sind vereinfacht als das Teilsystem `Kern` dargestellt.

Die Klasse `XMLEinstellungen` dient als Behälter, der alle Einstellungen aus Rheingold aufnehmen kann, die persistent gemacht werden können. Die Klasse `XMLSchreiber` hat zwei Aufgaben. Erstens sammelt sie die Einstellungen aus Rheingold und speichert diese in einem `XMLEinstellungen`-Behälter ab, zweitens überführt sie diese Einstellungen in einen XML-Baum, der mit Hilfe von Xerces geschrieben werden kann. Die Klasse enthält zwei Gruppen von Methoden, die indirekt über `XMLEinstellungen` kommunizieren, sich allerdings nicht gegenseitig aufrufen und von außen nur nacheinander aufgerufen werden.

[1] http://xerces.apache.org/

Analog zur Klasse `XMLSchreiber` überführt die Klasse `XMLLeser` einen XML-Baum in einen `XMLEinstellungen`-Behälter und nimmt anschließend die Einstellungen in Rheingold vor.

Unter den Teilsystemen von Rheingold gibt es ein Teilsystem, das den Lade- und Speichervorgang steuert, indem es entweder `leseXMLDokument()` gefolgt von `setzeRGEinstellungen()` oder `leseRGEinstellungen()` gefolgt von `schreibeXMLDokument()` aufruft.

Die hier vorgestellte Systemstruktur enthält drei Strukturprobleme:

1. Eine zyklische Abhängigkeit zwischen dem Teilsystem `Kern` und dem Teilsystem `XML`. Methodenaufrufe von `Kern` zu `XMLLeser` und `XMLSchreiber` und Methodenaufrufe von `XMLLeser` und `XMLSchreiber` zu `Kern` verursachen diesen Zyklus. Zur Erkennung von Zyklen kann ein einfacher auf Tiefensuche basierender Graphalgorithmus verwendet werden.

2. Zwei Gottklassen, die mehr als eine Abstraktion kapseln: `XMLLeser` und `XMLSchreiber`. Die beiden Klassen enthalten sehr viele Methoden und bestehen aus zwei nicht kohäsiven Methodenblöcken. Automatisch können wir diese Gottklassen mit Hilfe einer Komplexitätsmetrik für Klassen (z.B. Anzahl der Methoden) und einer Kohäsionsmetrik für Klassen (z.B. LCOM [CK94]) erkennen.

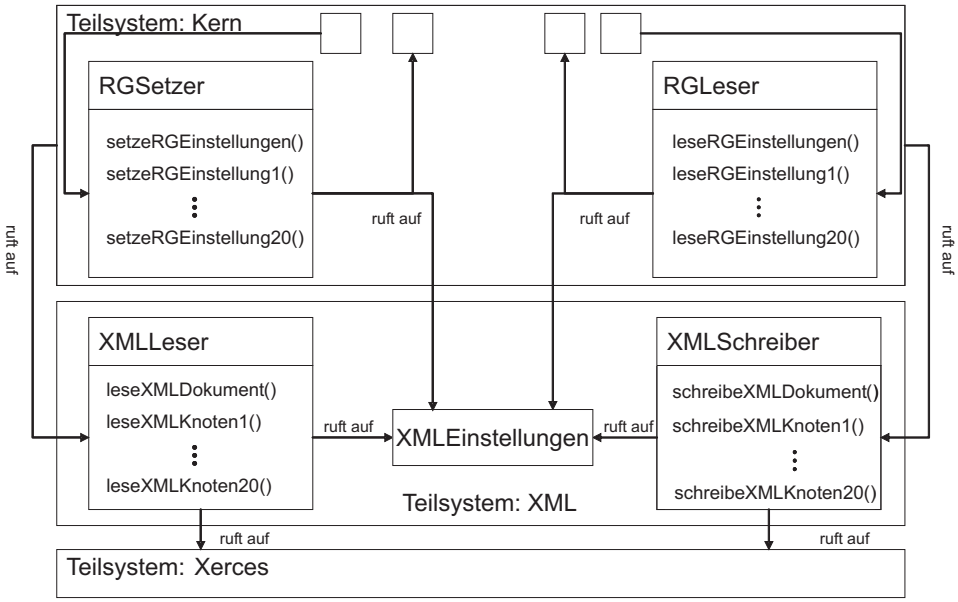


Abbildung 3.2.: Verbesserte Struktur der Referenzfallstudie

Diese Strukturprobleme erschweren das Verständnis des Systems. Das *Sammeln und Setzen* der Einstellungen und das *Schreiben und Lesen* der XML-Dateien wurde in einem Teilsystem vermischt. Somit können die beiden Teilsysteme `XML` und `Kern` nicht getrennt voneinander verstanden werden.

Da *Sammeln und Schreiben* und *Lesen und Setzen* sogar jeweils in einer einzigen Klasse stattfinden, sind diese Klassen sehr unübersichtlich und unverständlich. Kommt eine neue Einstellung hinzu, so muss sowohl der Quelltext für das Sammeln bzw. Setzen der Rheingold-Einstellungen und das Schreiben bzw. Lesen der XML-Daten in jeweils einer Klasse hinzugefügt werden.

Führt diese Anpassung nur eine Person durch, so muss diese wie oben schon erwähnt beide Teilsysteme kennen. Wird die Anpassung von zwei verschiedenen Leuten durchgeführt, so könnte eine Person für Rheingold-Einstellungen und eine Person für die XML-Persistenz zuständig sein. In diesem Fall besteht die Gefahr, dass die Änderung einer Klasse durch zwei Personen zu Konflikten und Problemen führt, z.B. da eine Person die Klasse zur Bearbeitung sperrt.

Um die gerade beschriebenen Anpassungen zu erleichtern können diese Strukturprobleme durch vier Restrukturierungen behoben werden.

1. Zunächst muss von der Klasse `XMLLeser` eine neue Klasse abgespalten werden. In diese Klasse werden alle Methoden übernommen, die Einstellungen in Rheingold setzen. Die neue Klasse nennen wir `RGSetzer`.

2. Analog zur Klasse `XMLLeser` muss von der Klasse `XMLSchreiber` eine neue Klasse abgespalten werden. Es werden alle Methoden, die Einstellungen von Rheingold auslesen, in die neue Klasse verschoben. Die neue Klasse nennen wir `RGLeser`

3. Wir verschieben die Klasse `RGSetzer` in das Teilsystem `Kern`.

4. Abschließend verschieben wir die Klasse `RGLeser` ebenfalls in das Teilsystem `Kern`.

Die resultierende Systemstruktur ist in Abbildung 3.2 zu sehen. Die Restrukturierungen 1 und 2 haben die Gottklassen beseitigt: Die entstandenen Klassen besitzen eine angemessene Größe, sind kohäsiv und stellen genau eine Abstraktion dar. Mit den Restrukturierungen 3 und 4 beseitigen wir den Zyklus zwischen den Teilsystemen.

Durch diese Restrukturierungen ist das Schreiben und Lesen der Einstellungen als XML-Dokument sauber vom Rest des Programms abgetrennt. Einzige Berührungspunkte sind der Behälter für die XML-Einstellungen und die Methoden für das Lesen und Schreiben des XML-Dokuments. Somit kann sich ein Softwareingenieur fast unabhängig von den restlichen Teilsystemen um die Persistenz der Einstellungen kümmern. Allerdings muss er die Schnittstellen mit den anderen Softwareingenieuren absprechen. Diese Restrukturierungen sorgen auch dafür, dass das Persistenzformat bei Bedarf einfacher ausgetauscht werden kann, falls z.B. eine zusätzliche binäre Speicherung geplant ist.

3.2. Verfahren zur allgemeinen Abschätzung der Auswirkung von Restrukturierungen

Verfahren zur allgemeinen Abschätzung der Auswirkung von Restrukturierungen bieten dem Softwareingenieur eine allgemeine Einschätzung, wie sich die Anwendung einer bestimmten Restrukturierung auf verschiedene Metriken auswirkt. Die Einschätzung wurde

zuvor empirisch abgeleitet, indem die unterstützten Restrukturierungen auf Beispielsystemstrukturen angewandt und die Auswirkungen auf ausgewählte Metriken bestimmt wurden.

Es ist problematisch, dass die vorliegende Systemstruktur überhaupt nicht berücksichtigt werden kann und die Effekte der Restrukturierungen somit anders als vorausgesagt ausfallen können. Einen Softwareingenieur können sie nur eingeschränkt bei der Restrukturierung eines Systems unterstützen.

Insgesamt konnten wir drei Verfahren identifizieren, die die gerade beschriebene Vorgehensweise anwenden. Tahvildari und Kontogiannis stellen in [TK03] einen dieser Ansätze vor. Sie schätzen für 6 Restrukturierungen die Auswirkung auf sieben Entwurfsprinzipien ab. Zum Beispiel stellen die Autoren fest, dass die Kapselung eines Systems verbessert werden kann, indem ein Umwickler eingebaut wird. Die in diesem Papier betrachteten Restrukturierungen sind alle grobgranularer als die in [Fow99] beschriebenen. Für keine der beschriebenen Restrukturierungen geben die Autoren jedoch an, wie diese in den vorhandenen Quelltext eingebaut werden können.

Du Bois und Mens gehen in [BM03] einer ähnlichen Fragestellung nach. Sie untersuchen für die drei Restrukturierungen *Attribut kapseln*, *Methode in Oberklasse verschieben* und *Methode extrahieren* den Einfluss auf die fünf Metriken *Anzahl der Methoden*, *Anzahl der Unterklassen*, *Kopplung zwischen Objekten*, *Antwortmenge einer Klasse* und *Mangel an Methodenkohäsion*. Die Autoren geben an, dass beispielsweise die Restrukturierung *Attribut kapseln* die Metrik *Anzahl der Methoden* erhöht. Sie vernachlässigen jedoch die konkret vorliegende Systemstruktur und können somit nicht alle betrachteten Entwurfsheuristiken berücksichtigen.

Der dritte Ansatz aus dieser Kategorie stammt von Sahraoui, Godin und Miceli [Sah00]. Die Autoren betrachten in ihrem Papier drei Restrukturierungen und ihre Auswirkung auf 12 Kopplungs- und Vererbungsmetriken. Diese Auswirkungen werden wieder unabhängig von der vorliegenden Systemstruktur vorgeschlagen. Es ist nicht garantiert, dass sich insgesamt die Qualität der Systemstruktur verbessert und somit ist auch dieser Ansatz gemäß unserer Definition nicht multikriteriell.

3.2.1. Diskussion

Die hier vorgestellten Verfahren können für keines der in unserem Referenzbeispiel vorhandenen Strukturprobleme Restrukturierungen vorschlagen.

Alle Ansätze dieser Kategorie sind nicht *multikriteriell*: Sie geben nur allgemeine Empfehlungungen für Restrukturierungen, die nicht von der konkreten Systemstruktur abhängen und können somit nicht garantieren, dass sich die Qualität der Struktur insgesamt verbessert. Die Verfahren laufen nicht *automatisch* ab, da ein Softwareingenieur selbständig entscheiden muss, welche Restrukturierungen er zur Verbesserung der Qualität der Struktur einsetzt. Die Verfahren berücksichtigen nicht, wie die Restrukturierungen auf das vorliegende System angewandt werden können und stellen so auch nicht sicher, dass das *extern beobachtbare Verhalten* unverändert bleibt. Strukturen, die *Entwurfsheuristiken* bewusst verletzen, werden von diesen Verfahren nicht besonders betrachtet und somit auch nicht bewahrt. Verfahren dieser Kategorie arbeiten jedoch auf

allen wesentlichen *Strukturierungseinheiten* und sind auf *reale objektorientierte Systeme* anwendbar.

3.3. Manuelle analytische Verfahren

Ansätze aus der Kategorie *Manuelle analytische Verfahren* arbeiten lokal, behandeln eine stark eingeschränkte Menge von Entwurfsheuristiken gleichzeitig und können diese mit Nutzereingaben entfernen.

In [Fow99] beschreibt Fowler einen Katalog von Refaktorisierungen. Passend zu diesen Refaktorisierungen gibt er eine Reihe von *schlechten Gerüchen* an, die auf eine schlechte Qualität der Struktur hinweisen und die mit Hilfe der Refaktorisierungen entfernt werden können. Fowler definiert den Begriff Refaktorisierung wie folgt: Eine Refaktorisierung ist eine Änderung der Struktur eines Softwaresystems, mit der Folge, dass das System anschließend einfacher zu verstehen und kostengünstiger zu verändern ist, ohne das von außen beobachtbare Verhalten zu verändern.

Vergleichen wir die Bedeutung der Begriffe Restrukturierung und Refaktorisierung [Gen04] so stellen wir fest, dass beide das beobachtbare Verhalten nicht verändern, Refaktorisierungen im Gegensatz zu Restrukturierungen aber einen positiven Effekt auf die Qualität der Struktur haben müssen: Das System muss anschließend einfacher zu verstehen und kostengünstiger zu verändern sein. Im Gegensatz dazu können Restrukturierungen die Qualität der Struktur durchaus verschlechtern.

Der Katalog von Fowler spiegelt den Erfahrungsschatz der Autoren wieder. Sie verbinden die *schlechten Gerüche* allerdings nur informell mit den Refaktorisierungen. Die Autoren stehen einer werkzeuggestützten Erkennung von Strukturproblemen skeptisch gegenüber. Sie vertrauen auf die Intuition eines Softwareingenieurs und geben textuelle Handlungsanweisungen, wie dieser schlechte Gerüche entfernen kann. Insgesamt enthält das Buch 22 schlechte Gerüche und über 70 Restrukturierungen.

Dudzikan und Wlodka stellen in [DW02] einen integrierten Ansatz vor, mit dessen Hilfe Java-Programme restrukturiert werden können. Der Ansatz erkennt Kandidaten für Strukturprobleme automatisch und schlägt dem Softwareingenieur Restrukturierungen vor, um diese zu entfernen. Dabei lässt das Verfahren allerdings die konkrete Situation unberücksichtigt, in der das Strukturproblem auftritt. Für jedes Strukturproblem schlägt das Verfahren immer die gleiche Menge von Restrukturierungen vor. Ein Softwareingenieur muss aus diesen Vorschlägen selbst den passenden auswählen. Das Verfahren wurde implementiert und in die Entwicklungsumgebung NetBeans[2] integriert. In der aktuellen Version können 15 Strukturprobleme erkannt und für 13 Strukturprobleme Behebungsvorschläge gemacht werden.

Tourwé und Mens gehen in [TM03] ähnlich zu den beiden oben beschriebenen Ansätzen vor: Sie erkennen Strukturprobleme mit Hilfe logischer Metaprogrammierung und geben zu den so gefundenen Strukturproblemen eine Liste von möglicherweise passenden Lösungsvorschlägen an, aus denen ein Softwareingenieur manuell den

[2] http://www.netbeans.org/

geeignetsten auswählen muss. Die Autoren erkennen also, dass mehrere verschiedene Restrukturierungen ein Strukturproblem beheben können. Sie sind aber nicht in der Lage, dem Softwareingenieur mit ihrem Verfahren bei der Auswahl zu helfen. Er muss alleine die richtige Restrukturierung auswählen. Die Restrukturierungsvorschläge werden nach Möglichkeit an die Umgebung angepasst, in der das Strukturproblem auftritt. Beispielsweise wird bei einem unbenutzten Parameter p, die zugehörige Restrukturierung *Lösche Parameter* entsprechend konfiguriert, so dass dem Benutzer als Vorschlag *Lösche Parameter p* präsentiert wird. Die Autoren beschreiben in ihrem Papier auch kaskadierende Restrukturierungen. Sie verstehen darunter die an eine Restrukturierung anschließenden und zu empfehlenden Folgerestrukturierungen. Tritt beispielsweise ein unbenutzter Parameter innerhalb einer Methode auf, so sollte ihrer Meinung nach an den Aufrufstellen der Methode ebenfalls untersucht werden, ob der Parameter dort bereits unbenutzt ist. Die Autoren erkennen zwar, dass alle Auswirkungen von Restrukturierungen betrachten werden müssen, können dies in ihrem Ansatz aber nicht berücksichtigen.

Trifu, Seng und Genssler stellen in [TSG04] einen weiteren Ansatz aus der Kategorie manuelle analytische Verfahren vor. Der in diesem Papier beschriebene Ansatz findet Strukturprobleme und bietet isoliert für jedes Strukturproblem Vorschläge zu dessen Behebung. Dieser Ansatz stellt in zwei Punkten eine Verbesserung gegenüber den obigen Verfahren dar. Erstens kann ein Softwareingenieur mit Hilfe eines Qualitätsmodells die Menge der für ihn relevanten Strukturprobleme bestimmen und zweitens unterstützt ihn der Ansatz dabei, die richtige Restrukturierung auszuwählen. Der Ansatz nutzt für letzteres so genannte *Correction Strategies*, die Expertenwissen formalisieren und den Softwareingenieur durch eine Kombination von Fragen und automatisierten Analysen zur passenden Restrukturierung führen. Dieser Ansatz ist nicht multikriteriell, da immer nur ein Strukturproblem und somit eine Entwurfsheuristik betrachtet wird.

Die bisher beschriebenen Ansätze können in unserer Referenzfallstudie die vorhandenen Strukturprobleme erkennen. Sie können aber nur allgemeine Restrukturierungsvorschläge machen, ohne das konkret vorliegende System zu berücksichtigen. Insbesondere schlagen sie pro Strukturproblem mehr als eine Lösung vor.

In der Struktur unserer Referenzfallstudie würden Verfahren dieser Kategorie, wie in Abbildung 3.3 dargestellt, die an den Strukturproblemen beteiligten Strukturelemente markieren.

Für jedes Strukturproblem würden die Verfahren dann eine Liste von allgemeinen Restrukturierungen vorschlagen:

- Teilsystemzyklus: *Teilsystem aufteilen* und *Klasse verschieben*.
- Gottklassen: *Klasse aufteilen* und *Methode verschieben*.

Ein Softwareingenieur müsste selbständig über die Reihenfolge entscheiden, in der die Strukturprobleme gelöst werden. Er müsste alleine entscheiden, welchen Restrukturierungstyp er einsetzt und wie er die Restrukturierung tatsächlich im Quelltext umsetzt.

Trifu beschreibt in [TM05] eine Weiterentwicklung des oben vorgestellten Ansatzes. Strukturprobleme sind für ihn nicht die eigentlichen Probleme, sondern nur Symptome,

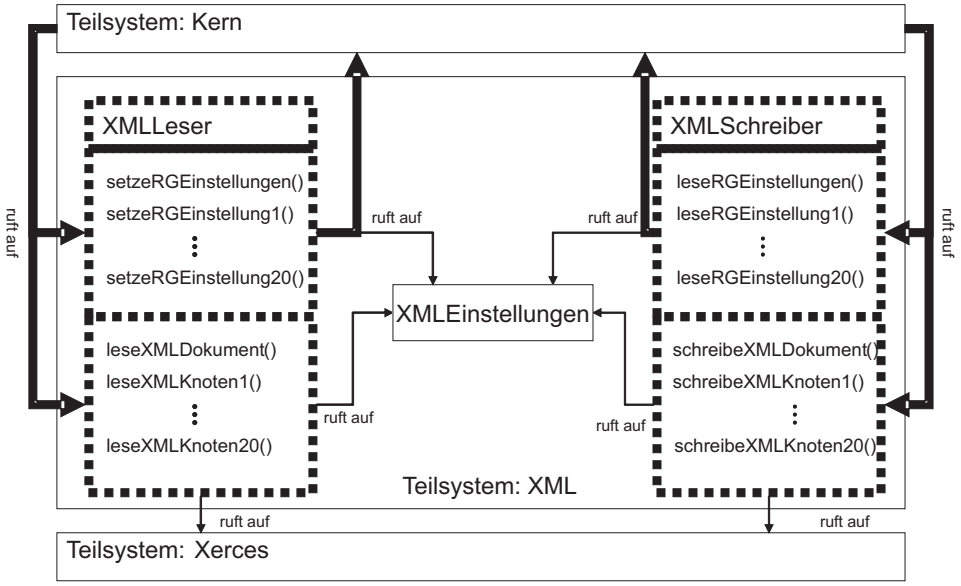


Abbildung 3.3.: Markierung der Strukturprobleme

die auf ein tieferliegendes Problem deuten. Das Problem selbst kann nur identifiziert werden, indem mehrere Symptome gleichzeitig betrachtet werden. Weiterhin behauptet der Autor, dass es zu jedem so identifizierten Problem genau eine passende Lösung gibt, und ein Softwareingenieur somit keine Auswahl treffen muss.

Seine Behauptung begründet der Autor mit einer Analogie aus der Medizin: Hat ein Arzt eine Krankheit diagnostiziert, so gibt es genau eine richtige Behandlung der Krankheit. Dies ist so nicht richtig, da für eine Krankheit durchaus verschiedene Behandlungen existieren können. Meist kann grundsätzlich zwischen Naturheilverfahren und Methoden der Schulmedizin ausgewählt werden, so dass auch hier noch eine Entscheidung getroffen werden muss.

Auch für Softwaresysteme darf bezweifelt werden, dass es für alle Strukturprobleme genau eine Lösung gibt. Das Verfahren stellt trotzdem einen Fortschritt dar, da es nicht mehr die eingeschränkte Sicht auf ein einzelnes Strukturproblem sondern eine Menge von Strukturproblemen und somit Entwurfsheuristiken hat. Trotzdem ist dieser Ansatz nicht multikriteriell, da nur ein kleiner Teil des Systems bei der Auswahl der Lösung betrachtet wird.

Dieser Ansatz würde in unserer Referenzfallstudie vorschlagen, die Klasse `XMLSchreiber` und `XMLLeser` in zwei Klassen aufzuteilen, da der Ansatz sie als *Klasse mit mehreren Zuständigkeiten* erkennen würde. Abbildung 3.4 zeigt dies.

Der Ansatz würde den damit zusammenhängenden Zyklus zwischen den zwei Teilsystemen `Kern` und `XML` weder erkennen noch beheben können, da er aktuell für Teilsysteme keine Strukturprobleme behandelt. Selbst wenn er dies tun würde, ist es unwahrscheinlich, dass der Ansatz für die in der Referenzfallstudie beschriebene Konstellation von Strukurproblemen ein Gesamtproblem definieren würde. Ein Softwareingenieur müsste sich also selbständig entscheiden, in welcher Reihenfolge die Strukturprobleme behoben

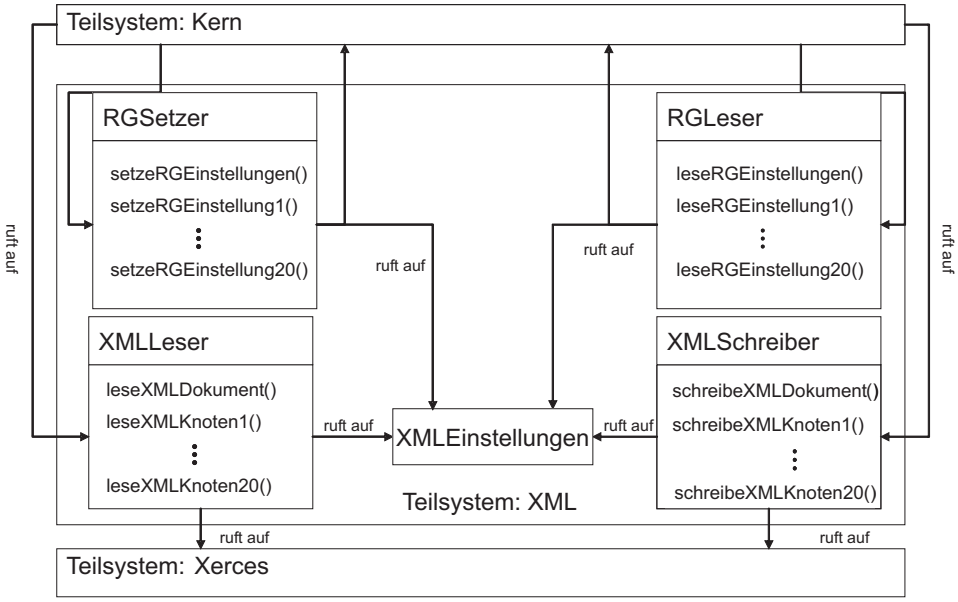


Abbildung 3.4.: Entfernung der Gottklassen

werden sollten.

3.3.1. Diskussion

Alle Verfahren dieser Kategorie sind nicht *multikriteriell*. Sie berücksichtigen zwar eine Menge von Entwurfsheuristiken, garantieren aber nicht, dass sich die Qualität der Struktur insgesamt verbessert, da sie nur einen Teil des Systems betrachten und Auswirkungen auf andere Systemteile unberücksichtigt lassen. Die Verfahren laufen gemäß unserer Definition nicht *automatisch* ab, da der Benutzer selbst entscheiden muss, welche Restrukturierungen er anwenden möchte. Die Verfahren dienen ihm hier nur als Entscheidungshilfe. Strukturen, die *Entwurfsheuristiken* bewusst verletzen, werden nicht gesondert betrachtet und müssen vom Softwareingenieur selbst identifiziert werden.

Das *extern beobachtbare Verhalten* wird durch die Ansätze nicht verändert, da sie zur Behebung von Strukturproblemen verhaltensbewahrende Restrukturierungen einsetzen. Die Ansätze sind auf *reale objektorientierte Systeme* anwendbar und berücksichtigen bis auf ein Verfahren alle wesentlichen *Strukturierungseinheiten*.

3.4. Automatisierte Spezialverfahren

Automatisierte Spezialverfahren verbessern die gesamte Struktur eines Softwaresystems unter Berücksichtigung genau einer Entwurfsheuristik.

Lieberherr gibt in [Lie95] ein Verfahren an, das die Struktur eines objektorientierten Systems in eine verhaltensgleiche Systemstruktur überführt, die gemäß dem Gesetz von Demeter aufgebaut ist. Dieses Gesetz besagt, dass ein Objekt nur Kenntnisse über sich

selbst und seine direkten Nachbarn haben soll. Insbesondere werden lange Zugriffsketten auf diese Art und Weise verhindert.

Moore beschreibt in [Moo96] ein Verfahren für die Programmiersprache *Self*, das eine Vererbungshierarchie in eine verhaltensgleiche Vererbungshierarchie überführt, in der kein Merkmal mehrfach deklariert wird, so dass es ein anderes Merkmal überdeckt. Merkmale sind in diesem Papier sowohl Attribute als auch Methoden. Aus Methoden können sogar gemeinsame Ausdrücke ausfaktorisiert werden, um Quelltext-Kopien zu eliminieren.

Streckenbach und Snelting geben in [SS03] eine Variante des Algorithmus von Snelting und Tip für Java-Programme an. Dieser Algorithmus garantiert, dass jede Abstraktion nur die Merkmale enthält, die ein Klient wirklich benötigt.

3.4.1. Diskussion

Die hier vorgestellten Spezialverfahren können keines der Strukturprobleme unserer Referenzfallstudie lösen.

Sie sind nicht *multikriteriell*, da sie sich auf genau eine Entwurfsheuristik konzentrieren und andere Entwurfsheuristiken nicht beachten. Strukturen, die *Entwurfsheuristiken* bewusst verletzen, werden von Ihnen nicht gesondert betrachtet. Diese Strukturen werden zerstört, wenn dadurch die betrachtete strukturelle Eigenschaft verbessert wird. Durch die Beschränkung auf eine strukturelle Eigenschaft sind die Verfahren auch nicht in der Lage, alle wesentlichen *Strukturierungseinheiten* zu adressieren.

Die Verfahren bestimmen allerdings *automatisch* Restrukturierungen zur Verbesserung der Qualität der Struktur, sind auf *reale objektorientierte Systeme* anwendbar und garantieren, dass sich das beobachtbare Ein- und Ausgabeverhalten des Systems nicht ändert.

3.5. Automatisierte Verfahren zur Analyse von Teilsystemzerlegungen

Verfahren aus dieser Kategorie haben entweder das Ziel, für ein existierendes System eine nicht mehr existente Teilsystemzerlegung wiederzugewinnen, oder eine optimale Teilsystemzerlegung zu berechnen. Obwohl diese Verfahren nicht direkt das Ziel haben, eine bestehende Systemstruktur zu verbessern, können ihre Ergebnisse zur Strukturverbesserung verwendet werden. Ein Softwareingenieur kann mit ihren Ergebnissen die nötigen Restrukturierungen nachträglich bestimmen, um eine existierende Teilsystemzerlegung in eine optimale Teilsystemzerlegung zu überführen. Er benötigt dazu lediglich die Restrukturierungen *Teilsystem erzeugen*, *Teilsystem löschen* und *Klasse verschieben*.

Alle vorliegenden Ansätze setzen zur Lösung des Problems Optimierungstechniken ein: Sie verwenden entweder Ballungsanalysen oder suchbasierte Verfahren.

Den auf Ballungsanalysen basierenden Ansätzen ist gemeinsam, dass sie ein Ähnlichkeitsmaß zwischen Strukturelementen definieren. Zwei Strukturelemente sind um so

ähnlicher, je mehr statisch beobachtbare Beziehungen zwischen ihnen existieren. Solche Ähnlichkeitsmaße führen dazu, dass Teilsysteme mit hoher innerer Kohäsion und geringer Kopplung erzeugt werden.

Als erster setzte Wiggerts Verfahren zur Ballungsanalyse für die Migration von prozeduralen zu objektorientierten Systemen ein [Wig97]. Mancoridis [MMR98] und Mitchell [Mit02] wendeten Ballungsanalysen zum ersten Mal zur Wiedergewinnung von Teilsystemstrukturen in objektorientierten Systemen an. Sie verwendeten zunächst nur Quelldateien und `include`-Beziehungen, um das Ähnlichkeitsmaß zu berechnen, später experimentierten sie auch mit Klassen und Aufrufbeziehungen. Rayside [RRHK00], Trifu [Tri01] und Abreu [eAPS00] beschreiben weitere Varianten von Ballungsanalysen, um eine Teilsystemstruktur zu bestimmen. Sie verwenden komplexere Ähnlichkeitsmaße, die Vererbungsbeziehungen, Methodenaufrufe, Attributzugriffe und Typabhängigkeiten berücksichtigen.

Mancoridis und Mitchell beschreiben in [DMM99] den ersten Ansatz zur Bestimmung der Teilsystemstruktur eines Systems, der einen suchbasierten Algorithmus einsetzt. Als Zielfunktion verwenden sie eine Linearkombination, die sich aus einer Kopplungs- und einer Kohäsionsmetrik zusammensetzt. Die eingesetzten Mutationen sind *Teilsystem zerlegen*, *Teilsysteme verschmelzen* und *Klasse verschieben*.

Eine Reihe anderer Autoren nahm diese Vorgehensweise auf. Harman [HHP02] verwendet eine ähnliche Zielfunktion wie Mancoridis und Mitchel, gibt aber zusätzlich eine feste Zielgröße für die entstehenden Teilsysteme vor. Lutz [Lut01] verwendet ebenfalls einen suchbasierten Algorithmus, als Zielfunktion setzt er aber ein auf Informationsdichte basierendes Maß ein.

Bauer [BT04] beschreibt ein weiteres suchbasiertes Verfahren. Sein Verfahren unterscheidet sich von den anderen Verfahren in einem wesentlichen Punkt: Es berücksichtigt die verschiedenen Rollen von Strukturelementen. Bevor das eigentliche Suchverfahren beginnt, identifiziert es zunächst Entwurfsmuster und darauf aufbauend ermittelt es die vorhandene Architekturform. So gelingt es dem Verfahren beispielsweise, Bibliotheken zu identifizieren und diese gesondert zu behandeln.

3.5.1. Diskussion

Alle Ansätze würden unsere Referenzfallstudie bezüglich des Entwurfsprinzips *Kopplung und Kohäsion* optimieren. Die Klassen `XMLSchreiber` und `XMLLeser` würden aus zwei Gründen in das Teilsystem `Kern` verschoben werden. Erstens rufen die Klassen `XMLSchreiber` und `XMLLeser` das Teilsystem `Kern` oft auf und zweitens rufen umgekehrt Klassen aus dem Teilsystem `Kern` diese beiden Klassen auf. Die Klasse `XMLEinstellungen` würde ebenfalls in das Teilsystem `Kern` verschoben werden, da sie von den bereits verschobenen Klassen `XMLSchreiber` und `XMLLeser` häufig benutzt wird. Danach könnte das Teilsystem `XML` aufgelöst werden. Abbildung 3.5 zeigt die dadurch entstehende Systemstruktur.

Auf diese Art und Weise wäre der Teilsystemzyklus beseitigt. Allerdings wäre auch das aussagekräftige Teilsystem `XML` eliminiert und die Gottklassen bestünden unverändert.

Verfahren dieser Kategorie lassen das *extern beobachtbare Verhalten* des Systems

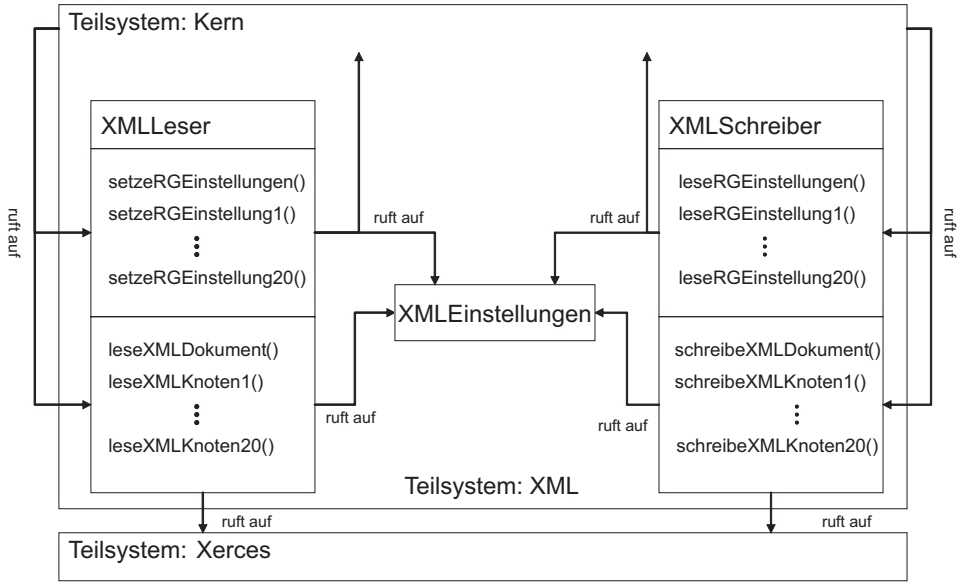


Abbildung 3.5.: Entfernung des Teilsystemzyklusses

unverändert, da sie nur auf Teilsystemstrukturen arbeiten. Sie sind auf *reale objektorientierte Systeme* anwendbar. Zumindest einer der vorgestellten Ansätze kann auch die besondere *Rolle von Strukturelementen* berücksichtigen.

Die Verfahren dieser Kategorie laufen gemäß unserer Definition nicht *automatisch* ab, da sie keine Restrukturierungen, sondern nur eine Systemstruktur mit einer höheren Qualität der Struktur bestimmen. Ein Softwareingenieur muss die nötigen Restrukturierungen selbständig ableiten, um die vorhandene Struktur in die berechnete Struktur zu überführen.

Die Verfahren sind eingeschränkt *multikriteriell*, da sie nur sicherstellen, dass Kopplung und Kohäsion für das gesamte System verbessert werden. Andere Entwurfsheuristiken, wie die Vermeidung von Zyklen, werden nicht direkt berücksichtigt. Prinzipiell könnten diese weiteren Entwurfsheuristiken zwar von den Verfahren berücksichtigt werden, die Autoren geben aber an, dass eine Berücksichtigung von Kopplung und Kohäsion ausreichend sei. Die Verfahren berücksichtigen nicht alle wesentlichen *Strukturierungseinheiten*, da sie die Klassenstruktur nicht verändern können.

3.6. Suchbasierte Verfahren zur Verbesserung der Klassenstruktur

O'Keeffe und O Cinnèide beschreiben in [OC04] und [OC06] einen suchbasierten Ansatz, mit dessen Hilfe ein Softwareingenieur die Klassenstruktur eines Systems verbessern kann.

Das Verfahren arbeitet mit zwei verschiedenen Suchverfahren: Hill-Climbing und Simuliertes Tempern. In jedem Schritt wählt das Suchverfahren eine der Restrukturie-

rungen *Verschiebe Methode/Attribut in Oberklasse*, *Verschiebe Methode/Attribut in Unterklasse* und *Extrahiere und verschmelze Hierarchie* aus und wendet diese auf einen Teil der Systemstruktur an. Das Verfahren bewertet die so ausgeführte Restrukturierung, indem es den Wert einer Zielfunktion, die aus 11 Einzelmetriken besteht, vor und nach der Restrukturierung berechnet. Die verwendeten Metriken wurden einem hierarchischen Qualitätsmodell entnommen [BD02]. Bei der Hill-Climbing-Variante wird eine Restrukturierung nur akzeptiert, wenn sich dadurch der Wert der Zielfunktion verbessert hat, bei der Variante Simuliertes Tempern werden auch zuerst große und später kleiner werdende Verschlechterungen des Wertes der Zielfunktion akzeptiert. Wird die Restrukturierung akzeptiert, so wird sie einer Liste von sinnvollen Restrukturierungen hinzugefügt. Anderenfalls wird die Restrukturierung rückgängig gemacht und verworfen. Das Ergebnis des Verfahrens ist eine Liste von Restrukturierungen, mit denen die Systemstruktur verbessert werden kann.

3.6.1. Diskussion

Das beschriebene suchbasierte Verfahren kann nicht alle Strukturprobleme aus unserer Referenzfallstudie lösen, da es keine Restrukturierungen für Teilsysteme kennt. Entfernen könnte es die beiden Gottklassen, was in Abbildung 3.6 zu sehen ist. Allerdings wären die Klassen über eine Vererbungsbeziehung verbunden, was bei den Aufgaben, die die Klassen erfüllen, nicht sinnvoll ist. Geeignet ist eine Assoziationsbeziehung zwischen den Klassen, da die Klasse `RGSetzer` die Klasse `XMLLeser` und die Klasse `RGLeser` die Klasse `XMLSchreiber` verwendet.

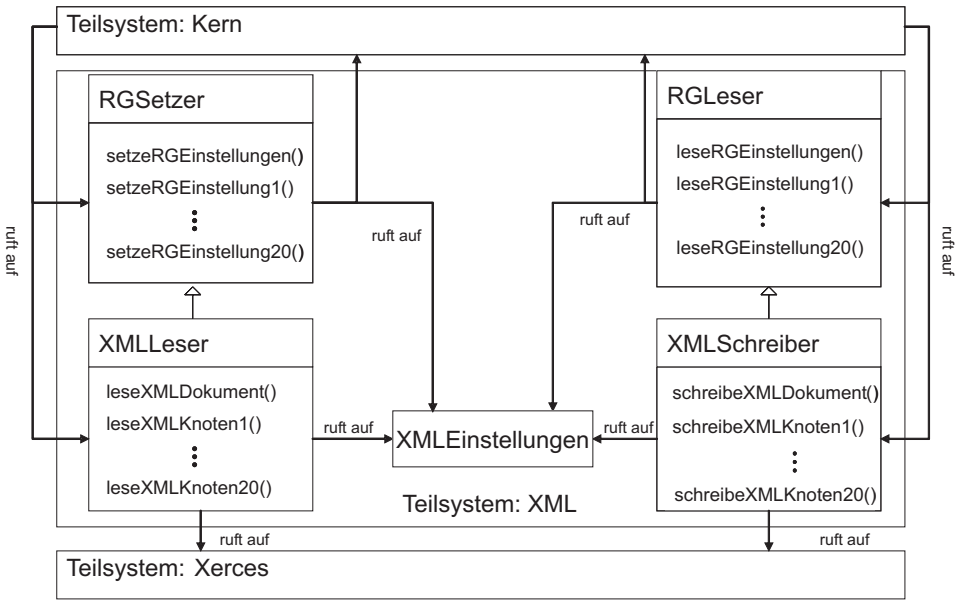


Abbildung 3.6.: Bearbeitung der Referenzfallstudie durch suchbasierte Verfahren

Das Verfahren ist *multikriteriell*, da es mehrere Entwurfsheuristiken berücksichtigt und die Qualität des Gesamtsystems betrachtet, während es nach geeigneten Restrukturie-

rungen sucht. Das Verfahren berechnet eine Reihe von ausführbaren Restrukturierungen und läuft somit *automatisch* ab. Die eingesetzten Restrukturierungen wurden so konzipiert, dass das Verfahren das *extern beobachtbare Verhalten* des Systems unverändert lässt.

Allerdings kann das Verfahren keine Strukturen bewahren, die *Entwurfsheuristiken* bewusst verletzen, da es keine Klassifikation der Strukturelemente vornimmt und alle Strukturen gleich behandelt. Das Verfahren hat ausschließlich das Ziel, Vererbungsstrukturen zu verbessern. Es verändert die Einteilung in Teilsysteme nicht und berücksichtigt somit nicht alle wesentlichen *Strukturierungseinheiten*. Die Autoren haben bisher noch nicht nachgewiesen, dass ihr Verfahren *auf reale objektorientierte* Systeme anwendbar ist. Sie haben es lediglich auf zwei Spielbeispiele aus einer Benchmarksuite angewendet.

3.7. Zusammenfassung

In diesem Kapitel haben wir gezeigt, dass keines der Verfahren aus dem Stand der Technik alle von uns geforderten Kriterien erfüllt. In Tabelle 3.7 ist noch einmal im Überblick zu sehen, ob Verfahren einer Kategorie eine Anforderung erfüllen oder nicht. Wir betrachten eine Anforderung als erfüllt, wenn mindestens ein Ansatz einer Kategorie die Anforderung erfüllt.

Verfahren zur allgemeinen Abschätzung der Auswirkung von Restrukturierungen berücksichtigen das vorliegende System nicht. Somit können sie nicht garantieren, dass sich insgesamt die Qualität der Struktur verbessert und sind also nicht multikriteriell. Sie sind nicht automatisch, da ein Softwareingenieur selbst entscheiden muss, welche Restrukturierungen er anwendet und sie bewahren keine Strukturen, die Entwurfsheuristiken bewusst verletzen.

Ansätze aus der Kategorie *manuelle analytische Verfahren* können nicht garantieren, dass sich die Qualität der Struktur insgesamt verbessert und sind deshalb nicht multikriteriell. Sie sind nicht automatisch, da sie dem Softwareingenieur mehrere Vorschläge zur Behebung eines Strukturproblems machen, aus denen er einen auswählen muss. Strukturen, die bewusst Entwurfsheuristiken bewusst verletzen, werden nicht gesondert betrachtet.

Existierende *automatisierte Spezialverfahren* verbessern die Qualität der Struktur automatisch bezüglich einer Entwurfsheuristik. Diese Verfahren berücksichtigen jedoch nicht alle wesentlichen Strukturierungseinheiten und beachten nicht die Auswirkungen auf andere Entwurfsheuristiken, die bei der Verbesserung bezüglich einer Entwurfsheuristik auftreten können. Auch diese Verfahren betrachten Strukturen, die bewusst Entwurfsheuristiken verletzen, nicht gesondert.

Vorhandene Ansätze zur *Analyse bzw. Verbesserung von Teilsystemzerlegungen* lassen die Klassenstruktur unberücksichtigt und betrachten bei der Berechnung der Qualität Systemstruktur nur wenige Entwurfsheuristiken. Sie sind nicht automatisch, da sie keine Liste von Restrukturierungen sondern nur die verbesserte Struktur als Ausgabe haben.

Existierende *Suchbasierte Verfahren* sind nur in der Lage, die Qualität der Klassenstruktur zu verbessern und können nicht auf reale, objektorientierte Systeme angewen-

det werden. Auch diese Verfahren betrachten Strukturen, die bewusst Entwurfsheuristiken verletzen, nicht gesondert. Nachdem wir in diesem Kapitel den Stand der Technik ausführlich besprochen haben, stellen wir im nächsten Kapitel unseren Ansatz zur Verbesserung der Qualität der Struktur existierender Systeme vor.

Das Verfahren muss	Manuelle analytische Verfahren	Verfahren zur allgemeinen Abschätzung der Auswirkungen von Restrukturierungen	Automatisierte Spezialverfahren	Automatisierte Verfahren zur Analyse von Teilsystemzerlegungen	Suchbasierte Verfahren zur Verbesserung Klassenstruktur
multikriteriell sein	⊖	⊖	⊖	⊖	⊕
automatisch ablaufen	⊖	⊖	⊕	⊖	⊕
das extern beobachtbare Verhalten des Systems unverändert lassen	⊕	⊖	⊕	⊕	⊕
Strukturen bewahren, die Entwurfsheuristiken bewusst verletzen	⊖	⊖	⊖	⊕	⊖
auf reale objektorientierte Systeme anwendbar sein	⊕	⊕	⊕	⊕	⊖
die wesentlichen Strukturierungseinheiten berücksichtigen	⊕	⊕	⊖	⊖	⊖

Tabelle 3.1.: Verfahren und Kriterien im Überblick

Kapitel 4.

Ein evolutionärer Algorithmus zur Strukturverbesserung

Im vorherigen Kapitel haben wir den vorhandenen Stand der Technik systematisch analysiert und konnten feststellen, dass keines der existierenden Verfahren einen zufriedenstellenden Ansatz zur Verbesserung der Qualität der Struktur eines gegebenen Systems bietet. Entweder berücksichtigen die analysierten Ansätze nicht alle Strukturierungseinheiten, oder sie sind nicht multikriteriell.

Die Bestimmung von Restrukturierungen zur Verbesserung der Qualität der Struktur eines Systems ist, wie wir bereits in Kapitel 1 beschrieben haben, ein Suchproblem. Um eine Näherungslösung zu bestimmen, verwenden wir einen evolutionären Algorithmus. Evolutionäre Algorithmen haben den Vorteil, dass sie eine Population von möglichen Lösungen betrachten, um so den Suchraum besser durchqueren zu können.

Unser Verfahren bestimmt zufällig Restrukturierungen, die eine existierende Systemstruktur in eine Struktur höherer Qualität überführen können, ohne dass sich das beobachtbare Verhalten des Systems ändert. Die höhere Qualität der Struktur führt dazu, dass Änderungen am System anschließend mit weniger Aufwand durchgeführt werden können.

In den folgenden zwei Abschnitten skizzieren wir die Hauptbestandteile unseres Verfahrens: das Metamodell und den evolutionären Algorithmus. Darauf aufbauend beschreiben wir im letzten Abschnitt dieses Kapitels den Ablauf unseres Verfahrens im Überblick. Die Hauptbestandteile beschreiben wir ausführlich in den Kapiteln 5 und 6.

4.1. Modellbildung

Unser Verfahren arbeitet nicht direkt auf dem Quelltext eines Systems, sondern simuliert die Restrukturierungen auf einem abstrakten Modell des Quelltextes. Wir haben uns für diesen Ansatz entschieden, weil unser Verfahren somit sprachunabhängig für ausdrucksbasierte, stark typisierte, objektorientierte Sprachen geeignet ist.

Der Aufbau eines Modells wird durch unser Metamodell festgelegt (siehe auch Abschnitt 5.2). Unser Metamodell enthält nur die Details, die wir zur Simulation der Restrukturierungen wirklich benötigen. Wir modellieren zwar alle wesentlichen Strukturelemente wie Teilsysteme, Klassen und Methoden und innerhalb von Methoden sogar

Zugriffe auf andere Strukturelemente. Kontrollflusselemente werden beispielsweise jedoch nicht modelliert.

Das Metamodell definiert Mengen und Elementaroperationen, die Informationen über ein Modell bereitstellen. Mit diesen Abfrageoperationen und den Mengen kann die Qualität der Struktur bewertet werden. Insbesondere können bestimmte strukturelle Muster gesucht und Metriken berechnet werden, um Entwurfsheuristiken zu überprüfen.

Weiterhin bietet das Metamodell eine Reihe von Elementaroperationen, mit denen Modelle verändert werden können. Vereinfacht gesagt können diese Operationen Modellelemente löschen, hinzufügen oder verschieben. Übertragen wir diese verändernden Elementaroperationen auf das ursprüngliche System, so können wir nicht garantieren, dass das beobachtbare Verhalten gleich bleibt.

Deshalb bietet das Metamodell eine weitere Klasse von Operationen: Modellrestrukturierungen, die ebenfalls die Struktur von Modellen verändern (siehe auch Abschnitt 5.2.5). Modellrestrukturierungen überprüfen Vor- und Nachbedingungen und setzen sich aus Elementaroperationen zusammen. Sie haben die Eigenschaft, dass sie auf das ursprüngliche System übertragen werden können und garantieren, dass die vorgenommenen Veränderungen eine Restrukturierung darstellen, also gemäß Abschnitt 2.6 sich das beobachtbare Verhalten nicht ändert. Diese Eigenschaft gilt auch, falls mehrere Modellrestrukturierungen hintereinander ausgeführt werden.

Unser Metamodell stellt folgende 10 Modellrestrukturierungen zur Verfügung:

1. Extrahiere Teilsystem
2. Integriere Teilsystem
3. Verschiebe Klasse
4. Extrahiere Klasse
5. Integriere Klasse
6. Verschiebe Methode
7. Verschiebe Methode in Oberklasse
8. Verschiebe Attribut in Oberklasse
9. Verschiebe Methode in Unterklasse
10. Verschiebe Attribut in Unterklasse

Diese Modellrestrukturierungen orientieren sich an den aus [Fow99] und [Kut02] bekannten Restrukturierungen zur Veränderung der Struktur eines Systems. Wir müssen diese sehr allgemeinen Restrukturierungen einschränken, da wir Vor- und Nachbedingungen sprachunabhängig überprüfen wollen und keine Nutzereingaben benötigen wollen. Eine Erweiterung des Metamodells um zusätzliche Modellrestrukturierungen ist denkbar. Mit den oben angegebenen Modellrestrukturierungen können wir allerdings zeigen, dass wir alle Strukturierungseinheiten adressieren können. Insbesondere können wir auch in Vererbungshierarchien Veränderungen der Struktur vornehmen.

4.2. Die Ausgestaltung eines evolutionären Algorithmus zur Strukturverbesserung

Für einen evolutionären Algorithmus müssen wir wie in Abschnitt 2.7 beschrieben vier Dinge klären: Erstens müssen wir festlegen, wie wir das Problem repräsentieren. Zweitens müssen wir Operatoren entwerfen, mit denen wir den Suchraum durchqueren. Drittens müssen wir die Zielfunktion festlegen, mit der wir die Elemente unserer Population bewerten. Als letztes müssen wir die Selektionsstrategie festlegen, mit der wir die Elemente auswählen, die im nächsten Schritt der Evolution weiter berücksichtigt werden sollen.

4.2.1. Repräsentation

Die Repräsentation legt den Phänotyp und den Genotyp unseres Algorithmus fest. Ein Modell als Ausprägung des in Abschnitt 4.1 beschriebenen Metamodells stellt unseren Phänotyp dar. Der evolutionäre Algorithmus arbeitet jedoch nicht direkt auf dem Phänotyp, sondern auf dem Genotyp unseres Problems. Unser Genotyp besteht aus einer geordneten Liste von Modellrestrukturierungen. Jede Modellrestrukturierung wird inklusive ihrer tatsächlichen Parameter abgespeichert. Diese Art der Repräsentation hat zwei Vorteile: Erstens können die notwendigen Schritte zur Veränderung der Struktur nach Ende des Verfahrens direkt aus dem Genotyp ausgelesen werden. Zweitens ermöglicht erst diese Form des Genotyps einen nicht destruktiven Rekombinationsoperator, wie wir im nächsten Abschnitt noch sehen werden. Der zu einem Genotyp gehörige Phänotyp entsteht durch sukzessive Anwendung der gespeicherten Modellrestrukturierungen auf das ursprüngliche Modell. Die Reihenfolge der Modellrestrukturierungen spielt eine Rolle, da manche der Modellrestrukturierungen erst die Voraussetzungen für nachfolgende Modellrestrukturierungen schaffen. Die Repräsentation beschreiben wir ausführlich in Abschnitt 6.1.

4.2.2. Operatoren

Unser Mutationsoperator variiert den Genotyp, indem er eine weitere Modellrestrukturierung an das Ende der Liste von Modellrestrukturierungen hinzufügt. Eine Rekombination kombiniert die Genotypen zweier Individuen, indem sie Teile der Listen von Modellrestrukturierungen miteinander verknüpft. Da durch die Rekombination Listen von Modellrestrukturierungen kombiniert werden, ist das Ergebnis wieder eine Liste von Modellrestrukturierungen. Somit ist garantiert, dass auch nach der Rekombination die Modellrestrukturierungen wieder auf das ursprüngliche System übertragen werden können und sich dabei das von außen beobachtbare Verhalten nicht ändert. Mutationen und Rekombination lassen den Genotyp im Laufe der Evolution kontinuierlich anwachsen. Die Operatoren behandeln wir detailliert in Abschnitt 6.2.

4.2.3. Zielfunktion

Die Entwurfsheuristiken zur Bewertung der Qualität der Struktur formulieren wir als Zielfunktion. Wir erfassen mit der Zielfunktion die drei Entwurfsprinzipien aus Kapitel 2: *Geheimnisprinzip und Schnittstellen*, *Kopplung und Kohäsion* und *Komplexität*. Diese Prinzipien überprüfen wir mit Hilfe von Entwurfsheuristiken. Ob und wie gut eine Entwurfsheuristik eingehalten wird, können wir mit Metriken messen. Die Zielfunktion bildet die Qualität der Struktur eines Systems auf eine reelle Zahl ab, indem sie die Metrikwerte auf die Größe des Systems normiert, gewichtet und anschließend linear kombiniert. Mit den Gewichten können wir verhindern, dass eine Metrik andere Metriken zu stark dominiert. Als Alternative zu einer Linearkombination könnten wir eine Paretofunktion verwenden. Allerdings kann es auch bei Paretofunktionen vorkommen, dass Lösungen berechnet werden, bei denen einige Metriken stark verbessert sind, andere Metriken aber stark verschlechtert sind. Diese Einschätzung konnten wir durch Experimente bestätigen. Unsere Zielfunktion beschreiben wir ausführlich in Abschnitt 6.3.

4.2.4. Selektion

Durch Mutation und Rekombination entstehen neue potentielle Lösungen. Die Selektion bestimmt die Elemente der Population, die in den nächsten Evolutionsschritt übernommen werden sollen und verwirft die restlichen Elemente wieder. Die Zielfunktion dient als Kriterium zur Auswahl der zu verwerfenden Elemente. Der Algorithmus wählt allerdings nicht zwangsläufig die bezüglich der Zielfunktion besten Lösungselemente aus. Die Auswahl findet immer in Gruppen von n Lösungselementen statt, n wird üblicherweise zwischen 2 und 8 gewählt. Der Algorithmus wählt die bezüglich der Zielfunktion besten Lösungselemente dieser Kleingruppe aus. Dieses Vorgehen wird Turnierauswahl [BNKF98] genannt und hat den Vorteil, dass neu erzeugte Lösungselemente beibehalten werden können, obwohl sie eine niedrigere Fitness als alle bisher in der Population enthaltenen Lösungselemente haben. Mit dieser Selektionsstrategie können lokale Extrema besser überwunden werden. Unsere Selektionsstrategie behandeln wir detailliert in Abschnitt 6.4.

4.3. Ablauf des Verfahrens

Nachdem wir die Hauptbestandteile unseres Verfahren beschrieben haben, stellen wir nun den gesamten Ablauf unseres Verfahrens vor. Dieser Ablauf ist schematisch in Abbildung 4.1 dargestellt. Bevor das Verfahren startet, kann ein Softwareingenieur es mit verschiedenen Parametern an seine Bedürfnisse anpassen.

Da er in der Regel weiß, welche Änderungen er bald durchführen muss und welche Systemteile er nicht ändern darf, kann er die in der Zielfunktion verwendeten Metriken unterschiedlich gewichten und bestimmte Systemteile von unserem Verfahren ausschließen. Es kann zum Beispiel vorkommen, dass ein System bereits besonders gute Kopplungswerte hat, da nur sehr wenige Teilsysteme existieren. Dementsprechend wird ein

Softwareingenieur die Kohäsionsmetriken stärker gewichten und sogar eine Verschlechterung der Kopplungswerte akzeptieren.

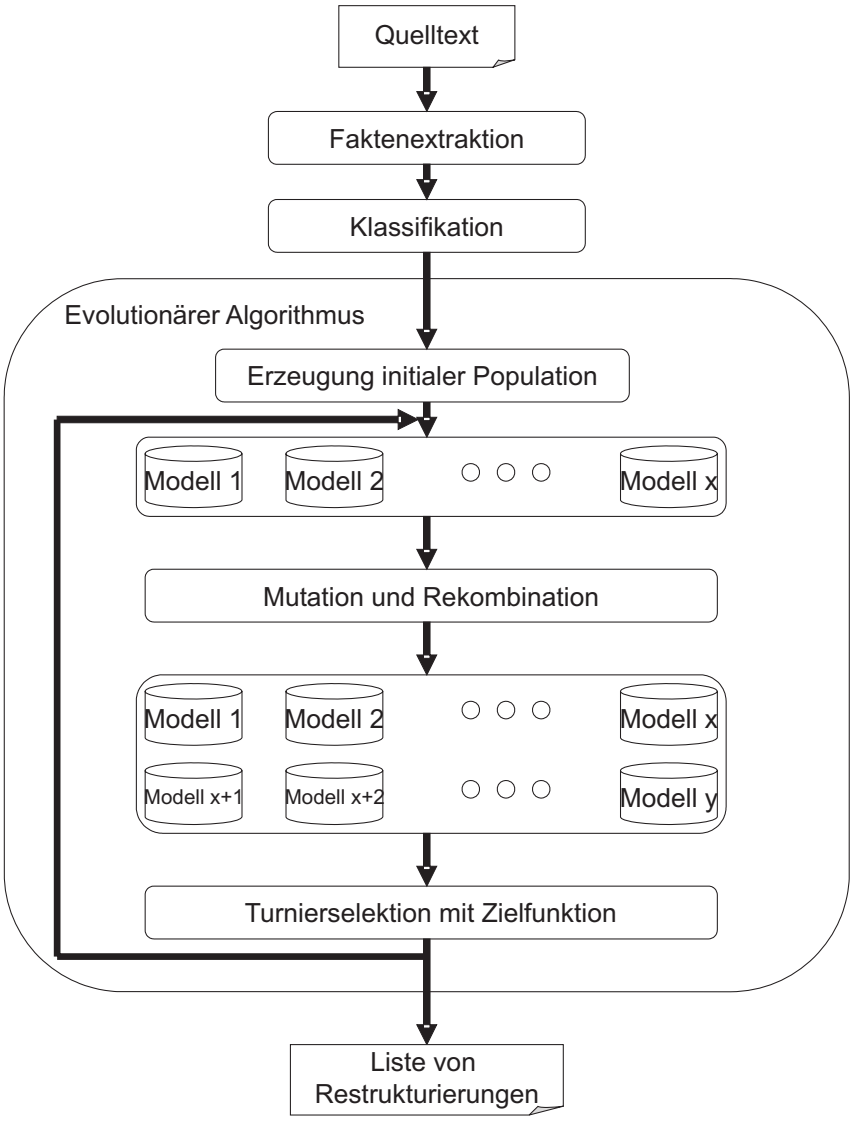


Abbildung 4.1.: Schematischer Ablauf des Verfahrens

Im ersten Schritt des Verfahrens - der Faktenextraktion - führt ein Faktenextraktor auf dem Quelltext des Systems die aus Übersetzungsvorgängen bekannten Vorgänge lexikalische, syntaktische und semantische Analyse durch. Wir erhalten als Zwischenergebnis einen attributierten Syntaxbaum mit Querverweisen, der den Quelltext des Systems repräsentiert. Dieser Baum wird in unser Modell - unseren Phänotyp - überführt. Dieser Schritt ist sprachabhängig, d.h. für jede unterstützte Sprache muss ein separater Faktenextraktor geschrieben werden.

Unser Verfahren muss Strukturen bewahren, die gängige Entwurfsheuristiken bewusst verletzen (siehe Kapitel 2.4). Diese Strukturen werden im nächsten Schritt, den wir

als *Klassifikation* bezeichnen, identifiziert. Kandidaten für solche Strukturen identifizieren wir werkzeuggestützt, da eine ausschließlich manuelle Erkennung zu aufwändig ist. Zur werkzeuggestützten Erkennung nutzen wir Kenntnisse über den Aufbau dieser Strukturen, die als Bedingungen formuliert werden. Mit Hilfe dieser Bedingungen und deduktivem Schließen können wir diese Strukturen bestimmen. Unser Verfahren kann problemlos durch andere Verfahren ersetzt werden, die zusätzlich dynamische Informationen verarbeiten [HML03]. Durch die dynamische Information können diese Verfahren weitere Strukturen wie z.B. zusätzliche Entwurfsmuster identifizieren.

Der nächste Schritt unseres Verfahrens ist die Ausführung des evolutionären Algorithmus. Zuerst erzeugen wir unsere Startpopulation. Diese besteht aus x Elementen, die jeweils aus einem Genotyp und einem Phänotyp bestehen. Der Genotyp ist zu Anfang leer, weil noch keine Modellrestrukturierungen ausgeführt wurden. Somit ist auch der Phänotyp bei allen Elementen identisch und zwar gleich dem ursprünglichen Systemmodell. Der aktuelle Phänotyp könnte zwar anhand des Genotyps und des ursprünglichen Phänotyps berechnet werden, es ist jedoch effizienter, den veränderten Phänotyp zusammen mit dem veränderten Genotyp zu speichern.

Anschließend beginnt die eigentliche Evolution. Der Algorithmus erzeugt neue mögliche Lösungen mit Hilfe der Operatoren *Mutation* und *Rekombination*. Die Populationsgröße steigt durch mehrfache Anwendung dieser Operatoren auf y Elemente an. Mit der in Abschnitt 4.2.4 beschriebenen Turnierauswahl wählen wir die x Elemente aus, die wir in die nächste Generation übernehmen.

Die Evolution bricht nach einer vorgegebenen Anzahl von Schritten ab. Diese Anzahl hängt von der Menge der eingesetzten Modellrestrukturierungen, der Größe des Systemmodells und der initialen Qualität der Struktur ab. Ein Nutzer kann nachträglich feststellen, ob die Anzahl der Evolutionsschritte zu hoch oder zu niedrig war, indem er den Verlauf der Werte der Zielfunktion betrachtet. Aus diesen Werten kann er ablesen, ob das Verfahren bereits konvergiert. Allerdings kann aus dem Verlauf der Zielfunktion nicht zweifelsfrei ermittelt werden, ob das Verfahren konvergiert hat, da durch die zufällige Auswahl der Modellrestrukturierungen theoretisch immer noch weitere Verbesserungen erfolgen können.

Das Ergebnis unseres Verfahrens ist eine Liste von Modellrestrukturierungen, die sich aus dem Genotyp des besten Elements der Population ergibt. Ein Softwareingenieur kann die durch die Modellrestrukturierungen vorgegebenen Restrukturierungen entweder manuell oder werkzeuggestützt durchführen. Es existieren viele Werkzeuge zur automatisierten Quelltexttransformation. Im Zusammenhang mit der vorliegenden Arbeit wurde beispielsweise ein prototypischer Assisstent entwickelt, mit dessen Hilfe ein Nutzer Restrukturierungen von in Java geschriebenen Programmen ausführen kann [Mül05]. Erforderliche Parameter, die normalerweise ein Nutzer angeben muss, können ebenfalls von den Modellrestrukturierungen übernommen werden. Eventuell müssen nachträglich noch aussagekräftige Namen vergeben und Kommentare angepasst werden [Cal88].

Ein Softwareingenieur kann die Modellrestrukturierungen noch einmal überprüfen, bevor er sie auf dem eigenen System ausführt. Ist er mit den Modellrestrukturierungen nicht einverstanden, so kann dies zum einen daran liegen, dass er die Parameter des Verfahrens noch nicht richtig eingestellt hat. Eventuell muss er diese Parameter anpassen,

weitere Strukturen vom Verfahren ausnehmen und das Verfahren erneut starten. Zum anderen treten manche Beziehungen zwischen Strukturelementen erst zur Laufzeit auf und können deshalb nur mit dynamischen Analysen exakt ermittelt werden. Nach diesem Überblick über den Ablauf unseres Verfahren stellen wir im nächsten Kapitel unser Metamodell vor.

Kapitel 5.

Modellbildung

Unser Verfahren arbeitet nicht direkt auf dem Quelltext eines Systems, sondern auf einem Modell des Quelltextes. Dieses Modell nennen wir auch Systemmodell. Wir legen mit einem Metamodell fest, welche Elemente des Quelltextes von einem Systemmodell modelliert werden. Im weiteren Verlauf dieses Kapitels beschreiben wir zunächst den Aufbau unseres Metamodells. Zum Abschluss des Kapitels beschreiben wir, wie wir Strukturelemente klassifizieren. Die durch die Klassifikation gewonnene Information benötigen wir, um Strukturelemente bewahren zu können, die Entwurfsprinzipien bewusst verletzen.

5.1. Anforderungen an das Metamodell

Unser Metamodell muss eine Reihe von Anforderungen erfüllen, damit wir daraus erzeugte Systemmodelle zur Strukturverbesserung einsetzen können. Es muss:

1. Operationen anbieten, die die Struktur des Modells verändern. Diese Operationen müssen auf das ursprüngliche System übertragen werden können und garantieren, dass das von außen beobachtbare Ein- / Ausgabeverhalten des Systems gleich bleibt.

2. Informationen anbieten, mit denen gängige Metriken zur Bewertung der Qualität der Systemstruktur berechnet werden können.

3. möglichst leichtgewichtig sein. Das Modell darf keine unnötigen Details enthalten, es soll nur die Strukturelemente modellieren, die zwingend benötigt werden.

4. sprachunabhängig für stark typisierte, ausdrucksbasierte objektorientierte Sprachen geeignet sein, um unser Verfahren nicht unnötig einzuschränken.

5.2. Aufbau des Metamodells

Um den Aufbau unseres Metamodells festlegen zu können, müssen wir folgende Fragen beantworten:

1. Welche Strukturelementtypen müssen wir betrachten?

2. Welche Kooperationstypen zwischen Objekten müssen wir berücksichtigen?

Unser Metamodell soll Operationen anbieten, die die Struktur eines Softwaresystems verhaltensbewahrend verändern. Diese Operationen müssen also alle Strukturelemente verändern können, die beim Programmieren-im-Großen von Interesse sind: Sie verändern erstens die Zuordnung von *Klassen* zu *Teilsystemen* und zweitens die Zuordnung von *Methoden* und *Attributen* zu *Klassen*. Diese Strukturelementtypen müssen somit auf jeden Fall von unserem Metamodell unterstützt werden. In Abschnitt 5.2.5 werden wir sehen, dass *Lokale Variablen* und *Formale Parameter* ebenfalls verändert werden müssen, so dass wir auch diese Strukturelementtypen in unser Metamodell aufnehmen müssen.

5.2.1. Unterstützte Strukturelementtypen

Für *Teilsysteme* gibt es in den meisten objektorientierten Programmiersprachen kein eigenes Sprachelement, weshalb Teilsysteme oft durch Konventionen festgelegt werden. In *Java* zum Beispiel bestimmen oft *Pakete* die Grenzen von Teilsystemen, d.h. alle Klassen innerhalb eines Pakets gehören zu demselben Teilsystem. Der Name des Teilsystems ist gleich dem Paketnamen. Ebenso verbreitet ist die Konvention, Teilsystemgrenzen mit Hilfe von Verzeichnissen zu definieren. Alle in einem Verzeichnis enthaltenen Dateien, bzw. die in diesen Dateien definierten Klassen, gehören zu demselben Teilsystem. Der Name des Teilsystems ist gleich dem Namen des Verzeichnisses. Diese Variante wird häufig für in *C++* implementierte Systeme verwendet. In unserem Metamodell muss ein Teilsystem mindestens eine Klasse enthalten. Leere Teilsysteme sind bedeutungslos und werden deshalb von unserem Metamodell nicht unterstützt.

Klassen liefern Baupläne für die elementaren Teilsysteme: die Objekte. Das Verhalten der Objekte wird durch die in den Klassen enthaltenen Attribute und Methoden festgelegt. Enthält eine Klasse nur für einen Teil ihrer Methoden Ablaufpläne, dann muss sie als abstrakt gekennzeichnet werden und es können zur Laufzeit keine Objekte vom Typ dieser Klasse erzeugt werden, da Ihr Bauplan unvollständig ist. Manche objektorientierte Sprachen wie z.B. *Java* unterstützen reine Schnittstellenklassen. Diese enthalten ausschließlich Methodendeklarationen ohne Ablaufpläne und gegebenenfalls noch Konstantendefinitionen.

Unser Metamodell unterstützt Vererbungsbeziehungen zwischen Klassen. Die drei bekannten Vererbungstypen *Trennung zwischen Schnittstelle und Implementierung*, *Spezialisierung* und *Implementierungsvererbung* behandeln wir gleich.

Softwaresysteme werden selten vollständig neu implementiert, sondern verwenden oft vorhandene Teilsysteme in Form von Bibliotheken oder Rahmenwerken. Die verwendeten *Fremdteilsysteme* und *Fremdklassen* modellieren wir und bezeichnen sie als Bibliotheksteilsysteme bzw. Bibliotheksklassen. Wir können die Fremdstrukturen nicht verändern, allerdings benötigen wir sie für ein vollständiges Systemmodell. Strukturelemente, die in einer Bibliotheksklasse enthalten sind, sind ebenfalls Fremdstrukturen. Teilsysteme sind Bibliotheksteilsysteme, wenn sie ausschließlich Bibliotheksklassen enthalten.

Unser Metamodell unterstützt drei verschiedene Arten von Variablen: *Lokale Variablen*, *Formale Parameter* und *Attribute*. Jede Variable besitzt einen bestimmten Typ,

der durch eine der Klassen definiert ist.

Eine geordnete Menge von Variablen mit gemeinsamen Namen nennen wir *Reihung*. Die Einzelwerte der Reihung können mit Indizierung ausgewählt werden [Goo96]. Wir modellieren Reihungen nicht als eigene Strukturelemente, sondern durch besonders gekennzeichnete Variablen.

Attribute sind die durch Werte erfassbaren Eigenschaften eines Objekts. Unter statischen oder Klassenattributen verstehen wir Attribute, die für alle Instanzen einer Klasse denselben Wert besitzen. *Lokale Variablen* sind temporäre Variablen und werden im Rumpf einer Methode vereinbart. Auf sie kann nur innerhalb dieses Methodenrumpfs zugegriffen werden. *Formale Parameter* legen die Eingabeschnittstelle einer Methode fest und können innerhalb der Methode wie lokale Variablen verwendet werden.

Methoden enthalten Ablaufpläne für das Verhalten der Objekte. Die Signatur einer Methode besteht aus ihrem Namen und einer geordneten Menge von *formalen Parametern*. Eine Methode kann einen Wert mit einem bestimmten Typ zurückliefern. Konstruktoren sind besondere Methoden, die Objekte erzeugen. Im Gegensatz zu Konstruktoren zerstören Destruktoren Objekte. Initialisierer von Attributen bilden wir ebenfalls auf Methoden ab. Viele objektorientierte Sprachen unterstützen statische Methoden. Diese greifen ausschließlich auf statische Attribute der sie umgebenden Klasse zu. Sie können aufgerufen werden, ohne dass ein Objekt der sie umgebenden Klasse erzeugt wurde. Legt eine Methode nur eine Schnittstelle in Form von Signatur und Rückgabetyp fest, ohne einen Ablaufplan zu definieren, bezeichnen wir sie als abstrakte Methode.

5.2.2. Unterstützte Kooperationstypen

Zusätzlich zu den gerade beschriebenen Strukturelementen und elementaren Beziehungen muss unser Metamodell weitere Beziehungen unterstützen, die die Kooperation zwischen Objekten modellieren. Wir modellieren alle statisch beobachtbaren Beziehungen zwischen Objekten, da jede dieser Beziehungen die Qualität der Struktur mitbestimmt.

Könnten wir zusätzlich Beziehungen berücksichtigen, die erst zur Laufzeit auftreten, würde unser Verfahren noch mächtiger werden. Allerdings können wir diese dynamischen Informationen nur mit viel Aufwand gewinnen und verzichten deshalb auf sie. Bauer hat diese Tatsache bereits in [Bau05] beschrieben: Dynamische Beziehungen können auf zwei verschiedene Arten ermittelt werden. Die erste Alternative besteht darin, den existierenden Quelltext eines Systems mit zusätzlichem Quelltext zu instrumentieren, der Informationen über die dynamischen Beziehungen ausgibt. Das System muss also ausgeführt werden, um die Informationen zu gewinnen. Bei der Ausführung müssen alle relevanten Ausführungspfade abgedeckt werden, was im Allgemeinen nur sehr schwer zu erreichen ist. Die zweite Variante, um dynamische Informationen zu gewinnen, ist die abstrakte Interpretation. Um zum Beispiel Interaktionsprotokolle in objektorientierten Systemen zu ermitteln, ist eine interprozedurale Analyse des Daten- und Steuerflusses erforderlich. Dies ist sehr rechenzeit- und speicheraufwändig, und trotz einiger Fortschritte in letzter Zeit können diese Informationen für große Systeme noch nicht berechnet werden.

Jede statisch beobachtbare Beziehung modellieren wir in unserem Metamodell als Zugriff.

Zugriffe erfolgen innerhalb von Methodenrümpfen oder Initialisierern. In objektorientierten Programmiersprachen können Zugriffe auf verschiedene Arten aneinander gereiht werden. In Java wird hierzu der Punktoperator „.“ verwendet, in C++ der Pfeiloperator „→“und der Punktoperator „.“. Auf diese Art und Weise aneinander gereihte Zugriffe fassen wir zu einer geordneten *Zugriffskette* zusammen.

Ein *Klassenzugriff* modelliert den direkten Zugriff auf den Namen einer Klasse. Klassenzugriffe werden zum Beispiel verwendet, um auf ein statisches Attribut oder eine statische Methode zugreifen zu können.

Ein *Methodenzugriff* modelliert einen Aufruf einer Methode. Die aufgerufene Methode speichern wir als Ziel des Methodenzugriffs.

Zugriffe auf Variablen werden je nach Art der zugegriffenen Variable als *LokalerVariablenZugriff*, *ParameterZugriff* oder *Attributzugriff* modelliert. Analog zu den Methodenzugriffen modellieren wir als Ziel des Zugriffs die zugegriffene Variable.

Objekte können eine Referenz auf sich selbst durch einen Zugriff auf `this` erhalten. In unserem Metamodell modellieren wir einen solchen Zugriff folglich als *thisZugriff*. Mit einem *SuperZugriff* kann ein direkter Zugriff auf die Oberklasse ausgedrückt werden, wie er in vielen objektorientierten Sprachen erlaubt ist. Einen Zugriff auf ein Literal modellieren wir als *LiteralZugriff*.

Die Argumente von Methodenaufrufen können beliebige Ausdrücke sein. Das bedeutet, dass ein Argument nicht immer als eine einzige Zugriffskette modelliert werden kann, da Zugriffsketten mit Operatoren verknüpft werden können. Um trotzdem jedes Argument als eine Zugriffskette betrachten zu können, verwenden wir für diese Operatoren das Modellelement *Operatorzugriff*. Ein Operatorzugriff kann beliebig viele Zugriffsketten verknüpfen. Wir müssen Argumente von Methodenaufrufen explizit modellieren, da wir die Signatur von Methoden und die verwendeten Argumente verändern wollen.

Abbildung 5.1 und Abbildung 5.2 zeigen die Strukturelemente unseres Metamodells und die Beziehungen zwischen ihnen im Überblick.

5.2.3. Vorgegebene Mengen

Im folgenden beschreiben wir unser Metamodell in Form von vorgegebenen Mengen. Diese Mengen repräsentieren die von uns in Abschnitt 5.2 festgelegten Strukturelemente und Beziehungen. Zusätzlich zu den vorgegebenen Mengen definieren wir Elementaroperationen, mit denen der Zustand einer Ausprägung des Metamodells verändert und abgefragt werden kann. Weiterhin beschreiben wir auf den Elementaroperationen aufbauende Modellrestrukturierungen, die verhaltensbewahrend auf ein ursprüngliches System übertragen werden können.

Gegeben sei ein **Softwaresystem** S. Dann ist

$\mathcal{T}$	Die Menge aller Teilsysteme (ohne Klassen).
$\mathcal{S}_t$	Die Menge aller Klassen, die zum Teilsystem t gehören.
$\mathcal{K}_S$	Die Menge der Systemklassen (ohne Bibliotheksklassen).

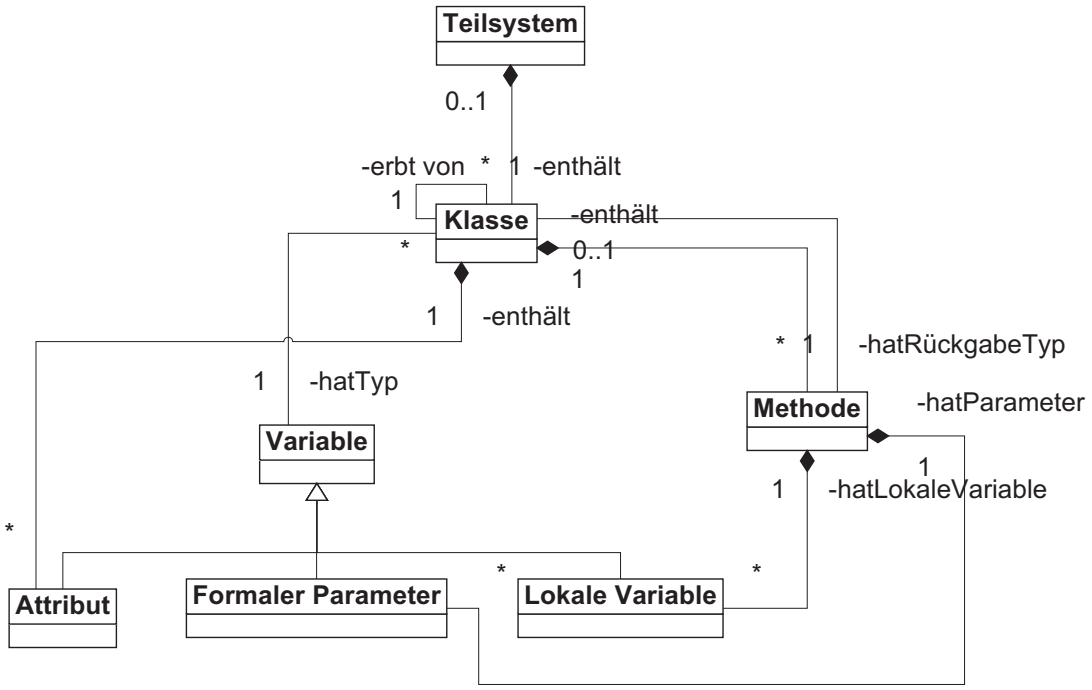


Abbildung 5.1.: Die Elemente unseres Metamodells

$\mathcal{K}_S^{bib}$	Die Menge der Bibliotheksklassen.
$\mathcal{K}_S^A$	Die Menge aller abstrakten Klassen.
$\mathcal{K}_S^I$	Die Menge der Schnittstellenklassen.
$\mathcal{T}^{Muster}$	Die Menge aller Teilsysteme des Systems, die während der Klassifikation markiert wurden.
$\mathcal{K}^{Muster}$	Die Menge aller Klassen des Systems, die während der Klassifikation markiert wurden.
$\mathcal{M}^{Muster}$	Die Menge aller Methoden des Systems, die während der Klassifikation markiert wurden.
$\mathcal{A}^{Muster}$	Die Menge aller Attribute des Systems, die während der Klassifikation markiert wurden.
$\mathcal{Z}_S^*$	Die Menge aller Methoden- und Attributzugriffe (aller Methoden) des Systems.
$\mathcal{Z}$	Die Menge aller Zugriffe eines Systems.
$\mathcal{Z}_{lese}$	Die Menge aller Lesegriffe eines Systems.
$\mathcal{Z}_{schreib}$	Die Menge aller Schreibzugriffe eines Systems.
$\mathcal{ZK}$	Die Menge aller Zugriffsketten eines Systems.

Gegeben sei eine **Klasse** k. Dann ist

$\mathcal{M}_k$	Die Menge der von k definierten, nichtstatischen Methoden.
$\mathcal{M}_k^s$	Die Menge der von k definierten, statischen Methoden.
$\mathcal{M}_k^a$	Die Menge der von k abstrakt deklarierten Methoden.

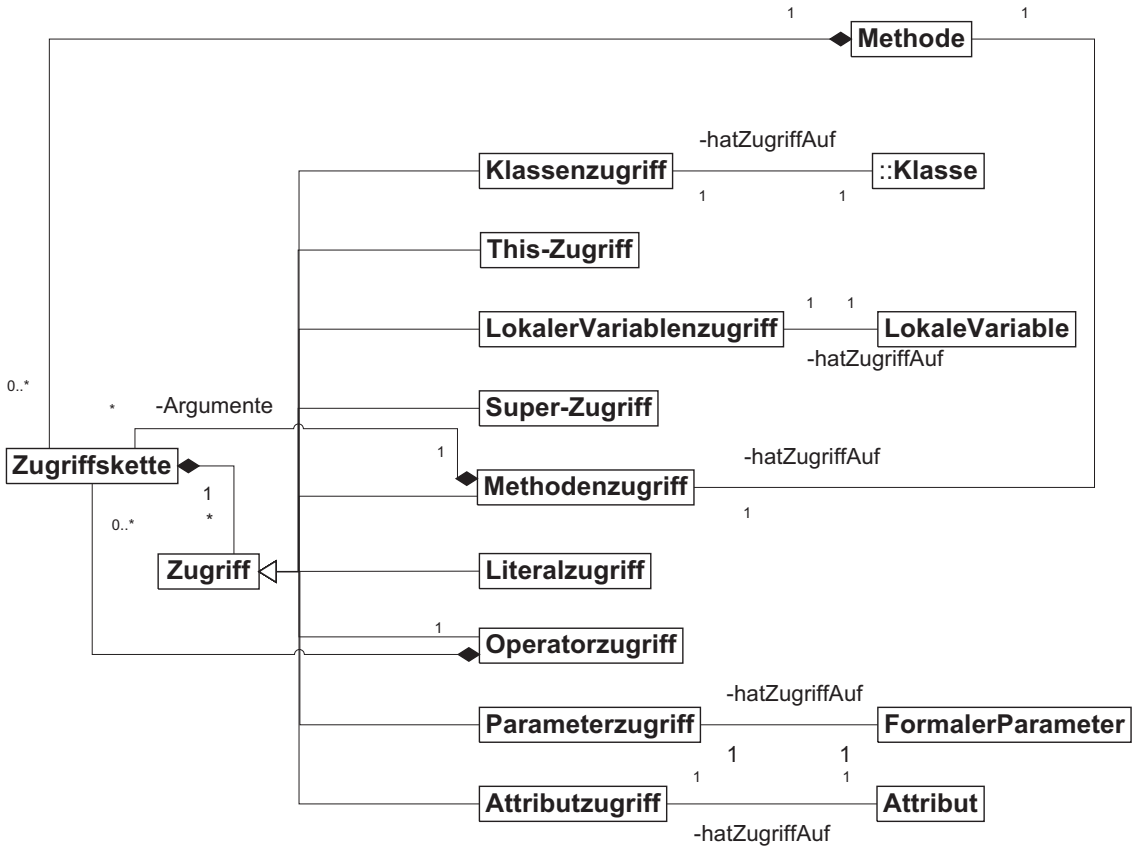


Abbildung 5.2.: Die Beziehungen zwischen den Elementen eines Systems

(Es gilt: $\mathcal{M}_k^a \cap \mathcal{M}_k = \emptyset$)

$\mathcal{M}_k^{Kon}$ Die Menge der Konstruktoren von k.
(Es gilt: $\mathcal{M}_k^{Kon} \cap \mathcal{M}_k = \emptyset$)

$\mathcal{M}_k^{Des}$ Die Menge der Destruktoren von k.
(Es gilt: $\mathcal{M}_k^{Des} \cap \mathcal{M}_k = \emptyset$)

$\mathcal{M}_k^{Ini}$ Die Menge der Initialisierer von k.
(Es gilt: $\mathcal{M}_k^{Ini} \cap \mathcal{M}_k = \emptyset$)

$\mathcal{A}_k$ Die Menge der von k deklarierten, nichtstatischen Attribute.

$\mathcal{A}_k^s$ Die Menge der von k deklarierten, statischen Attribute.

O_k Die Menge der direkten Oberklassen von k

$\mathcal{O}_k^*$ Die Menge aller Oberklassen von k.

$\mathcal{U}_k$ Die Menge der direkten Unterklassen von k.

$\mathcal{U}_k^*$ Die Menge aller Unterklassen von k.

Gegeben sei eine **Methode** m. Dann ist

$\mathcal{R}_m$ Der Rückgabetyp der Methode.

$\mathcal{P}_m$ Die geordnete Menge der Parameter von m.

$\mathcal{LV}_m$ Die Menge der lokalen Variablen von m.

$\mathcal{Z}_m$	Die Menge aller von m zugegriffenen Elemente. Zugriffe erfolgen (wie in 5.2.2 vorgestellt) auf Klassen, Methoden, Attribute, Parameter, lokale Variablen, Literale, `this` und `super`.
$\mathcal{Z}_m^A$	Die Menge aller von m zugegriffenen Attribute.
$\mathcal{Z}_m^M$	Die Menge aller von m zugegriffenen Methoden.
$\mathcal{Z}_m^K$	Die Menge aller von m direkt referenzierten Klassen (in Form einer Referenz auf den Klassennamen).
$\mathcal{Z}_m^P$	Die Menge aller von m zugegriffenen Parameter.
$\mathcal{Z}_m^{LV}$	Die Menge aller von m zugegriffenen lokalen Variablen.
$\mathcal{ZK}_m$	Die Menge aller Zugriffsketten einer Methode m.

Gegeben sei eine Zugriffskette zk, dann ist

$\mathcal{Z}_{zk}$	Die geordnete Menge aller Zugriffe einer Zugriffskette zk.

Gegeben sei ein Operatorzugriff oz, dann ist

$\mathcal{ZK}_{oz}$	Die geordnete Menge aller Zugriffsketten des Operatorzugriffs.

Gegeben sei ein Methodenzugriff mz, dann ist

$\mathcal{ZK}_{mz}$	Die geordnete Menge aller Zugriffsketten des Methodenzugriffs. Dies sind die Zugriffsketten innerhalb der Aufrufparameter.

5.2.3.1. Hilfsmengen

Zusätzlich zu den bereits definierten Mengen leiten wir folgende Hilfsmengen ab:

$\mathcal{A}_k^g$	Die Menge aller Attribute einer Klasse k. $\mathcal{A}_k^g = \mathcal{A}_k \cup \mathcal{A}_k^S$
$\mathcal{M}_k^g$	Die Menge aller Methoden einer Klasse k. (inkl. Konstruktoren, Destruktoren und Initialisierern) $\mathcal{M}_k^g = \mathcal{M}_k \cup \mathcal{M}_k^S \cup \mathcal{M}_k^{Kon} \cup \mathcal{M}_k^{Des} \cup \mathcal{M}_k^{Ini}$
$\mathcal{M}_S^*$	Die Menge aller nichtstatischen Methoden aller Klassen des Systems S. $\mathcal{M}_S^* := \bigcup_{k \in \mathcal{K}_S} \mathcal{M}_k$
$\mathcal{C}_m$	Die Menge aller nichtstatischen Methoden, die die Methode m aufrufen. $\mathcal{C}_m := \{m' \mid m' \in \mathcal{M}_S^* \wedge m \in \mathcal{Z}_{m'}^M\}$
$\mathcal{C}_a$	Die Menge aller nichtstatischen Methoden, die das Attribut a benutzen. $\mathcal{C}_a := \{m \mid m \in \mathcal{M}_S^* \wedge a \in \mathcal{Z}_m^A\}$
$\mathcal{MZ}_m$	Die Menge aller Zugriffe auf die Methode m. $\mathcal{MZ}_m := \{z \mid z \in \mathcal{Z}_S^* \wedge l(z) = m\}$ l(z) gibt für einen Zugriff das zugegriffene Element zurück. (siehe nächster Abschnitt)
$\mathcal{AZ}_a$	Die Menge aller Zugriffe auf das Attribut a.

$$\mathcal{AZ}_a := \{z | z \in \mathcal{Z}_S^* \wedge l(z) = a\}$$

5.2.4. Elementaroperationen

5.2.4.1. Abfrageoperationen

Um Informationen über das Modell zu erhalten, definieren wir folgende Operationen:

$istReihung(v)$	Liefert `wahr` zurück, falls v eine Reihung ist.
$typ(p)$	Liefert für einen Parameter, eine lokale Variable oder für ein Attribut p dessen statischen Typ zurück.
$name(e)$	Liefert für eine Klasse, ein Attribut, eine Methode, einen Parameter oder eine lokale Variable e den vollqualifizierten Namen zurück.
$l(z)$	Gibt für einen Zugriff z das zugegriffene Element zurück (z kann ein Zugriff auf eine Klasse, eine Methode, ein Attribut, ein Parameter, eine lokale Variable, `this` oder `super` sein)
$inKlasse(z)$	Die Klasse, in der die Methode implementiert ist, aus der der Zugriff z erfolgt.
$typUeber(z)$	Liefert für Zugriffe auf Methoden und Attribute den Typ bzw. die Klasse des Objekts, über den der Zugriff erfolgt.
$zvor(z)$	Gibt den Zugriff zurück, der in der z enthaltenden Zugriffskette vor z steht.
$praefix(z)$	Liefert alle Zugriffe der z enthaltenden Zugriffskette, die vor z stehen.
$argument(z_m, p_i)$	Liefert für einen Methodenzugriff z_m die Zugriffskette, die als Argument für den Parameter p_i übergeben wird.
$hatpraefix(zk, zk_{pr})$	Liefert den Wert `wahr` zurück, falls die Zugriffskette zk mit der Zugriffskette zk_{pr} beginnt.
$inMethode(z)$	Liefert die Methode zurück, in der sich der Zugriff z befindet.
$zKette(z)$	Liefert die Zugriffskette, in der sich der Zugriff z befindet.
$=_{SR}(m_1, m_2)$	Liefert `wahr`, falls die Methoden m_1 und m_2 die gleiche Signatur und den gleichen Rückgabetyp haben $=_{SR}(m_1, m_2) \Longleftrightarrow$ $\mathcal{P}_{m_1} = \mathcal{P}_{m_2} \wedge name(m_1) = name(m_2) \wedge \mathcal{R}_{m_1} = \mathcal{R}_{m_2}$
$zugriffsTyp(z)$	Gibt für einen Zugriff z den Typ des Zugriffs zurück.

Für Klassen definieren wir eine weitere Operation, mit der wir bestimmen können, ob eine Klasse k_1 eine andere Klasse k_2 'benutzt'. Eine Klasse k_1 benutzt eine andere Klasse k_2, wenn sie bzw. ihre Bestandteile die Klasse k_2 oder deren Bestandteile referenzieren. Basierend auf den von uns definierten Mengen benutzt k_1 eine Klasse k_2 wenn:

- k_1 ein Attribut vom Typ k_2 hat.

- Eine Methode aus k_1 eine lokale Variable vom Typ k_2 hat.
- Eine Methode aus k_1 einen Parameter vom Typ k_2 hat.
- k_2 eine direkte Oberklasse von k_1 ist.
- In einer Methode aus k_1 direkt auf die Klasse k_2 zugegriffen wird.
- Eine Methode aus k_1 eine Methode von k_2 aufruft.

Formal können wir diese Operation wie folgt beschreiben:

$$
\begin{aligned}
benutzt(k_1, k_2) \Longleftrightarrow \\
&\exists a \in \mathcal{A}^g_{k_1} \wedge typ(a) = k_2 \\
&\vee \exists m, p : m \in \mathcal{M}^g_{k_1} \wedge p \in \mathcal{P}_m \wedge typ(p) = k_2 \\
&\vee \exists m, lv : m \in \mathcal{M}^g_{k_1} \wedge lv \in \mathcal{LV}_m \wedge typ(lv) = k_2 \\
&\vee k_2 \in \mathcal{O}_{k_1} \\
&\vee \exists m : m \in \mathcal{M}^g_{k_1} : k_2 \in \mathcal{Z}_m \\
&\vee \exists m, m_2 : m \in \mathcal{M}^g_{k_1} \wedge m_2 \in \mathcal{M}^g_{k_2} \wedge m_2 \in \mathcal{Z}^M_m
\end{aligned}
$$

Als weitere Operation definieren wir die Kopplung zwischen einer Klasse k_1 und einer Klasse k_2 als die Summe

- der Anzahl der Attribute in k_1 vom Typ k_2 und der Anzahl der Zugriffe auf diese Attribute und
- der Anzahl der Parameter einer Methode in k_1 vom Typ k_2 und der Anzahl der Zugriffe auf diesen Parameter und
- der Anzahl der lokalen Variablen einer Methode in k_1 vom Typ k_2 und der Anzahl der Zugriffe auf diese lokalen Variablen und
- der Anzahl der Aufrufe einer Methode aus k_2 durch Methoden von k_1 und
- der Anzahl der direkten Zugriffe auf k_2 aus k_1 und
- falls k_2 eine Oberklasse von k_1 ist der Summe aller direkten Verwendungen der Oberklasse mit `super`.

Formal können wir die Kopplung wie folgt berechnen:

$$
\begin{aligned}
kopplung(k_1, k_2) \quad := \quad & \\
& |\{a | a \in \mathcal{A}^g_{k_1} \wedge typ(a) = k_2\}| \\
+ \; & |\{z | z \in \mathcal{Z} \wedge inKlasse(z) = k_1 \wedge l(z) \in \mathcal{A}^g_{k_1} \wedge typ(l(z)) = k_2\}| \\
+ \; & |\{p | \exists m : m \in \mathcal{M}^g_{k_1} \wedge p \in \mathcal{P}_m \wedge typ(p) = k_2\}| \\
+ \; & |\{z | \exists m : m \in \mathcal{M}^g_{k_1} \wedge z \in \mathcal{Z} \wedge inKlasse(z) = k_1 \wedge l(z) \in \mathcal{P}_m \\
& \wedge typ(l(z)) = k_2\}| \\
+ \; & |\{lv | \exists m : m \in \mathcal{M}^g_{k_1} \wedge lv \in \mathcal{LV}_m \wedge typ(lv) = k_2\}| \\
+ \; & |\{z | \exists m : m \in \mathcal{M}^g_{k_1} \wedge z \in \mathcal{Z} \wedge inKlasse(z) = k_1 \\
& \wedge l(z) \in \mathcal{LV}_m \wedge typ(l(z)) = k_2\}| \\
+ \; & |\{z | \exists m, m_2 : m \in \mathcal{M}^g_{k_1} \wedge m_2 \in \mathcal{M}^g_{k_2} \wedge m_2 \in \mathcal{Z}^M_m \\
& \wedge inMethode(z) = m \wedge l(z) = m_2\}| \\
+ \; & |\{z | inKlasse(z) = k_1 \wedge l(z) = k_2\}| \\
& \text{Falls } k_2 \in \mathcal{O}_{k_1} + 1 + |\{z | inKlasse(z) = k_1 \wedge l(z) = \texttt{super}\}|
\end{aligned}
$$

Aufbauend auf diesen beiden Operationen stellen wir analoge Operationen für Teilsysteme zur Verfügung. Ein Teilsystem t_1 benutzt ein Teilsystem t_2, wenn eine Klasse aus t_1 eine Klasse aus t_2 benutzt.

$$benutzt(t_1, t_2) \Longleftrightarrow \exists k_1, k_2 : k_1 \in \mathcal{S}_{t_1} \wedge k_2 \in \mathcal{S}_{t_2} \wedge benutzt(k1, k2)$$

Die gerichtete Kopplung zwischen einem Teilsystem t_1 und einem Teilsystem t_2 berechnen wir, indem wir die Kopplung aller möglichen Klassenpaare aus t_1 und t_2 berechnen.

$$kopplung(t_1, t_2) : \sum_{k_i \in S_{t_1}} \sum_{k_j \in S_{t_2}} kopplung(k_i, k_j)$$

5.2.4.2. Verändernde Operationen

Um das Modell zu verändern, definieren wir zusätzlich zu üblichen Mengenoperationen folgende Operationen:

$haengeVor(z^*, z)$	Fügt den Zugriff z^* in der z enthaltenden Zugriffskette direkt vor z ein.
$ersetzeDurch(z^*, z)$	Ersetzt den Zugriff z^* in der umgebenden Zugriffskette durch den Zugriff z.
$loesche(z)$	Löscht den Zugriff z aus der z enthaltenden Zugriffskette.
$ersetze(z_1, z_2, zkette)$	Dieser Operator ersetzt in der Zugriffskette $zkette$ alle Vorkommen der Zugriffskette z_1 durch die Zugriffskette z_2.
$loesche(zk_1, zk_2)$	Löscht die Zugriffskette zk_1 aus der Zugriffskette zk_2, falls sie in dieser vorhanden ist.

$setzeArg(z_m, p_i, zk)$	Ersetzt für den Methodenaufruf z_m das Argument für den Parameter p_i durch die Zugriffskette zk.
$cl(zkette)$	Erzeugt eine Kopie der Zugriffskette $zkette$.
$erzeugeP(name, typ)$	Erzeugt einen Parameter mit dem Namen $name$ und dem Typ typ.
$erzeugeZ(element)$	Erzeugt einen neuen Zugriff auf das Modellelement $element$. $element$ kann eine Klasse, ein Attribut, eine Methode, ein Parameter, eine lokale Variable, `this` oder `super` sein.

5.2.5. Modellrestrukturierungen

In diesem Abschnitt beschreiben wir unsere Modellrestrukturierungen, indem wir zunächst die zu erfüllenden Vorbedingungen und anschließend die einzelnen Schritte der Restrukturierung beschreiben. Unsere Vor- und Nachbedingungen orientieren sich an den Vor- und Nachbedingungen aus [Kut02].

Für alle Modellrestrukturierung muss folgende Vorbedingung gelten: Alle verwendeten Parameter dürfen nicht während der Klassifikation markiert worden sein, d.h. falls

- ein Parameter ein Teilsystem t ist, muss gelten:
  $t \notin \mathcal{T}^{Muster}$
- ein Parameter eine Klasse k ist, muss gelten:
  $k \notin \mathcal{K}^{Muster}$
- ein Parameter eine Methode m ist, muss gelten:
  $m \notin \mathcal{M}^{Muster}$
- ein Parameter ein Attribut a ist, muss gelten:
  $a \notin \mathcal{A}^{Muster}$

Weiterhin dürfen die Parameter keine Fremdstrukturen sein (siehe Abschnitt 5.2.1). Ob ein Strukturelement eine Fremdstruktur ist, kann mit Hilfe der Menge $\mathcal{K}_S^{bib}$ bestimmt werden.

Die ersten drei Modellrestrukturierungen verändern die Einteilung der Klassen eines Systems in Teilsysteme. Alle weiteren Modellrestrukturierungen verändern den Aufbau der Klassen eines Systems.

5.2.5.1. Extrahiere Teilsystem

Parameter:

1. Teilsystem t, das aufgeteilt werden soll
2. Menge der Klassen B, die zu dem neuen Teilsystem gehören sollen

Mit der Modellrestrukturierung *Extrahiere Teilsystem* können wir ein Teilsystem t_E von einem Teilsystem t abspalten.

Wir führen die Modellrestrukturierung folgendermaßen aus:

1. Als erstes erzeugen wir das neue Teilsystem t_E.

$$\mathcal{T} := \mathcal{T} \cup \{t_E\}$$

2. Dann entfernen wir die Klassen B aus dem Teilsystem t.

$$\mathcal{S}_t := \mathcal{S}_t \setminus B$$

3. Anschließend fügen wir die Klassen B zu t_E hinzu.

$$\mathcal{S}_{t_E} := B$$

4. Im letzten Schritt weisen wir t_E einen eindeutigen, zufälligen Namen zu, der noch nicht existiert.

5.2.5.2. Integriere Teilsystem

Parameter:

1. Teilsystem t_I, das integriert werden soll
2. Teilsystem t, in das t_I integriert werden soll

Die Modellrestrukturierung *Integriere Teilsystem* verschiebt die Klassen eines Teilsystems t_I in ein Teilsystem t und löscht anschließend das zu integrierende Teilsystem t_I.

Die eigentliche Restrukturierung führen wir folgendermaßen aus:

1. Zuerst fügen wir die Klassen von t_I den Klassen des Teilsystems t hinzu.

$$\mathcal{S}_t = \mathcal{S}_t \cup \mathcal{S}_{t_I}$$

2. Danach löschen wir das Teilsystem t_I.

$$\mathcal{T} = \mathcal{T} \setminus \{t_I\}$$

5.2.5.3. Verschiebe Klasse

Parameter:

1. die zu verschiebende Klasse k
2. das Quellteilsystem t_1
3. das Zielteilsystem t_2

Die Modellrestrukturierung *Verschiebe Klasse* verschiebt eine Klasse k von einem Teilsystem t_1 in ein Teilsystem t_2.
Wir führen die Modellrestrukturierung folgendermaßen aus:

1. Zunächst entfernen wir k aus der Menge der Klassen von t_1.

$$\mathcal{S}_{t_1} := \mathcal{S}_{t_1} \setminus \{k\}$$

2. Abschließend fügen wir k zur Menge der Klassen von t_2 hinzu.

$$\mathcal{S}_{t_2} := \mathcal{S}_{t_2} \cup \{k\}$$

5.2.5.4. Verschiebe Methode

Parameter:

1. die zu verschiebende Methode m
2. die Quellklasse k_q
3. die Zielklasse k_z
4. die Strategie, um auf das alte Quellobjekt zugreifen zu können
5. die Strategie, um ein neues Zielobjekt zu erhalten

Die Restrukturierung *Verschiebe Methode* verschiebt eine Methode m außerhalb der Klassenhierarchie von einer Klasse k_q in eine Klasse k_z. Damit wir die Modellrestrukturierung ausführen können, müssen folgende Vorbedingungen erfüllt sein:

1. k_z und k_q stehen nicht in einer Vererbungsbeziehung.
2. m ist nicht polymorph.
3. m enthält keine expliziten `super`-Zugriffe.

4. Die Zielklasse ist keine Schnittstellenklasse

5. Falls m nicht statisch ist, muss die Zielklasse einem Parametertyp von m oder einem Attributtyp von k_q entsprechen.

Formal können wir die Vorbedingungen folgendermaßen beschreiben:

1. $k_z \notin O^*_{k_q} \wedge k_z \notin U^*_{k_q}$
2. $\forall o \in O^*_k : \forall m_o \in \mathcal{M}^a_o \cup \mathcal{M}_o : (=_{SR}(m, m_o) = \texttt{falsch})$
   $\wedge \forall u \in U^*_k : \forall m_u \in \mathcal{M}_u : (=_{SR}(m, m_u) = \texttt{falsch})$
3. $\forall z \in \mathcal{Z}_m : l(z) \neq \texttt{super}$
4. $k_z \notin \mathcal{K}^I_S$
5. $m \notin \mathcal{M}^s_{k_q} \rightarrow (\exists p \in \mathcal{P}_m : typ(p) = k_z \vee \exists a \in \mathcal{A}_{k_q} : typ(a) = k_z)$

Unsere Modellrestrukturierungen läuft in drei Schritten ab:

1. Wir fügen zuerst neue Parameter zur Methode m hinzu, damit wir auf das alte k_q-Objekt zugreifen können und passen die Zugriffe innerhalb von m entsprechend an (T1).

2. Anschließend ersetzen wir die Zugriffskette, die Präfix des Methodenaufrufs von m ist, durch eine Zugriffskette, die das korrekte Objekt der Zielklasse referenziert (T2). Wir können eine solche Zugriffskette angeben, da wir die potentiellen Zielklassen für nichtstatische Methoden wie in Vorbedingung 5 beschrieben eingeschränkt haben.

3. Zuletzt kopieren wir m in die Klasse k_z und löschen m aus der Menge der Methoden von k_q (T3).

T1
Wir müssen drei verschiedene Fälle beachten, um entscheiden zu können, ob und wieviele Parameter in T1 zur Signatur der Methode m hinzugefügt werden müssen:

1. m ruft Methoden von k_q auf und greift auf Attribute von k_q schreibend zu.

$$(\mathcal{Z}^M_m \cap \mathcal{M}^g_{k_q} \neq \emptyset) \wedge (\exists z, \exists a : inMethode(z) = m \wedge a \in \mathcal{A}^g_{k_q} \wedge l(z) = a \wedge z \in \mathcal{Z}_{schreib})$$

2. m greift nur auf Attribute von k_q zu, und zwar ausschließlich lesend.

$$(\mathcal{Z}^M_m \cap \mathcal{M}^g_{k_q} = \emptyset) \wedge (\nexists z, \nexists a : inMethode(z) = m \wedge a \in \mathcal{A}^g_{k_q} \wedge l(z) = a \wedge z \in \mathcal{Z}_{schreib})$$

3. m greift auf kein nichtstatisches Attribut und auf keine nichtstatische Methode von k_q zu.

$$(\mathcal{Z}_m^A \cap \mathcal{A}_{k_q} = \emptyset) \wedge (\mathcal{Z}_m^M \cap \mathcal{M}_{k_q} = \emptyset)$$

Abhängig davon, welche der obigen Voraussetzung erfüllt ist, wird eine der nachfolgenden Strategien angewandt, um das alte k_q-Objekt verfügbar zu machen.

1. Wir übergeben eine Referenz auf das vollständige alte Objekt als zusätzliches Argument beim Aufruf der Methode m (T1-1).

2. Wir übergeben Referenzen auf alle von m gelesenen Attribute des k_q-Objekts als zusätzliche Argumente (T1-2). Diese Variante ist wünschenswerter als die erste Variante, da wir so die Abhängigkeiten zwischen den beiden Klassen abschwächen können. Statt der Rückreferenz und der damit verbundenen zyklischen Abhängigkeit zwischen Quell- und Zielklasse entstehen nur Abhängigkeiten zu den tatsächlich gelesenen Attributen.

3. Wir übergeben keine Referenz auf das alte k_q-Objekt, da dies nicht erforderlich ist (T1-3).

T1-1
Dieser Schritt der Modellrestrukturierung sieht im Detail wie folgt aus:

1. Zunächst erzeugen wir einen neuen Parameter q vom Typ k_q.

$$erzeugeP(q, k_q)$$
$$\mathcal{P}_m := \mathcal{P}_m \cup \{q\}$$

2. Als nächstes ersetzen wir implizite und explizite Zugriffe auf `this` durch Zugriffe auf q.

$$\forall zk \in \mathcal{ZK}_m, \forall z \in \{z_2 \mid z_2 \in zk \wedge l(z_2) \ = \ \texttt{this}\} :$$
$$ersetze(z, z_3)\ mit\ l(z_3) = q$$

3. Dann fügen wir für alle Methodenaufrufe von m die Zugriffskette bis zum Zugriff auf m als zusätzliches Argument hinzu.

$$\forall z \in \{z_2 | z_2 \in \mathcal{Z} \wedge l(z_2) \ = m\} :$$
$$zk_{neu} := cl(praefix(z))$$
$$setzeArg(z, q, zk_{neu})$$

4. Falls sich der Zugriff z innerhalb der Klasse k_q befindet, fügen wir einen expliziten `this`-Zugriff an den Beginn der geklonten Zugriffskette.

$$inKlasse(z) \ = k_q \rightarrow haengeVor(\texttt{this}, zk_{neu})$$

T1-2

Falls innerhalb der Methode nur lesende Attributzugriffe und keine schreibenden Attributzugriffe und Methodenaufrufe stattfinden, so muss nicht das komplette Objekt vom Typ k_q übergeben werden, sondern nur die gelesenen Attribute.

Die Modellrestrukturierung führen wir folgendermaßen aus:

1. Als erstes fügen wir für jedes gelesene Attribut einen neuen Parameter zur Methode m hinzu.

$$\begin{aligned}&\forall a \in \mathcal{Z}_m^a : \\ &p_a := erzeugeParameter(p_a, typ(a)) \\ &\mathcal{P}_m := (\mathcal{P}_m \cup \{p_a\})\end{aligned}$$

2. Dann ersetzen wir alle Lesezugriffe auf Attribute von k_q innerhalb von m durch Zugriffe auf die neu erzeugten Parameter.

$$\begin{aligned}&\forall z \in \{zu | inMethode(zu) = m \wedge l(zu) \in \{A_{k_q} \cup A_{k_q}^S\}\} \ : \\ &hatPraefix(zkette(z), [\texttt{this}]) \rightarrow loesche([\texttt{this}], zkette(z)]) \\ &ersetze([z_a], [z_{q_a}], zkette(z)) \\ &mit\ l(z_a) = a\ und\ l(z_{q_a}) = q_a\end{aligned}$$

3. Anschließend passen wir die Methodenaufrufe an, indem wir jeden Aufruf mit zusätzlichen Argumenten versehen. Für jedes innerhalb von m gelesene Attribut muss ein neues Argument hinzugefügt werden.

$$\begin{aligned}&\forall z \in \mathcal{MZ}_m : \forall a \in \{at | at \in Z_m^a \wedge at \in (\mathcal{A}_{k_q} \cup \mathcal{A}_{k_q}^S)\} : \\ &zkette_{neu} := cl(praefix(z)) \\ &zkette_{neu} := zkette_{neu} \cup [z_a]\ mit\ l(z_a) = a \\ &setzeArgument(m, p_a, zkette_{neu})\end{aligned}$$

T1-3

Falls die Methode m auf keine nichtstatischen Attribute und Methoden von k_q zugreift, benötigt m in der Zielklasse k_z keine Referenz auf ein Objekt vom Typ k_q. Alle verwendeten Attribute und Methoden können über einen statischen Zugriff auf die Klasse k_q erreicht werden.

Sind die Vorbedingungen erfüllt, führen wir die Modellrestrukturierung wie folgt aus.

1. Im ersten Schritt löschen wir alle Zugriffe, die vor dem Methodenaufruf an die (pseudo)statische Methode m stehen.

$$\forall z \in \mathcal{MZ}_m : \forall z_1 \in praefix(z) : loesche(z_1)$$

2. Im zweiten Schritt fügen wir vor jedem Methodenaufruf an m einen Typzugriff auf den Typ k_z ein.

$$\forall z \in \mathcal{MZ}_m : haengeVor(erzeugeZ(k_z), z)$$

T2

In Transformationsschritt 2 benötigen wir eine neue Zugriffskette, um ein Objekt zu referenzieren, auf dem die verschobene Methode aufgerufen werden kann. Diese Zugriffskette können wir je nach gewählter Zielklasse festlegen:

1. Bei der ersten Variante (T2-P) verschieben wir die Methode m in eine ihrer Parameterklassen k_z. An den Aufrufstellen von m existiert als Argument für den Parameter vom Typ k_z eine Zugriffskette p. Das über diese Zugriffskette verfügbare Objekt können wir verwenden, um die Methode m in der Zielklasse aufzurufen.

2. In der zweiten Variante (T2-A) verwenden wir als Zielklasse den Typ eines Attributs von k_q. An den Aufrufstellen von m kann so über das k_q-Objekt auch ein k_z-Objekt angesprochen werden, an das der Methodenaufruf nach der Verschiebung gesendet wird. Wir müssen in diesem Fall allerdings voraussetzen, dass alle Attribute bereits initialisiert sind.

T2-P

Formal beschreiben wir die Anpassung der Aufrufstellen in **T2-P** wie folgt:
Sei $p_i \in \mathcal{P}_m$ der Parameter, in dessen zugehörige Klasse m verschoben werden soll.

1. Zunächst ersetzen wir die Zugriffskette vor dem Methodenaufruf durch die Zugriffskette, die durch das jeweilige Argument für den Parameter p_i festgelegt ist.

$$\forall z \in \mathcal{ZM}_m : ersetze(praefix(z), argument(z, p_i), zkette(z))$$

2. Anschließend ersetzen wir die Zugriffe auf den Parameter p_i innerhalb von m durch einen Zugriff auf `this`.

$$\forall z_p \in \{z | l(z) = p_i \wedge methode(z) = m\} : ersetzeDurch(z_p, \texttt{this})$$

3. Im letzten Schritt löschen wir den nicht mehr benötigten Parameter p_i aus der Signatur der Methode. Damit werden automatisch alle Argumente an den Aufrufstellen gelöscht.

$$\mathcal{P}_m := \mathcal{P}_m \setminus \{p_i\}$$

T2-A
Formal beschreiben wir diese Variante folgendermaßen:

1. Wir verlängern alle Zugriffsketten vor einem Methodenzugriff auf m um einen Attributzugriff auf a. a ist das von uns ausgewählte Attribut von k_q vom Typ k_z.

$$\forall z \in \mathcal{MZ}_m : haengeVor(z_a, z)$$

2. Die Zugriffe auf das Attribut a innerhalb von m ersetzen wir durch this-Zugriffe. Abhängig von der für T1 gewählten Variante ersetzen wir also entweder Zugriffsketten der Form $q.a$ (T1-1) oder q_a (T1-2).

$$\begin{aligned}&\forall z \in \{z_m | methode(z_m) = m\}\\&(T1-1) : ersetze([z_q, z_a], [\texttt{this}])\\&(T1-2) : ersetze([z_{p_a}], [\texttt{this}])\end{aligned}$$

T3
Abschließend entfernen wir die Methode m aus der Menge der Methoden von k_q und fügen m zur Menge der Methoden von k_z hinzu.

$$\mathcal{M}_{k_q} := \mathcal{M}_{k_q} \setminus \{m\} \wedge \mathcal{M}_{k_z} := \mathcal{M}_{k_z} \cup \{m\}$$

5.2.5.5. Verschiebe Attribut in Oberklasse

Parameter:

1. das zu verschiebende Attribut a
2. die Quellklasse k_q

3. die Zielklasse k_z

Wir können Attribute nicht beliebig zwischen einer Quell- und Zielklasse verschieben. Nur wenn zwischen den Objekten der Quell- und Zielklasse zur Laufzeit eine 1-zu-1-Beziehung besteht, verändert sich das beobachtbare Verhalten nach einer Verschiebung nicht. Verschieben wir ein Attribut in eine Ober- oder Unterklasse der Quellklasse, besteht dieses Problem nicht.

Die Modellrestrukturierung *Verschiebe Attribut in Oberklasse* verschiebt ein Attribut $a \in \mathcal{A}_{k_q}$ in eine direkte Oberklasse k_z von k_q. Ein Attribut kann nicht in eine Oberklasse verschoben werden, die mehrere Stufen in der Hierarchie über k_q steht. Eine solche Verschiebung können wir allerdings erreichen, indem wir die Modellrestrukturierung *Verschiebe Attribut in Oberklasse* mehrfach ausführen.

Folgende Vorbedingungen müssen erfüllt sein:

1. k_z ist eine direkte Oberklasse von k_q:

$$k_z \in O_{k_q}$$

2. Die Zielklasse ist keine Schnittstellenklasse

$$k_z \notin K_S^I$$

Die Modellrestrukturierung führen wir folgendermaßen aus:

1. Entferne a aus der Menge der in k_q vereinbarten Attribute $\mathcal{A}_{k_q}$:

$$\mathcal{A}_{k_q} := \mathcal{A}_{k_q} \setminus \{a\}$$

2. Füge a zur Menge der in k_z vereinbarten Attribute $\mathcal{A}_{k_z}$ hinzu:

$$\mathcal{A}_{k_z} := \mathcal{A}_{k_z} \cup \{a\}$$

Eine Verschiebung eines Attributes nach oben in der Hierarchie schränkt den Zugriff auf das Attribut nicht ein, sondern erweitert diesen. Somit verändert diese Modellrestrukturierung das beobachtbare Verhalten nicht.

5.2.5.5.1. Überdeckung anderer Attribute Ein verschobenes Attribut kann an seiner neuen Position ein Attribut gleichen Typs und gleichen Namens in einer der Unterklassen überdecken. In diesem Fall benötigen wir das Attribut in der Unterklasse nicht mehr, an seiner Stelle kann das in die Oberklasse verschobene Attribut verwendet werden. Bei der Initialisierung kann es keine Konflikte geben, da die Attribute an getrennten Stellen initialisiert werden. Werden die Attribute in Initialisierern gesetzt, so verschieben wir die dort vorhandenen Zugriffe in entsprechende Konstruktoren.

5.2.5.6. Verschiebe Attribut in Unterklasse

Parameter:

1. das zu verschiebende Attribut a
2. die Quellklasse k_q
3. die Zielklasse k_z

Die Modellrestrukturierung verschiebt ein Attribut a von einer Quellklasse k_q in eine Zielklasse k_z, die eine direkte Unterklasse von k_q ist. Diese Modellrestrukturierung ist nicht möglich, falls eine weitere Unterklasse z_2 von k_q das Attribut a benutzt, sich aber nicht im selben Vererbungsbaum wie k_z befindet. In diesem Fall müssten wir das Attribut a in beide Vererbungsbäume kopieren. Die damit verbundene Redundanz ist aber ebenfalls eine Entwurfsschwäche, weshalb wir die Modellrestrukturierung in diesem Fall nicht ausführen.

Folgende Vorbedingungen müssen also erfüllt sein:

1. k_z ist eine direkte Unterklasse von k_q:

$$k_z \in U_{k_q}$$

2. Keine Methode aus k_q greift auf a zu:

$$\forall m \in \mathcal{M}_{k_q} : a \notin \mathcal{Z}_m^A$$

3. Alle Zugriffe auf a innerhalb der Hierarchie von k_q erfolgen aus dem Teilbaum der Hierarchie von Q, dessen Wurzel k_z ist:

$$m \in (\mathcal{C}_a \cap (\bigcup_{u' \in \mathcal{U}_{k_q}^*} \mathcal{M}_{u'})) \Rightarrow m \in (\mathcal{M}_{k_z} \cup (\bigcup_{u'' \in \mathcal{U}_{k_z}^*} \mathcal{M}_{u''}))$$

4. Wird außerhalb der Hierarchie auf a zugegriffen, so darf dieser Zugriff nur über ein Objekt aus dem k_z-Teilbaum erfolgen:

$$z \in \mathcal{AZ}_a : inKlasse(z) \notin \mathcal{U}_{k_q}^* \Rightarrow typUeber(z) \in \{k_z \cup \mathcal{U}_{k_z}^*\}$$

Sind die Vorbedingungen erfüllt, führen wir die Modellrestrukturierung folgendermaßen aus:

1. Entferne a aus der Menge der in k_q deklarierten Attribute $\mathcal{A}_{k_q}$:

$$\mathcal{A}_{k_q} := \mathcal{A}_{k_q} \setminus \{a\}$$

2. Füge a der Menge der in k_z deklarierten Attribute $\mathcal{A}_{k_z}$ hinzu:

$$\mathcal{A}_{k_z} := \mathcal{A}_{k_z} \cup \{a\}$$

5.2.5.7. Verschiebe Methode in Oberklasse

Parameter:

1. die zu verschiebende Methode m
2. die Quellklasse k_q
3. die Zielklasse k_z

Analog zu Attributen verschieben wir auch nichtstatische Methoden innerhalb der Vererbungshierarchie. Statische Methoden sind nicht an Vererbungshierarchien gebunden, wir können sie bereits mit der Modellrestrukturierung *Verschiebe Methode* verschieben.

Weiterhin verschieben wir keine Methoden, die eine Schnittstelle implementieren oder eine Methode aus einer Oberklasse überschreiben. In diesen Fällen könnte für manche Klassen nach der Verschiebung eine andere Implementierung als vor der Verschiebung gültig sein. Somit würde sich das beobachtbare Verhalten des Systems ändern.

Um eine nichtstatische Methode $m \in \mathcal{M}_{k_q}$ aus einer Klasse k_q in eine direkte Oberklasse k_z von k_q verschieben zu können, müssen folgende Vorbedingungen erfüllt sein:

1. k_z ist eine direkte Oberklasse von k_q:
$$k_z \in O_{k_q}$$
2. k_z ist keine Schnittstellenklasse
$$k_z \notin \mathcal{K}_S^I$$
3. Die Methode m implementiert keine abstrakte Methode einer Schnittstelle oder einer abstrakten Klasse und überschreibt keine Implementierung einer Oberklasse:
$$\forall o \in \mathcal{O}_{k_q}^*, \forall m_O \in (\mathcal{M}_o^a \cup \mathcal{M}_o) : (=_{SR} (m, m_O) = \texttt{falsch})$$
4. Die Methode nutzt nur vererbte Methoden und Attribute und nicht in k_q implementierte Methoden oder Attribute:
$$\mathcal{Z}_m \cap (\mathcal{M}_{k_q}^g \cup \mathcal{A}_{k_q}^g) = \varnothing$$

Sind alle Vorbedingungen erfüllt, so können wir die Modellrestrukturierung wie folgt durchführen:

1. Entferne m aus der Menge der in k_q implementierten Methoden:
$$\mathcal{M}_{k_q} := \mathcal{M}_{k_q} \setminus \{m\}$$
2. Füge m zur Menge der von k_z implementierten Methoden hinzu:
$$\mathcal{M}_{k_z} := \mathcal{M}_{k_z} \cup \{m\}$$

Analog zur Attributverschiebung ist die Methode m nach wie vor über die bisher benutzten Objekte erreichbar. Durch die Verschiebung zur Oberklasse wurde die Menge an Objekten erweitert, über die diese Methode aufrufbar ist.

5.2.5.8. Verschiebe Methode in Unterklasse

Parameter:

1. die zu verschiebende Methode m
2. die Quellklasse k_q
3. die Zielklasse k_z

Wollen wir eine Methode $m \in \mathcal{M}_{k_q}$ einer Klasse k_q in eine direkte Unterklasse $k_z \in \mathcal{U}_{k_q}$ verschieben, so müssen folgende Vorbedingungen erfüllt sein:

1. Die Methode implementiert keine abstrakte Methode einer Schnittstelle oder einer abstrakten Klasse und überschreibt keine Implementierung einer Oberklasse:
$$\forall o \in \mathcal{O}^*_{k_q} : \forall m_O \in (\mathcal{M}^a_o \cup \mathcal{M}_o) : (=_{SR} (m, m_O) = \texttt{falsch})$$
2. Die Methode wird in keiner Unterklasse von k_q überschrieben:
$$\forall U \in \mathcal{U}^*_{k_q}, \forall m_U \in \mathcal{M}_U : (=_{SR} (m, m_U) = \texttt{falsch})$$
3. Keine Methode aus k_q greift auf m zu:
$$\forall m' \in \mathcal{M}_{k_q} : m \notin \mathcal{Z}_{m'}$$
4. Alle Zugriffe auf m innerhalb der Hierarchie von k_q erfolgen aus dem Teilbaum der Hierarchie von k_q, dessen Wurzel k_z ist:
$$m' \in (\mathcal{C}_m \cap (\bigcup_{u' \in \mathcal{U}^*_{k_q}} \mathcal{M}_{U'})) \Rightarrow m' \in (\mathcal{M}_{k_z} \cup \bigcup_{u'' \in \mathcal{U}^*_{k_z}} \mathcal{M}_{u''})$$
5. Wird außerhalb der Hierarchie auf m zugegriffen, so darf dieser Zugriff nur über ein Objekt aus dem k_z-Teilbaum erfolgen.
$$z \in \mathcal{MZ}_m : inKlasse(z) \notin \mathcal{U}^*_{k_q} \Rightarrow typUeber(z) \in \{U \cup \mathcal{U}^*_{k_z}\}$$

Sind die Vorbedingungen für m, k_q und k_z erfüllt, können wir die Verschiebung in folgenden Schritten ausführen:

1. Entferne m aus der Menge der in k_q implementierten Methoden $\mathcal{M}_{k_q}$:
$$\mathcal{M}_{k_q} := \mathcal{M}_{k_q} \setminus \{m\}$$
2. Füge m der Menge der implementierten Methoden $\mathcal{M}_{k_z}$ von k_z hinzu:
$$\mathcal{M}_{k_z} := \mathcal{M}_{k_z} \cup \{m\}$$

Das beobachtbare Verhalten bleibt gleich, da eine Methode nur in eine Unterklasse verschoben wird, wenn sie lediglich über Objekte dieses Teilbaums des Vererbungsbaums angesprochen wird.

5.2.5.9. Extrahiere Klasse

Parameter:

1. die Klasse k, die aufgeteilt werden soll
2. die Menge der Attribute der neuen Klasse
3. die Menge der Methoden der neuen Klasse

Die Modellrestrukturierung *Extrahiere Klasse* entfernt Attribute und Methoden aus einer Klasse, und fasst diese anschließend zu einer neuen Klasse zusammen. Den Namen der neuen Klassen wählen wir zufällig.

Sei V_2 die Menge der Attribute und Methoden, die von der Quellklasse k abgespalten werden sollen und V_1 die Menge der Attribute und Methoden, die in der Quellklasse verbleiben. Folgende Methoden einer Klasse müssen in V_1 enthalten sein:

- Methoden, die in Schnittstellen deklarierte oder abstrakte Methoden implementieren oder eine Methode aus einer Oberklasse überschreiben.

$$\{m | m \in \mathcal{M}_k \wedge (\exists m_2, k_o : k_o \in \mathcal{O}_k^* \wedge m_2 \in (\mathcal{M}_{k_o} \cup \mathcal{M}_{k_o}^a) \ \wedge \ (=_{SR}(m, m_2) = \texttt{wahr}))\}$$

- Methoden, die von einer Methode einer Unterklasse von K überschrieben werden.

$$\{m | m \in \mathcal{M}_k \wedge (\exists m_2, k_u : k_u \in \mathcal{U}_k^* \wedge m_2 \in \mathcal{M}_{k_u} \wedge \ =_{SR}(m, m_2) = \texttt{wahr})\}$$

Die Modellrestrukturierung können wir wie folgt durchführen: Gegeben ist die Quellklasse $k \in \mathcal{K}_S$ und eine Aufteilung der Attribute und Methoden von k in zwei disjunkte Teilmengen V_1 und V_2:

$$V_1 \cap V_2 = \varnothing \wedge V_1 \cup V_2 = \mathcal{M}_k \cup \mathcal{A}_k$$

1. Zunächst erstellen wir eine neue Klasse k_{spalt} und fügen zu dieser Klasse die Elemente aus V_2 hinzu. Unterklassen von k_{spalt} gibt es nicht ($\mathcal{U}_{k_{spalt}} = \mathcal{U}_{k_{spalt}}^* = \varnothing$):

$$\begin{aligned} \mathcal{K}_S &:= \mathcal{K}_S \cup \{k_{spalt}\} \\ \mathcal{M}_{k_{spalt}} &:= \mathcal{M}_k \cap V_2 \\ \mathcal{A}_{k_{spalt}} &:= \mathcal{A}_k \cap V_2 \end{aligned}$$

2. Dann entfernen wir die Menge V_2 aus K:

$$\mathcal{M}_k := \mathcal{M}_k \setminus (\mathcal{M}_k \cap V_2)$$
$$\mathcal{A}_k := \mathcal{A}_k \setminus (\mathcal{A}_k \cap V_2)$$

3. Zur Quellklasse k fügen wir ein Attribut ref mit dem Typ k_{spalt} hinzu. Dadurch können alle Methoden aus k und alle Klienten von k auf die abgespaltenen Methoden und Attribute zugreifen:

$$\mathcal{A}_k := \mathcal{A}_k \cup \{ref\}$$
$$typ(ref) := k_{spalt}$$

4. Alle externen Zugriffe z^e auf Methoden von k_{spalt} verwenden bisher ein Objekt vom Typ k, d.h. $typUeber(z^e) = k$. Über dieses Objekt ist auch die Referenz ref erreichbar. Somit muss zwischen allen Zugriffen z^e und dem jeweiligen Zugriff auf das k-Objekt jeweils ein neuer Zugriff z^{neu} auf dieses Attribut ref eingefügt werden:

$$\forall z^e \in \bigcup_{m \in \mathcal{M}_{k_{spalt}}} \{z \in \mathcal{MZ}_m : inKlasse(z) \neq k_{spalt}\} :$$
$$haengeVor(z^{neu}, z^e) \; mit \; l(z^{neu}) = ref$$

   Ebenso für Attribute von k_{spalt}:

$$\forall z^e \in \bigcup_{a \in \mathcal{A}_{k_{spalt}}} \{z \in \mathcal{AZ}_a : inKlasse(z) \neq k_{spalt}\} :$$
$$haengeVor(z^{neu}, z^e) \; mit \; l(z^{neu}) = ref$$

5. Falls Elemente aus V_2 auf Elemente aus V_1 zugreifen, müssen wir einen Rückverweis von k_{spalt} auf k hinzufügen. In diesem Fall legen wir ein Attribut $rref$ der Klasse k_{spalt} mit dem Typ k an.

$$\mathcal{A}_{K_{spalt}} := \mathcal{A}_{K_{spalt}} \cup \{rref\}$$
$$typ(rref) := k$$

Der Rückverweis sorgt dafür, dass Methoden aus k_{spalt} Zugriff auf Elemente aus k haben. Deshalb muss vor jedem Zugriff z^i in k_{spalt} auf Methoden von k jeweils ein neuer Zugriff z^{neu} auf das Attribut $rref$ eingefügt werden:

$$\forall z^i \in \bigcup_{m \in \mathcal{M}_k} \{z \in \mathcal{MZ}_m : inKlasse(z) = k_{spalt}\} :$$
$$haengeVor(z^{neu}, z^i) \; mit \; l(z^{neu}) = rref$$

Das gleiche gilt für Attribute von k, die von Methoden aus k_{spalt} verwendet werden:

$$\forall z^i \in \bigcup_{a \in \mathcal{A}_k} \{z \in \mathcal{AZ}_a : inKlasse(z) = k_{spalt}\} :$$
$$haengeVor(z^{neu}, z^i) \; mit \; l(z^{neu}) = rref$$

Die Konstruktoren und Destruktoren der Quellklasse behandeln wir bei der Aufteilung folgendermaßen. Die beiden neuen Klassen haben eine 1 zu 1 Beziehung und treten zur Laufzeit auch immer paarweise auf. An den Stellen, an denen bisher das ursprüngliche Objekt erzeugt bzw. gelöscht wurde, fügen wir Aufrufe des Konstruktors bzw. des Destruktors der neuen Klasse hinzu.

Eine aufgespaltene Klasse teilen wir nicht erneut auf, da sonst in ungünstigen Fällen sehr lange Zugriffsketten über die einzelnen Referenzen der Teilklassen entstehen könnten. Um eine Klasse trotzdem in mehr als zwei Klassen aufteilen zu können, zerlegen wir als Variante von *Extrahiere Klasse* eine Klasse auch in drei oder mehr Klassen. Diese Modellrestrukturierung kann analog zu *Extrahiere Klasse* ausgeführt werden, mit dem einzigen Unterschied, dass die Menge der Attribute und Methoden in mehr als zwei Mengen aufgeteilt werden muss.

5.2.5.10. Integriere Klasse

Parameter: die zu integrierenden Klassen k und k_{spalt}.

Integriere Klasse ist die zu *Extrahiere Klasse* zugehörige inverse Modellrestrukturierung, die eine Klasse wieder in ihre ursprüngliche Form überführen kann. Sie kann nur auf Klassen angewendet werden, die zuvor mit *Extrahiere Klasse* aufgespalten wurden, da nur in diesem Fall sichergestellt ist, dass die Objekte zur Laufzeit in einer 1-zu-1-Beziehung stehen.

Um die Klassen zusammenlegen zu können, müssen wir bei den Zugriffen der Methoden von k_{spalt} auf Methoden und Attribute aus k den jeweilige Zugriff auf den Rückverweis $rref$ aus der entsprechenden Zugriffskette entfernen. Alle Methoden und Attribute

von k_{spalt} müssen wieder nach k verschoben werden (außer dem Rückverweis, falls dieser existiert). Dann können wir k_{spalt} aus dem Modell entfernen:

$$\begin{aligned}
&\forall z \in \{z^* | z^* \in \mathcal{Z}_S^* \wedge l(z^*) = rref\} : \\
&loesche(z) \\
&\mathcal{M}_k := \mathcal{M}_k \cup \mathcal{M}_{k_{spalt}} \\
&\mathcal{A}_k := \mathcal{A}_k \cup (\mathcal{A}_{k_{spalt}} \setminus \{rref\}) \\
&\mathcal{K}_S := \mathcal{K}_S \setminus \{k_{spalt}\}
\end{aligned}$$

Den Verweis ref auf k_{spalt} können wir aus k löschen, ebenso müssen wir alle Zugriffe auf diesen Verweis entfernen. Da sich alle Methoden und Attribute von k_{spalt}, auf die über diesen Verweis zugegriffen wurde, wieder in der Quellklasse k befinden und ein k-Objekt an diesen Stellen verfügbar ist, ist somit wieder ein gültiger Zustand des Systems hergestellt.

$$\begin{aligned}
&\forall z \in \{z^* | z^* \in \mathcal{Z}_S^* : l(z^*) = ref\} : \\
&loesche(z) \\
&\mathcal{A}_k := \mathcal{A}_k \setminus \{ref\}
\end{aligned}$$

5.2.6. Interaktionen von Modellrestrukturierungen

In den vorherigen Abschnitten haben wir die Modellrestrukturierungen isoliert voneinander vorgestellt. Betrachten wir Interaktionen zwischen Modellrestrukturierungen, so können wir manche der Modellrestrukturierungen wie folgt abändern.

5.2.6.1. Methodenverschiebungen

Die verschiedenen Arten der Methodenverschiebung können im Zusammenhang mit dem Extrahieren und Integrieren von Klassen Zustände im Systemmodell erzeugen, die wir gesondert berücksichtigen, da nach unserer in den vorherigen Abschnitten beschriebenen Vorgehensweise unnötige Rückreferenzen im Modell erzeugt werden.

Für die weiteren Fälle sei $k \in \mathcal{K}_S$ eine Klasse, k_{spalt} sei die durch *Extrahiere Klasse* erzeugte, von k abgespaltene Klasse. Die Referenz von k auf k_{spalt} sei ref und die Rückreferenz $rref$.

5.2.6.1.1. Fall 1 Verschieben wir eine Methode $m \in \mathcal{M}_{k_{spalt}}$ in eine Klasse $k', k' \neq k$, so können wir einen Parameter p vom Typ k_{spalt} erzeugen. Werden die beiden Klassen wieder zusammengelegt, so müssen wir den Parameter in den Typ k umwandeln, da die über diesen Parameter zugegriffenen Elemente wieder in der Klasse k vorhanden sind. An den Aufrufstellen übergeben wir automatisch das richtige k-Objekt, indem wir den Zugriff auf die Rückreferenz löschen.

Die Anpassung muss also beim Zusammenlegungsoperator erfolgen:

$$typ(p) = k_{spalt} \Rightarrow typ(p) := k$$

5.2.6.1.2. Fall 2 Verschieben wir eine Methode $m \in k$ nach k_{spalt}, so entsteht in der Methode ein Parameter p vom Typ k (falls m auf Elemente von k zugreift und die entsprechende Art der Methodenverschiebung erfolgt). Dies ist unnötig, da die referenzierten Elemente auch über die Rückreferenz $rref$ erreicht werden könnten. Daher wird in diesem Fall der Parameter gelöscht und jeder Zugriff auf den Parameter durch einen neuen Zugriff z_{rref} auf $rref$ ersetzt:

$$\begin{aligned}
&\mathcal{P}_m := \mathcal{P}_m \setminus \{p\} \\
&\forall z \in \{z' | z' \in \mathcal{Z}_m \wedge l(z') = p\} \\
&ersetzeDurch(z, z_{rref}) \; mit \; l(z_{rref}) = rref
\end{aligned}$$

5.2.6.1.3. Fall 3 Wird eine Methode $m \in k_{spalt}$ nach k verschoben, so entsteht in der Methode ein Parameter p vom Typ k_{spalt}. Dies ist analog zu Fall 2 unnötig, hier können die über den Parameter referenzierten Elemente über die Referenz ref erreicht werden. Analog zu Fall 2 muss nun der Parameter gelöscht und jeder Zugriff auf den Parameter durch einen neuen Zugriff z_{ref} auf ref ersetzt werden:

$$\begin{aligned}
&\mathcal{P}_m := \mathcal{P}_m \setminus \{p\} \\
&\forall z \in \{z' | z' \in \mathcal{Z}_m \wedge (z') = p\} : \\
&ersetzeDurch(z, z_{ref}) \; mit \; (z_{ref}) = ref
\end{aligned}$$

Zusätzlich ergibt sich bei Fall 3 eine ungünstige Zugriffskette bei Zugriffen auf die Methode m. Dabei wird nacheinander auf die Referenz und auf die Rückreferenz zugegriffen, was keinen Sinn ergibt, da hier wieder das ursprüngliche Objekt referenziert wird.

Die Zugriffskette eines Zugriffs auf die Methode vor der Verschiebung ist hier beispielhaft angegeben (`o` sei ein Objekt vom Typ k):

```
o.ref.m()
```

Und nach der Verschiebung (die Parameteranpassung ist hier schon erfolgt):

```
o.ref.rref.m()
```

Daher müssen bei Zugriffen auf die Methode außerhalb von k_{spalt} die Zugriffe auf ref und $rref$ gelöscht werden:

$$\begin{aligned}
&\forall z \in \{z' | z' \in \mathcal{MZ}_m \wedge inKlasse(z') \neq k_{spalt}\} : \\
&loesche(zvor(zvor(z))) \\
&loesche(zvor(z))
\end{aligned}$$

Diese Anpassung würde obiges Beispiel wie folgt verändern:

```
o.m()
```

5.2.7. Vereinfachungen und Einschränkungen

Um unser Modell sprachunabhängig und nicht unnötig kompliziert zu gestalten, verzichten wir darauf, Sichtbarkeiten zu modellieren. In vielen objektorientierten Sprachen kann die Sichtbarkeit von Sprachelementen festgelegt werden. Wir gehen vereinfachend davon aus, dass Strukturen überall sichtbar und somit zugreifbar sind. Somit kann eine Modellrestrukturierung nicht immer ohne Zusatzrestrukturierungen auf das ursprüngliche System übertragen werden, da eventuell Elemente nicht sichtbar sind. In einem solchen Fall kann aber mit Hilfe eines Werkzeuges eine automatische Restrukturierung ausgeführt werden, die die Sichtbarkeit des Elements erhöht und eventuell entstehende Namenskollisionen auflöst.

Eine weitere Vereinfachung betrifft die Namen von Strukturelementen. Unser Metamodell legt fest, dass Strukturelemente immer einen global eindeutigen Namen haben müssen. Diese Annahme kann bei der Abbildung von Modellrestrukturierungen auf Restrukturierungen eventuell Umbenennungen erfordern. Diese Umbenennungen können zwar automatisch durchgeführt werden, allerdings sollte ein Softwareingenieur diese Namen kontrollieren und in sprechende Namen ändern.

Unser Metamodell unterstützt keine Reflexion. Wird in einem System Reflexion eingesetzt, so können keine verlässlichen Aussagen über referenzierte Klassen, Methoden und Attribute mehr gemacht werden. Systeme, in denen Reflexion eingesetzt wird, können wir zwar bearbeiten, allerdings können wir für diese Systeme nicht garantieren, dass die Modellrestrukturierungen auf das ursprüngliche System übertragen werden können, ohne das extern beobachtbare Verhalten zu ändern. Wir können nicht einmal garantieren, dass das System noch lauffähig ist.

5.3. Faktenextraktion

Ein unserem Metamodell entsprechendes Modell gewinnen wir ausschließlich anhand des Quelltexts eines existierenden Systems. Den Aufbau eines Modells auf Basis des Quelltexts nennen wir Faktenextraktion [TKS05].

Im Gegensatz zu unserem Metamodell ist die Faktenextraktion sprachabhängig. Im ersten Schritt der Faktenextraktion wird ein abstrakter Syntaxbaum des Quelltexts aufgebaut. Im zweiten Schritt wird dieser Syntaxbaum in ein Modell überführt.

Für den ersten Schritt der Faktenextraktion verwenden wir übliche Techniken aus dem Übersetzerbau. Wir benötigen für jede zu analysierende Programmiersprache einen Zerteiler und Teile der semantischen Analyse, insbesondere Namens- und Typanalyse. Mit der Typanalyse können wir in objektorientierten Systemen den statischen, deklarierten Typ von Variablen und Ausdrücken bestimmen und somit Variablenzugriffe und Methodenaufrufe ermitteln.

Wie ein abstrakter Syntaxbaum in unser Modell überführt werden kann, erklären wir

anhand der Programmiersprache *Java* in folgendem Beispiel.

Beispiel 6: *Abbildung von* Java *auf unser Metamodell*
Die meisten Elemente wie Klassen, Methoden lassen sich direkt auf unser Metamodell übertragen. Es gibt jedoch ein paar Sonderfälle, die wir im Folgenden näher betrachten wollen. Diese Sonderfälle werden von den Modellrestrukturierungen nicht verändert. Durch sie entstehen allerdings durchaus Abhängigkeiten, die bei der Bewertung der Qualität der Struktur berücksichtigt werden müssen.

- Basistypen

  Java unterstützt Basistypen wie `char` oder `int` [AG96]. Um die Struktur eines Systems vollständig modellieren zu können, bilden wir jeden Basistyp auf eine Bibliotheksklasse ab.

- Ausnahmebehandlung

  `throws-Anweisungen` werden in eine Zugriffskette der Methode überführt, die die Ausnahme auslösen kann. Jede Ausnahmeklasse wird in der Zugriffskette als Klassenzugriff modelliert.

  Im Rahmen der Ausnahmebehandlung müssen wir auch auf `catch`-Blöcke gesondert eingehen. Die von einem `catch`-Block behandelten Ausnahmen modellieren wir als lokale Variablen. Die Inhalte der `catch`-Blöcke modellieren wir als gewöhnliche Bestandteile der den `catch`-Block umgebenden Methode.

- Typkonversion

  In *Java* können mit Hilfe des `CAST`-Operators Typkonversionen vorgenommen werden. Wir bilden diese Operatoren auf einfache Klassenzugriffe ab.

- Typabfragen zur Laufzeit

  In *Java* kann zur Laufzeit mit Hilfe von `instanceof` überprüft werden, ob ein Objekt einen bestimmten Typ besitzt. Diese Überprüfung modellieren wir als Klassenzugriff.

- Innere Klassen und Anonyme Klassen

  Die von *Java* unterstützten inneren und anonymen Klassen bilden wir auf Klassen ab. Da beide nur Hilfskonzepte für die sie umgebenden Klassen oder Methoden darstellen, verändern wir sie nicht. Wir markieren sie deshalb als unveränderliche Strukturelemente.

- Templates

  Ab der Version 5 unterstützt *Java* Templates. Für Templates existieren bisher weder Restrukturierungen noch Entwurfsprinzipien, die die damit verbundenen Besonderheiten berücksichtigen. Wir bilden Templates deshalb in unserem Metamodell ebenfalls auf normale Klassen ab, verändern diese Klassen aber nicht und markieren sie deshalb als unveränderlich.

Für andere objektorientierte Sprachen wie z.B. *C++* müssen eigene Faktenextraktoren und Abbildungen auf das Metamodell festgelegt werden. In [STT06] werden drei Faktenextraktoren beschrieben, die den Quelltext eines existierenden Systems auf ein ähnliches Metamodell abbilden.

5.4. Erzeugen einer Graphrepräsentation

Ein Modell als Ausprägung unseres Metamodells lässt sich anschaulich als gerichteter, getypter Multigraph darstellen. Ein solcher Graph $G = (E, K)$ besteht aus einer Menge von Knoten E und Kanten K. Den Startknoten einer Kante referenzieren wir mit K_{Start}, den Endknoten mit K_{Ende}. Da wir eine Graphdarstellung von Modellen im weiteren Verlauf dieser Arbeit benutzen werden, geben wir nun eine Konstruktionsvorschrift an, mit der Modelle in Graphen überführt werden können.

Zunächst erzeugen wir für alle Strukturelemente Knoten und modellieren mit Kanten, dass Strukturelemente andere Strukturelemente enthalten können.

1. Wir erzeugen für alle Teilsysteme aus $\mathcal{T}$ einen Knoten vom Typ *Teilsystem*.
2. Wir erzeugen für alle Mengen $\mathcal{S}_t$ für jede darin enthaltene Klasse einen Knoten vom Typ *Klasse* und fügen zwischen dem Knoten, der das Teilsystem t repräsentiert und jedem Klassenknoten eine Kante vom Typ *enthält* ein.
3. Für alle Mengen $\mathcal{M}_k^q$ führen wir folgendes durch: Wir erzeugen für jede in der Menge enthaltene Methode einen Knoten vom Typ *Methode* und eine Kante vom Typ *enthält* zwischen dem Methodenknoten und dem Klassenknoten, der k repräsentiert.
4. Für alle Mengen $\mathcal{A}_k^q$ führen wir folgendes durch: Wir erzeugen für jedes in der Menge enthaltene Attribut einen Knoten vom Typ *Attribut* und eine Kante vom Typ *enthält* zwischen dem Attributknoten und dem Klassenknoten, der k repräsentiert.
5. Für alle Mengen $\mathcal{P}_m$ führen wir folgendes durch: Wir erzeugen für jeden in der Menge enthaltenen Parameter einen Knoten vom Typ *Parameter* und eine Kante vom Typ *hatParameter* zwischen dem Parameterknoten und dem Methodenknoten, der m repräsentiert.
6. Für alle Mengen $\mathcal{LV}_m$ führen wir folgendes durch: Wir erzeugen für jede in der Menge enthaltene lokale Variable einen Knoten vom Typ *lokale Variable* und eine Kante vom Typ *enthält* zwischen dem neu erzeugten Knoten und dem Methodenknoten, der m repräsentiert.
7. Für alle Mengen $\mathcal{ZK}_m$ führen wir folgendes durch: Wir erzeugen für jede in der Menge enthaltene Zugriffskette einen Knoten vom Typ *Zugriffskette* und eine Kante vom Typ *enthält* zwischen dem Methodenknoten, der m repräsentiert, und dem Zugriffskettenknoten.

8. Für alle Mengen $\mathcal{Z}_{zk}$ führen wir folgendes durch: Wir erzeugen für jeden in der Menge enthaltenen Zugriff einen Knoten vom Typ *Zugriff* und eine Kante vom Typ *enthält* zwischen dem Zugriffskettenknoten, der zk repräsentiert, und dem Zugriffsknoten.

9. Für alle Mengen $\mathcal{ZK}_{oz}$ führen wir folgendes durch: Wir erzeugen für jede in der Menge enthaltene Zugriffskette einen Knoten vom Typ *Zugriffskette* und eine Kante vom Typ *enthält* zwischen dem Knoten, der den Operatorzugriff oz repräsentiert, und dem Zugriffskettenknoten.

10. Für alle Mengen $\mathcal{ZK}_{mz}$ führen wir folgendes durch: Wir erzeugen für jede in der Menge enthaltene Zugriffskette einen Knoten vom Typ *Zugriffskette* und eine Kante vom Typ *enthält* zwischen dem Knoten, der den Methodenzugriff mz repräsentiert, und dem Zugriffskettenknoten.

Als nächstes fügen wir dem Graph zusätzliche Kanten hinzu, die Zugriffe, Vererbungsbeziehungen und Typabhängigkeiten modellieren.

1. Für jeden Zugriff z aus $\mathcal{Z}$ bestimmen wir mit Hilfe von $l(z)$ das Ziel des Zugriffs. Wir bestimmen die beiden Knoten, die den Zugriff und das Ziel des Zugriffs modellieren und fügen zwischen diesen beiden Knoten eine Kante vom Typ *hatZugriffAuf* hinzu. Für jeden Zugriff auf `super` fügen wir eine Kante von dem Knoten, der die Unterklasse repräsentiert, zum Knoten, der die Oberklasse repräsentiert, hinzu. Zugriffe auf `this` modellieren wir aus Gründen der Vereinfachung nicht, da wir schon das über `this` angesprochene Merkmal der Klasse als Ziel eines Zugriffs modellieren.

2. Für jede Variable v (Attribut, lokale Variable oder Methodenparameter) bestimmen wir mit $typ(v)$ die Klasse, die den Typ der Variablen repräsentiert und fügen zwischen den zugehörigen Knoten im Graph eine Kante vom Typ *hatTyp* hinzu.

3. Jede Methode hat einen Rückgabetyp $\mathcal{R}_m$. Dieser Typ wird durch einen Klassenknoten repräsentiert. Zwischen dem Methodenknoten und dem Klassenknoten fügen wir eine Kante vom Typ *hatRückgabetyp* hinzu.

4. Für jede Oberklasse einer Klasse fügen wir zwischen den Klassen eine Kante vom Typ *erbtVon* hinzu.

Mit den aus [Ciu01] bekannten Abstraktions- und Selektionstechniken können abstrakte Graphen erzeugt werden. Knoten können in unserem Fall anhand der *enthält*-Beziehungen verschmolzen werden.

5.5. Beispielmodellierung

Anhand des Teilsystems `XML` unserer Referenzfallstudie aus Abschnitt 3.1 erklären wir nun, wie wir auf Basis unseres Metamodells ein Systemmodell erstellen können. Die

Struktur des Teilsystems ist noch einmal vereinfacht in Abbildung 5.3 in Form eines herkömmlichen UML-Diagramms dargestellt. Neu hinzugekommen ist die Fassadenklasse `XML`, die einen einfacheren Zugriff von außen ermöglicht.

Das Teilsystem `XML` enthält vier Klassen: `XML.XML`, `XML.XMLEinstellungen`, `XML.XMLLeser` und `XML.XMLSchreiber`. Die Namen der Klassen erhalten den Teilsystemnamen als Präfix, damit sie eindeutig sind. Die Klasse `XML.XML` enthält die Methode `XML.XML.leseEinstellungen`, die einen Parameter vom Typ `String` und einen Rückgabewert vom Typ `XML.XMLEinstellungen` besitzt.

Aus der Abbildung lässt sich ablesen, dass durch das System folgende Mengen definiert sind:

- $\mathcal{T} = \{\texttt{XML}\}$
- $\mathcal{S}_{XML} = \{\texttt{XML.XML}, \texttt{XML.XMLLeser}, \texttt{XML.XMLEinstellungen}, \texttt{XML.XMLSchreiber}\}$
- $\mathcal{M}^g_{XML} = \{\texttt{XML.XML.leseEinstellungen}\}$
- $\mathcal{M}^g_{XMLLeser} = \{\texttt{XML.XMLLeser.leseEinstellungen}\}$
- $\mathcal{P}_{XML.XML.leseEinstellungen} = \{(\texttt{datei}, \texttt{String})\}$
- $\mathcal{P}_{XML.XMLLeser.leseEinstellungen} = \{(\texttt{datei}, \texttt{String})\}$
- $\mathcal{R}_{XML.XML.leseEinstellungen} = \{\texttt{XMLEinstellungen}\}$
- $\mathcal{R}_{XML.XMLLeser.leseEinstellungen} = \{\texttt{XMLEinstellungen}\}$

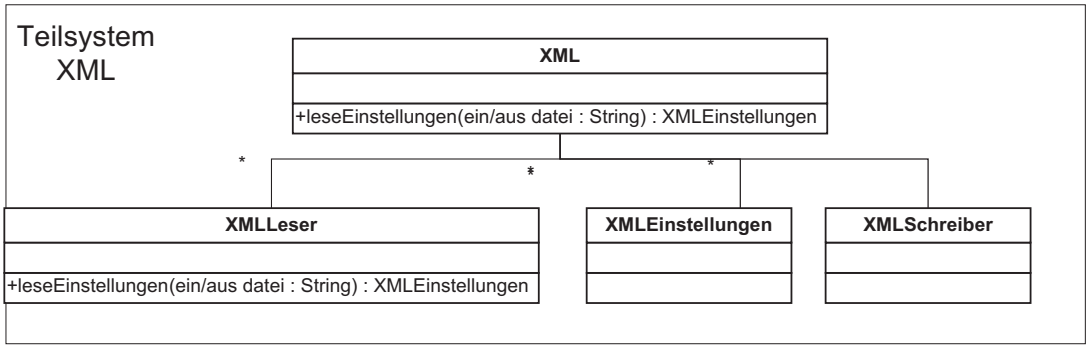


Abbildung 5.3.: Teilsystem XML mit Fassadenklasse (UML)

Abbildung 5.4 zeigt das Systemmodell in der von uns in Abschnitt 5.4 beschriebenen Graphrepräsentation.

Aus dem Beispiel geht bis jetzt noch nicht hervor, wie wir die Kooperation zwischen Objekten modellieren. Deshalb betrachten wir nun den in Abbildung 5.5 dargestellten Quelltextausschnitt, der einen Teil der Implementierung der Klasse `XML` enthält.

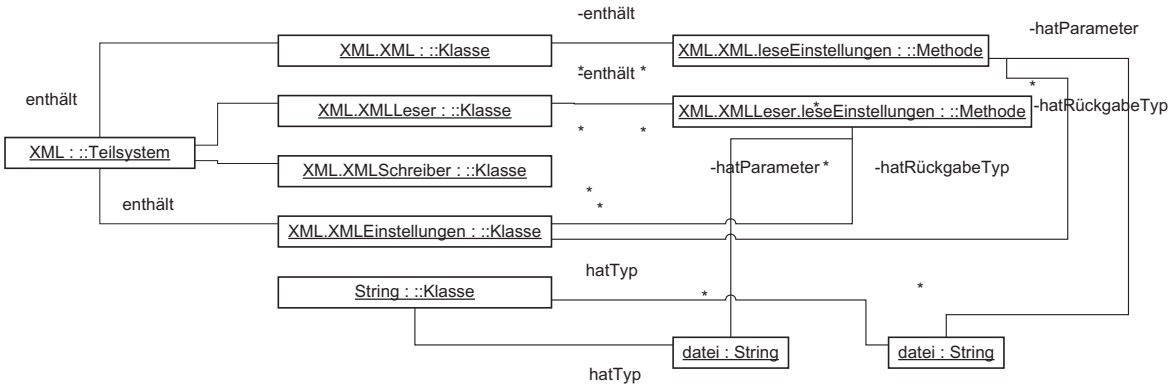


Abbildung 5.4.: Teilsystem XML mit Fassadenklasse (EVOModell)

```
class XML {

    XMLLeser leser          = new XMLLeser();
    XMLSchreiber schreiber  = new XMLSchreiber();

    public XMLEinstellungen leseEinstellungen(String datei) {
        return leser.leseXMLEinstellungen(datei);
    }
}
```

Abbildung 5.5.: Quelltextausschnitt aus der Klasse XML

Zugriffe treten in diesem Beispiel in der Methode `leseEinstellungen` auf. Die Methode hat nur eine Aufgabe: Sie delegiert eingehende Aufrufe an das Objekt `leser` der Klasse `XMLLeser`.

Aus diesem Quelltextausschnitt können wir die bereits vorgestellten Mengen wie folgt erweitern und um neue Mengen ergänzen:

- $\mathcal{M}^g_{XML} = \{\texttt{XML.XML.leseEinstellungen}, \texttt{XML.XML.ini1}, \texttt{XML.XML.ini2}\}$
- $ZK_{XML.XML.leseEinstellungen} = \{zp1\}$
- $Z_{zp1} = \{az1, mz1\}$
- $ZK_{mz1} = \{zp2\}$
- $Z_{zp2} = \{pz1\}$

Die Zugriffsketten der beiden Initialisierer haben wir aus Gründen der Übersichtlichkeit weggelassen. Abbildung 5.6 zeigt unsere Graphrepräsentation, insbesondere die Zugriffe der Methode `leseEinstellungen`.

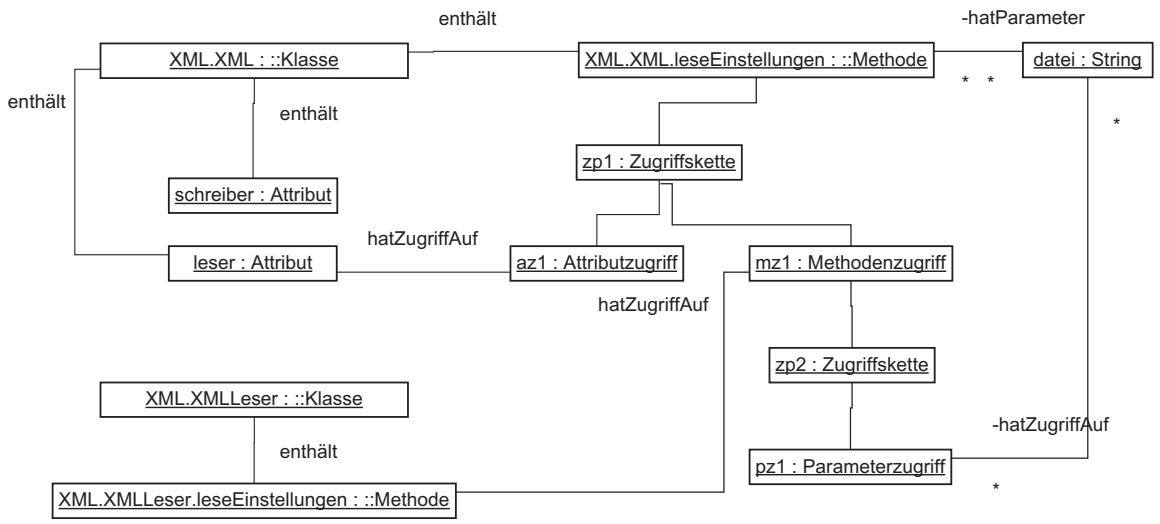


Abbildung 5.6.: Zugriffe als Systemmodell

Die Abbildung zeigt, dass die Methode `leseEinstellungen` eine Zugriffskette besitzt, die aus einem Attributzugriff auf das Attribut `leser` und einem Methodenzugriff auf die Methode `leseEinstellungen` der Klasse `XML.XMLLeser` besteht. Der Zugriff auf den Parameter `datei` der umgebenden Methode wird als separate Zugriffskette modelliert, die aus einem einzigen Parameterzugriff besteht.

5.6. Diskussion

Am Anfang des Kapitels haben wir vier Anforderungen an unser Metamodell gestellt.

Das Metamodell bietet Modellrestrukturierungen an, die auf das ursprüngliche System übertragen werden können und dabei garantieren, dass sich das beobachtbare Verhalten nicht verändert. Auf Basis der Elementaroperationen können weitere Modellrestrukturierungen definiert werden.

Mit Hilfe der vorgegebenen Mengen und der Elementaroperationen zur Abfrage von Informationen bietet unser Metamodell alle Informationen an, die typische Metamodelle zur Bewertung der Qualität der Struktur anbieten.

Aus dem Aufbau des Metamodells geht hervor, dass es zumindest leichtgewichtiger als ein abstrakter Syntaxbaum ist, da keine Kontrollflussanweisungen und Zuweisungen modelliert werden.

Unser Metamodell ist wie gefordert sprachunabhängig, da ausdrucksbasierte, stark typisierte objektorientierte Sprachen auf das Metamodell abgebildet werden können. Wir haben dies exemplarisch für die Programmiersprache *Java* in Abschnitt 5.3 durchgeführt. In [TKS05] wird ein Metamodell beschrieben, das eine Obermenge unseres Metamodells darstellt. Auch dieses Modell ist sprachunabhängig für stark typisierte, ausdrucksbasierte objektorientierte Sprachen geeignet.

Um den Quelltext eines Systems zu modellieren, existieren bereits verschiedene Meta-

modelle. Das UML-Metamodell[1] zur Darstellung von Klassendiagrammen ist der wohl bekannteste Vertreter. Dieses Metamodell ist für uns aber nicht geeignet, da innerhalb von Methodenrümpfen keine Strukturelemente modelliert werden und keine Modellrestrukturierungen unterstützt werden.

Abstrakte Syntaxbäume bieten eine weitere Möglichkeit, um Softwaresysteme als Systemmodell darzustellen. Abstrakte Syntaxbäume haben allerdings zwei große Nachteile: Erstens sind sie sprachabhängig, zweitens enthalten sie zu viele Details, wie Schleifen oder Zuweisungen, die wir nicht benötigen.

Metamodelle zur Bewertung der Systemstruktur und zur Analyse von Teilsystemstrukturen, wie sie Ciupke [Ciu01] und Bauer [Bau05] vorstellen, bieten keine Modellrestrukturierungen an. Sie sind sehr grobgranular und können deshalb nicht um Modellrestrukturierungen für Klassen erweitert werden. Beide Metamodelle unterstützen keine Zugriffsketten, die wir für Modellrestrukturierungen zwingend benötigen.

Genssler [Gen04] stellt ein Metamodell zur Adaption objektorientierter Programme vor. Dieses Metamodell besitzt eine sprachunabhängige Schnittstelle und Modellrestrukturierungen. Allerdings ist es für unsere Zwecke nicht leichtgewichtig genug, da es z.B. Zuweisungen, Reihungen und Ausnahmebehandlungen separat modelliert.

5.7. Klassifikation von Teilstrukturen

Manche Teilstrukturen verletzen bewusst Entwurfsprinzipien. Die von uns verwendeten Metriken identifizieren solche Teilstrukturen fälschlicherweise als Strukturprobleme. Unser Verfahren wird versuchen diese Teilstrukturen zu verändern, um die vermeintlichen Strukturprobleme zu beheben.

Wir müssen diese falschen Veränderungen ausschließen, indem wir die an bewussten Verletzungen von Entwurfsprinzipien beteiligten Elemente vorab erkennen. In einem Klassifikationsschritt teilen wir die Strukturelemente deshalb in zwei Gruppen ein: In eine erste Gruppe, der alle veränderlichen Strukturelemente angehören und in eine zweite Gruppe, in der die unveränderlichen Strukturelemente enthalten sind.

Prinzipiell kann jede beliebige Teilstruktur zu den unveränderlichen Elementen gehören. Nur ein Softwareingenieur, der das zu untersuchende System kennt, könnte somit die Klassifikation perfekt durchführen. Wir konnten allerdings bereits eine Reihe von immer wiederkehrenden Teilstrukturen identifizieren, die Entwurfsprinzipien verletzen. Diese Strukturen - darunter viele Entwurfsmuster - identifizieren wir vorab automatisch und ordnen die beteiligten Strukturelemente den unveränderlichen Elementen zu.

Zusätzlich zu Entwurfsmustern könnte die Organisationsform eines Unternehmens in Konflikt zu den von uns beschriebenen Entwurfsprinzipien stehen. Die Aufteilung von Softwareingenieuren in feste Gruppen, die sich um bestimmte Teilsysteme kümmern, könnte beispielsweise verhindern, dass manche Teilsysteme zusammengelegt werden. In so einem Fall müssen die entsprechenden Teilsysteme manuell als unveränderlich markiert werden.

[1] http://www.uml.org/#UML2.0

Unsere Vorgehensweise zur automatischen Erkennung von Entwurfsmustern erfolgt analog zu dem von Bauer [Bau05] vorgestellten Ansatz. Wir klassifizieren die Strukturelemente eines Systems Bottom-Up, indem wir zuerst die Methoden - die elementaren Strukturelemente - in verschiedene Rollen einteilen. Mit Hilfe der Rollen von Methoden und weiterer struktureller Beziehungen können wir auf die Rolle einer Klasse schließen. Auf Basis der Rollen der Klassen innerhalb eines Teilsystems klassifizieren wir Teilsysteme.

Grundsätzlich nutzen wir folgende Informationsarten aus, um Strukturelemente klassifizieren zu können:

1. Aufbau der Struktur: Wir werten aus, in welchen Beziehungen die einzelnen Strukturelemente zueinander stehen. Diese Bedingungen formulieren wir prädikatenlogisch.

2. Namenskonventionen: Namenskonventionen geben Hinweise auf die Rolle von Strukturelementen. Sie müssen aber mit Strukturinformationen kombiniert werden, um sinnvolle Ergebnisse zu erzielen.

Es ist allgemein bekannt, dass nicht alle Entwurfsmuster auf Basis dieser zwei Informationsarten identifiziert werden können. Um z.B. ein Beobachterentwurfsmuster zuverlässig zu identifizieren, benötigen wir die Aufrufprotokolle von Methoden. Diese Information können wir aber nur mit Hilfe dynamischer Analysen gewinnen, was für große, industrielle Systeme jedoch nicht praktikabel ist.

Für unsere Zwecke reichen die Ergebnisse, die wir mit Hilfe von statischen Analysen gewinnen können, jedoch völlig aus. Falsch positive Entwurfsmuster können wir manuell identifizieren, bevor der evolutionäre Algorithmus startet. Übersehen wir dabei ein falsch positives Muster, so kann unser Verfahren eventuell nicht das beste erreichbare Ergebnis erzielen.

Falsch negative Entwurfsmuster könnten durch unsere Verfahren zerstört werden. Dies ist allerdings unwahrscheinlich, da ein Softwareingenieur die Modellrestrukturierungen manuell noch einmal kontrollieren kann, bevor er die dazu passenden Restrukturierungen ausführt. Er wird entdecken, dass ein Entwurfsmuster durch eine Restrukturierung zerstört werden würde, und diese deshalb verwerfen.

Im folgenden beschreiben wir die Methoden- und Klassenrollen, die wir erkennen können und zeigen am Beispiel der Rolle Fassadenklasse, wie wir Entwurfsmuster auf Basis unseres Metamodells formalisieren können.

Wir erkennen die folgenden Methodenrollen:

1. Zugriffsmethode: Eine Zugriffsmethode ist stark an die Attribute gekoppelt, die sie liest oder schreibt. Sie sollte nicht ohne diese Attribute in eine andere Klasse verschoben werden.

2. Behältermethode: Eine Behältermethode ist eine kurze Methode, die ein Objekt in einen Behälter hinzufügt oder aus ihm entnimmt. Sie ist ebenfalls eine Art Zugriffsmethode und sollte deshalb nicht ohne den Behälter verschoben werden, auf den sie den Zugriff ermöglicht.

3. Statusmethode: Eine Statusmethode führt einfache Überprüfungen durch, die den Zustand eines Objekts charakterisieren. Sie ist stark an die anderen Methoden und Attribute in derselben Klasse gekoppelt.

4. Delegationsmethode: Eine Delegationsmethode bietet keine eigene Funktionalität an, sie leitet eingehende Anfragen nur weiter. Delegationsmethoden sind nur schwach an die anderen Methoden und Attribute der Klasse, in der sie definiert sind, gekoppelt. Delegationsmethoden sind Teil vieler Entwurfsmuster.

5. Fabrikmethode: Eine Fabrikmethode erzeugt Objekte eines bestimmten Typs und verwendet oft keine Attribute der sie umgebenden Klasse. Somit besteht die Gefahr, dass sie auf Grund der starken Kopplung in die Klasse der Objekte verschoben wird, die sie erzeugt.

Aufbauend auf diesen Methodenrollen können wir die folgenden Entwurfsmuster identifizieren: Fassade, Besucher, Proxy, Adapter, Kompositum, Strategie und Abstrakte Fabrik.

Beispiel 7: Fassadenklassen
Fassadenklassen bieten eine wohldefinierte Schnittstelle für ein Teilsystem an. Grundsätzlich können sie dadurch charakterisiert werden, dass sie nahezu ausschließlich aus Delegationsmethoden bestehen. Unsere Strategie zu ihrer Erkennung sieht demnach wie folgt aus: Zunächst identifizieren wir alle Delegationsmethoden eines Systems. Danach klassifizieren wir alle Klassen als Fassadenklassen, bei denen der Anteil der Delegationsklassen einen bestimmten Prozentsatz $p_{fassade}$ übersteigt.
Delegationsmethoden können wir auf unserem Systemmodell wie folgt mit Hilfe unserer Operationen und Mengen beschreiben:

$$\begin{aligned} delegationsMethode(x) \rightarrow & |\mathcal{ZK}_x| <= 5 \\ & \wedge \exists m \in \mathcal{Z}_x^M : \mathcal{R}_m = \mathcal{R}_x \wedge \mathcal{P}_m = \mathcal{R}_m \\ & \wedge inKlasse(x) \notin \{O^*_{inKlasse(m)} \cup U^*_{inKlasse(m)}\} \end{aligned}$$

Eine Delegationsmethode x ist also dadurch charakterisiert, dass sie höchstens 5 Zugriffsketten enthält, eine andere Methode m aufruft, die formalen Parameter und der Rückgabetyp von m und x identisch sind, und die Methoden nicht in der gleichen Vererbungshierarchie liegen. Fassadenklassen können wir aufbauend auf Delegationsmethoden folgendermaßen formal beschreiben:

$$fassadenKlasse(x) \rightarrow \frac{|\{m|m \in M_x \wedge delegationsMethode(x)\}|}{|\{\mathcal{M}_x \cup \mathcal{M}_x^s \cup \mathcal{M}_x^a\}|} > p_{fassade}$$

$p_{fassade}$ wählen wir zwischen 0.8 und 0.9.

5.8. Zusammenfassung

In diesem Kapitel haben wir den ersten grundlegenden Teil unseres Ansatzes vorgestellt: Unser Metamodell bestimmt den Aufbau der Systemmodelle und legt Modellrestrukturierungen fest, mit denen wir Restrukturierungen simulieren können. Weiterhin haben wir gezeigt, wie anhand eines Systemmodells Instanzen von Entwurfsmustern als typische Strukturen bestimmt werden können, die Entwurfsheuristiken bewusst verletzen. Im nächsten Kapitel stellen wir auf diesem Kapitel aufbauend unseren evolutionären Algorithmus im Detail vor.

Kapitel 6.

Ausgestaltung des evolutionären Algorithmus

In diesem Kapitel beschreiben wir unseren evolutionären Algorithmus im Detail. Zunächst stellen wir die von uns gewählte Repräsentation vor. Im Anschluss daran beschreiben wir unsere Mutations- und Rekombinationsoperatoren, unsere Zielfunktion und unsere Selektionsstrategie.

6.1. Repräsentation des Problems

Wir müssen einen geeigneten Genotyp und die Abbildung des Genotyps auf den Phänotyp (die Repräsentation) festlegen, um einen evolutionären Algorithmus auf unser Problem anwenden zu können. Den Phänotyp haben wir bereits in Kapitel 5 in Form des Systemmodells vorgestellt. Unser Genotyp ist eine Liste von Modellrestrukturierungen. Einen Phänotyp bestimmen wir, indem wir die Modellrestrukturierungen des Genotyps nacheinander auf den ursprünglichen Phänotyp anwenden.

Es können nur dann alle im Genotyp gespeicherten Modellrestrukturierungen ausgeführt werden, wenn die durch den Genotyp vorgegebene Reihenfolge eingehalten wird. Andernfalls ist nicht sicher, dass die Vorbedingungen aller Modellrestrukturierungen erfüllt sind.

Abbildung 6.1 zeigt schematisch den Aufbau des Genotyps.

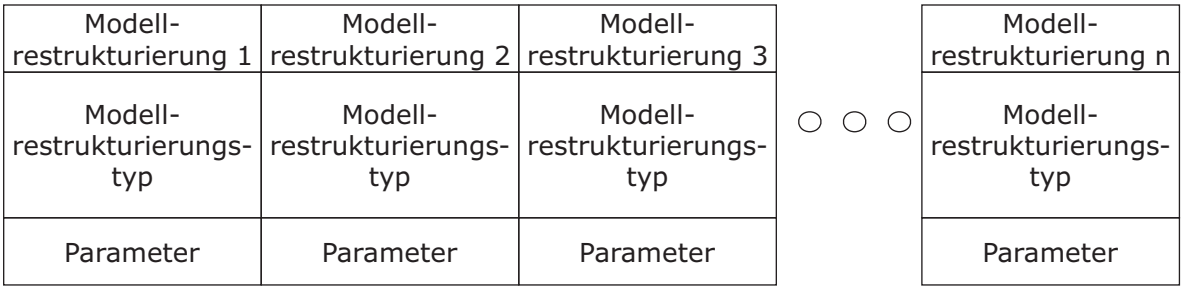


Abbildung 6.1.: Genotyp

Jedes Element der Liste besteht aus einem Tupel, das erstens den Typ der Modellrestrukturierung und zweitens die benötigten Parameter enthält. Die möglichen Modellre-

strukturierungstypen haben wir in Abschnitt 5.2.5 vorgestellt.

Die hier vorgestellte Repräsentation weicht von traditionell in evolutionären Algorithmen verwendeten Repräsentationen ab. Wir kodieren im Genotyp nicht wie üblich ein aktuelles Lösungselement, sondern die nötigen Veränderungen, um ein Anfangselement in ein aktuelles Lösungselement zu überführen. Für uns ist diese Repräsentation jedoch die einzig mögliche Repräsentation, mit der wir auch bei der Rekombination eine Veränderung des extern beobachtbaren Verhaltens im Phänotyp ausschließen können. Zusätzlich können wir durch diese Form der Kodierung die nötigen Schritte zur Veränderung der Struktur leicht ablesen. Wir müssen nur die Liste von Modellrestrukturierungen ausführen.

6.2. Evolutionäre Operatoren

In diesem Abschnitt beschreiben wir zuerst unsere evolutionären Operatoren: den Mutationsoperator und den Rekombinationsoperator. Wir beschreiben anschließend, wie wir die Vorbedingungen der Modellrestrukturierungen aus Abschnitt 5.2.5 verschärfen, um Entwurfsprinzipien bereits bei der Wahl der Parameter für die Modellrestrukturierungen berücksichtigen zu können. Anschließend beschreiben wir, wie wir Parameter von Modellrestrukturierungen reparieren können, falls sich durch unseren Rekombinationsoperator die Voraussetzungen für manche Modellrestrukturierungen geändert haben. Zum Schluss des Abschnitts skizzieren wir, wie wir Genotypen vereinfachen können und damit vermeiden, dass überflüssige Information in den Genotypen kodiert ist.

6.2.1. Mutationsoperator

Mit dem Mutationsoperator variieren wir einen vorhandenen Genotyp leicht, um neue Bereiche des Suchraums zu erkunden. In unserem Fall besteht die Mutation darin, den Genotyp um eine weitere Modellrestrukturierung zu erweitern. Welche Modellrestrukturierung dies ist, können wir nur mit Hilfe des Phänotyps festlegen. Anhand des Phänotyps muss überprüft werden, ob die von uns ausgewählte Mutation überhaupt ausführbar ist.

Zunächst wählen wir zufällig aus, welchen Typ die auszuführende Modellrestrukturierung hat. Jede der zehn Modellrestrukturierungstypen wird mit einer vorgegebenen Wahrscheinlichkeit p_{mtyp_i} ausgewählt. Insgesamt muss gelten:

$$\sum_{i=1}^{10} p_{mtyp_i} = 1$$

Wir betrachten alle Modellrestrukturierungstypen als gleichwertig und setzen alle p_{mtyp_i} deshalb auf $\frac{1}{10}$. Anschließend wählen wir wiederum zufällig aus, welche Strukturelemente verändert werden sollen, und wie sie verändert werden sollen. Diese Auswahl kann nicht vollständig zufällig erfolgen, da die Vorbedingungen für die Modellrestrukturierungen die möglichen Parameter bereits einschränken.

Wir schränken die Menge der möglichen Parameter zusätzlich ein, indem wir bei der Wahl der Parameter Domänenwissen in Form der Entwurfsprinzipien berücksichtigen. Diese Vorgehensweise beschleunigt das Verfahren und schließt schlechte Mutationen von vorne herein aus. In Abschnitt 6.2.3 gehen wir genauer auf das von uns berücksichtigte Domänenwissen ein.

Um das Verfahren zusätzlich zu beschleunigen, berechnen wir bereits vor Beginn der Evolution für jeden Modellrestrukturierungstyp, welche Elemente die Vorbedingungen (siehe Abschnitt 5.2.5) erfüllen. Diese Information aktualisieren wir, sobald wir den Phänotyp durch eine neue Modellrestrukturierung verändern.

Abbildung 6.2 zeigt noch einmal schematisch, wie eine neue Modellrestrukturierung zu einem Genotyp hinzugefügt wird.

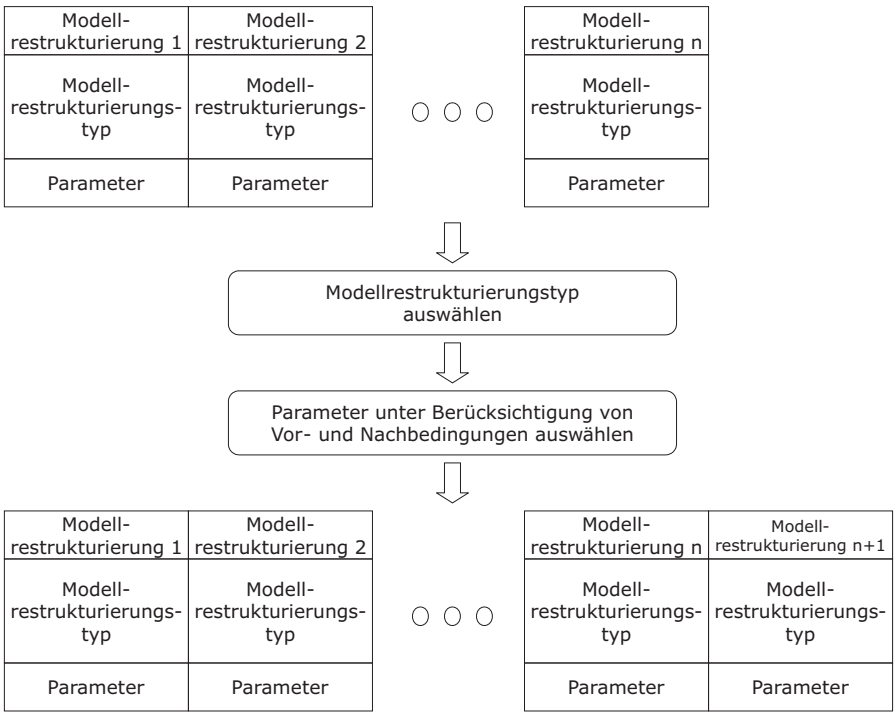


Abbildung 6.2.: Schematische Darstellung einer Mutation

Aus dem Aufbau unseres Mutationsoperators folgern wir, dass die Länge des Genotyps unbeschränkt ist und theoretisch beliebig lange Listen von Modellrestrukturierungen entstehen können. In der Praxis ist die Anzahl der vorgeschlagenen Modellrestrukturierungen im Verhältnis zu den möglichen Modellrestrukturierungen jedoch sehr klein (siehe Abschnitt 7.3.4).

Jedoch dauert es mit wachsendem Genotyp immer länger, den aktuellen Phänotyp zu bestimmen. Dieses Problem entschärfen wir, indem wir den zu einem Genotyp gehörigen Phänotyp zwischenspeichern und den Genotyp wie in Abschnitt 6.2.5 beschrieben vereinfachen.

6.2.2. Rekombinationsoperator

Mit dem Rekombinationsoperator kombinieren wir zwei existierende Genotypen, die wir Eltern-Genotypen nennen. Wir gehen davon aus, dass durch die Kombination zweier bereits guter Genotypen ein noch besserer Genotyp (ein Kind-Genotyp) entstehen kann. Besser bedeutet in diesem Zusammenhang, dass der Phänotyp des Kind-Genotyps eine höhere Fitness als die Phänotypen der Eltern-Genotypen besitzt.

Ein Rekombination führen wir folgendermaßen durch:

1. Zunächst bestimmen wir eine Zahl n, wobei n mindestens 1 und höchstens kleiner gleich der Länge des ersten Eltern-Genotyps ist.

2. Wir kopieren die ersten n Elemente aus Eltern-Genotyp 1 in den Kind-Genotyp.

3. Anschließend versuchen wir, so viele Modellrestrukturierungen wie möglich aus dem zweiten Eltern-Genotypen zu übernehmen. Wir müssen alle Modellrestrukturierungen ausführen, um entscheiden zu können, ob wir sie in den Kind-Genotyp übernehmen können.

Der große Vorteil unseres Rekombinationsoperators ist, dass er nicht destruktiv ist. Seine Anwendung führt zu einer neuen Liste von Modellrestrukturierungen, die alle ausgeführt werden können. Somit können diese Modellrestrukturierungen wieder auf das ursprüngliche System übertragen werden, ohne das von außen beobachtbare Verhalten zu verändern. Zusätzlich erwarten wir, dass seine Anwendung zu bezüglich der Zielfunktion besseren Phänotypen, d.h. Systemmodellen führt.

Prinzipiell sind für diesen Rekombinationsoperator weitere Varianten denkbar. Zum Beispiel könnten wir auch von Elternteil 2 nur einen zufälligen Anteil der Modellrestrukturierungen auswählen, oder den Kind-Genotyp dadurch erzeugen, dass wir die Eltern-Genotypen vollständig kopieren und aneinanderhängen. Es hängt von der Fallstudie ab, welche Variante besser ist. Prinzipiell sollte es jedoch jeder Variante gelingen, die Konvergenz zu beschleunigen. Abbildung 6.3 zeigt noch einmal schematisch anhand eines fiktiven Beispiels den Ablauf einer Rekombination.

Aus dem Eltern-Genotyp 1 werden die ersten drei Modellrestrukturierungen übernommen. Aus dem Eltern-Genotyp 2 werden nur die Modellrestrukturierungen 6 und 9 übernommen, da die Vorbedingungen für 7 und 8 nicht erfüllt sind.

Der in seiner Länge unbeschränkte Genotyp führt bei der Rekombination dazu, dass die Laufzeit des Operators im Verlauf der Evolution ansteigt. Jedoch hat sich wie bereits erwähnt in der Praxis gezeigt, dass die Anzahl der guten Modellrestrukturierungen im Vergleich zur Systemgröße relativ klein ist.

In unserem Verfahren wenden wir pro Evolutionsschritt den Rekombinationsoperator mit einer festgelegten Wahrscheinlichkeit $p_{Rekombination}$ an. Meist ist diese Wahrscheinlichkeit deutlich kleiner, als die Wahrscheinlichkeit $p_{Mutation}$ eine Mutation auszuführen. Mutationen sind gerade am Anfang der Evolution wichtiger, da alle Elemente einer Population zunächst gleich sind, und zunächst variiert werden müssen.

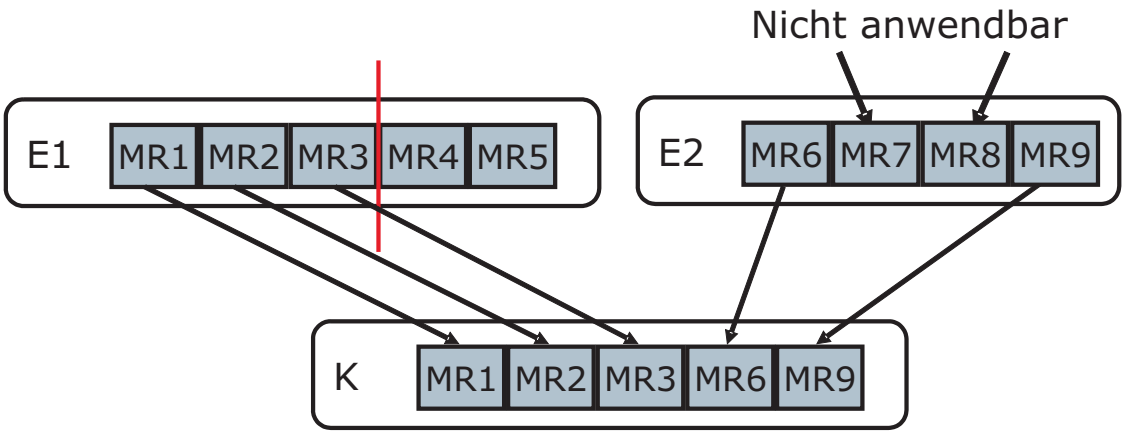


Abbildung 6.3.: Schematische Darstellung einer Rekombination

Wir könnten die Elemente der Startpopulation zwar vorab variieren, indem wir zufällig Modellrestrukturierungen ausführen und dabei die Zielfunktion nicht beachten. Allerdings würden die Genotypen dann bereits Modellrestrukturierungen enthalten, die mit hoher Wahrscheinlichkeit die Qualität der Struktur verschlechtern und den Genotyp somit unnötig verlängern würden.

6.2.2.1. Rekombination und Interaktionen von Modellrestrukturierungen

Wie wir in Abschnitt 5.2.6 beschrieben haben, können ausgeführte Modellrestrukturierungen Auswirkungen auf nachfolgende Modellrestrukturierungen haben. Bei den von uns betrachteten Modellrestrukturierungen sind dies Kombinationen, bei denen zunächst eine Klasse extrahiert wird, anschließend eine Methode aus dieser Klasse oder der Ursprungsklasse verschoben wird, und eventuell die Klasse abschließend wieder integriert wird.

Während der Rekombination werden nicht immer alle dieser aufeinanderfolgenden Modellrestrukturierungen übernommen. Die zuerst ausgeführten Modellrestrukturierungen haben Auswirkungen auf die danach ausgeführten Modellrestrukturierungen, werden während der Rekombination eventuell aber verworfen. Somit könnte man vermuten, dass der Rekombinationsoperator nicht gewährleistet, dass das beobachtbare Ein- und Ausgabeverhalten gleich bleibt.

Allerdings können wir in allen drei von uns zu berücksichtigenden Fällen zeigen, dass das Ein- und Ausgabeverhalten tatsächlich gleich bleibt. Führen wir während der Rekombination die Modellrestrukturierungen, die mit den nachfolgenden interagieren, nicht aus, so können die nachfolgenden Modellrestrukturierungen entweder gar nicht, oder ohne die Auswirkungen der vorangehenden Modellrestrukturierungen zu berücksichtigen, ausgeführt werden.

In dem von uns beschriebenen Fall 1 werden folgende drei Modellrestrukturierungen betrachtet:

1. *Extrahiere Klasse*: Aus einer Klasse k wird eine Klasse k_{spalt} extrahiert.

2. *Verschiebe Methode*: Eine Methode m wird aus k_{spalt} in eine Klasse k' ($k' \neq k$) verschoben.

3. *Integriere Klasse*: k_{spalt} wird wieder in k integriert.

Nach der letzten Modellrestrukturierung muss der Typ des während der zweiten Modellrestrukturierung erzeugten Parameters von k_{spalt} in k umgewandelt werden. Wird während der Rekombination die erste Modellrestrukturierung nicht ausgeführt, dann können die beiden folgenden Modellrestrukturierungen ebenfalls nicht ausgeführt werden. Da die Klasse k_{spalt} nicht existiert, sind die Vorbedingungen für die letzten beiden Modellrestrukturierungen nicht erfüllt. Entfällt die zweite Modellrestrukturierung, so hat dies keine Auswirkungen auf die Vorbedingungen für die dritte Modellrestrukturierung. Diese setzt nur voraus, dass es Klassen k und k_{spalt} gibt. Die Anpassung des Typs entfällt.

In dem von uns beschriebenen Fall 2 werden folgende zwei Modellrestrukturierungen betrachtet.

1. *Extrahiere Klasse*: Aus einer Klasse k wird eine Klasse k_{spalt} extrahiert.

2. *Verschiebe Methode*: Eine Methode m wird aus k in die Klasse k_{spalt} verschoben.

Im Normalfall muss der Methode m ein neuer Parameter vom Typ k hinzugefügt werden. In diesem Fall kann jedoch die während *Extrahiere Klasse* erzeugte Rückreferenz $rref$ verwendet werden. Wird die Restrukturierung *Extrahiere Klasse* während der Rekombination nicht ausgeführt, so kann auch die zweite Modellrestrukturierung nicht ausgeführt werden, da die Zielklasse k_{spalt} nicht existiert. Somit ist auch in diesem Fall gewährleistet, dass das beobachtbare Ein- und Ausgabeverhalten unverändert bleibt.

Analog zu Fall 2 können wir auch bei Fall 3 ausschließen, dass während der Rekombination das beobachtbare Ein- und Ausgabeverhalten verändert wird.

6.2.3. Zusätzliche Vorbedingungen für die Modellrestrukturierungen

Wir könnten die Parameter der Modellrestrukturierungen vollständig zufällig auswählen. Es ist nur eine Frage der Zeit, bis eine bessere Systemstruktur gefunden wird, da schlechte Modellrestrukturierungen wieder verworfen werden. Ungeschickt gewählte Parameter stellen also kein prinzipielles Problem dar, können aber die Anzahl der erforderlichen Evolutionsschritte unnötig erhöhen.

Wir haben uns deshalb entschlossen, die Evolution zu beschleunigen, indem wir wie in Abschnitt 6.2.1 beschrieben bei der Wahl der Parameter für eine Modellrestrukturierung Domänenwissen berücksichtigen. In manchen Fällen schränken wir die bereits gültigen Vorbedingungen zusätzlich ein. Da wir so die bereits vorhandenen Vorbedingungen verschärfen, können wir weiterhin garantieren, dass die Modellrestrukturierungen auf das ursprüngliche System übertragen werden können und sich das beobachtbare Verhalten nicht ändert. Im Folgenden beschreiben wir das für die einzelnen Modellrestrukturierungen eingesetzte Domänenwissen.

- *Extrahiere Teilsystem*: Wir zerlegen die Klassen eines Teilsystems unter Berücksichtigung von Kopplung und Kohäsion in zwei Teilmengen. Die Teilmengen gewinnen wir mit Hilfe eines klassischen Algorithmus zur Ballungsanalyse [Bau05].

- *Integriere Teilsystem*: Wir verschmelzen keine voneinander unabhängigen Teilsysteme. Sei t_I das zu integrierende Teilsystem und t das Teilsystem, in das t_I integriert werden soll (siehe Abschnitt 5.2.5.2). Dann fügen wir folgende zusätzliche Vorbedingung für die Modellrestrukturierung hinzu:

  $$benutzt(t, t_I) \vee benutzt(t_I, t)$$

  Diese Vorbedingung können wir anhand der vordefinierten Mengen und Elementaroperationen prüfen.

- *Verschiebe Klasse*: Wir verschieben eine Klasse k nur in Teilsysteme, in denen sich mindestens eine Klasse k_2 befindet, die k benutzt oder von der k abhängt. Als zusätzliche Vorbedingung für die Modellrestrukturierung aus Abschnitt 5.2.5.3 muss also gelten:

  $$\exists k_2 : k_2 \in \mathcal{S}_{t_2} \wedge (benutzt(k_2, k) \vee benutzt(k, k_2))$$

- *Verschiebe Methode*: Hier berücksichtigen wir bereits implizit Domänenwissen, da wir Methoden nur in die Klasse von Parametern der Methode oder von Attributen der umgebenden Klasse verschieben. Diese Vorbedingungen sind aber im Gegensatz zu den oben beschriebenen Vorbedingungen für *Verschiebe Klasse* und *Integriere Teilsystem* nötig, um zu garantieren, dass die Modellrestrukturierungen auf das ursprüngliche System übertragen werden können.

- *Verschiebe Attribut in Oberklasse*: Wir verschieben ein Attribut a vom Typ t_a nur in eine Oberklasse k_z, falls t_a keine Unterklasse von k_z ist. Ansonsten würde die Oberklasse von der Unterklasse abhängen. Die zusätzliche Vorbedingung lautet wie folgt:

  $$t_a \notin \mathcal{U}^*_{k_z}$$

- *Extrahiere Klasse(n)*: Die Attribute und Methoden werden mit Hilfe einer Ballungsanalyse bezüglich der Entwurfsprinzipien Kopplung und Kohäsion aufgeteilt. Die Ballungsanalyse ist ähnlich zur Ballungsanalyse, die wir für die Modellrestrukturierung *Extrahiere Teilsystem* verwenden.

Bei den restlichen Modellrestrukturierungen verwenden wir kein Domänenwissen.

6.2.4. Reparatur der Parameter

Wir müssen die Parameter für die Modellrestrukturierungen korrekt angeben, da bei falschen Parametern nicht alle Vorbedingungen der Modellrestrukturierung erfüllt sind und die Modellrestrukturierung somit nicht durchgeführt werden kann. In unserem evolutionären Algorithmus tritt bei der Rekombination oft der Fall ein, dass Parameter nicht mehr korrekt sind.

Beispiel 8: ***Rekombination und Extrahiere Klasse***
Ist in einem Genotyp beispielsweise eine Modellrestrukturierung *Extrahiere Klasse* enthalten, die eine Klasse K aufteilt, so muss mit den Mengen V_1 und V_2 festgelegt werden, welche Attribute und Methoden in der Ursprungsklasse verbleiben, und welche in die neue Klasse verschoben werden. Durch die Rekombination kann es vorkommen, dass Attribute und/oder Methoden der Klasse K vor Ausführung von *Extrahiere Klasse* in andere Klassen verschoben oder Methoden und Attribute zusätzlich nach K verschoben wurden. So enthalten die Mengen V_1 und V_2 eventuell Elemente, die gar nicht mehr in der Klasse enthalten sind, oder die Klasse enthält Elemente, die in keiner der Mengen enthalten sind. Beide Fälle verletzen eine Vorbedingung von *Extrahiere Klasse*, die Modellrestrukturierung kann somit nicht mehr ausgeführt werden.

Wir möchten jedoch so viele Modellrestrukturierungen wie möglich beibehalten, da diese bis zum Zeitpunkt der Rekombination fester Bestandteil der kombinierten Genotypen waren, und wir erwarten, dass sich durch ihr Entfernen die Fitness der Struktur wieder verschlechtert. Dies gilt insbesondere für Modellrestrukturierungen, von denen viele Strukturelemente betroffen sind, wie z.B. *Extrahiere Klasse* und *Extrahiere Teilsystem*. Deshalb reparieren wir die Parameter dieser beiden Modellrestrukturierungen.

6.2.4.1. Extrahiere Klasse

Um die Modellrestrukturierung in jedem Fall ausführen zu können, reparieren wir die Parameter wie folgt: Die Parameter enthalten die Ursprungsklasse und die Aufteilung der Klassenelemente. Aus dieser Aufteilung müssen alle Elemente entfernt werden, die nicht mehr in der Ursprungsklasse des Systems existieren. Neue Elemente in der Ursprungsklasse müssen der Aufteilung hinzugefügt werden, damit die Aufteilung eine vollständige Abbildung der Klasse bildet.

Sei k die Ursprungsklasse, V_1 und V_2 die beiden Mengen der Aufteilung aus dem gegebenen Parameter, so dass V_1 die Menge der Elemente von k und V_2 die von k_{spalt} repräsentiert. Zusätzlich sei hier $\mathcal{EL}_k := \mathcal{M}_k \cup \mathcal{A}_k$. Dann müssen V_1 und V_2 wie folgt angepasst werden:

$$V_1 := \{V_1 \cap \mathcal{EL}_k\} \cup \{\mathcal{EL}_k \setminus \{V_1 \cup V_2\}\}$$
$$V_2 := V_2 \cap \mathcal{EL}_k$$

6.2.4.2. Extrahiere Teilsystem

Als Parameter für die Modellrestrukturierung *Extrahiere Teilsystem* müssen wir die Menge der Klassen B angeben, die aus dem Ursprungsteilsystem t entfernt und zu einem neuen Teilsystem t_E hinzugefügt werden sollen.

Den Parameter B können wir reparieren, indem wir alle Klassen aus B entfernen, die nicht mehr in t enthalten sind:

$$B = \mathcal{S}_t \cap B$$

Klassen, die neu zum Teilsystem t hinzugekommen sind, verbleiben in t.

6.2.5. Vereinfachung des Genotyps

Wie wir in den vorherigen Abschnitten beschrieben haben, ist die Länge unseres Genotyps unbeschränkt. In ersten Experimenten konnten wir beobachten, dass unser Genotyp sehr oft überflüssige Modellrestrukturierungen enthält. In vielen Fällen gab es Paare von Modellrestrukturierungen, die sich gegenseitig aufheben. Zum Beispiel wurden Methoden oft erst in eine Oberklasse verschoben und direkt danach wieder zurück in die Unterklasse verschoben.

Der Grund für diese überflüssigen Modellrestrukturierungen ist unsere Selektionsstrategie, die auch leichte Verschlechterungen zulässt. So kann es zum Beispiel sein, dass ein Genotyp um eine Modellrestrukturierung $M1$ ergänzt wird, die Fitness sinkt, der Genotyp aber dank Turnierauswahl trotzdem in den nächsten Evolutionsschritt übernommen wird. Wird nun eine Modellrestrukturierung $M2$ ausgeführt, die $M1$ rückgängig macht, so steigt die Fitness wieder auf den ursprünglichen Wert an und der Genotyp lebt weiter.

Die Modellrestrukturierungen $M1$ und $M2$ sind somit überflüssig. Sie sind sogar aus zwei Gründen störend: Erstens verlangsamt sich unser Verfahren durch sie unnötig, da der aktuelle Phänotyp bestimmt wird, indem alle Modellrestrukturierungen des Genotyps auf den ursprünglichen Phänotyp angewendet werden. Zweitens stören diese Modellrestrukturierungen später den Softwareingenieur, der die Modellrestrukturierungen als Restrukturierungen auf dem eigentlichen System ausführt.

Deshalb versuchen wir nach jeder Mutation und Rekombination, die Genotypen zu vereinfachen. Wir suchen also nach direkt aufeinanderfolgenden Modellrestrukturierungen, die sich gegenseitig aufheben. Dieser Schritt kostet uns etwas Laufzeit, allerdings sparen wir durch ihn bei jeder Rekombination wieder Laufzeit ein.

In Tabelle 6.4 haben wir die Modellrestrukturierungen und die dazu inversen Modellrestrukturierungen dargestellt. Fügen wir neue Modellrestrukturierungen zu unserem Verfahren hinzu, so müssen wir diese Liste gegebenenfalls ergänzen.

Modellrestrukturierung	Inverse Modellrestrukturierung
Extrahiere Teilsystem(t,t_E,B)	Integriere Teilsystem (t,t_E)
Extrahiere Klasse(k,k_{spalt},V)	Integriere Klasse(k,k_{spalt})
Verschiebe Klasse(k,t_q,t_z)	Verschiebe Klasse(k,t_z,t_q)
Verschiebe Attribut in Oberklasse(a,k_q,k_z)	Verschiebe Attribut in Unterklasse(a,k_z,k_q)
Verschiebe Methode in Oberklasse(m,k_q,k_z)	Verschiebe Methode in Unterklasse(m,k_z,k_q)
Verschiebe Methode(m,k_q,k_z)	Verschiebe Methode(m,k_z,k_q)

Abbildung 6.4.: Modellrestrukturierungen und inverse Modellrestrukturierungen

6.3. Eine Zielfunktion zur werkzeuggestützten Bestimmung der Qualität der Systemstruktur

Mit Hilfe unserer Zielfunktion bewerten wir die Qualität der Struktur eines Systems. Wie wir bereits in Abschnitt 2.2 beschrieben haben, gibt es mehrere Entwurfsprinzipien, nach denen die Struktur eines Systems entworfen sein sollte. Diese Entwurfsprinzipien können wir durch die Überprüfung von Entwurfsheuristiken messen. Hierzu verwenden wir Metriken, die entweder strukturelle Eigenschaften direkt oder die Anzahl der Verletzungen von Regeln (siehe Abschnitt 2.3) messen.

Abbildung 6.5 zeigt die Entwurfsprinzipien und einen Ausschnitt aus den von uns verwendeten Metriken.

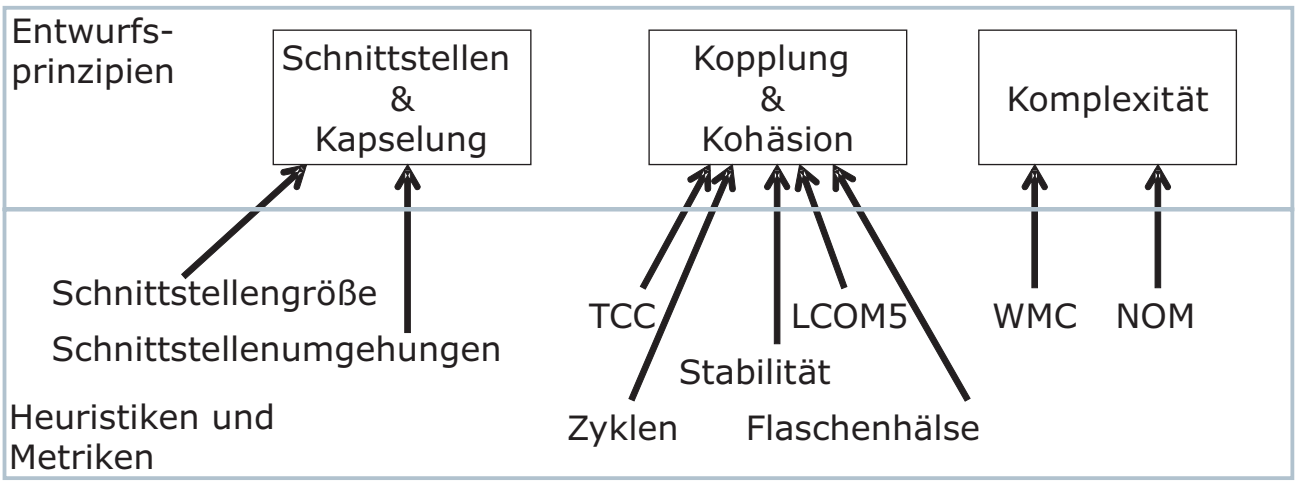


Abbildung 6.5.: Ausschnitt aus der Zielfunktion

Anhand der Abbildung wird deutlich, dass wir ein einfaches Qualitätsmodell aufgestellt haben: Die Entwurfsprinzipien stellen die geforderten Qualitätseigenschaften dar, und die Metriken machen die Qualitätseigenschaften messbar [MSS05].

In den folgenden Abschnitten stellen wir die von uns ausgewählten Metriken vor. Bei der Auswahl der Metriken haben wir insbesondere die Metriken berücksichtigt, die sich in dem BMBF-Forschungsprojekt QBench[1] als besonders relevant herausgestellt haben.

Wir beschreiben die von uns an den Metriken vorgenommenen Anpassungen und erklären, wie wir die Metriken mit unserem Metamodell berechnen können. Ein Softwareingenieur kann die Zielfunktion jederzeit um neue Metriken ergänzen bzw. vorhandene Metriken streichen. Wir teilen die Metriken in zwei Kategorien ein: Metriken der ersten Kategorie bewerten die Qualität der Struktur auf Klassenebene, Metriken der zweiten Kategorie die Qualität auf Teilsystemebene. Innerhalb dieser Kategorien ordnen wir die Metriken anhand der Entwurfsprinzipien, die sie überprüfen.

Im Normalfall berücksichtigen wir in der Zielfunktion auch die während der Klassifikation markierten Elemente. In manchen Fällen werden diese Strukturelemente besonders behandelt, z.B. werden Zugriffsmethoden bei der Metrik K-Kohäsion (LCOM) (siehe Abschnitt 6.3.1.5) als Attributzugriffe betrachtet. Fremdstrukturen, d.h. Strukturelemente, die zu Bibliotheken oder Rahmenwerken gehören, berücksichtigen wir nicht in unserer Zielfunktion, da wir diese Strukturen nicht verändern können.

6.3.1. Metriken für Klassenstrukturen

Wir beschreiben nun die Metriken zur Bewertung der Qualität von Klassen. Alle Metriken sind auf die Anzahl der Klassen normiert, da sich diese im Laufe der Evolution ändern kann. Die Metrikwerte müssen unabhängig von der Anzahl der Klassen sein.

6.3.1.1. K-Kopplung (RFC)

Die Metrik *Antwortmenge einer Klasse*[2] ist eine bekannte Kopplungsmetrik [BDW99] (K-Kopplung (RFC)). Für eine Klasse k ist K-Kopplung (RFC) als die Anzahl der Methodenaufrufe, die aus in k vereinbarten Methoden erfolgen, definiert.

K-Kopplung (RFC) gibt es in verschiedenen Varianten. Zum einen haben wir uns entschlossen, keine transitiven Aufrufbeziehungen zu betrachten, da in vielen Systemen transitiv jede Methode von einer beliebigen Methode aus erreichbar ist. Zum anderen verwenden wir die nichtpolymorphe Variante dieser Metrik, d.h. wir zählen keine Methoden, die die statisch ermittelte Zielmethode überschreiben und somit potentiell aufgerufen werden könnten. Wir wandeln die Metrik zusätzlich ab, indem wir nur die Aufrufe von Methoden berücksichtigen, die nicht in derselben Klasse wie die aufrufende Methode definiert sind.

Der Wert von K-Kopplung (RFC) einer Klasse sollte so klein wie möglich sein, da in diesem Fall die Kopplung zu anderen Klassen sehr gering ist. Für ein System ergibt sich K-Kopplung (RFC) aus der Summe der Werte von K-Kopplung (RFC) aller Klassen normiert durch die Anzahl der Klassen:

[1] http://www.qbench.de
[2] englisch: Response for Class (RFC)

$$K - Kopplung(RFC)(k) := \sum_{m \in \mathcal{M}_k^g} |\{m_2 | m_2 \in \mathcal{Z}_m^M \wedge m_2 \notin \mathcal{M}_k^g\}|$$

$$K - Kopplung(RFC)(S) := \frac{\sum_{k \in \mathcal{K}_\mathcal{S}} K - Kopplung(RFC)(k)}{|\mathcal{K}_\mathcal{S}|}$$

6.3.1.2. K-Kopplung (ICP)

Die *datenflussbasierte Kopplungsmetrik*[3] [BDW99] misst ähnlich wie K-Kopplung (RFC) die Kopplung zwischen Klassen mit Hilfe der Methodenaufrufe. Sie berücksichtigt allerdings zusätzlich die Anzahl der Methodenparameter. Gezählt werden wie bei der Metrik K-Kopplung (RFC) nur Aufrufe von Methoden, die nicht in derselben Klasse definiert sind.

Der Wert von K-Kopplung (ICP) einer Methode m ist gleich der Anzahl der von m aufgerufenen Methoden multipliziert mit der Anzahl der Methodenparameter der aufgerufenen Methode.

Der Wert von K-Kopplung (ICP) einer Klasse k ist gleich der Summe der Werte für K-Kopplung (ICP) aller Methoden.

$$K - Kopplung(ICP)(k) := \sum_{m \in \mathcal{M}_k^g} \sum_{(m_2 \in \mathcal{Z}_m^M \wedge m_2 \notin \mathcal{M}_k^g)} |\mathcal{P}_{m_2}|$$

Der Wert von K-Kopplung (ICP) für ein System S ist die Summe aller Werte von K-Kopplung (ICP) der Klassen geteilt durch die Anzahl der Klassen.

$$K - Kopplung(ICP)(S) := \frac{\sum_{k \in \mathcal{K}_S} K - Kopplung(ICP)(k)}{|\mathcal{K}_S|}$$

6.3.1.3. K-Stabilität

Die Stabilitätsmetrik *K-Stabilität* bewertet ebenfalls die Kopplung zwischen Klassen. Wir haben sie analog zur Stabilitätsmetrik für Teilsysteme von Martin [Mar02] definiert. Sie bewertet die Kopplung zwischen Klassen, indem jeder Klasse ein Instabilitätswert zugewiesen wird. Die Kopplung zwischen Klassen ist optimal, wenn es nur Abhängigkeiten in Richtung stabilerer Klassen gibt, und nicht umgekehrt. Der Instabilitätswert einer Klasse berechnet sich wie folgt:

$$I(k) := \frac{C_e(k)}{C_a(k) + C_e(k)}$$

[3] englisch: Information Flow based Coupling

$C_e(k)$ steht für die Anzahl der Klassen, die k über Attribut- oder Methodenzugriffe benutzt. $C_a(k)$ steht für die Anzahl der Klassen, welche Methoden oder Attribute aus k benutzen.

Formal können wir $C_a(k) + C_e(k)$ wie folgt beschreiben:

$$C_e(k) := |\{k_2 | benutzt(k, k_2)\}|$$
$$C_a(k) := |\{k_2 | benutzt(k_2, k)\}|$$

Anschließend berechnen wir die Anzahl der Klassenpaare (k_1, k_2), bei denen k_1 von k_2 abhängt und die die Stabilitätsbedingung verletzten.

$$PV(S) := |\{(k_1, k_2) | benutzt(k_1, k_2) \wedge I(k_1) < I(k_2)\}|$$

Die Anzahl der maximal möglichen Verletzungen ist gleich der gerichteten Anzahl von Klassenpaaren, zwischen denen eine Abhängigkeit besteht.

$$PA(S) := |\{(k_1, k_2) | benutzt(k_1, k_2)\}|$$

Die Stabilitätsmetrik für ein System S ist dann definiert als:

$$K - Stabilitaet(S) := 1 - \frac{PV(S)}{PA(S)}$$

6.3.1.4. K-Kohäsion (TCC)

Die Metrik K-Kohäsion (TCC)[4] ist eine Kohäsionsmetrik [Mar01], die den Zusammenhalt einer Klasse misst und sowohl Methoden als auch Attribute berücksichtigt. Die Metrik ist definiert als die Anzahl der Methodenpaare, die auf dasselbe Attribut zugreifen, geteilt durch die Anzahl aller Methodenpaare. Der Zusammenhalt einer Klasse k ist demnach umso größer, je mehr Methoden aus k auf dieselben Attribute aus k zugreifen. Der Wert der Metrik K-Kohäsion (TCC) für ein System S entsteht durch aufsummieren aller Werte von K-Kohäsion (TCC) für die Klassen und anschließender Division durch die Anzahl der Klassen. Formal beschreiben wir die Metrik wie folgt:

Anzahl der Methodenpaare in einer Klasse k, die auf ein gemeinsames Attribut der umgebenden Klasse zugreifen:

$$MG(k) := |\{(m_1, m_2) | \exists a : a \in \mathcal{A}_k^g \wedge a \in \mathcal{Z}_{m_1}^A \wedge a \in \mathcal{Z}_{m_2}^A \wedge m_1, m_2 \in \mathcal{M}_K^g\}|$$

[4] englisch: Tight Class Cohesion

K-Kohäsion (TCC) für eine Klasse k:

$$K - Kohaesion(TCC)(k) := \frac{MG(k)}{\frac{|\mathcal{M}_k^g|!}{2!(|\mathcal{M}_k^g|-2)!}}$$

K-Kohaesion (TCC) für ein System S:

$$K - Kohaesion(TCC)(S) := \frac{\sum_{k \in K_S} K - Kohaesion(TCC)(k)}{|\mathcal{K}_S|}$$

6.3.1.5. K-Kohäsion (LCOM)

Die Metrik K-Kohäsion (LCOM)[5] [CK94] misst, im Gegensatz zur TCC-Metrik, mangelnde Kohäsion innerhalb einer Klasse. Für jedes Attribut einer Klasse k bestimmt sie die Anzahl der Methoden von k, die dieses Attribut verwenden. Werden innerhalb einer Klasse ausschließlich Zugriffsmethoden, also `getter-` und `setter`-Methoden verwendet, um auf Attribute zuzugreifen, so ist K-Kohäsion (LCOM) wertlos. Wir interpretieren deshalb `getter-` und `setter`-Aufrufe als Attributzugriffe, um aussagekräftige Metrikwerte zu erhalten. Zugriffsmethoden erkennen wir während des Klassifikationsschritts.

$$\begin{aligned} LHilf(k) &:= \frac{\sum_{a \in \mathcal{A}_k^g} |\{m | m \in \mathcal{M}_k^g \wedge a \in \mathcal{Z}_m^A\}|}{|\mathcal{A}_k|} \\ K - Kohaesion(LCOM)(k) &:= \frac{|\mathcal{M}_k^g| - LHilf(k)}{|\mathcal{M}_k^g| - 1} \\ K - Kohaesion(LCOM)(S) &:= \frac{\sum_{k \in \mathcal{K}_S} K - Kohaesion(LCOM)(k)}{|\mathcal{K}_S|} \end{aligned}$$

6.3.1.6. K-Komplexität (WMC) und K-Komplexität (NOM)

Mit der Metrik K-Komplexität (WMC)[6] können wir feststellen, ob die Komplexität im System gleichmäßig auf alle Klassen verteilt ist. Die Komplexität einer Klasse ergibt sich aus der aufsummierten Komplexität der in einer Klasse definierten Methoden.

Die Komplexität einer Methode messen wir auf drei verschiedene Arten:

1. als zyklomatische Komplexität [FP97].
2. als Anzahl der Quelltextzeilen.
3. als konstant 1.

[5] englisch: Lack of Cohesion of Methods
[6] englisch: Weighted Method Count

Bei letzterer Variante ergibt sich als Spezialfall die Metrik K-Komplexität (NOM)[7].

Ein Softwareingenieur muss die für ihn akzeptablen Werte für die Komplexität einer Klasse festlegen. Hierzu definiert er eine Trapezfunktion (siehe Abbildung 6.6), die einen Komplexitätswert einer Klasse k auf einen Zielfunktionswert zwischen 0 und 1 abbildet.

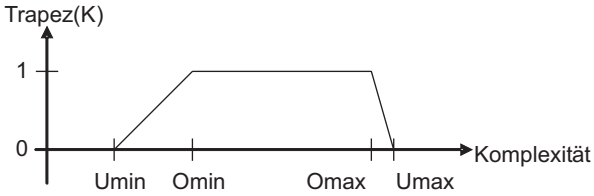


Abbildung 6.6.: Trapezfunktion zur Bewertung der Komplexität einer Klasse

Die Trapezfunktion besteht aus vier Stützstellen. Zwischen den Stützstellen o_{min} und o_{max} wird die Komplexität als optimal angesehen. Das bedeutet, dass der Wert der Trapezfunktion in diesem Intervall 1 ist. Außerhalb dieses optimalen Intervalls gibt es noch die Grenzintervalle $[u_{min}, o_{min}]$ und $[o_{max}, u_{max}]$, in denen die Werte der Trapezfunktion linear interpoliert werden. Auf diese Art und Weise erzielen wir weichere Übergänge und vermeiden, dass der Metrikwert 0 wird, wenn die Komplexität nur minimal außerhalb des optimalen Intervalls liegt:

$$K - Komplexitaet(S) := \frac{\sum_{k \in \mathcal{K}_S} trapez(\sum_{m \in \mathcal{M}_k \cup \mathcal{M}_k^S} komplexitaet(m))}{|\mathcal{K}_S|}$$

Beispiel 9: *Parameter für Trapezfunktion*

Für K-Komplexität (NOM) können wir Parameter für die Trapezfunktion beispielsweise anhand der Problemmuster *Klässchen* und *Gottklasse* aus [SSM06] ableiten. *Klässchen* sind Klassen, die weniger als 3 Methoden haben. *Gottklassen* definieren mehr als 50 Methoden. Somit können wir u_{min} auf 3 und u_{max} auf 50 festlegen. Außerhalb dieses Intervalls ist die Komplexität einer Klasse inakzeptabel und der Metrikwert somit 0. Für die weichen Übergänge können wir o_{min} auf 13 und o_{max} auf 40 festlegen.

6.3.1.7. Geheimnisprinzip und Schnittstellen: K-Schnittstellenumgehungen

Sei m eine Methode, die in einer Klasse k_q definiert ist. Alle Aufrufe an m, die über ein Objekt vom Typ einer Unterklasse von k_q erfolgen, sind Schnittstellenumgehungen. Methoden sollten in den Klassen der Hierarchie definiert werden, in denen sie auch tatsächlich benötigt werden. Der Metrikwert für ein System S ergibt sich aus der Summe aller im System vorkommenden Schnittstellenumgehungen. Als Schnittstellenumgehung wird jede Methode gezählt, in der eine Methode über die falsche Abstraktion angesprochen wird:

[7] englisch: Number of Methods

$$K - Schnittstellenumgehungen(\mathcal{K}_S) := |\{m|m \in \mathcal{M}_S^* : sumgehungen(m) > 0\}|$$

Die Anzahl der Schnittstellenumgehungen einer Methode berechnet sich wie folgt (K sei die Klasse, in der die Methode implementiert ist):

$$sumgehungen(m) := |\{z|z \in \mathcal{MT}_m^{außerhalb} \wedge typ(z) \in \mathcal{U}_k^*\}|$$
$$\mathcal{MZ}_m^{außerhalb} := \{z|z \in \mathcal{MZ}_m \wedge z_k \notin \mathcal{K} \cup \mathcal{U}_K^*\}$$

6.3.2. Metriken für Teilsystemstrukturen

6.3.2.1. TS-Kopplung

Die Kopplung zwischen zwei Teilsystemen sollte möglichst gering sein, da nur voneinander weitgehend unabhängige Teilsysteme getrennt verstanden, weiterentwickelt und getestet werden können. Zunächst bestimmen wir für jedes Teilsystem t_i die Kopplung zu den anderen Teilsystemen. Hierzu zählen wir, wieviele Abhängigkeitskanten zwischen Klassen aus t_i zu Klassen außerhalb von t_i verlaufen. Diese Metrik ähnelt den typischerweise in Verfahren zur Analyse von Teilsystemstrukturen verwendeten Kopplungsmetriken [BT04].

Für ein Teilsystem t_i berechnet sich die Menge der ausgehenden Abhängigkeitskanten wie folgt:

$$k_{aus}(t_i) := \sum_{t_a \in (\mathcal{T} \setminus t_i)} kopplung(t_i, t_a)$$

Die Kopplung des Systems berechnen wir mit den Kopplungswerten der einzelnen Teilsysteme, indem wir diese Werte aufsummieren und durch die Anzahl aller Abhängigkeitskanten dividieren. Zusammengefasst teilen wir die Anzahl der Abhängigkeiten, die zwischen Teilsystemen verlaufen, durch die Anzahl aller Abhängigkeiten.

Für ein System S bestehend aus n Teilsystemen ist

$$TS - Kopplung(S) := 1 - \frac{\sum_{i=1}^{n} k_{aus}(t_i)}{\sum_{k_1 \in \mathcal{T}} \sum_{k_2 \in \mathcal{T}} kopplung(k_1, k_2)} \quad mit\ k_1 \neq k_2$$

der Kopplungswert des Gesamtsystems.

6.3.2.2. TS-Flaschenhälse

Ein Teilsystem t_i bezeichnen wir als Flaschenhals, wenn

1. Viele andere Teilsysteme von t_i abhängen (Eingangsgrad),

2. t_i von vielen anderen Teilsystemen abhängt (Ausgangsgrad).

Im Idealfall einer Schichtenarchitektur ist der Eingangsgrad 0, falls das Teilsystem auf der obersten Schicht angesiedelt ist. Der Ausgangsgrad ist 0, falls das Teilsystem in der untersten Schicht angesiedelt ist. Befindet sich das Teilsystem auf einer mittleren Ebene, so sollten Eingangs- und Ausgangsgrad in etwa ausgeglichen, aber nicht zu hoch sein. Deshalb bestimmen wir den Flaschenhalswert eines Teilsystems als das Minimum aus Ein- und Ausgangsgrad. Je niedriger der Wert ist, desto weniger Flaschenhälse liegen vor. Die Formel zur Berechnung lautet wie folgt:

Eingangsgrad eines Teilsystems t_i:

$$einGrad(t_i) := |\{t_j | benutzt(t_j, t_i) \wedge t_i \neq t_j\}|$$

Ausgangsgrad eines Teilsystems t_i:

$$ausGrad(t_i) := |\{t_j | benutzt(t_i, t_j) \wedge t_i \neq t_j\}|$$

Als $maxGrad$ bezeichnen wir den aktuell höchsten Ein- oder Ausgangsgrad. Für ein System S bestehend aus n Teilsystemen berechnen wir die TS-Flaschenhälse wie folgt:

$$TS - Flaschenhaelse(S) := 1 - \sum_{i=1}^{n} \frac{min(einGrad(t_i), ausGrad(t_i))}{maxGrad * |\mathcal{T}|}$$

6.3.2.3. TS-Zyklen

Zyklische Abhängigkeiten zwischen Teilsystemen werden allgemein als Strukturproblem betrachtet, da die beteiligten Teilsysteme nicht mehr getrennt weiterentwickelt, getestet und ausgeliefert werden können. Je mehr Teilsysteme an einem Zyklus beteiligt sind, desto schwerer sind die Abhängigkeiten zu verstehen.

TS-Zyklen berechnen wir, indem wir unser Modell in einen Graph überführen (siehe Abschnitt 5.4). In diesem Graph berechnen wir alle vorhandenen starken Zusammenhangskomponenten mit Hilfe eines Standardalgorithmus [Sed92].

Eine starke Zusammenhangskomponente ist ein Teilgraph, in dem man von jedem beliebigen Startknoten aus jeden beliebigen anderen Knoten des Teilgraphs erreichen kann.

Im Idealfall enthält der Graph ausschließlich starke Zusammenhangskomponenten der Größe 1, d.h. es sind keinerlei zyklische Abhängigkeiten vorhanden. Die Größe einer starken Zusammenhangskomponente gewichten wir mit der Anzahl der in den beteiligten Teilsystemen enthaltenen Klassen.

Die Formel für TS-Zyklen lautet wie folgt:

$$TS - Zyklen(S) := 1 - \frac{\sum_{i=1}^{n} (groesse(szk(i))^k}{|\mathcal{T}|^k}$$

$szk(i)$ steht für die i-te starke Zusammenhangskomponente, $groesse(szk(i))$ gibt die Anzahl der Knoten zurück, die Teil der starken Zusammenhangskomponente sind. Gezählt werden nur starke Zusammenhangskomponenten, die größer als 1 sind. k ist ein Gewichtungsfaktor der größere starke Zusammenhangskomponenten schwerer bestraft.

6.3.2.4. TS-Stabilität

R. Martin entwickelte eine Stabilitätsmetrik für Teilsyteme [Mar02]. Die Grundidee ist, dass jedem Teilsystem ein Stabilitätswert zugewiesen wird und nach Möglichkeit kein stabileres von einem instabilerem Teilsystem abhängt. Der Instabilitätswert eines Teilsystems berechnet sich wie folgt: Sei $C_a(t_i)$ die Anzahl der Klassen außerhalb von t_i, die von Klassen innerhalb von t_i abhängen:

$$C_a(t_i) := |\{(k_1, k_2) | k_1 \notin S_{t_i} \wedge k_2 \in S_{t_i} \wedge benutzt(k_1, k_2)\}|$$

und sei $C_e(t_i)$ die Anzahl der Klassen aus t_i, die von Klassen außerhalb von t_i abhängen:

$$C_e(t_i) := |\{(k_1, k_2) | k_1 \notin S_{t_i} \wedge k_2 \in S_{t_i} \wedge benutzt(k_2, k_1)\}|$$

Dann ist der Instabilitätswert eines Teilsystems t_i:

$$I(t_i) := \frac{C_e(t_i)}{C_e(t_i) + C_a(t_i)}$$

Wir definieren ausgehend von den Instabilitätswerten eine Verletzungszahl, die in unsere Zielfunktion eingehen kann. Die Verletzungszahl legen wir fest als die Summe aller Verletzungen der Stabilitätsbedingungen. Normalisiert wird die Metrik mit der maximal möglichen Anzahl von Verletzungen der Stabilitätsbedingung:

$$TS - Stabilitaet(S) := \frac{|\{(t_i, t_j) | t_i \neq t_j \wedge benutzt(t_i, t_j) \wedge I(t_i) < I(t_j)\}|}{|\{(t_i, t_j) | t_i \neq t_j \wedge benutzt(t_i, t_j)|}$$

6.3.2.5. TS-Kohäsion

Die Kohäsion sollte innerhalb eines Teilsystems möglichst hoch sein, da das Teilsystem sonst zwei oder mehr voneinander unabhängige Aufgaben erfüllt, die in getrennten Teilsystemen gekapselt werden sollten.

TS-Kohäsion eines Teilsystems t_i berechnen wir wie folgt. Zunächst bestimmen wir die Anzahl der Abhängigkeitskanten, die innerhalb von t_i verlaufen:

$$intraab(t_i) := |\{(k_1, k_2) | k_1 \in t_i \wedge k_2 \in t_i \wedge benutzt(k_1, k_2)\}|$$

Die Anzahl der Klassen eines Teilsystems t_i ist: $|\mathcal{S}_{t_i}|$
TS-Kohäsion eines Teilsystems t_i ist dann:

$$TS - Kohaesion(t_i) := \begin{cases} 0 & |\mathcal{S}_{t_i}| = 0 \\ \frac{intraab(t_i)}{|\mathcal{S}_{t_i}|*(|\mathcal{S}_{t_i}|-1)} & \text{sonst} \end{cases}$$

TS-Kohäsion für das Gesamtsystem S, das aus n Teilsystemen besteht ist:

$$TS - Kohaesion(S) := \sum_{i=1}^{n} \frac{TS - Kohaesion(t_i)}{n}$$

6.3.2.6. TS-Komplexität (NOC) und TS-Komplexität (WMC)

Wir bewerten die Komplexität eines Teilsystems auf zwei Arten:

1. Indem wir die Anzahl der Klassen innerhalb eines Teilsystems zählen: TS-Komplexität (NOC).
2. Indem wir die WMC-Werte der in diesem Teilsystem enthaltenen Klassen aufsummieren: TS-Komplexität (WMC).

Für jede Variante muss ein Softwareingenieur mit Hilfe einer Zielfunktion festlegen, in welchem Bereich die Komplexität als optimal angesehen wird. TS-Komplexität (NOC) für ein System S bestehend aus n Teilsystemen berechnen wir wie folgt:

$$TS - Komplexitaet(NOC)(S) := \frac{\sum_{i=1}^{n} NOCGuete(|\{k|k \in \mathcal{S}_{T_i}\}|)}{n}$$

NOCGuete(x) ist die Zielfunktion, die einer Anzahl von Klassen einen Fitnesswert zwischen 0 und 1 zuweist. Die Komplexität für das gesamte System S berechnen wir wie folgt:

$$TS - Komplexitaet(WMC)(S) = \frac{\sum_{i=1}^{n} TWMCGuete(\sum_{j=1}^{|\mathcal{S}_{t_i}|} WMC(k_j))}{n}$$

6.3.2.7. Geheimnisprinzip und Schnittstellen

Die Schnittstelle eines Teilsystems sollte möglichst schmal und wohldefiniert sein. Diese Eigenschaft können wir mit dem Export-Quotienten eines Teilsystems (EQT) messen. Der EQT eines Teilsystems t_i ist die Anzahl der Klassen aus t_i, die außerhalb von t_i verwendet werden, geteilt durch die Anzahl der Klassen von t_i. Den EQT-Wert eines Teilsystems können wir formal wie folgt beschreiben:

$$H(t_i) := |\{k_1 | \exists k_2 : k_1 \in t_i \wedge k_2 \notin t_i \wedge benutzt(k_2, k_1)\}|$$
$$EQT(t_i) := \frac{H(t_i)}{|\mathcal{S}_{t_i}|}$$

Den EQT-Wert eines Systems S bestimmen wir, indem wir alle EQT-Werte aufsummieren und durch die Anzahl der Teilsysteme dividieren ($n = |\mathcal{T}|$).

$$EQT(S) := 1 - \frac{\sum_{i=1}^{n} EQT(t_i)}{n}$$

6.3.3. Metrikberechung am Beispiel der Referenzfallstudie

Für die Referenzfallstudie zeigen wir in diesem Beispiel, wie die Metriken für Teilsysteme berechnet werden können. Da wir nicht alle Klassen des Systems vorgestellt haben verzichten wir auf die Metriken für Klassen und die Komplexitätsmetrik für Teilsysteme.

Zunächst stellen wir die Referenzfallstudie in vereinfachter Form als Graph dar (siehe Abschnitt 5.4). Die Knoten unserer Graphen sind Teilsysteme oder Klassen. Die Beispielgraphen enthalten zwei Arten von Kanten. Erstens modellieren Kanten, das ein Teilsystem eine Klasse enthält, und zweitens Methodenaufrufe. Da wir die Methoden mit den sie enthaltenden Klassen zusammengefasst haben, verlaufen die *ruftAuf*-Kanten zwischen den Klassen. Aus Gründen der Übersichtlichkeit haben wir hier eine weitere Vereinfachung vorgenommen und nur einen Teil der *ruftAuf*-Kanten übernommen.

Abbildung 6.7 zeigt den auf diese Art und Weise gewonnenen Graph. Abbildung 6.8 zeigt ebenfalls einen Graphen, der die Referenzfallstudie visualisiert. Hier wurden die Klassen-Knoten anhand der *enthält*-Kanten mit den Teilsystem-Knoten verschmolzen.

Zur Berechnung der Metrik TS-Zyklen müssen wir die starken Zusammenhangskomponenten im Graph 6.8 bestimmen. In unserem einfachen Beispiel sehen wir sofort, dass es genau eine starke Zusammenhangskomponente der Größe 2 gibt. Die Zyklenmetrik berechnet sich also wie folgt:

$$TS - Zyklen(S) = 1 - \frac{2^k}{2^k} = 0$$

Ein Metrikwert von 0 ist hier berechtigt, da die einzigen beiden Teilsysteme zyklisch voneinander abhängen.

Die Metrik TS-Kopplung berechnen wir, indem wir für jedes Teilsystem zählen, wie viele *ruft auf*-Kanten zwischen Teilsystemen verlaufen und diese Anzahl durch die Gesamtzahl der *ruft auf*-Kanten teilen:

$$\begin{aligned} TS - Kopplung(S) &= 1 - \frac{Kopplung(XML, Kern) + Kopplung(Kern, XML)}{10} \\ &= 1 - \frac{2+4}{10} = 0.4 \end{aligned}$$

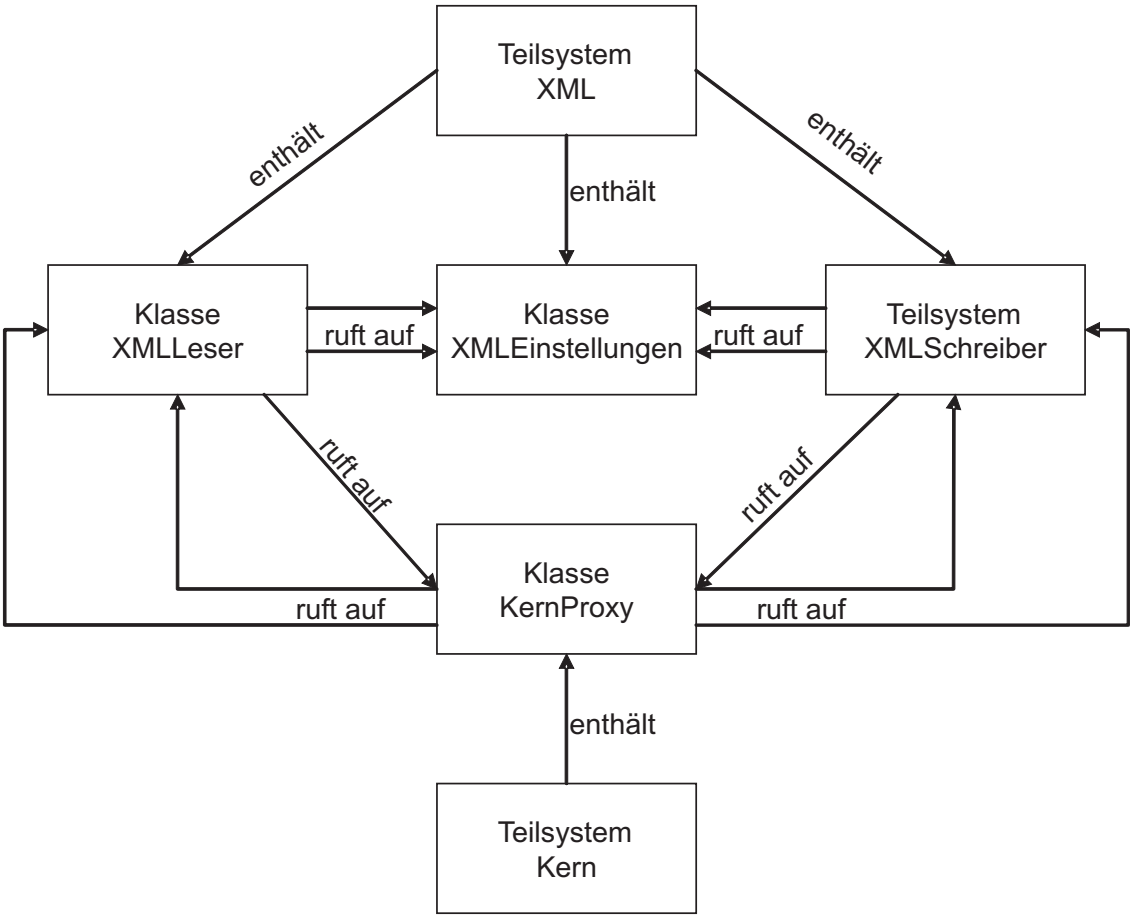


Abbildung 6.7.: Vereinfachter Graph für die Klassen und Teilsysteme der Referenzfallstudie

Der Metrikwert von 0.6 zeigt uns, dass mehr Abhängigkeiten zwischen den Teilsystemen als innerhalb der Teilsysteme verlaufen.

Die Metrik TS-Flaschenhälse berechnen wir, indem wir die Ein- und Ausgangsgrade für die beiden Teilsysteme bestimmen. Für jedes Teilsystem ist der Ein- und Ausgangsgrad 1, so dass sich die Metrik wie folgt berechnet:

$$TS - Flaschenhaelse(S) = 1 - \frac{min(1,1) + min(1,1)}{1 * 2} = 0$$

Die Kohäsionsmetrik können wir nur unvollständig berechnen, da wir nicht alle Klassen des Teilsystems Kern kennen. Wenn wir annehmen, dass es außer der Klasse `KernProxy` keine anderen Klassen gibt, berechnet sich die Metrik wir folgt:

$$TS - Kohaesion(S) = \frac{\frac{2}{3*2} + \frac{0}{1}}{2} = \frac{1}{6}$$

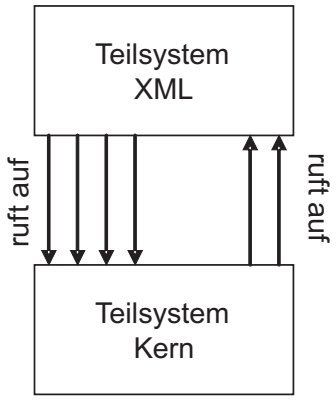


Abbildung 6.8.: Vereinfachter Graph für die Teilsysteme der Referenzfallstudie

Der Wert der Metrik TS-Kohaesion ist nicht besonders hoch, ist aber auf Grund des nur unvollständig angegebenen Teilsystems *Kern* nicht aussagekräftig.

Zur Berechnung der Metrik TS-Stabilität müssen wir zuerst die Hilfswerte bestimmen, um darauf aufbauend den eigentlichen Metrikwert berechnen zu können:

$$
\begin{aligned}
C_e(XML) &= 2 \\
C_a(XML) &= 1 \\
C_e(Kern) &= 1 \\
C_a(Kern) &= 1 \\
I(XML) &= \frac{2}{2+1} = \frac{2}{3} \\
I(Kern) &= \frac{1}{1+1} = \frac{1}{2} \\
TS - Stabilitaet(S) &= \frac{1}{2} = 0.5
\end{aligned}
$$

In unserem Beispiel haben wir zwei mögliche Verletzungen, da sowohl *XML* von *Kern* abhängt und umgekehrt. Durch den Zyklus muss naturgemäß eine Verletzung vorhanden sein, so dass der Wert der Metrik nicht besonders gut ist.

Als letztes berechnen wir die Metrik $EQT(S)$

$$
\begin{aligned}
EQT(XML) &= \frac{2}{3} \\
EQT(Kern) &= \frac{1}{1} \\
EQT(S) &= 1 - \frac{5}{6} = \frac{1}{6}
\end{aligned}
$$

Dieser Metrikwert ist sehr schlecht, da fast alle Klassen außerhalb der sie enthaltenden Teilsysteme verwendet werden. Auch diese Metrik ist aufgrund des nur unvollständig spezifizierten Teilsystems *Kern* nur begrenzt aussagekräftig.

Im nächsten Abschnitt beschreiben wir, wie wir aus den einzelnen Metriken unsere Zielfunktion zusammensetzen.

6.3.4. Form der Zielfunktion

In der Literatur wird beschrieben, dass Paretofunktionen [BNKF98] für Optimierungsprobleme mit mehreren Zielkriterien besser als Linearkombination geeignet sind. Bei Linearkombinationen besteht die Gefahr, dass ein Merkmal die anderen besonders stark dominiert. Paretofunktionen haben allerdings den Nachteil, dass ein Nutzer aus einer so genannten Front von Lösungen die für ihn geeignetste aussuchen muss. Diese Front ist in unserem Fall oft sehr groß und unübersichtlich, da wir sehr viele Metriken in unserer Zielfunktion verwenden. Insbesondere kann es bei pareto-optimalen Lösungen vorkommen, dass einige Merkmale stark verbessert werden und manche Merkmale komplett ignoriert werden bzw. sogar stark verschlechtert werden. Diese Tatsache deckt sich nicht mit dem subjektiven Qualitätsverständnis eines Softwareingenieurs, der sich eine Berücksichtigung aller Merkmale wünscht.

Wir haben uns deshalb entschieden, die Metriken als Linearkombination zu einer Zielfunktion zu kombinieren. Bei einer Linearkombination müssen wir die einzelnen Metriken gewichten, um zu verhindern, dass einzelne Metriken andere zu stark dominieren. Auf den ersten Blick scheint es sehr einfach zu sein, geeignete Gewichte zu wählen. Allerdings sind zum einen nicht alle Metriken zwischen 0 und 1 normiert, und zum anderen haben wir festgestellt, dass der Wertebereich während unserer Optimierungsläufe nur selten voll ausgeschöpft wurde. Um alle Metriken trotzdem gleich gewichten zu können, müssen wir diesen Wertebereich vorab kennen.

Um dieses Problem zu lösen, führen wir für jede einzelne Metrik $M_i(S)$ eine Voroptimierung durch, in der die Zielfunktion nur aus $M_i(S)$ besteht. Den am Ende erreichten Wert $M_{i_{max}}(S)$ speichern wir. Während der eigentlichen Optimierung berechnen wir für jede Metrik mit Hilfe des initialen Metrikwerts $M_{i_{init}}(S)$ und $M_{i_{max}}(S)$ den bereits erreichten Anteil des optimalen Werts. Es kann während der Optimierung bezüglich aller Metriken durch anders gewählte Modellrestrukturierungen vorkommen, dass der erzielte Wert größer als unser angenommenes Maximum ist. In unseren Experimenten kam dies allerdings so gut wie nicht vor. Weiterhin ist zu erwarten, dass in den meisten Fällen das Maximum nicht erreicht wird, da die Wechselwirkungen mit anderen Metriken dies verhindern.

Der aktuelle Metrikwert berechnet sich also wie folgt:

$$M_{i_{akt}}(S) = \frac{M_i(S) - M_{i_{init}}(S)}{M_{i_{max}}(S) - M_{i_{init}}(S)}$$

Ein Softwareingenieur kann jedoch für die unterschiedlichen Metriken noch unterschiedliche Gewichte w_i vergeben, um manche Metriken stärker zu berücksichtigen.

$$fitness(S) = \sum_{i=1} w_i * M_{i_{akt}}(S)$$

Wir haben in unseren Experimenten gute Erfahrungen damit gemacht, die w_i so zu wählen, dass alle Entwurfsprinzipien gleich stark gewichtet werden und jede Metrik für ein Entwurfsprinzip wiederum gleich stark ins Gewicht fällt. Selbstverständlich kann die Zielfunktion jederzeit um neue Metriken ergänzt werden.

6.4. Selektion

Während der Evolution erzeugen wir mit Mutation und Rekombination neue Elemente in unserer Population. Aus diesen neu erzeugten Elementen und den bereits vorhandenen Elementen müssen wir die vielversprechendsten auswählen und so die Populationsgröße wieder auf die vorgegebene Anzahl reduzieren. Die Populationsgröße $pop_{groesse}$ ist ein weiterer Parameter, den wir vor Beginn der Evolution festlegen müssen.

Elemente der Population sind vielversprechend, wenn ihre Fitness im Vergleich zu den anderen Elementen der Population hoch ist. Wir sortieren jedoch nicht alle Genotypen bezüglich ihrer Fitness, sondern führen wie mittlerweile in der Literatur üblich [BNKF98] eine Turnierselektion durch. So können wir auch leichte Verschlechterungen tolerieren und lokalen Extrema besser entkommen.

Wir legen also vor der Evolution unsere Turniergröße $t_{groesse}$ fest. Während der Selektion wählen wir aus $t_{groesse}$ zufällig gewählten Elementen so lange das Element mit der besten Fitness aus, bis wir die Populationsgröße $pop_{groesse}$ wieder erreicht haben. Üblicherweise arbeiten wir mit einer Turniergröße von vier. In Abbildung 6.9 ist eine Turnierselektion schematisch für $t_{groesse} = 2$ dargestellt.

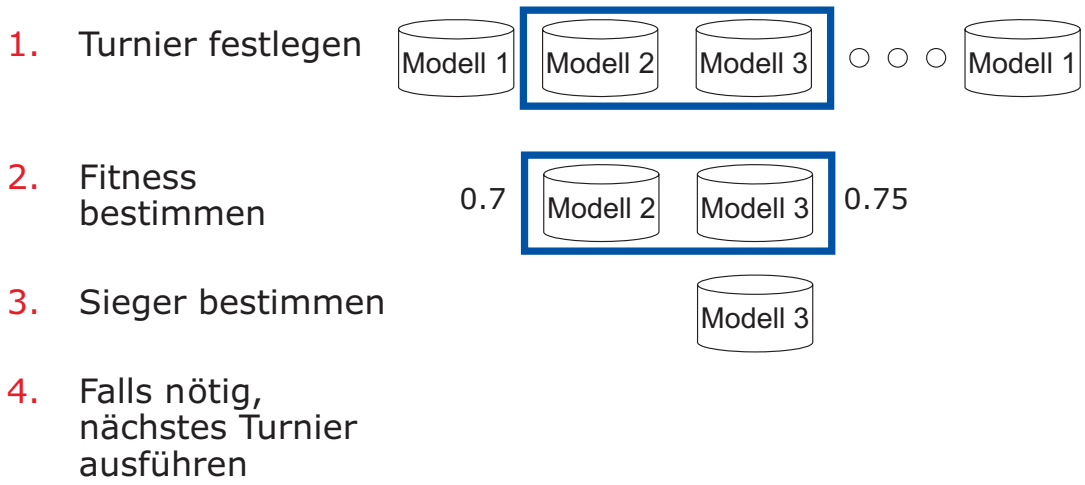


Abbildung 6.9.: Schematischer Ablauf einer Turnierselektion

6.5. Eingruppierung in das vorhandene Klassifikationsschema evolutionärer Algorithmen

Der von uns im Rahmen dieser Arbeit entwickelte Algorithmus ist ohne Zweifel ein evolutionärer Algorithmus, da er alle üblichen Eigenschaften eines solchen aufweist. Er arbeitet mit einer *Population möglicher Lösungen*, seine *Repräsentation* unterscheidet zwischen Genotyp und Phänotyp, *erneuernde* und *erhaltende Operatoren* verändern existierende Genotypen, und mit Hilfe einer *Zielfunktion selektiert* der Algorithmus die vielversprechendsten Elemente.

Allerdings passt unser Algorithmus in keine der in Abschnitt 2.7.3 vorgestellten Unterkategorien. Der Algorithmus fällt weder in die Kategorie *genetisches Programmieren* noch in die Kategorie *evolutionäres Programmieren*, da keine ausführbaren Programme Gegenstand der Optimierung sind. Somit kann die Fitness eines Individuums nicht durch Ausführung eines Programms bestimmt werden, was ein typisches Kennzeichen für Algorithmen dieser beiden Kategorien ist.

Zwei Gründe sprechen dagegen, unseren Algorithmus in die Kategorie *Genetische Algorithmen* einzugruppieren. Erstens verwenden wir hauptsächlich Mutationsoperatoren, und nicht wie bei genetischen Algorithmen üblich hauptsächlich Rekombinationsoperatoren. Zweitens repräsentieren wir potentielle Lösungen nicht als binäre Liste, sondern als Liste von komplexen Objekten.

Die Dominanz des Mutationsoperators ist ein Indikator dafür, dass unser Algorithmus den Evolutionsstrategien zuzuordnen ist. Allerdings verwenden Evolutionsstrategien als Repräsentation vorherrschend Vektoren aus reellen Zahlen.

Zusammenfassend stellen wir fest, dass wir unseren Algorithmus in keine der vier Untergruppen einordnen können. Deshalb bezeichnen wir ihn folgerichtig ausschließlich als besondere Spielart eines evolutionären Algorithmus.

6.6. Zusammenfassung

In diesem Kapitel haben wir unseren evolutionären Algorithmus zur Strukturverbesserung im Detail vorgestellt. Im Unterschied zu traditionellen evolutionären Algorithmen verwenden wir eine andere Repräsentation: Unser Genotyp besteht aus einer Liste von Mutationen, d.h. in unserem Fall Modellrestrukturierungen.

Wir haben insbesondere eine Reihe von Parametern identifiziert, die unser Verfahren beeinflussen:

- Populationsgröße: Legt die Anzahl der gleichzeitig betrachteten Systemmodelle fest und hat Einfluss auf den Rekombinationsoperator. Die Populationsgröße ist physikalisch durch den zur Verfügung stehenden Hauptspeicher beschränkt.

- Rekombinationswahrscheinlichkeit: Legt fest, mit welcher Wahrscheinlichkeit in einem Schritt eine Rekombination durchgeführt wird. Der Wertebereich liegt zwischen 0 und 1. Die Wahrscheinlichkeit für eine Mutation ergibt sich als 1 - Rekombinationswahrscheinlichkeit.

- Anzahl Evolutionsschritte: Die Evolution läuft für eine fest vorgegebene Anzahl von Schritten. Anhand des Verlaufs der Zielfunktion kann abgeschätzt werden, ob die Anzahl der Schritte zu hoch oder zu niedrig angesetzt wurde.
- Turniergröße: Legt den Selektionsdruck bei der Turnierselektion fest. Je größer die Turniergröße, desto größer der Druck.
- Gewichte der Metriken: Ermöglicht eine stärkere Gewichtung einzelner Metriken, bzw. unerwünschte Metriken können durch ein Gewicht von 0 ausgeblendet werden. Die Summe aller Gewichte muss 1 ergeben.

Im nächsten Kapitel evaluieren wir unser Verfahren anhand von fünf Fallstudien.

Kapitel 7.

Evaluation

In diesem Kapitel untersuchen wir, wie sich das von uns entwickelte Verfahren in der Praxis einsetzen lässt. Dazu beschreiben wir zunächst die prototypische Implementierung unseres Verfahrens: das Werkzeug EVO. Anschließend stellen wir alle von uns ausgewählten Fallstudien kurz vor. Im weiteren Verlauf des Kapitels stellen wir die Ergebnisse einiger Experimente vor, mit denen wir die nachfolgenden Fragen beantworten werden:

- F1: Wie wirken sich verschiedene Rekombinationswahrscheinlichkeiten und Populationsgrößen auf den Verlauf der Zielfunktion aus? Aus den Ergebnissen dieser Fragestellung leiten wir geeignete Rekombinationswahrscheinlichkeiten und Populationsgrößen für unsere weiteren Experimente ab.
- F2: Lohnt es sich in der Zielfunktion zusätzlich zu bekannten Kopplungs- und Kohäsionsmetriken weitere Metriken zu verwenden? Mit diesem Experiment beantworten wir die Frage, ob zusätzliche Entwurfsheuristiken berücksichtigt werden müssen und ob sich unterschiedliche Gewichte der einzelnen Metriken auf die Ergebnisse unseres Verfahrens auswirken.
- F3: Gelingt es unserem Verfahren, die Qualität der Systemstruktur zu verbessern? Anhand der Fallstudien untersuchen wir, ob wir die Qualität der Struktur verbessern können. Zusätzlich untersuchen wir, ob es immer möglich ist, alle Metriken zu verbessern, oder ob sich manche Metriken nur verbessern lassen, wenn sich dafür andere Metriken verschlechtern.
- F4: Sind die vorgeschlagenen Modellrestrukturierungen für einen Softwareingenieur akzeptabel? Wir untersuchen im Detail, wie die Struktur der Fallstudien verändert wurde, und welche Änderungen anschließend leichter durchführbar sind.
- F5: Kann unser Verfahren eine Referenzstruktur wieder herstellen, wenn die Systemstruktur von uns zuvor manuell verschlechtert wurde? Gelingt dies unserem Verfahren, so ist dies ein weiterer Hinweis, dass unser Verfahren für reale Systeme geeignet ist.

Jede dieser Fragen beantworten wir anhand einer oder mehrerer Fallstudien. Mit der ersten Frage bestimmen wir die wichtigsten Parameter unseres Verfahrens. Die Ergebnisse bezüglich der weiteren Fragestellungen werden zeigen, dass unser Verfahren, die von uns geforderten Kriterien erfüllt (siehe Kapitel 1).

7.1. Ein prototypisches Werkzeug zur suchbasierten Strukturverbesserung

Wir haben ein prototypisches Werkzeug namens *EVO* entwickelt, um zeigen zu können, dass unser Ansatz auch in der Praxis funktioniert. Die Architektur von *EVO* ist in Abbildung 7.1 schematisch dargestellt.

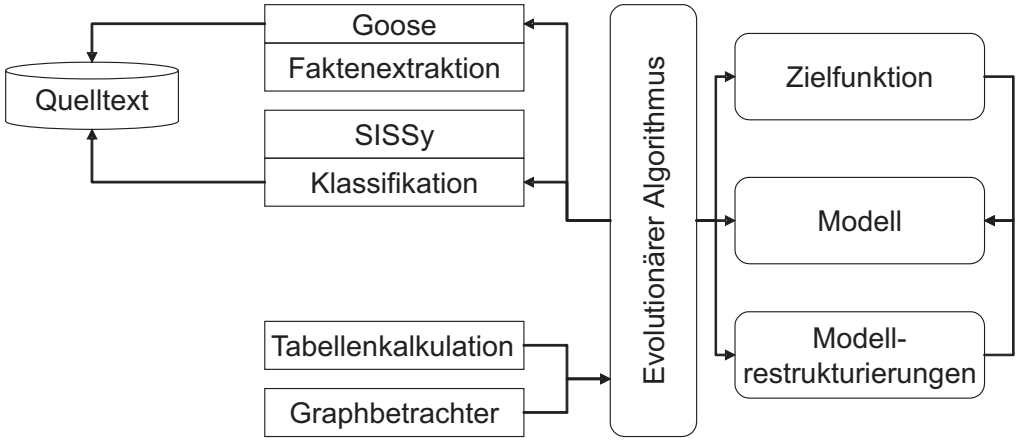


Abbildung 7.1.: Architektur von EVO

EVO besteht aus vier Teilsystemen:

1. `Evolutionärer Algorithmus`: Dieses Teilsystem steuert unser Verfahren zur Strukturverbesserung. Es liest benötigte Eingangsdaten ein und führt mit Hilfe der Teilsysteme `Modell`, `Zielfunktion` und `Modellrestrukturierungen` die Evolution durch. Ein Softwareingenieur kann EVO über eine Konfigurationsdatei steuern. In der Konfigurationsdatei kann er z.B. die Populationsgröße und die Anzahl der Evolutionsschritte festlegen. Ein Beispiel für eine vollständige Konfigurationsdatei ist in Anhang C zu sehen.

2. `Modell`: Das Teilsystem `Modell` enthält die Modellelemente und elementare Operationen, die ein Systemmodell aufbauen und verändern können (siehe Abschnitt 5.2.3). Diese Operationen fügen Modellelemente hinzu, löschen oder verschieben sie. Weitere elementare Operationen bieten Informationen über den Aufbau eines Systemmodells.

3. `Modellrestrukturierungen`: Das Teilsystem `Modellrestukturierungen` bietet die von uns in Abschnitt 5.2.5 beschriebenen Modellrestrukturierungen an. Die Modellrestrukturierungen bauen auf den elementaren Operationen des Teilsystems `Modell` auf. Die Modellrestrukturierungen überprüfen nötige Vorbedingungen und verändern ein Modell mit Elementaroperationen so, dass die erforderlichen Nachbedingungen erfüllt sind. Dies garantiert, dass das beobachtbare Ein- und Ausgabeverhalten gleich bleibt. Neue Modellrestrukturierungen können sehr leicht in diesem Teilsystem hinzugefügt werden.

4. `Zielfunktion`: Das Teilsystem `Zielfunktion` implementiert die in Abschnitt 6.3 beschriebenen Metriken. Diese Metriken können zu einer Zielfunktion in Form einer Linearkombination zusammengesetzt und unterschiedlich gewichtet werden. Zusätzlich zu den vorhandenen Metriken können jederzeit neue Metriken auf Basis des Teilsystems `Modell` implementiert werden.

Außer den entwickelten Teilsystemen sind in Abbildung 7.1 noch vier externe Programme zu sehen, die EVO zur Strukturverbesserung benötigt.

Die Programme *GOOSE*[1] und *SISSy*[2] verarbeiten den Quelltext eines Softwaresystems vor. GOOSE wurde im Rahmen des Forschungsprojektes FAMOOS am FZI entwickelt. Es führt die Faktenextraktion durch und überführt den Quelltext eines Softwaresystems in ein Modell. Dieses Modell wird in einem relationalen Format gespeichert, das von EVO eingelesen werden kann. Da nur der Java-Faktenextraktor das von uns benötigte Modell vollständig aufbauen kann, kann EVO zur Zeit nur in *Java*[3] geschriebene Programme mit dem vollen Funktionsumfang bearbeiten. Für *C++* existiert nur ein unvollständiger Faktenextraktor, der es uns nur ermöglicht, die Zerlegung in Teilsysteme zu verändern.

SISSy wurde ebenfalls am FZI entwickelt und hat *GOOSE* mittlerweile als Analysewerkzeug zur Bewertung von Softwarestrukturen abgelöst. Wir haben uns dafür entschieden, die Klassifikation als Teil von *SISSy* zu implementieren, damit sowohl *SISSy* als auch *EVO* von den Ergebnissen profitieren können. Dies ist möglich, weil das Metamodell von *EVO* eine Teilmenge des Metamodells von *SISSy* ist. Zur Klassifikation werden jedoch nur die Teile des Metamodells von *SISSy* benötigt, die auch im Metamodell von *EVO* vorhanden sind. Wir lesen die Ergebnisse der Klassifikation in Form einer XML-Datei in EVO ein und können somit besondere Strukturen markieren und unverändert lassen.

Das Ergebnis von EVO ist eine Liste von Modellrestrukturierungen. Diese Liste enthält zusätzlich Informationen über den Verlauf der Metrikwerte und den veränderten Quelltext in vereinfachter Form. Die Modellrestrukturierungen können manuell überprüft werden. Als Hilfsmittel verwenden wir einen Graphenbetrachter und eine Tabellenkalkulation. Mit Hilfe der Tabellenkalkulation können wir den Verlauf der Zielfunktion graphisch darstellen, und so z.B. beurteilen, ab welchem Punkt unser Verfahren keine wesentlichen Verbesserungen mehr vorschlägt.

Um die entstandene Struktur besser verstehen zu können, exportieren wir sie als Graph im GML-Format[4]. Diesen Graph können wir mit dem frei verfügbaren Werkzeug yEd[5] darstellen und die Auswirkungen der Modellrestrukturierungen untersuchen.

[1] http://esche.fzi.de/PROSTextern/software/goose/index.html
[2] http://sissy.fzi.de
[3] Bis einschließlich JDK 1.4
[4] http://infosun.fmi.uni-passau.de/Graphlet/GML/
[5] http://www.yworks.com

7.2. Vorstellung der Fallstudien

Wir verwenden insgesamt fünf Fallstudien. Anhand der kleineren aber überschaubaren Fallstudien *JHotDraw* und *JGraph* werden wir erstens messen, wie sich verschiedene Evolutionsparameter auswirken und zweitens die vorgeschlagenen Modellrestrukturierungen manuell überprüfen. Durch Experimente mit den Fallstudien *JFreeChart* und *JEdit* können wir zeigen, dass unser Verfahren auch für größere Systeme Restrukturierungen bestimmen kann. Anhand der Fallstudie *Rheingold* können wir zeigen, dass unser Verfahren - zumindest für die Teilsystemstruktur - auch für in *C++* geschriebene Softwaresysteme geeignet ist. Restrukturierungen der Klassen von Rheingold können wir leider nicht durchführen, da uns kein Faktenextraktor zur Verfügung steht, der die von uns benötigten Zugriffsketten extrahiert.

Die Größe der Fallstudien in Form vorhandener Artefakte ist in Abbildung 7.2 dargestellt. Für unser Verfahren ist insbesondere die Anzahl der Teilsysteme, Klassen, Attribute und Methoden interessant, da diese stärker als die Anzahl von Quelltextzeilen (LOC) die Größe des Suchraums bestimmen.

Name	LOC	#Teilsysteme	#Klassen	#Methoden	#Attribute
JGraph	23200	6	50	1183	384
JHotDraw	27626	15	209	2295	409
JEdit	148979	28	367	4631	2635
JFreechart	216297	33	521	6270	1843
Rheingold	105157	27	299	3220	1974

Abbildung 7.2.: Größe der Fallstudien

Abschließend beschreiben wir die Funktionalität der einzelnen Fallstudien im Überblick.

- JHotDraw[6]: *JHotDraw* ist ein JAVA-Rahmenwerk für technische und strukturierte Bilder. Ursprünglich wurde es als anschauliches Beispiel für eine gute Struktur konstruiert. Mittlerweile hat es diesen Zustand jedoch hinter sich gelassen und wird von einer Reihe von Anwendungen als Rahmenwerk genutzt. *JHotDraw* gilt als Musterbeispiel für den korrekten Einsatz von Entwurfsmustern, so dass wir an dieser Fallstudie unser Verfahren zur Klassifikation gut testen können. Bei *JHotDraw* stellte sich zusätzlich die Frage, ob es unserem Verfahren überhaupt gelingen kann, Verbesserungsvorschläge zu finden, da die Struktur von *JHotDraw* allgemein als gut angesehen wird.

- JGraph[7]: *JGraph* ist eine Bibliothek, um Graphen zeichnen zu können. Es wird von einer Reihe von Programmen eingesetzt, um z.B. Geschäftsprozesse oder UML-Diagramme zu visualisieren. Obwohl *JGraph* in *Java* programmiert ist, kann es

[6] http://www.jhotdraw.org - Version 5.3
[7] http://www.jgraph.com - Version 5.9.2.0

selbst große Graphen (mehr als 10000 Knoten und Kanten) noch performant darstellen.

- JFreeChart[8]: *JFreeChart* ist eine Bibliothek, mit der Diagramme erstellt werden können. *JFreeChart* unterstützt eine Vielzahl von Diagrammtypen wie z.B. Balken- oder Tortendiagramme. Diagramme können dabei in einer Vielzahl von Formaten wie z.B. als Swing-Komponenten, JPEG-Bild oder PDF-Dokument erstellt werden.

- JEdit[9]: *JEdit* ist ein ausgereifter Texteditor, der von mehreren Entwicklern programmiert wurde. Er zeichnet sich durch eine Vielzahl von Funktionen aus, die die Funktionalität vieler kommerzieller Editoren übersteigt.

Die am FZI entwickelte Fallstudie *Rheingold* haben wir bereits in Kapitel 3 ausführlich vorgestellt.

7.3. Ergebnisse

7.3.1. Klassifikation

In diesem Abschnitt stellen wir die Ergebnisse der Klassifikation unserer fünf Fallstudien vor. Die Tabelle in Abbildung 7.3 zeigt für jede Fallstudie, wie viele Methoden und Klassen mit einer bestimmten Rolle identifiziert werden konnten.

	JGraph	JHotDraw	JEdit	JFreeChart	Rheingold
Methoden insgesamt	1183	2295	4631	6270	3220
Zugriffsmethoden	404	377	856	2390	611
Behältermethoden	18	49	43	71	11
Statusmethoden	120	56	119	123	0
Delegationsmethoden	41	100	282	282	8
Fabrikmethoden	29	112	151	180	63
Besucher	1	1	0	3	0
Beobachter	3	2	1	10	0
Fassade	0	0	1	0	0

Abbildung 7.3.: Ergebnisse der Klassifikation

Die Klassifikation führt dazu, dass ein Großteil der Methoden der Fallstudien nicht mehr verändert werden muss. Der Suchraum wird somit beträchtlich verkleinert, so dass unser Verfahren weniger Evolutionsschritte benötigt, um eine gute Struktur zu finden.

Die Ergebnisse von *JHotDraw* haben wir manuell überprüft. Dabei stellten wir fest, dass unser einfaches Klassifikationsverfahren gut funktioniert. Wir konnten keine falsch

[8] http://www.jfree.org - Version 1.0.2
[9] http://www.jedit.org - Version 4.2

positiven Zugriffsmethoden, Behältermethoden, Statusmethoden und Delegationsmethoden, Besucher und Beobachter finden. Nur unter den als Fabrikmethoden klassifizierten Methoden befanden sich 11, die wir als falsch Positive identifizieren konnten.

Die Klassifikation von *JHotDraw* hat zusätzlich ergeben, dass im Teilsystem `Framework` nur Schnittstellenklassen und Klassen mit ausschließlich Delegationsmethoden vorhanden sind. Da somit keine Methoden der Klassen aus diesem Teilsystem von unserem Verfahren verschoben werden können, und die Klassen und die Schnittstellen auch keine Attribute besitzen, sind alle Klassen aus diesem Teilsystem unveränderlich. Wir haben dieses Teilsystem deshalb von der Strukturverbesserung ausgenommen. Das Teilsystem selbst ist ein Rahmenwerk für den Rest der Anwendung und muss getrennt bearbeitet werden.

7.3.2. F1: Wie wirken sich verschiedene Rekombinationswahrscheinlichkeiten und Populationsgrößen auf den Verlauf der Zielfunktion aus?

Unsere Experimente mit den Parametern Rekombinationswahrscheinlichkeit und Populationsgröße haben wir an der Fallstudie *JGraph* durchgeführt. Wir haben anhand von *JGraph* insgesamt fünf verschiedene Konfigurationen untersucht. Die Parameter der Konfigurationen sind in Abbildung 7.4 dargestellt.

Name	Populationsgröße	Rekombinationswahrscheinlichkeit
JGraphKon1	10	0.02
JGraphKon2	20	0.02
JGraphKon3	30	0.02
JGraphKon4	20	0
JGraphKon5	20	0.2

Abbildung 7.4.: JGraph Konfigurationen

In den Konfigurationen 1 bis 3 haben wir die Populationsgröße variiert und die Rekombinationswahrscheinlichkeit gleich gewählt. In den Konfigurationen 2, 4 und 5 haben wir die Rekombinationswahrscheinlichkeit verändert, die Populationsgröße ist konstant 20. Wir haben unser Werkzeug in jeder Konfiguration fünf Mal ausgeführt, um anschließend die Mediane der maximalen Fitness zu betrachten und Ausreißer somit besser ausblenden zu können.

In jeder Konfiguration haben wir 4000 Generationen und alle in den Abschnitten 6.3.1.1 und 6.3.2.1 beschriebenen Metriken in der Zielfunktion verwendet. Während der Evolution werden alle in Abschnitt 5.2.5 beschriebenen Modellrestrukturierungen mit gleicher Wahrscheinlichkeit eingesetzt.

Die Voroptimierung haben wir mit der Konfiguration JGraphKon2 durchgeführt. Die so erzielten maximalen Fitnesswerte für die Metriken der Zielfunktion haben wir für alle Konfigurationen verwendet, da unsere Zielfunktion die relative Verbesserung gegenüber

den ermittelten maximalen Werten bewertet und die Ergebnisse der einzelnen Konfigurationen sonst nicht vergleichbar wären.

Abbildung 7.5 zeigt für jede Konfiguration den Median der maximal erzielten Fitness aus fünf Versuchen.

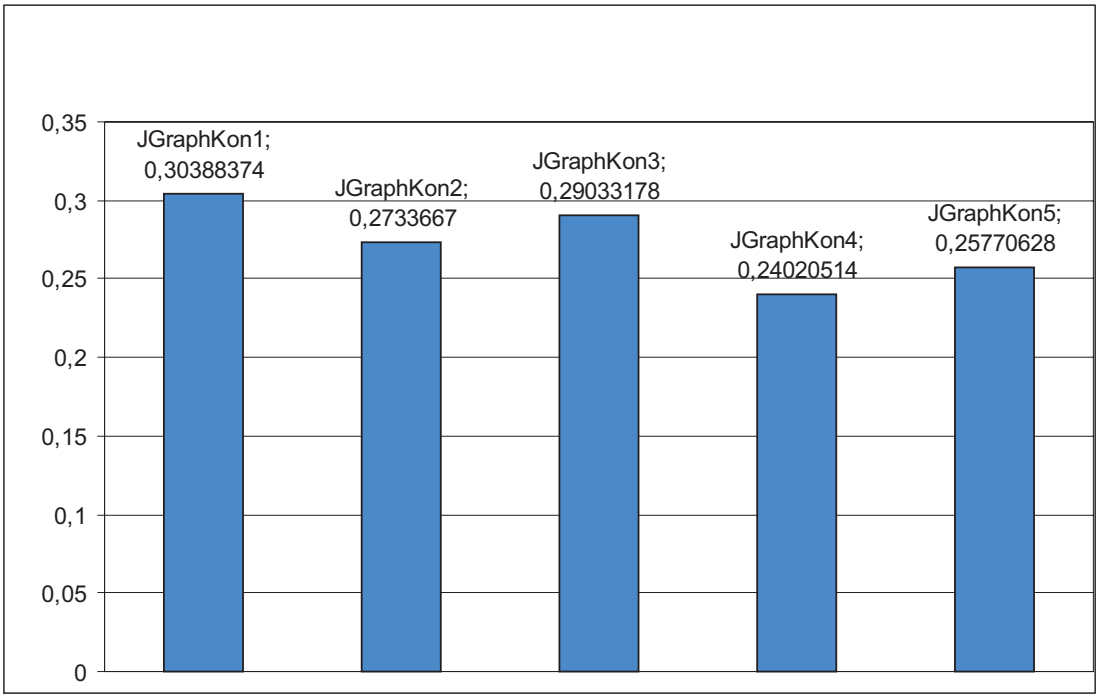


Abbildung 7.5.: Median der Maximalwerte

Wir stellen fest, dass mit den Konfigurationen JGraphKon1 (0.30), JGraphKon2 (0.27) und JGraphKon3 (0.29) die höchste maximale Fitness erzielt werden konnte. In diesen drei Konfigurationen haben wir den Rekombinationsoperator mit einer Wahrscheinlichkeit von 0.02 eingesetzt. In der Konfiguration JGraphKon4, in der wir keine Rekombination eingesetzt haben, liegt die maximale Fitness mit (0.24) deutlich unter den Fitnesswerten der Konfigurationen 1 bis 3. Allerdings stellen wir fest, dass in Konfiguration JGraphKon5 (0.26) eine zu hohe Rekombinationswahrscheinlichkeit ebenfalls zu niedrigeren maximalen Fitnesswerten führt. Bei dieser hohen Rekombinationswahrscheinlichkeit hatten die Individuen wahrscheinlich nicht genug Zeit, sich in verschiedene Richtungen zu entwickeln. Dies ist aber zunächst erforderlich, da in der Startpopulation alle Individuen gleich sind, und somit eine Rekombination zunächst keine großen Vorteile bringen kann.

Zusätzlich zur maximal erreichten Fitness haben wir die Konfigurationen bezüglich ihrer Konvergenzgeschwindigkeit verglichen. Hierzu haben wir in Abbildung 7.6 pro Konfiguration dargestellt, in welchem Evolutionsschritt eine Fitness von 0.1, 0.2 und 0.225 erreicht wurde. Aus der Abbildung geht hervor, dass diese Werte bei einer Rekombinationswahrscheinlichkeit von 0.02 am schnellsten erreicht wurden. Ohne Rekombination beziehungsweise mit einer zu hohen Rekombinationswahrscheinlichkeit wurden

mehr Schritte benötigt.

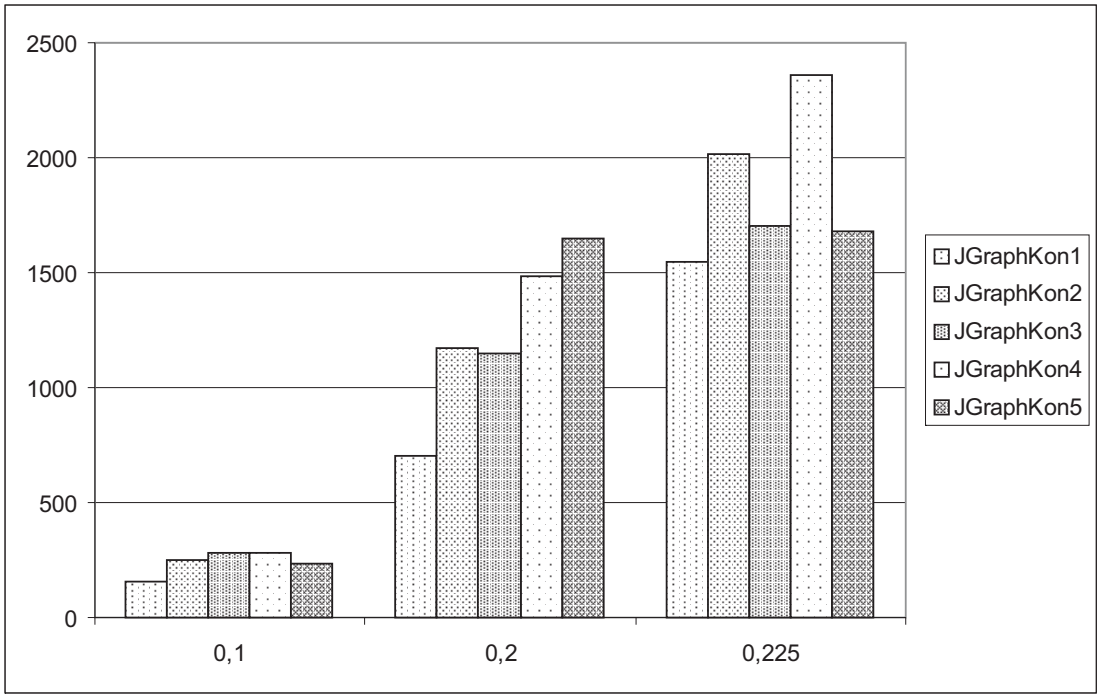


Abbildung 7.6.: Konvergenzgeschwindigkeit

Insgesamt stellen wir fest, dass ein Rekombinationsoperator mit kleiner Anwendungswahrscheinlichkeit positiven Einfluss auf die mit unserem Verfahren erzielbaren Ergebnisse hat (F1).

Die Populationsgröße scheint nur einen geringen Einfluss auf die Ergebnisse zu haben. Die beste maximale Fitness hat unser Verfahren bei einer Populationsgröße von 10 erreicht. Es könnte allerdings sein, dass bei den größeren Populationen die Individuen noch nicht genug Zeit hatten, zu mutieren und sich miteinander zu verbinden.

Als nächstes haben wir die verschiedenen Konfigurationen bezüglich der Standardabweichung der maximalen Fitness in den einzelnen Läufen verglichen. Aus dem Diagramm in Abbildung 7.7 geht hervor, dass die Ergebnisse am stabilsten sind, wenn wir keinen Rekombinationsoperator einsetzen. Die Konfigurationen mit einer Populationsgröße von 20 führen zu stabileren Ergebnissen als die Konfiguration mit einer Populationsgröße von 10. Bei einer Populationsgröße von 30 war die Standardabweichung allerdings am größten, so dass eine größere Populationsgröße nicht automatisch zu stabileren Ergebnissen führt.

Aus diesen Ergebnissen schließen wir, dass eine Populationsgröße von 10 für unsere weiteren Experimente ausreichend ist, und dass wir den Rekombinationsoperator mit einer Wahrscheinlichkeit von 0.02 einsetzen werden.

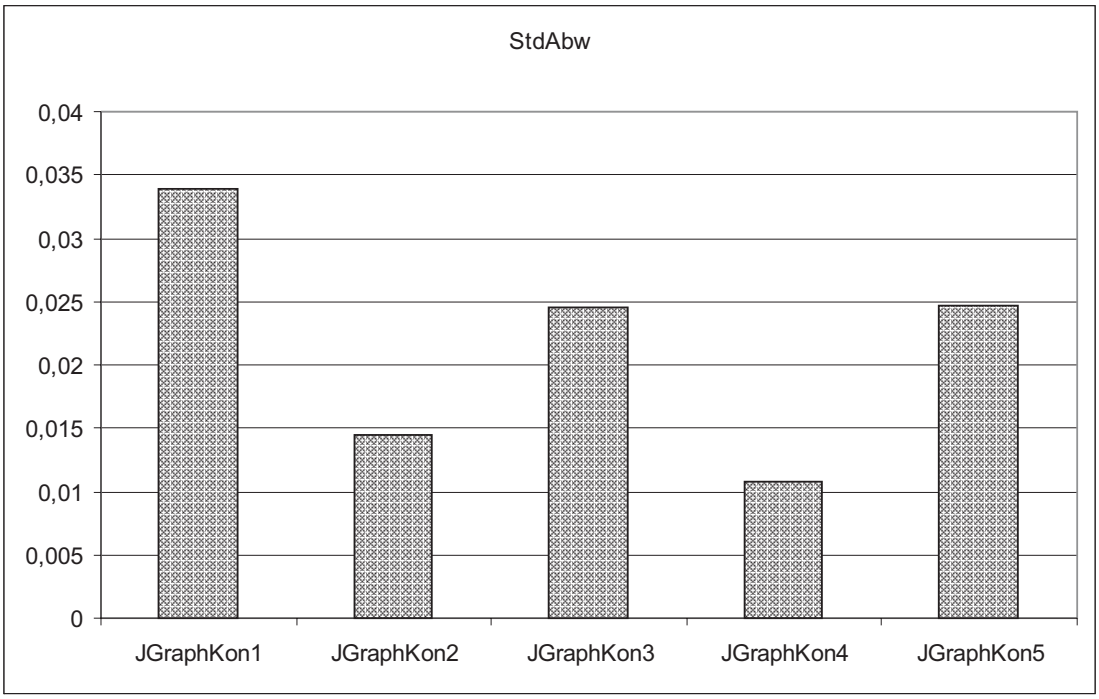


Abbildung 7.7.: Standardabweichung der maximalen Fitness

7.3.3. F2: Lohnt es sich in der Zielfunktion zusätzlich zu bekannten Kopplungs- und Kohäsionsmetriken weitere Metriken zu verwenden?

Nahezu alle existierenden Verfahren zur Untersuchung von Teilsystemstrukturen verwenden ausschließlich eine Kopplungs- und eine Kohäsionsmetrik, die fast identisch zu den von uns beschriebenen Metriken TS-Kopplung und TS-Kohäsion aus Abschnitt 6.3 sind. Anhand von *JHotDraw* haben wir untersucht, wie sich die Werte von TS-Komplexität (NOC), TS-Zyklen und TS-Flaschenhälse entwickeln, wenn die Zielfunktion ausschließlich aus der Kopplungs- und Kohäsionsmetrik aus Abschnitt 6.3.2.1 besteht [SBBP05][SP04]. Verbessern sich alle anderen Metriken in gleichem Maße mit, so ist es gerechtfertigt, die Zielfunktion auf zwei Metriken zu beschränken. Anderenfalls haben wir gezeigt, dass zusätzliche Metriken in der Zielfunktion berücksichtigt werden müssen.

Für dieses Experiment haben wir folgende Parameter verwendet:

1. Populationsgröße: 10
2. Evolutionsschritte: 4000
3. Rekombinationswahrscheinlichkeit: 0.02

Um zu untersuchen, wie sich zusätzliche Metriken auswirken, haben wir drei verschiedene Konfigurationen verwendet, die sich nur in der Zielfunktion unterscheiden:

1. JHotDrawK1: Die Zielfunktion besteht aus den Metriken TS-Kopplung und TS-Kohäsion.
2. JHotDrawK2: Die Zielfunktion besteht aus den Metriken TS-Kopplung, TS-Kohäsion und TS-Komplexität (NOC).
3. JHotDrawK3: Die Zielfunktion besteht aus den Metriken TS-Kopplung, TS-Kohäsion, TS-Komplexität (NOC), TS-Zyklen und TS-Flaschenhälse.

Wir haben unser Verfahren in jeder Konfiguration 10 Mal durchgeführt. Abbildung 7.8 zeigt für alle Konfigurationen die Mediane der erreichten maximalen Fitnesswerte. Um die Werte besser vergleichen zu können, haben wir für diese Experimente nicht die relative Verbesserung sondern die absoluten Metrikwerte dargestellt.

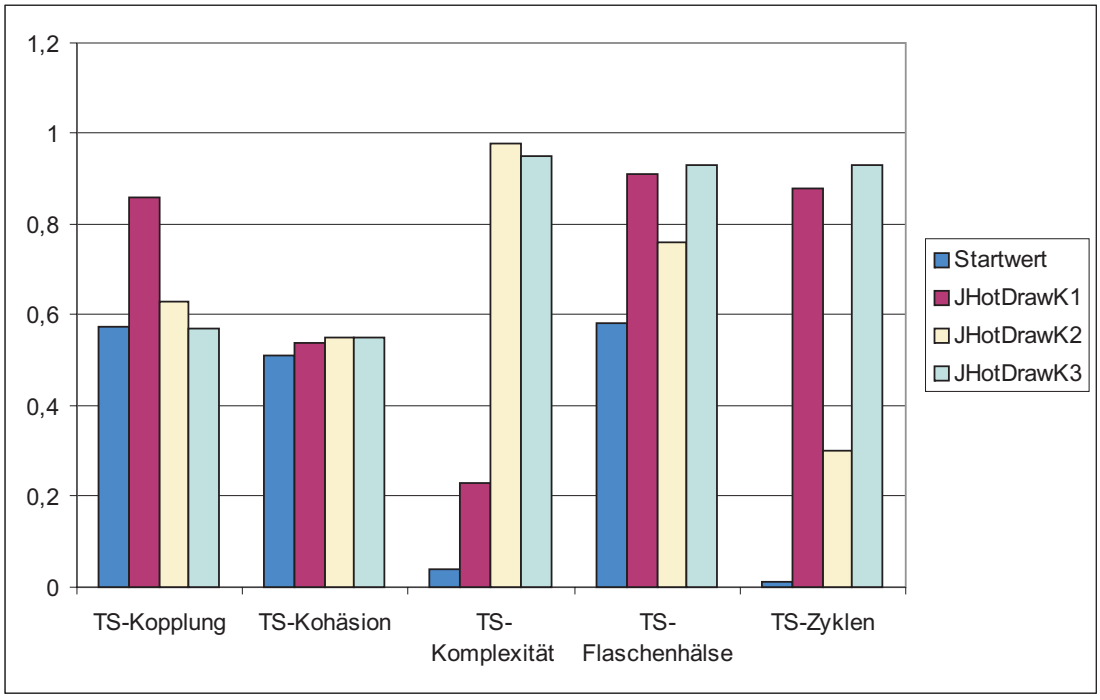


Abbildung 7.8.: Maximale Fitness bei verschiedenen Konfigurationen

Zunächst können wir aus dem Diagramm ablesen, dass eine Optimierung bezüglich TS-Kopplung und TS-Kohäsion die Werte für TS-Flaschenhälse und TS-Zyklen ebenfalls verbessert. Nur der Wert von TS-Komplexität (NOC) bleibt mit 0.23 immer noch sehr schlecht. Dieser schlechte Wert kommt dadurch zustande, dass ein sehr großes Teilsystem entstanden ist, in dem über 50 % aller Klassen enthalten sind. Die guten Werte für TS-Flaschenhälse und TS-Zyklen sind ebenfalls dadurch begründet, dass die meisten Klassen in einem Teilsystem liegen und die meisten Abhängigkeiten innerhalb dieses Teilsystems verlaufen.

Berücksichtigen wir zusätzlich zu TS-Kopplung und TS-Kohäsion TS-Komplexität (NOC), so berechnet unser Verfahren Modellrestrukturierungen, die den maximalen

Wert von TS-Komplexität (NOC) auf 0.98 verbessern. Die Maximalwerte für alle anderen Metriken steigen ebenfalls an, jedoch sind die Werte für TS-Flaschenhälse und TS-Zyklen nicht mehr so gut, wie bei Konfiguration JHotDrawK2. Somit können wir nicht alle Metriken verbessern, wenn wir zusätzlich eine Komplexitätsmetrik in der Zielfunktion verwenden.

Die Ergebnisse unseres Verfahrens bei Konfiguration JHotDrawK3 zeigen, dass in diesem Fall alle Metrikwerte steigen, bzw. bei TS- Kopplung zumindest gleich bleiben. Insbesondere sind die Werte für die TS-Flaschenhälse und TS-Zyklen mit jeweils 0.93 sehr gut.

Unsere Ergebnisse zeigen deutlich, dass alle von uns vorgeschlagenen Metriken in der Zielfunktion berücksichtigt werden müssen. Eine Optimierung bezüglich TS-Kopplung und TS-Kohäsion verbessert zwar die Werte von TS-Komplexität (NOC), TS-Zyklen und TS-Flaschenhälse. Die Verbesserung von TS-Komplexität (NOC) ist jedoch sehr gering und eine Analyse der entstandenen Teilsystemstruktur zeigt, dass diese nicht akzeptabel ist.

Unser Experiment zeigt auch, dass unterschiedliche Gewichte Einfluss auf die Ergebnisse unseres Verfahrens haben. Die Metriken TS-Komplexität, TS-Flaschenhälse und TS-Zyklen haben wir in verschiedenen Konfigurationen berücksichtigt, bzw. nicht berücksichtigt. Das bedeutet, dass die Gewichte für die Metriken 1 oder 0 waren. Wie in Abbildung 7.8 zu sehen ist, führen diese verschiedenen Gewichte zu unterschiedlichen Ergebnissen.

Diese Erkenntnisse verbessern den existierenden Stand der Technik, da alle existierenden Verfahren angenommen haben, dass eine Optimierung bezüglich Kopplungs- und Kohäsionsmetrik ausreichend ist.

Unsere Beobachtung haben wir anhand weiterer Fallstudien wiederholen können. Insbesondere konnten wir sie auch für Metriken zur Bewertung der Klassenstruktur zeigen. Eine Übersicht über diese Ergebnisse enthält eine von uns betreute Diplomarbeit [Pac05].

7.3.4. F3: Gelingt es unserem Verfahren, die Qualität der Systemstruktur zu verbessern?

Wir haben anhand von allen Fallstudien untersucht, ob es unserem Verfahren gelingt, die Qualität der Systemstruktur zu verbessern. Für alle Fallstudien haben wir eine Populationsgröße von 10 und eine Rekombinationswahrscheinlichkeit von 0.02 verwendet. Die Anzahl der Evolutionsschritte betrug bei *JGraph* 4000, bei *JHotDraw* 10000, bei *Rheingold* 40000, bei *JFreeChart* 50000 und bei *JEdit* 100000. Die nötigen Evolutionsschritte haben wir experimentell bestimmt, indem wir beobachtet haben, ab welchem Evolutionsschritt keine signifikante Veränderung der Fitness mehr auftritt. Die Anzahl der nötigen Evolutionsschritte hängt von der Anzahl der Strukturelemente und von der Anzahl der Modellrestrukturierungstypen ab. Es fällt auf, dass wir für *JFreeChart* weniger Evolutionsschritte als für *JEdit* benötigen, obwohl *JFreeChart* aus mehr Quelltextzeilen besteht und mehr Methoden enthält. Dies liegt teilweise daran, dass nach der Klassifikation nur noch 3224 der 6270 Methoden von *JFreeChart* verschoben werden können und der

Suchraum somit stark verkleinert wurde. Weiterhin hängt die Anzahl der nötigen Evolutionsschritte stark von der initialen Qualität der Struktur ab. Ist die Systemstruktur bereits von hoher Qualität, so kann unser Verfahren nur wenige Modellrestrukturierungen zur Verbesserung der Struktur finden und wird hierzu nur wenige Evolutionsschritte benötigen.

In die Zielfunktion haben wir pro Strukturierungseinheit folgende Metriken aufgenommen:

- Klassenstruktur (siehe Abschnitt 6.3.1.6) :

  1. K-Komplexität (WMC) (Gewicht: 1.5)
  2. K-Komplexität (NOM) (Gewicht: 1.5)
  3. K-Kopplung (RFC) (Gewicht: 1)
  4. K-Kopplung (ICP) (Gewicht: 1)
  5. K-Stabilität (Gewicht: 1)
  6. K-Kohäsion (TCC) (Gewicht: 1)
  7. K-Kohäsion (LCOM) (Gewicht: 1)
  8. K-Schnittstellenumgehungen (Gewicht: 2)

- Teilsystemstruktur:

  1. TS-Komplexität (NOC) (Gewicht: 3)
  2. TS-Kopplung (Gewicht: 2)
  3. TS-Zyklen (Gewicht: 0.5)
  4. TS-Flaschenhälse (Gewicht: 0.5)
  5. TS-Stabilität (Gewicht: 0.5)
  6. TS-Kohäsion (Gewicht: 3)

Wir haben fast alle der von uns beschriebenen Metriken aus den Abschnitten 6.3.1.3 und 6.3.2.1 verwendet. Die Metrik zur Bewertung der Schnittstelle eines Teilsystems haben wir weggelassen, da diese Metrik in ersten Versuchen zu nicht nachvollziehbaren Ergebnissen geführt hat, und somit nur bedingt geeignet zu sein scheint.

Die Gewichte der Metriken haben wir so gewählt, dass das Entwurfsprinzip *Kopplung und Kohäsion* ungefähr doppelt so stark gewichtet wird wie die Prinzipien *Komplexität* und *Geheimnisprinzip und Schnittstellen*. Diese Gewichtung hat in den von uns untersuchten Fallstudien zu aus unserer Sicht guten Ergebnissen geführt.

Die Laufzeit unseres Programms hängt stark von der gewählten Anzahl der Evolutionsschritte und der Größe der Modelle ab. Für die Fallstudie *JGraph* mit 4000 Evolutionsschritten benötigte unser Programm im Durchschnitt 19 Minuten. Für *JHotDraw* dauerte ein Programmlauf ca. 2 Stunden, *Rheingold* benötigte ca. 4 Stunden, *JEdit* ca.

24 Stunden und *JFreeChart* ca. 36 Stunden. Unsere Experimente führten wir auf einem Standard PC mit 3 GHz und 2 GB Arbeitsspeicher aus. Die Laufzeit für die großen Fallstudien ist sehr hoch. Trotzdem können bei diesen Laufzeiten Ergebnisse noch innerhalb von ca. 1,5 Tagen gewonnen werden, so dass unser Verfahren für Systeme dieser Größe durchaus geeignet ist. Die meiste Laufzeit verbrauchen Kopieroperationen, die vom ursprünglichen Modell eine Kopie erstellen müssen, bevor dieses mutiert werden kann. Diese Kopie wird in jedem Fall erstellt, auch wenn das entstehende Modell anschließend wieder verworfen wird. Eine geschicktere Kopierstrategie oder eine Parallelisierung unseres Verfahrens könnten eine Lösung sein, um die Laufzeit zu verbessern.

Abbildung 7.9 zeigt den Verlauf der Fitness für die Fallstudie JHotDraw.

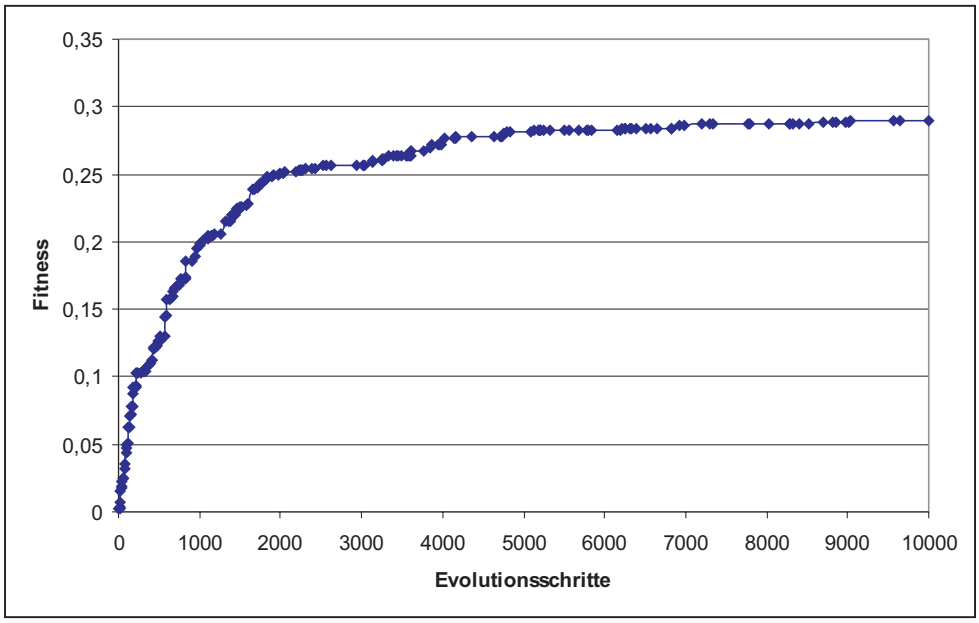


Abbildung 7.9.: Verlauf der Zielfunktion für *JHotDraw*

Ab Evolutionsschritt 5000 verbessert sich die Fitness nur noch minimal. Die Sprünge im Verlauf der Zielfunktion finden immer dann statt, wenn eine erfolgreiche Rekombination mehrere Modellrestrukturierungen eines anderen Elternteils zum vorhandenen Genotyp hinzugefügt hat.

In Abbildung 7.10 ist der Verlauf der Zielfunktion für die Fallstudie *Rheingold* zu sehen.

Ab Evolutionsschritt 20000 verbessert sich die Qualität der Struktur nur noch minimal. 255 der am Ende im Genotyp gespeicherten 286 Modellrestrukturierungen werden bis zum Evolutionsschritt 20000 gefunden, zwischen dem Evolutionsschritt 20000 und 40000 werden nur noch 31 weitere Modellrestrukturierungen gefunden.

Abbildung 7.11 zeigt für alle Fallstudien, wie sich die Metrikwerte relativ verändert haben. Bezugspunkt war jeweils die in den Voroptimierungen erreichte maximale Fitness. Eine positive Zahl bedeutet, dass sich die Metrik während der Optimierung verbessert hat, eine negative Zahl bedeutet, dass der Wert schlechter geworden ist. Für

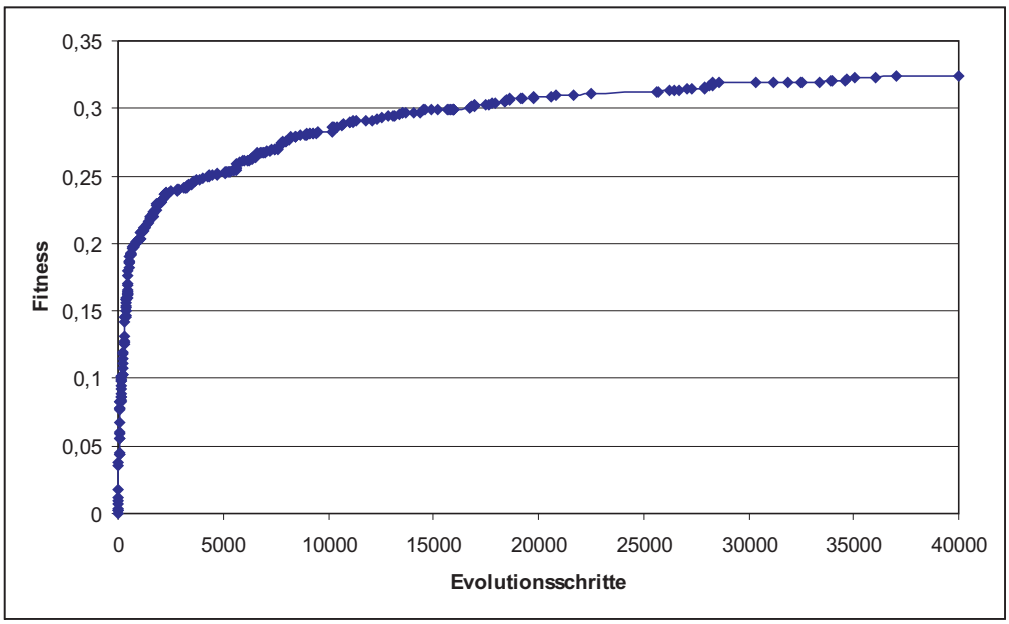


Abbildung 7.10.: Verlauf der Zielfunktion für *Rheingold*

JHotDraw bedeutet zum Beispiel der Wert 0.64325 in der Zeile TS-Komplexität (NOC), dass während der Optimierung mit einer aus allen Metriken bestehenden Zielfunktion 64.32 Prozent des Bezugspunktes dieser Metrik erreicht wurden.

54 von 62 Metrikwerten konnten entweder verbessert werden oder sind zumindest gleich geblieben. Nur 8 von 62 Metrikwerten verschlechterten sich. Der schlechte Wert der Metrik TS-Kopplung für *JGraph* lässt sich folgendermaßen erklären: Der Wert von TS-Kopplung beträgt ursprünglich 0.67 und kommt dadurch zustande, dass ca. 90 % aller Klassen in einem Teilsystem liegen und somit fast alle Beziehungen innerhalb dieses Teilsystems verlaufen. Aufgrund der unausgeglichenen Verteilung der Klassen auf die Teilsysteme ist im Gegenzug der Wert der Teilsystemkomplexität mit 0.12 sehr niedrig.

Unser Verfahren hat die Teilsystemstruktur so verändert, dass mehrere Teilsysteme ausgeglichener Größe und guter Kohäsion entstanden sind. Als Nachteil verlaufen in der neuen Struktur allerdings mehr Beziehungen zwischen den Teilsystemen, und somit verschlechtert sich der Wert der Metrik TS-Kopplung. Eine Verschlechterung bezüglich einer Metrik kann also durchaus gewünscht sein, wenn sich dadurch andere wichtigere Metriken verbessern.

Einen weiteren interessanten Effekt beobachten wir bei den Werten für K-Kopplung (ICP) und K-Kopplung (RFC) bei der Fallstudie *JGraph*. K-Kopplung (RFC) verschlechtert sich, während sich K-Kopplung (ICP) verbessert. Wir hätten erwartet, dass sich Metriken für ein Entwurfsprinzip in die gleiche Richtung verändern. Aus den Ergebnissen geht jedoch hervor, dass sich Metriken eines Entwurfsprinzips durchaus in verschiedene Richtungen entwickeln können. Wir können uns deshalb nicht auf eine Metrik pro Entwurfsprinzip beschränken. Diesen Effekt können wir auch für die Metriken K-Kohäsion (LCOM) und K-Kohäsion (TCC) bei der Fallstudie *JGraph* beobachten. Diese Metriken

	JGraph	JHotDraw	JEdit	JFreeChart	Rheingold
TS-Komplexität (NOC)	1	0.64325	0.21734	0.84103	0.41142
TS-Kopplung	-0.9718	0.26814	0.14223	-0.07580	0.12155
TS-Kohäsion	0.26157	0.50008	0.35526	0.16378	0.11797
TS-Zyklen	0	0.64438	0.30618	0.53748	0.49345
TS-Flaschenhälse	0.03062	-0.00866	-0.00757	-0.01049	0.00244
TS-Stabilität	0.16667	0.657	0.39456	0.81004	0.74797
K-Komplexität (WMC)	0.05183	0.29189	0.67654	0.01238	-
K-Komplexität (NOM)	0.02739	0.48928	0.73945	-0.01602	-
K-Kopplung (ICP)	0.00618	0.15508	0.50427	0.52972	-
K-Kopplung (RFC)	-0.02501	0.13528	0.59997	0.40360	-
K-Kohäsion (LCOM)	-0.01499	0.06439	0.50704	0.24899	-
K-Kohäsion (TCC)	0.10345	0.46893	1.25447	0.68589	-
K-Stabilität	0.44467	0.31045	0.85857	0.93772	-
K-Schnittstellen-umgehungen	0	0.6	0	0	-

Abbildung 7.11.: Relative Veränderung der Metrikwerte pro Fallstudie

entwickeln sich jedoch nicht immer in gegensätzliche Richtungen, wie die Ergebnisse für die Fallstudien *JHotDraw*, *JEdit* und *JFreeChart* zeigen. Für diese Fallstudien konnten alle K-Kopplungs- und K-Kohäsionsmetriken verbessert werden.

Die Flaschenhalsmetrik verschlechtert sich bei den Fallstudien *JHotDraw*, *JFreeChart* und *JEdit*. Die Verschlechterung liegt jedoch bei nur ca. 1 Prozent und ist somit zu vernachlässigen.

Für die Fallstudie *JFreeChart* verschlechterten sich zudem die Werte für die Metriken TS-Kopplung und K-Komplexität(NOM). Allerdings ist die Verschlechterung für K-Komplexität(NOM) mit 1 Prozent sehr gering und die ebenfalls geringe Verschlechterung der Metrik TS-Kopplung ermöglichte insbesondere eine Verbesserung der Metrik TS-Komplexität (NOC).

Die Metrik K-Kohäsion (TCC) zeigt für die Fallstudie *JEdit* mit einer Verbesserung von 1.25, dass in seltenen Fällen die Werte der Voroptimierung noch übertroffen werden können.

Für die Fallstudie *Rheingold* konnte unser Verfahren nur die Zuordnung von Klassen zu Teilsystemen verändern. Es gelang unserem Verfahren, alle Metriken gleichermaßen zu berücksichtigen und zu verbessern, TS-Flaschenhälse konnte allerdings nur minimal verbessert werden.

Abschließend stellen wir fest, dass es unserem Verfahren im besten Fall gelungen ist, für eine Fallstudie 13 von 14 Metriken und im schlechtesten Fall 11 von 14 Metriken zu verbessern oder unverändert zu lassen. Insgesamt konnte unser Verfahren die Qualität der Systemstruktur aller Fallstudien verbessern.

Bevor wir nun die berechneten Modellrestrukturierungen im nächsten Abschnitt detailliert betrachten, geben wir in Abbildung 7.12 eine quantitative Übersicht über die An-

zahl der berechneten verschiedenen Modellrestrukturierungen. Die Zeile *Summe* enthält die Länge der Liste von Modellrestrukturierungen pro Fallstudie und somit die Länge des jeweiligen Genotyps.

	JGraph	JHotDraw	JEdit	JFreeChart	Rheingold
Verschiebe Methode	18	32	745	517	-
Verschiebe Methode in Oberklasse	2	4	7	93	-
Verschiebe Methode in Unterklasse	0	5	1	40	-
Verschiebe Attribut in Oberklasse	10	5	81	26	-
Verschiebe Attribut in Unterklasse	0	0	0	0	-
Extrahiere Klasse	0	7	99	86	-
Integriere Klasse	0	0	9	11	-
Verschiebe Klasse	20	49	61	219	203
Integriere Teilsystem	4	12	31	29	34
Extrahiere Teilsystem	5	25	15	116	49
Summe	59	140	1049	1137	286

Abbildung 7.12.: Quantitative Übersicht über die Modellrestrukturierungen

Wir beobachten, dass keine Modellrestrukturierung *Verschiebe Attribut in Unterklasse* verwendet wurde. Diese Modellrestrukturierung scheint bei allen Fallstudien nicht anwendbar zu sein, was bedeutet, dass alle Attribute nur in den Unterbäumen der Vererbungshierarchie vorhanden sind, in denen sie auch tatsächlich benötigt werden. *Verschiebe Attribut in Unterklasse* ist dennoch eine wichtige Modellrestrukturierung, da *Verschiebe Attribut in Oberklasse* sonst nicht rückgängig gemacht werden könnte. Da unser evolutionärer Algorithmus auch Verschlechterungen tolerieren kann, kommt es sehr oft vor, dass ungünstige Modellrestrukturierungen wieder rückgängig gemacht werden.

Die anderen Modellrestrukturierungen werden jedoch angewandt. Insbesondere die Modellrestrukturierungen *Verschiebe Klasse* und *Verschiebe Methode* werden für die von uns bearbeiteten Fallstudien oft vorgeschlagen.

7.3.5. F4: Sind die vorgeschlagenen Modellrestrukturierungen für einen Softwareingenieur akzeptabel?

In diesem Abschnitt untersuchen wir für die Fallstudien *JGraph* und *JHotDraw* stichprobenhaft die vorgeschlagenen Modellrestrukturierungen, um zu zeigen, dass diese auch von Bedeutung für einen Softwareingenieur sind.

7.3.5.1. JHotDraw

Obwohl *JHotDraw* allgemein als vorbildlich strukturiertes System gilt, hat unser Verfahren einige Modellrestrukturierungen zur Strukturverbesserung vorgeschlagen: 53 Modellrestrukturierungen verändern die Klassenstruktur, 86 Modellrestrukturierungen verändern die Teilsystemstruktur.

Im Folgenden zeigen wir Beispiele für Modellrestrukturierungen, die eine Methode verschieben, ein Attribut verschieben, und eine Klasse aufspalten.

Unser Verfahren schlug vor, die Methode `writeColor()` der Klasse `FigureAttributes` in die Klasse `StorableOutput` und die Methode `readColor()` der Klasse `FigureAttributes` in die Klasse `StorableInput` zu verschieben. Der Quelltext der beiden Methoden ist in Abbildung 7.13 dargestellt. Wir stellen fest, dass die Methoden nur Methoden der Zielklassen und Methoden von Bibliotheksklassen, aber keine Methoden oder Attribute der sie umgebenden Klassen verwenden. Somit handelt es sich unserer Meinung nach um zwei deplatzierte Methoden (siehe Strukturproblem Feature Envy in [Fow99]), die korrekterweise verschoben wurden. Selbst die Entwickler von `JHotDraw` stimmen unserer Einschätzung indirekt zu: In den Zielklassen sind schon nahezu identische Methoden zum Speichern und Laden von Farben vorhanden. Der einzige Unterschied ist das Fehlen eines Namens für eine Farbe. Somit kann sogar in einer Zusatzrestrukturierung jeweils eine der beiden Methoden entfernt werden. Insgesamt begünstigen diese Restrukturierungen Änderungen, die die Persistenz von Attributen betreffen. Beispielsweise können Formatänderungen leichter vorgenommen werden, da der die Persistenz betreffende Quelltext auf zwei Klassen beschränkt ist.

```
public static void writeColor(StorableOutput dw
                              , String colorName
                              , Color color) {
   if (color != null) {
        dw.writeString(colorName);
        dw.writeInt(color.getRed());
        dw.writeInt(color.getGreen());
        dw.writeInt(color.getBlue());
    }
}

public static Color readColor(StorableInput dr) throws IOException {
    return new Color(dr.readInt(), dr.readInt(), dr.readInt());
}
```

Abbildung 7.13.: Quelltext der Methoden writeColor() and readColor()

Unser Verfahren hat weiterhin vorgeschlagen, das Attribut `fLocator` vom Typ `Locator` aus der Klasse `PolygonHandle` in die Klasse `AbstractHandle` zu verschieben. Die Ausgangssituation ist im oberen Drittel von Abbildung 7.14 dargestellt.

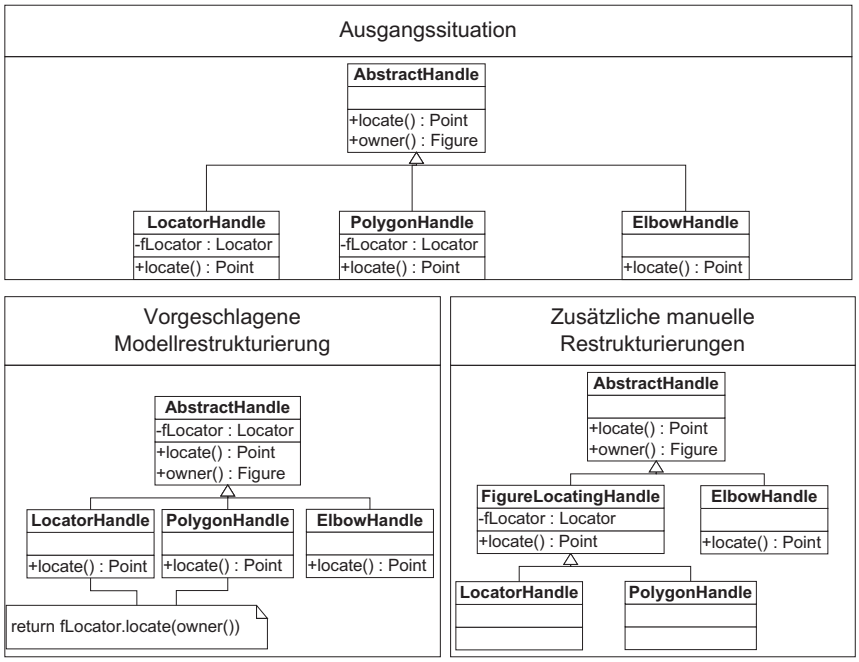


Abbildung 7.14.: Manuelle Inspektion einer Modellrestrukturierung

Aus der Abbildung geht ebenfalls hervor, dass die Klasse `LocatorHandle` ein identisches Attribut gleichen Typs und gleichen Namens besitzt. Unser Verfahren schlägt deshalb korrekterweise vor, das in die Oberklasse verschobene Attribut auch in dieser Unterklasse zu verwenden. Der einzige Nachteil besteht in der Tatsache, dass eine dritte Unterklasse (`ElbowHandle`) dieses neue Attribut der Oberklasse nicht benötigt. Jedoch ist diese Restrukturierungen vorteilhaft, da ein redundantes Attribut entfernt werden kann.

Eine genauere Analyse des Quelltextes ergab jedoch, dass die Klassen `LocatorHandle` und `PolygonHandle` die Methode `locate()` identisch implementieren. Somit könnte ein Softwareingenieur die von unserem Verfahren vorgeschlagene Modellrestrukturierung folgendermaßen ergänzen: Er erzeugt eine neue Klasse `FigureLocatingHandle` und verschiebt in diese Klasse das Attribut `fLocator` und die Methode `locate()`. Auf diese Art und Weise gibt es keinerlei Duplikation von Methoden und Attributen mehr. Attribute und Methoden sind nur in den Teilbäumen der Vererbungshierarchie sichtbar, in denen sie auch wirklich benötigt werden. Unser Verfahren konnte diesen Vorschlag in dieser Form nicht machen, weil wir nicht feststellen können, ob zwei Methoden identisch implementiert sind. Unserem Verfahren ist es aber gelungen, mehrfach vorhandene Attribute zu entfernen. (siehe Strukturproblem *Unvollständige Vererbung* [SSM06]). Es ist durchaus denkbar, unser Verfahren so zu erweitern, dass auch duplizierte Methoden berücksichtigt werden können.

Als letztes stellen wir eine Modellrestrukturierung vor, die die Klasse `UndoManager` aufteilt. Die Methoden dieser Klasse sind in Abbildung 7.15 zu sehen. Die Aufgabe der Klasse ist es, Kommandos sowohl rückgängig zu machen, als auch erneut auszuführen,

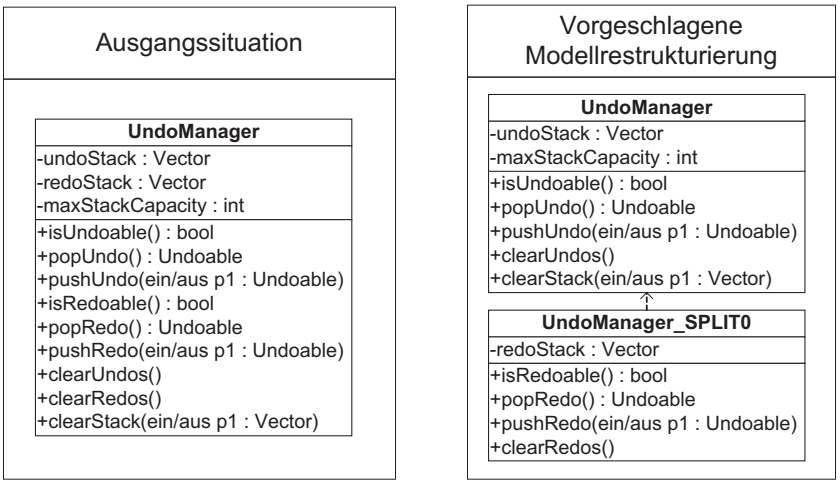


Abbildung 7.15.: Modellrestrukturierung *Extrahiere Klasse*

wenn sie bereits rückgängig gemacht wurden. Diese Klasse ist also nicht wie der Name suggeriert ausschließlich für ein, sondern für zwei Konzepte zuständig.

EVO schlägt vor, aus dieser Klasse eine neue Klasse zu extrahieren. Die resultierenden Klassen sind ebenfalls in Abbildung 7.15 dargestellt. Von der Klasse `UndoManager` wurden alle Methoden und Attribute abgespalten, die sich um das Redo-Konzept kümmern. Die neue Klasse verwendet nur das Attribut `maxStackSize` und die Hilfsmethode `clearStack`, und hängt deshalb von der ursprünglichen Klasse ab. Die Klasse `UndoManager` hängt von der neuen Klasse nicht ab. Mit dieser Modellrestrukturierung hat unser Verfahren eine Instanz des Strukturproblems *Große Klasse* (siehe [Fow99]) beseitigt. Nach dieser Restrukturierung sind die beiden Konzepte in zwei Klassen aufgeteilt. Eine Änderung an einem der beiden Konzepte ist nun leichter möglich, da der Quelltext für die beiden Konzepte nicht mehr in einer Klasse vermischt ist.

Unser Verfahren hat die Teilsystemstruktur mit 69 Modellrestrukturierungen verändert. In der Ausgangssituation hat *JHotDraw* 14 Teilsysteme (das `Framework`-Teilsystem betrachten wir wie in Abschnitt 7.3.1 beschrieben nicht). Wenn wir die Metrikwerte betrachten, fällt uns auf, dass der Wert der Metrik TS-Komplexität mit 0.07 sehr klein ist. Der schlechte Metrikwert kommt dadurch zustande, dass vier Teilsysteme 165 der 187 Klassen enthalten (`util`: 41, `standard`: 65, `contrib` 25, und `figures` 34) und die Größe dieser Teilsysteme außerhalb des von uns tolerierten Bereichs liegt (wir betrachten die Komplexität eines Teilsystems als optimal, wenn es zwischen 5 und 15 Klassen enthält). Die anderen 10 Teilsysteme sind dementsprechend sehr klein. Die Teilsysteme sind zudem nicht sehr kohäsiv (TS-Kohäsion = 0.08), TS-Kopplung ist mit 0.2 ebenfalls niedrig.

Unserem Verfahren gelingt es Modellrestrukturierungen vorzuschlagen, die bis auf die Flaschenhalsmetrik alle Werte der Teilsystemmetriken verbessern. Allerdings verschlechtert sich die Flaschenhalsmetrik nur geringfügig um weniger als 1 Prozent. Grundsätzlich führen die Modellrestrukturierungen zu einer ausgeglicheneren Verteilung der Klassen auf die Teilsysteme.

Die entstandene Einteilung in Teilsysteme betrachten wir exemplarisch anhand des Teilsystems `util` genauer. Abbildung 7.16 zeigt die neuen Teilsysteme, die nach den Modellrestrukturierungen Klassen aus `util` enthalten. Anhand der Visualisierung erkennen wir, dass die Teilsysteme geschichtet sind, und es keine Zyklen gibt. Unserem Verfahren ist es also gelungen, das große Teilsystem `util` in kleinere Teilsysteme zu zerlegen, und dabei die Zielfunktion zu beachten.

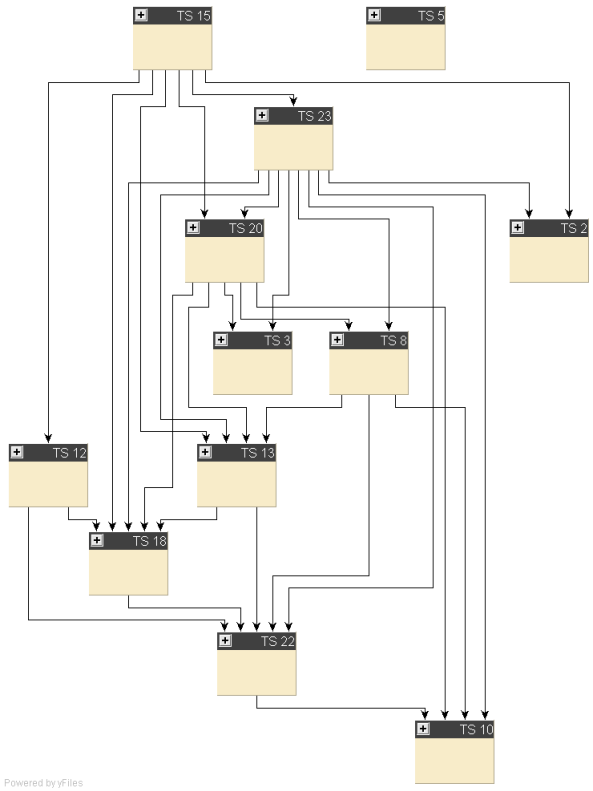


Abbildung 7.16.: Aufteilung des Teilsystems Util in mehrere Teilsysteme

Im folgenden betrachten wir drei dieser Teilsysteme genauer. In einem Teilsystem sind die folgenden Klassen enthalten:

1. `util.SerializationStorageFormat`
2. `util.StandardStorageFormat`
3. `util.StandardVersionControlStrategy`
4. `util.StorageFormat`
5. `util.StorageFormatManager`

Wie sich bereits anhand der Namen der Klassen erahnen lässt, legen diese Klassen das Speicherformat einer Zeichnung fest und bilden somit eine sinnvolle Einheit. Die

Klasse `StandardVersionControlStrategy` erscheint auf den ersten Blick nicht zu den anderen Klassen zu passen. Allerdings wird diese Klasse nur von zwei anderen Klassen verwendet: `DrawApplet` und `DrawApplication`. Letztere verwendet sie zusammen mit den Klassen `StandardStorageFormat` und `StorageFormatManager`, so dass die in diesem Teilsystem vorhandenen Klassen oft gemeinsam verwendet werden. Soll in Zukunft ein neues Speicherformat unterstützt werden, so können die dazu nötigen Änderungen lokal in diesem Teilsystem erfolgen. Das Teilsystem enthält keine Klassen, die für andere Konzepte zuständig sind und somit einen Entwickler verwirren könnten.

In einem weiteren Teilsystem sind die Klassen `util.GraphLayout` und `util.GraphLayoutNode` zusammengefasst. Dieses Teilsystem erscheint auf den ersten Blick zu klein. Allerdings verwenden diese Klassen keine anderen Klassen aus dem ursprünglichen Teilsystem `util`, so dass sie zu keiner dieser anderen Klassen passen. Sie selbst verwenden sich gegenseitig, werden aber von keiner Klasse aus *JHotDraw* verwendet. Es ist also sinnvoll, diese Klassen getrennt vom Rest des Systems zu halten, da sie zum Verständnis des Systems und für typische Änderungen nicht benötigt werden. Erst wenn in Zukunft Layoutalgorithmen unterstützt werden sollen, kann die Arbeit an diesem Teilsystem fortgesetzt werden.

Als letztes betrachten wir das Teilsystem, das aus folgenden Klassen besteht:

1. `util.RedoCommand`
2. `util.UndoCommand`
3. `util.UndoManager`
4. `util.UndoManager_SPLIT0`
5. `util.UndoManagerRedoActivity`
6. `util.UndoAble`
7. `util.UndoableAdapter`
8. `standard.TextHolder`
9. `standard.StandardFigureSelection`

Dieses Teilsystem verwaltet ausgeführte Befehle und ermöglicht es einem Nutzer, diese rückgängig zu machen oder erneut auszuführen. Anhand der Klassen `TextHolder` und `ConnectedTextTool` erkennen wir, dass die gefundene Aufteilung nicht immer unserer Erwartung entspricht. Diese beiden Klassen verwenden die anderen Klassen des Teilsystems nicht, und werden auch nicht von diesen verwendet. Demnach würde ein Softwareingenieur es ablehnen, diese Klassen zu verschieben und sie an ihrer ursprünglichen Position belassen. Unser Verfahren hat diese Klassen in dieses Teilsystem verschoben, um den Wert der Metrik TS-Komplexität(NOC) zu verbessern. Eine geringere Gewichtung dieser Metrik würde dies verhindern.

7.3.5.2. JGraph

Auch für die kleinste Fallstudie *JGraph* konnte unser Verfahren Modellrestrukturierungen zur Verbesserung der Struktur berechnen.

Die vorhandene Teilsystemstruktur lässt sich folgendermaßen charakterisieren. Es gibt ein paar wenige Zyklen, die Kohäsion ist mit 0.06 sehr schlecht, die Kopplung ist gut (0.67) und der Wert der Komplexitätsmetrik ist mit 0.12 ebenfalls niedrig.

In Abbildung 7.17 sind die Teilsysteme und ihre Abhängigkeiten zu sehen. Gut zu erkennen sind insbesondere die vier direkten Zyklen zwischen den Teilsystemen. Auf der rechten Seite der Abbildung sind die Teilsysteme als Quadrate und die Klassen des Systems als Kreise dargestellt. Zwischen Quadraten und Kreisen existiert eine Kante, falls das Teilsystem diese Klasse enthält. Wir erkennen in diesem Teil der Abbildung den Grund für den schlechten Wert für die Metrik TS-Komplexität: Fast alle Klassen sind in einem Teilsystem enthalten.

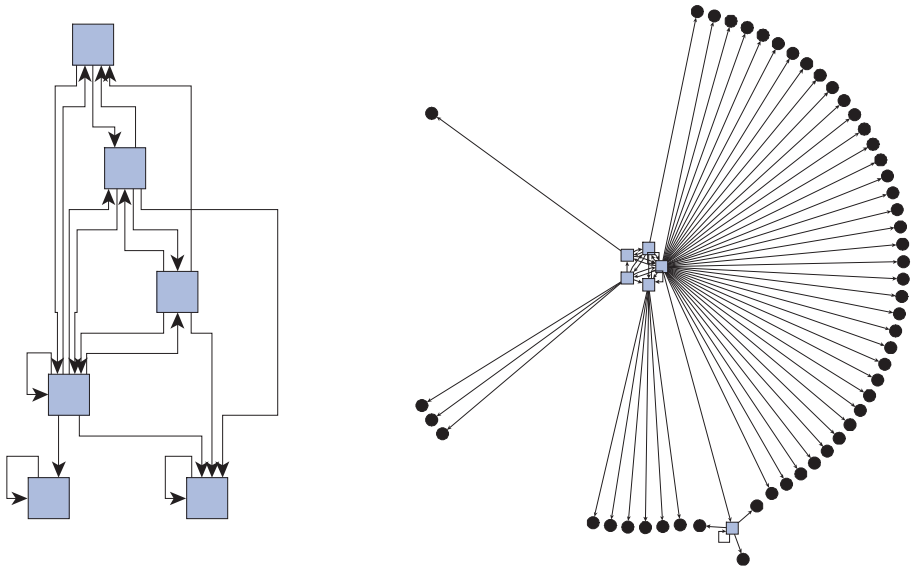


Abbildung 7.17.: Ursprüngliche Teilsystemstruktur

Die Modellrestrukturierungen führen zu einer Struktur, die in Abbildung 7.18 zu sehen ist.

Wir sehen, dass die Klassen gleichmäßig auf die Teilsysteme verteilt sind, die Anzahl der Zyklen jedoch nicht zugenommen hat. Der Wert der Metrik TS-Kopplung hat sich erwartungsgemäß verschlechtert, da die Klassen jetzt auf mehr als ein Teilsystem verteilt sind, und somit mehr Abhängigkeiten als vorher zwischen den Teilsystemen verlaufen.

Als Beispiel stellen wir ein entstandenes Teilsystem vor, das aus folgenden Klassen besteht:

1. `event.GraphModelListener`
2. `event.GraphLayoutCacheEvent`

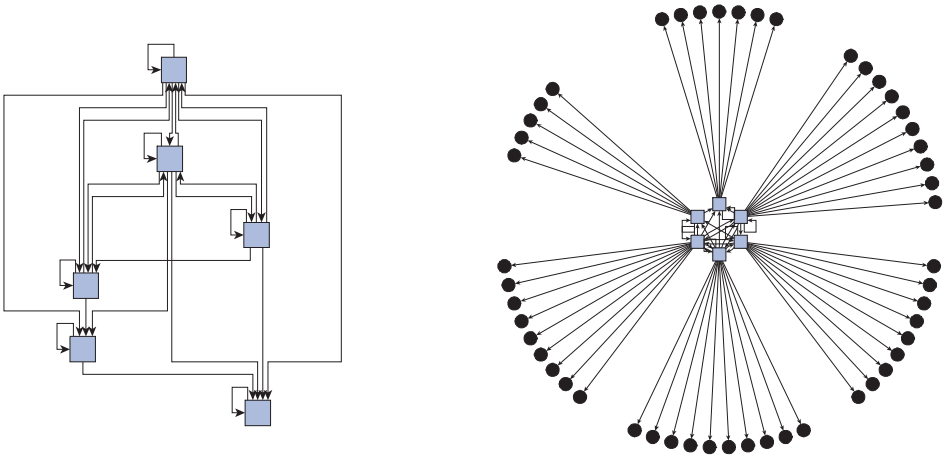


Abbildung 7.18.: Neue Teilsystemstruktur

3. `event.GraphLayoutCacheListener`
4. `event.GraphModelEvent`
5. `basic.BasicGraphTransferable`
6. `graph.GraphUndoManager`
7. `util.Spline`

Die Klasse `GraphModelListener` verwendet `GraphLayoutCacheEvent` und sonst keine anderen Klassen des Systems. Vier der Klassen waren schon in der Ausgangsstruktur in demselben Teilsystem enthalten und gehören logisch zusammen, da sie Ereignisse modellieren. Die Klasse `GraphUndoManager` wird von keiner anderen Klasse direkt verwendet. Sie selbst verwendet außer Bibliotheksklassen nur die Klasse `GraphLayoutCacheEvent`. Mit Hilfe dieser Klasse registriert `GraphUndoManager` Änderungen am Graph, und ist deshalb sinnvollerweise Bestandteil dieses Teilsystems.

`BasicGraphTransferable` ist eine Hilfsklasse, die von `GraphTransferable` verwendet wird. `util.Spline` ist ebenfalls eine Hilfsklasse. Sie wird nur von `util.Spline2D` verwendet, verwendet selbst aber keine Klassen aus dem System. Insgesamt wurden in diesem Teilsystem Hilfsklassen zusammengruppiert. Die Klassen modellieren nicht genau ein Konzept, wurden allerdings sinnvollerweise zusammengelegt, da sonst Teilsysteme entstanden wären, die zu klein sind. Eventuell würde ein Softwareingenieur sich dafür entscheiden, `util.Spline` nicht von `util.Spline2D` zu trennen.

Unser Verfahren hat zusätzlich eine Reihe von Modellrestrukturierungen vorgeschlagen, die Attribute und Methoden verschieben.

Beispielsweise wird die Methode `getPerimeterPoint()` von der Klasse `VertexRenderer` in die Klasse `VertexView` verschoben. Diese Modellrestrukturierung erscheint sinnvoll, da die Methode nur Methoden von `VertexView` verwendet, wie aus dem Quelltextausschnitt in 7.19 hervorgeht.

```
public Point2D getPerimeterPoint(VertexView view, Point2D source, Point2D p) {
    Rectangle2D bounds = view.getBounds();
    double x = bounds.getX();
    double y = bounds.getY();
    double width = bounds.getWidth();
    double height = bounds.getHeight();
    double xCenter = x + width / 2;
    double yCenter = y + height / 2;
    double dx = p.getX() - xCenter; // Compute Angle
    double dy = p.getY() - yCenter;
    double alpha = Math.atan2(dy, dx);
    double xout = 0, yout = 0;
    double pi = Math.PI;
    double pi2 = Math.PI / 2.0;
    double beta = pi2 - alpha;
    double t = Math.atan2(height, width);
    if (alpha < -pi + t || alpha > pi - t) { // Left edge
        xout = x;
        yout = yCenter - width * Math.tan(alpha) / 2;
    } else if (alpha < -t) { // Top Edge
        yout = y;
        xout = xCenter - height * Math.tan(beta) / 2;
    } else if (alpha < t) { // Right Edge
        xout = x + width;
        yout = yCenter + width * Math.tan(alpha) / 2;
    } else { // Bottom Edge
        yout = y + height;
        xout = xCenter + height * Math.tan(beta) / 2;
    }
    return new Point2D.Double(xout, yout);
}
```

Abbildung 7.19.: Quelltextbeispiel aus JGraph

Weiterhin wurden einige Modellrestrukturierungen vorgeschlagen, die Attribute in Oberklassen verschieben. Diese Verschiebungen erscheinen oft nicht sinnvoll, da das Attribut sonst in Unterklassen zur Verfügung steht, die es nicht benötigen. Diese Modellrestrukturierungen kommen durch die Komplexitätsmetriken zu Stande, die eine gleichmäßige Verteilung der Attribute auf die Klassen besser bewerten. Um dies zu verhindern, könnte die Zielfunktion um weitere Metriken ergänzt werden, die beispielsweise messen, in wie vielen Unterklassen ein Attribut einer Oberklasse verwendet wird.

7.3.5.3. Rheingold

Für die Fallstudie *Rheingold* konnten alle Teilsystemmetriken verbessert werden. Allgemein sorgte unser Verfahren für eine ausgeglichenere Verteilung der Klassen auf die Teilsysteme. Abbildung 7.20 zeigt noch einmal vereinfacht die Struktur von *Rheingold*. Gut zu erkennen ist die zyklische Abhängigkeit zwischen den Teilsystemen `Kern` und `XML`.

Die beiden Gottklassen `XMLLeser` und `XMLSchreiber` haben wir bereits vor diesem Experiment (siehe Abschnitt 3.1) aufgeteilt, da unser Verfahren aufgrund des unvollständigen Faktenextraktors für in *C++* geschriebene Systeme keine Restrukturierungen von Klassen unterstützt. Zwei der entstandenen Klassen (`XMLeser` und `XMLSchreiber`) wurden weiter unterteilt. Für die verschiedenen Funktionalitäten von *Rheingold* wurde jeweils eine eigene Klasse extrahiert, die für das Lesen bzw. das Schreiben zuständig ist. Aus `XMLLeser` entstanden die fünf Klassen `XMLAnsichtsLeser`, `XMLDatenmodellLeser`, `XMLKnotenleser`, `XMLSchiffsLeser` und `XMLECDISLeser`. Aus `XMLSchreiber` wurden die fünf

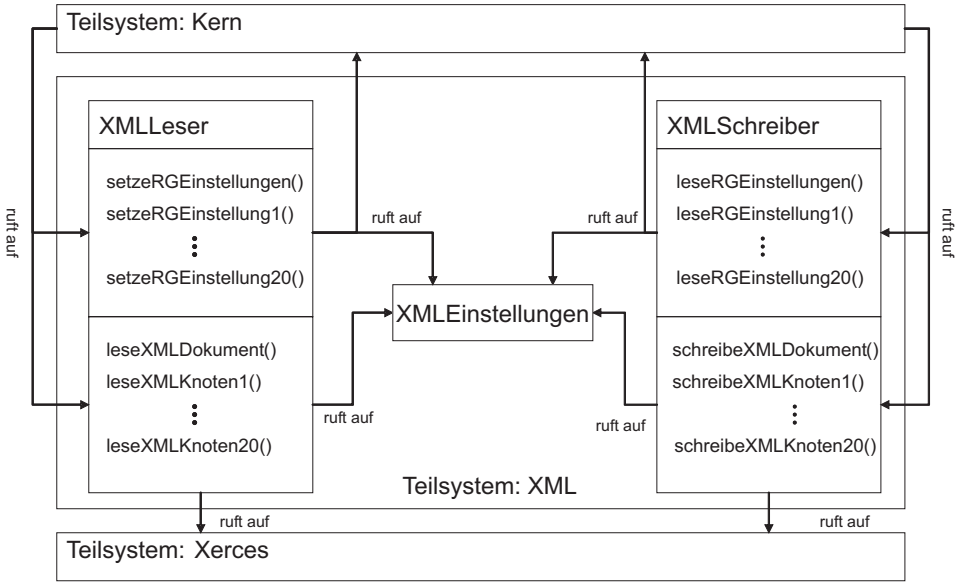


Abbildung 7.20.: Referenzfallstudie

Klassen `XMLAnsichtsSchreiber`, `XMLDatenmodellSchreiber`, `XMLKnotenSchreiber`, `XMLSchiffsSchreiber` und `XMLECDISSchreiber` extrahiert.

Zum Startzeitpunkt unseres Verfahrens waren also 13 Klassen im Teilsystem `XML` enthalten. Unser Verfahren hat die Klassen `RGSetzer` und `RGLeser` aus dem Teilsystem `XML` in ein Teilsystem verschoben, in dem sich weitere Klassen zum Lesen und Setzen von Einstellungen in anderen Formaten befinden.

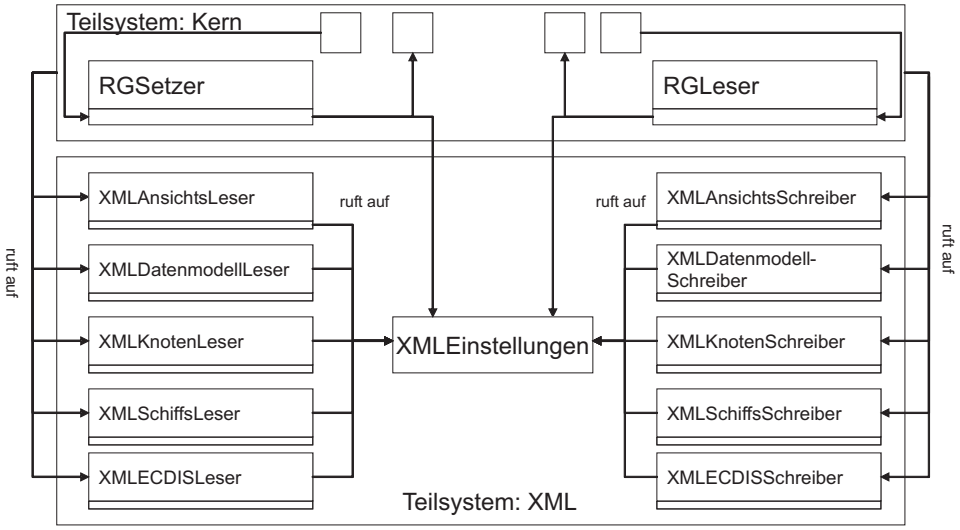


Abbildung 7.21.: Neue Struktur von Rheingold

Diese Verschiebungen haben die zyklische Abhängigkeit entfernt. Die entstandene Struktur ist vereinfacht in Abbildung 7.21 dargestellt.

7.3.6. F5: Kann unser Verfahren eine Referenzstruktur wieder herstellen, wenn die Systemstruktur von uns zuvor manuell verschlechtert wurde?

In einem weiteren Experiment haben wir untersucht, ob unser Verfahren eine Referenzstruktur wieder herstellen kann [SSB06]. Als Fallstudie haben wir erneut JHotDraw verwendet, da die Struktur dieses Systems - wie bereits in Abschnitt 7.2 erwähnt - allgemein als gut erachtet wird, und somit als Referenzstruktur verwendet werden kann. Um die Referenzstruktur zu zerstören, haben wir 10 Methoden manuell ausgesucht und verschoben.

Die Namen der von uns verschobenen Methoden sind in der linken Spalte der Tabelle in Abbildung 7.22 zu sehen. Unser Verfahren haben wir mit den üblichen Parametern (Populationsgröße 10, 10000 Generationen, Rekombinationswahrscheinlichkeit 0.02) insgesamt 10 Mal auf die so veränderte Fallstudie angewandt. Als Zielfunktion haben wir uns auf die Metriken beschränkt, die die Klassenstruktur bewerten, da wir in diesem kleinen Experiment ausschließlich die Modellrestrukturierung *Methode verschieben* benötigen.

Die rechte Spalte von Abbildung 7.22 zeigt die Anzahl die Programmläufe, in denen unser Verfahren die deplatzierte Methode wieder zurück in die Ausgangsposition verschoben hat.

Methode	verschoben in Klasse	# zurück verschoben
contrib.MDIDesktopManager.resizeDesktop	MDIDesktopPane	10
util.Geom.distanceFromLine	PolygonFigure	1
util.Geom.ovalAngelToPoint	ChopEllipseConnector	10
figures.ImageFigure.drawGhost	FigureAttributes	10
contrib.PolygonFigure.findSegment	FigureAttributes	10
contrib.DiamondFigure.getPolygon	FigureAttributes	10
figures.AttributeFigure.writeObject	FigureAttributes	10
standard.CompositeFigure.bringToFront	QuadTree	10
standard.CompositeFigure.findFigure	QuadTree	10
samples.pert.PertFigure.layout	QuadTree	10

Abbildung 7.22.: Manuell deplatzierte Methoden

Insgesamt konnte unser Verfahren jede Methode mindestens einmal an die ursprüngliche Stelle zurück verschieben. 9 der 10 Methoden wurden sogar in jedem Programmlauf an die Ursprungsposition verschoben. Somit konnten wir zeigen, dass unser Verfahren eine Referenzstruktur wieder herstellen kann. Zusätzlich zu diesen Rückverschiebungen schlug unser Verfahren erneut die in Abschnitt 7.3.4 erwähnten 32 Instanzen von *Verschiebe Methode* vor.

7.4. Zusammenfassung

In diesem Kapitel konnten wir anhand unserer prototypischen Implementierung zeigen, dass unser Verfahren die von uns in Kapitel 1 geforderten Kriterien erfüllt.

Das Werkzeug hat *automatisch* eine Liste von Modellrestrukturierungen für jede Fallstudie bestimmt. Wir mussten nur vorab verschiedene Parameter wie Populationsgröße und Anzahl der Evolutionsschritte von Hand festlegen. Es ist empfehlenswert, dass ein Softwareingenieur die Modellrestrukturierungen noch einmal manuell überprüft, da diese nur so gut wie die von ihm aufgestellte Zielfunktion sein können.

Wir konnten zeigen, dass unser Verfahren *multikriteriell* ist. In den fünf Fallstudien verbesserten sich 54 von 62 Metrikwerten oder blieben gleich. Nur 8 von 62 Metrikwerten verschlechterten sich. In drei Fällen waren die Verschlechterungen nur minimal (1 Prozent oder weniger), in drei Fällen konnten andere Metriken, die dasselbe Entwurfsprinzip messen, dafür verbessert werden. Im zwei Fällen wurde die Verschlechterung der Metrik TS-Kopplung bewusst in Kauf genommen, da sich dafür die Werte aller anderen Teilsystemmetriken verbessern ließen, und die Qualität der Struktur sich insgesamt verbessert hat.

Die Modellrestrukturierungen verändern das *extern beobachtbare Verhalten eines Systems nicht*. Wir haben sie so konstruiert, dass sie die aus der Literatur [Kut02] bekannten Vorbedingungen erfüllen.

In Abschnitt 7.3.1 haben wir gezeigt, dass mit Hilfe unseres einfachen Klassifikationsverfahrens *Strukturen, die Entwurfsheuristiken bewusst verletzen*, gefunden und somit von Veränderungen ausgeschlossen werden können.

Anhand von fünf Fallstudien haben wir gezeigt, dass unser Verfahren *auf reale objektorientierte Systeme anwendbar ist*. Keine der Fallstudien ist ein Spielsystem, alle werden von einer Vielzahl von Nutzern direkt oder indirekt verwendet. Unsere aktuelle Implementierung ist momentan auf Systeme bis ca. 300000 Quelltextzeilen beschränkt, da sehr viel Hauptspeicher verbraucht wird. Ein Ausweg könnte die Parallelisierung des Programms sein. Dies würde sowohl das Speicherplatzproblem beheben, als auch die Ausführungsgeschwindigkeit verbessern. Für die Programmiersprache *Java* können wir unser Modell vollständig aufbauen, für die Programmiersprache *C++* können wir nur einen Teil des Modells aufbauen, da uns hier kein geeigneter Faktenextraktor zur Verfügung steht.

Unsere Implementierung und somit auch unser Verfahren *berücksichtigen die wesentlichen Strukturierungseinheiten*. Das Werkzeug konnte Modellrestrukturierungen berechnen, die die Qualität der Teilsystem- und der Klassenstruktur verbessern. Insbesondere werden beide Strukturierungseinheiten verzahnt betrachtet.

Kapitel 8.

Zusammenfassung und Ausblick

Das *Ziel der vorliegenden Arbeit* war ein Verfahren, das Restrukturierungen bestimmt, die die Qualität der Struktur eines gegebenen Systems verbessern. Die höhere Qualität der Struktur erleichtert Anpassungen des Systems an eine sich verändernde Umwelt.

Das von uns entwickelte Verfahren besteht aus einem evolutionären Algorithmus, der Restrukturierungen zufallsgesteuert mit Modellrestrukturierungen simuliert und mit einer Zielfunktion bewertet. Das Ergebnis des Verfahrens ist eine Liste von Modellrestrukturierungen, die auf das ursprüngliche System als Restrukturierungen übertragen werden können. Es ist garantiert, dass eine Anwendung der Restrukturierungen das von außen beobachtbare Verhalten des Systems nicht ändert.

Die vorliegende Arbeit stellt eine Verallgemeinerung des existierenden Stands der Technik dar. Zum einen berücksichtigt unser Verfahren im Vergleich zu existierenden Verfahren mehr Entwurfsheuristiken und mehr Restrukturierungen gleichzeitig. Zum anderen gelingt es uns mit der vorliegenden Arbeit zu zeigen, dass ein evolutionärer Algorithmus auch zur Verbesserung von Klassenstrukturen realer Systeme eingesetzt werden kann.

Im weiteren Verlauf dieses Kapitels fassen wir zunächst unseren Ansatz noch einmal zusammen. Anschließend bewerten wir ihn, indem wir überprüfen, ob er die in Kapitel 1 formulierten Kriterien erfüllt. Zum Abschluss der Arbeit stellen wir ausgewählte Fragestellungen vor, die an die Ergebnisse unserer Arbeit anknüpfen.

8.1. Ergebnisse der Arbeit

In der vorliegenden Arbeit haben wir einen evolutionären Algorithmus entwickelt, der Restrukturierungen zur Verbesserung der Qualität der Struktur eines Systems bestimmen kann.

Ausgangspunkt unseres Verfahren ist der existierende Quelltext eines Systems. Diesen Quelltext überführen wir mit Hilfe eines Faktenextraktors in ein abstraktes Modell. Auf diesem Modell können wir Restrukturierungen mit Modellrestrukturierungen simulieren und die Qualität der Struktur mit Hilfe einer Zielfunktion messen. Unsere Zielfunktion besteht aus Metriken, die die Entwurfsprinzipien *Kopplung und Kohäsion*, *Geheimnisprinzip und Schnittstellen* und *Komplexität* messen.

Bevor unser Verfahren mit der Evolution startet, sucht es nach Strukturen, die Entwurfsprinzipien bewusst verletzen. Das Verfahren lässt diese Strukturen unverändert, da

sie mit gängigen Metriken nicht richtig bewertet werden können.

Unser Verfahren betrachtet während der Evolution eine Population von möglichen Lösungen. Lösungen sind in unserem Fall die Modelle und die Modellrestrukturierungen, die das ursprüngliche Modell in das aktuelle Modell überführen. Aus diesen Lösungen erzeugen die Operatoren Mutation und Rekombination neue Lösungselemente. Die Zielfunktion und die Selektionsstrategie legen fest, welche Elemente beibehalten und welche verworfen werden sollen.

Das Verfahren terminiert nach einer vorgegebenen Anzahl von Evolutionsschritten. Als Ausgabe erhält ein Softwareingenieur das Modell mit dem höchsten Zielfunktionswert inklusive der Liste von Modellrestrukturierungen, die die ursprüngliche Systemstruktur in die verbesserte überführen.

Der Softwareingenieur kann die Vorschläge noch einmal überprüfen, da die Ergebnisse nur so gut wie die von ihm parametrisierte Zielfunktion sein können. Er könnte mit Modellrestrukturierungen nicht einverstanden sein, weil unser Verfahren im aktuellen Zustand nur statische Beziehungen erfasst, manche Beziehungen zwischen Strukturelementen aber erst zur Laufzeit auftreten. Weiterhin gibt es Beziehungen, die sich nicht an statischen oder dynamischen Beobachtungen festmachen lassen, wie z.B. organisatorische Zwänge, die verhindern, dass Teilsysteme zusammengelegt werden können.

Für vier in *Java* geschriebene Open-Source-Softwaresysteme und für ein in *C++* geschriebenes System konnten wir zeigen, dass unser Verfahren die Qualität der Struktur verbessern kann, und dass die gemachten Vorschläge nahezu vollständig von einem Softwareingenieur bestätigt werden konnten.

8.2. Bewertung der Arbeit

Unser Verfahren erfüllt alle Anforderungen aus Kapitel 1. Es ist *multikriteriell*, da in der Zielfunktion beliebig viele Metriken verwendet werden können. Aktuell umfasst unsere Zielfunktion bereits eine Vielzahl relevanter Metriken. Verschlechtert sich ein Metrikwert, so wirkt sich dies sofort auf den Wert der Zielfunktion und somit auf die Gesamtqualität aus. Wenn sich die Gesamtqualität der Struktur verbessert, toleriert das Verfahren allerdings auch, dass sich eine oder mehr Metriken verschlechtern. Diesen Sachverhalt konnten wir in Abschnitt 7.3.4 beobachten. Der Wert der Metrik TS-Kopplung war bereits zu Anfang außergewöhnlich hoch und verschlechterte sich zugunsten der Metriken TS-Kohäsion- und TS-Komplexität. Unser Metamodell enthält alle relevanten Strukturelemente und deren statisch beobachtbare Beziehungen untereinander, so dass zusätzliche Metriken zur Bewertung der Systemstruktur hinzugefügt werden können.

Wir konnten zeigen, dass unser Verfahren *automatisch abläuft*. Ein Softwareingenieur muss das Verfahren zwar zu Anfang parametrisieren und somit festlegen, was für ihn Qualität der Struktur bedeutet. Danach kann unser Verfahren jedoch automatisch eine Liste von durchführbaren Modellrestrukturierungen berechnen, die die Qualität der Systemstruktur verbessern.

Wir haben unsere Modellrestrukturierungen so konstruiert, dass sie auf das ursprüngliche System übertragen werden können und sich *das extern beobachtbare Verhalten des*

Systems nicht verändert. Dies können wir garantieren, da eine Modellrestrukturierung auf dem Modell alle nötigen Vorbedingungen prüft und alle nötigen Modelländerungen zur Erfüllung der Nachbedingungen durchführt, die auch für das ursprüngliche System nötig sind.

Strukturen, die Entwurfsheuristiken bewusst verletzen, werden von unserem Verfahren beibehalten. Wir identifizieren diese Strukturen in einem Vorverarbeitungsschritt mit bekannten Techniken und markieren sie in unserem Systemmodell. Unser Verfahren schließt diese Strukturen von der Optimierung aus, um keine Struktur fälschlicherweise zu zerstören.

In Kapitel 7 haben wir mit unserem Verfahren vier Open-Source-Fallstudien und ein am FZI entwickeltes System bearbeitet. Jede der Fallstudien wird von vielen Nutzern eingesetzt. Auf diese Art und Weise konnten wir zeigen, dass unser Verfahren *auf reale objektorientierte Systeme anwendbar ist.*

Wir haben in Kapitel 2 beschrieben, dass die wichtigsten Strukturierungseinheiten objektorientierter Systeme Teilsysteme und Klassen sind. Wir berücksichtigen diese Strukturierungseinheiten in unserem Metamodell, den Modellrestrukturierungen und der Zielfunktion. Somit unterstützt unser Verfahren wie von uns gefordert *alle wesentlichen Strukturierungseinheiten.*

Wie wir in Kapitel 3 zeigen konnten, erfüllt kein existierendes Verfahren alle diese Kriterien. *Verfahren zur allgemeinen Abschätzung der Auswirkung von Restrukturierungen* berücksichtigen das vorliegende System nicht. Sie können nicht auf die konkrete Struktur eingehen und sind nicht multikriteriell. Ansätze aus der Kategorie *manuelle analytische Verfahren* können nicht garantieren, dass sich die Qualität der Struktur insgesamt verbessert, da sie nur lokal arbeiten. Existierende *automatisierte Spezialverfahren* verbessern die Qualität der Struktur automatisch bezüglich einer Entwurfsheuristik. Diese Verfahren berücksichtigen jedoch nicht alle wesentlichen Strukturierungseinheiten und sind nicht multikriteriell, da sie nur eine Entwurfsheuristik betrachten. Vorhandene Ansätze zur *Analyse bzw. Verbesserung von Teilsystemzerlegungen* lassen die Klassenstruktur unberücksichtigt und betrachten bei der Berechnung der Qualität der Systemstruktur nur wenige Entwurfsheuristiken. Existierende *Suchbasierte Verfahren* sind nur in der Lage, die Qualität der Klassenstruktur zu verbessern und können nicht auf reale, objektorientierte Systeme angewendet werden.

8.3. Ausblick

Das von uns entwickelte Verfahren bietet eine gut funktionierende, prinzipielle Vorgehensweise, mit der die Qualität der Struktur eines Softwaresystems verbessert werden kann. Bei der Konstruktion des Verfahrens haben wir darauf geachtet, an einigen Stellen Erweiterungsmöglichkeiten vorzusehen.

8.3.1. Unterstützung zusätzlicher Modellrestrukturierungen

Die von uns in dieser Arbeit beschriebenen Modellrestrukturierungen lassen sich sehr leicht um weitere Modellrestrukturierungen erweitern. Mit Hilfe der Elementaroperationen kann eine Vielzahl von weiteren Modellrestrukturierungen entworfen werden, die den Aufbau von Klassen und Teilsystemen verändern. Beispielsweise könnte eine Modellrestrukturierung hinzugefügt werden, die aus einer existierenden Klasse eine Unterklasse extrahiert. Da diese Modellrestrukturierung ein Spezialfall von *Extrahiere Klasse* ist, kann sie basierend auf unseren Elementaroperationen formuliert werden.

Um zusätzliche Modellrestrukturierungen zu ermöglichen, könnte unserem Verfahren ein weiterer Vorverarbeitungsschritt hinzugefügt werden, der Methoden in möglichst kleine Methoden aufteilt. Auf diese Art und Weise könnte unser Verfahren den Aufbau von Klassen noch feingranularer verändern. Insbesondere könnten so Methoden, die zu viele Quelltextzeilen enthalten und zu viele Aktionen durchführen, besser bearbeitet werden.

In diesem Zusammenhang muss die Zielfunktion ebenfalls erweitert werden. Beispielsweise könnten wir während der Aufteilung der Methoden automatisch Quelltext-Duplikation erkennen. Mit dieser Information könnte unser Verfahren kopierte Methoden eliminieren und somit die Komplexität des Gesamtsystems verringern. Wie wir in Kapitel 7 gesehen haben, könnte dies insbesondere bei der Restrukturierung von Vererbungshierarchien nützlich sein, da in Unterklassen oft identische Methoden existieren.

8.3.2. Behandlung von Sonderfällen

Momentan behandeln wir Strukturen, die Entwurfsheuristiken bewusst verletzen, konservativ: Wir verändern sie und die in ihnen enthaltenen Strukturen nicht. Unser Verfahren könnte allerdings in solchen Fällen Modellrestrukturierungen in eingeschränkter Form durchführen. Mit nur einem Teil der Modellrestrukturierungstypen und beschränkten Parametern könnte es unserem Verfahren gelingen, auch diese Strukturen zu verbessern. Betrachten wir erneut Fassadenklassen als Beispiel: Auch Fassadenklassen können groß und unhandlich werden. Es liegt Nahe, solche Fassaden aufzuspalten. Die Struktur der entstehenden Fassaden sollte sich zum einen daran orientieren, welche Methoden der Fassade gemeinsam verwendet werden. Zum anderen sollte berücksichtigt werden, an welche Klassen die Fassade eingehende Aufrufe delegiert.

8.3.3. Art der Zielfunktion

Unser Ansatz betrachtet Strukturverbesserung als multikriterielles Optimierungsproblem. Jede Metrik der Zielfunktion repräsentiert genau ein Kriterium. Da wir eine Linearkombination verwenden, kann es unter Umständen vorkommen, dass ein Kriterium andere Kriterien stark dominiert. In diesem Fall muss ein Softwareingenieur die Gewichte anpassen und das Verfahren erneut ausführen. Eine Alternative könnte darin bestehen, eine Paretofunktion zu verwenden und dem Nutzer eine Front von veränderten Systemstrukturen zu präsentieren. Eine solche Front von Systemstrukturen enthält

alle Systemstrukturen, die bezüglich gleich vieler Kriterien am besten sind. Anhand dieser Front kann ein Benutzer nachträglich entscheiden, welche Kriterien für ihn die wichtigsten sind. Allerdings ist damit zu rechnen, dass (siehe Abschnitt 6.3.4) bei vielen Systemstrukturen aus der Front manche der Kriterien ignoriert bzw. stark verschlechtert werden.

8.3.4. Vereinfachung der Restrukturierungsliste

Die Liste von Modellrestrukturierungen kann eine existierende Struktur in die Zielstruktur überführen. Jedoch ist diese Liste nicht die einzige und vor allem nicht immer die kürzeste Liste. Während der Evolution konnten wir beobachten, dass manche der Modellrestrukturierungen nur zu Zwischenzuständen führen, die durch weitere Modellrestrukturierungen wieder verworfen werden. Wird eine Methode m beispielsweise von einer Klasse A zuerst in eine Klasse B und abschließend in eine Klasse C verschoben, so enthält die Liste der vorgeschlagenen Modellrestrukturierungen zwei Verschiebungen: *Verschiebe m von A nach B* und *Verschiebe m von B nach C*. Die endgültige Position könnte allerdings mit einer einzigen Verschiebung (*Verschiebe m von A nach C*) erreicht werden. Unser Verfahren könnte um einen Nachbearbeitungsschritt ergänzt werden, der diese Zwischenzustände eliminiert und dem Softwareingenieur die kürzeste Liste präsentiert.

8.3.5. Parallelverarbeitung

Die Laufzeit und der Speicherbedarf der Implementierung unseres Verfahrens sind für Systeme mit bis zu 300000 Quelltextzeilen noch akzeptabel. Beides könnte allerdings verbessert werden, da unser Verfahren für eine parallele Ausführung bestens geeignet ist. In unserer aktuellen Implementierung betrachten wir zwar eine Population von Modellen, wir können daraus aber keine Geschwindigkeitsvorteile erzielen, da wir die parallele Suche sequentiell auf einer Maschine mit einem Prozessor durchführen. In einem Netz von mehreren Rechnern könnte jeder Rechner für einen Teil der Population zuständig sein und diese durch Rekombination und Mutation verändern. Der Datenfluss zwischen den Rechnern könnte sich auf die Werte der Zielfunktion und die Listen von Modellrestrukturierungen beschränken. Der durch die Kommunikation nötige Zusatzaufwand würde sich folglich in Grenzen halten. Ein zentraler Rechner könnte den Austausch von Individuen zwischen den Populationen verschiedener Rechner leicht steuern.

8.3.6. Andere Anwendungsbereiche

Unser Verfahren kann außer zur Verbesserung der Qualität der Systemstruktur auch zum Verständnis eines Systems verwendet werden. Gerade bei älteren Systemen ist die Teilsystemstruktur oft nicht explizit festgelegt und die Klassenstruktur zwar vorhanden, aber nicht sehr aussagekräftig. Mit unserem Verfahren kann sozusagen die eigentliche Struktur berechnet werden, in der zusammengehörige Strukturelemente gruppiert und unzusammenhängende Strukturelemente getrennt sind. Diese Struktur kann ein

Softwareingenieur verwenden, um ein besseres Verständnis des Systems zu bekommen. Da unser Verfahren mehr Entwurfsheuristiken als andere Verfahren betrachtet, ist uns auch in diesem Bereich eine Verbesserung gegenüber dem existierenden Stand der Technik gelungen.

Das von uns entwickelte Verfahren ist für objektorientierte Softwaresysteme geeignet, die keine Reflexion benutzen und nicht verteilt ablaufen. Bei Systemen, die Reflexion einsetzen, können wir nicht garantieren, dass das von außen beobachtbare Verhalten gleich bleibt. Um Verteilungsgesichtspunkte zu berücksichtigen und damit die Kommunikationskosten zu optimieren, sind die von uns betrachteten statischen Beziehungen nicht ausreichend. Die Kommunikationskosten hängen von Benutzungsprofilen ab, die nur durch Beobachtung des Systems zur Laufzeit gewonnen werden können.

Das Prinzip unseres Verfahrens ist jedoch beispielsweise auch auf prozedurale Systeme übertragbar. Hierzu müsste die Menge der Modellrestrukturierungen und die Menge der Metriken angepasst werden, da z.B. in prozeduralen Systemen keine Klassen existieren.

Es ist auch denkbar, unser Verfahren auf modellgetrieben entwickelte Systeme zu übertragen. Auch diese Systeme besitzen eine Struktur, die im Laufe der Zeit zerfällt und Änderungen erschwert. Für Systeme dieser Bauart müssten zusätzliche Metriken und Restrukturierungen für die Modelle definiert werden. Insbesondere müsste auch das Zusammenspiel zwischen Modellen, Generatoren und manueller Implementierung berücksichtigt werden. Erste Schritte in diese Richtung werden in der Dissertation von Christoph Andriessens [And07] beschrieben. Der Autor stellt mit seiner Arbeit ein Verfahren vor, das zur Bewertung von modellgetrieben entwickelten Systemen eingesetzt werden kann.

Generell sind die Erkenntnisse unserer Arbeit immer dann von Nutzen, wenn eine existierende Struktur in eine andere Struktur überführt werden soll und sich die Qualität der Struktur mit Hilfe einer Zielfunktion messen lässt. Dank unser neuartigen Repräsentation, in der nicht die Struktur selbst, sondern die durchgeführten Veränderungen im Genotyp protokolliert werden, liefert unser Verfahren direkt die nötigen Transformationschritte.

Anhang A.

Metrikwerte im Detail

Die Metrikwerte für die Fallstudien im Detail.

Metrik	Startwert	Voroptimierungs-wert	Endwert	Prozent des Voroptimierungs-werts
TS-Komplexität (NOC)	0.1200	1.0000	1.0000	1.0000
TS-Kopplung	0.6714	0.9788	0.3727	-0.9718
TS-Kohäsion	0.0626	0.5485	0.1897	0.2616
TS-Zyklen	0.3976	1.0000	0.3976	0.0000
TS-Flaschenhälse	0.0968	0.6667	0.1142	0.0306
TS-Stabilität	0.6667	1.0000	0.7222	0.1667
K-Komplexität (WMC)	0.6066	0.6851	0.6107	0.0518
K-Komplexität (NOM)	0.6903	0.7943	0.6931	0.0274
K-Kohäsion (LCOM)	-0.9349	-0.8534	-0.9361	-0.0150
K-Kohäsion (TCC)	0.2457	0.2869	0.2500	0.1035
K-Stabilität	0.8476	0.8734	0.8591	0.4447
K-Kopplung (ICP)	-214.7200	-140.2949	-214.2600	0.0062
K-Kopplung (RFC)	-41.7800	-29.7838	-42.0800	-0.0250
K-Schnittstellenumgehungen	-2.0000	-2.0000	-2.0000	0.0000

Abbildung A.1.: Metrikwerte für JGraph

Metrik	Startwert	Voroptimierungs-wert	Endwert	Prozent des Voroptimierungs-werts
TS-Komplexität (NOC)	0.0718	0.7773	0.5256	0.6433
TS-Kopplung	0.2415	0.9949	0.4435	0.2681
TS-Kohäsion	0.0867	0.5109	0.2988	0.5001
TS-Zyklen	0.6842	1.0000	0.8877	0.6444
TS-Flaschenhälse	0.0110	0.6667	0.0053	-0.0087
TS-Stabilität	0.9024	0.9907	0.9604	0.6570
K-Komplexität (WMC)	0.7289	0.7760	0.7426	0.2919
K-Komplexität (NOM)	0.7907	0.8167	0.8034	0.4893
K-Kohäsion (LCOM)	-0.9328	-0.8518	-0.9276	0.0644
K-Kohäsion (TCC)	0.2131	0.2495	0.2302	0.4689
K-Stabilität	0.9375	0.9452	0.9399	0.3105
K-Kopplung (ICP)	-59.9330	-48.0981	-58.0977	0.1551
K-Kopplung (RFC)	-86.5598	-68.0151	-84.0512	0.1353
K-Schnittstellenumgehungen	-13.0000	-8.0000	-10.0000	0.6000

Abbildung A.2.: Metrikwerte für JHotDraw

Metrik	Startwert	Voroptimierungs-wert	Endwert	Prozent des Voroptimierungs-werts
TS-Komplexität (NOC)	0.2278	1.0000	0.3956	0.2173
TS-Kopplung	0.6547	1.0000	0.6239	0.1422
TS-Kohäsion	0.1615	0.8365	0.4097	0.3553
TS-Zyklen	0.5099	0.9471	0.6598	0.3061
TS-Flaschenhälse	0.0189	0.6667	0.0062	-0.0076
TS-Stabilität	0.7922	1.0000	0.8830	0.3946
K-Komplexität (WMC)	0.6205	0.7625	0.7166	0.6765
K-Komplexität (NOM)	0.7583	0.8556	0.8302	0.7396
K-Kohäsion (LCOM)	-0.9167	-0.7270	-0.7836	0.5070
K-Kohäsion (TCC)	0.3779	0.4563	0.4763	1.2544
K-Stabilität	0.86743	0.89023	0.88701	0.85857
K-Kopplung (ICP)	-94.4087	-80.0046	-63.4964	0.5043
K-Kopplung (RFC)	-38.0382	-24.7937	-30.0919	0.5999
K-Schnittstellenumgehungen	0.0000	0.0000	0.0000	0.0000

Abbildung A.3.: Metrikwerte für JEdit

Metrik	Startwert	Voroptimierungs-wert	Endwert	Prozent des Voroptimierungs-werts
TS-Komplexität (NOC)	0.2376	0.9403	0.8286	0.8410
TS-Kopplung	0.2529	1.0000	0.1802	-0.0758
TS-Kohäsion	0.0669	0.8346	0.2011	0.1638
TS-Zyklen	0.6950	0.9601	0.8676	0.5375
TS-Flaschenhälse	0.0085	0.6667	0.0019	-0.0105
TS-Stabilität	0.8248	0.9756	0.9466	0.8100
K-Komplexität (WMC)	0.6620	1.0000	0.6662	0.0124
K-Komplexität (NOM)	0.8462	1.0000	0.8438	-0.0160
K-Kohäsion (LCOM)	-0.9368	-0.8077	-0.8874	0.2490
K-Kohäsion (TCC)	0.2343	0.2780	0.2891	0.6859
K-Stabilität	0.9634	0.9682	0.9678	0.9377
K-Kopplung (ICP)	-165.2507	-135.7718	-149.6352	0.5297
K-Kopplung (RFC)	-139.6760	-83.3893	-116.9588	0.4036
K-Schnittstellenumgehungen	0.0000	0.0000	0.0000	0.0000

Abbildung A.4.: Metrikwerte für JFreeChart

Metrik	Startwert	Voroptimierungs-wert	Endwert	Prozent des Voroptimierungs-werts
TS-Komplexität (NOC)	0.2561	1.0000	0.5622	0.4114
TS-Kopplung	0.4143	1.0000	0.4855	0.1216
TS-Kohäsion	0.0677	0.6694	0.1387	0.1180
TS-Zyklen	0.2343	1.0000	0.6121	0.4935
TS-Flaschenhälse	0.0104	0.6667	0.0120	0.0024
TS-Stabilität	0.7373	0.9864	0.9236	0.7480

Abbildung A.5.: Metrikwerte für Rheingold

Anhang B.

Auszug aus der Liste der Modellrestrukturierungen für JHotDraw

```
::MUTATIONLOG::

10) PDM (CH.ifa.draw.applet.DrawApplet.drawing(),
        CH.ifa.draw.samples.javadraw.JavaDrawApplet)
        CH.ifa.draw.applet.DrawApplet ->
        CH.ifa.draw.samples.javadraw.JavaDrawApplet

11) MS (CH.ifa.draw.contrib, CH.ifa.draw.application ) ->
        CH.ifa.draw.contrib

14) PDM (CH.ifa.draw.standard.DecoratorFigure.peelDecoration(),
        CH.ifa.draw.samples.javadraw.AnimationDecorator)
        CH.ifa.draw.standard.DecoratorFigure ->
        CH.ifa.draw.samples.javadraw.AnimationDecorator

23) PDM (CH.ifa.draw.util.PaletteButton.select(),
        CH.ifa.draw.standard.ToolButton)  CH.ifa.draw.util.PaletteButton
        CH.ifa.draw.standard.ToolButton

27) PDM (CH.ifa.draw.standard.CompositeFigure.figureAt(int),
        CH.ifa.draw.samples.pert.PertFigure)
        CH.ifa.draw.standard.CompositeFigure ->
        CH.ifa.draw.samples.pert.PertFigure

36) PUA (CH.ifa.draw.standard.LocatorHandle.fLocator,
        CH.ifa.draw.standard.AbstractHandle)
        CH.ifa.draw.standard.LocatorHandle ->
        CH.ifa.draw.standard.AbstractHandle
```

```
40) MC (CH.ifa.draw.util.UndoableHandle)  CH.ifa.draw.util ->
        CH.ifa.draw.figures

44) MMP1 (CH.ifa.draw.figures.FigureAttributes.write(
        CH.ifa.draw.util.StorableOutput), CH.ifa.draw.util.StorableOutput)
        CH.ifa.draw.figures.FigureAttributes ->
        CH.ifa.draw.util.StorableOutput

50) MMS (CH.ifa.draw.samples.pert.PertFigure.readTasks(
        CH.ifa.draw.util.StorableInput), CH.ifa.draw.util.StorableInput)
        CH.ifa.draw.samples.pert.PertFigure ->
        CH.ifa.draw.util.StorableInput

59) PDM (CH.ifa.draw.util.PaletteButton.reset(),
        CH.ifa.draw.standard.ToolButton)  CH.ifa.draw.util.PaletteButton
        -> CH.ifa.draw.standard.ToolButton

69) MMS (CH.ifa.draw.util.ColorMap.colorIndex(java.awt.Color),
        CH.ifa.draw.util.ColorEntry)  CH.ifa.draw.util.ColorMap ->
        CH.ifa.draw.util.ColorEntry

84) SPC CH.ifa.draw.standard.CompositeFigure
        {(._addToQuadTree(CH.ifa.draw.framework.Figure));
        (._clearQuadTree());
        (._nHighestZ);
        (._nLowestZ);
        (._removeFromQuadTree(CH.ifa.draw.framework.Figure));
        (._theQuadTree);
        (.add(CH.ifa.draw.framework.Figure));
        (.addAll(CH.ifa.draw.framework.FigureEnumeration));
        (.addAll(java.util.Vector));
        (.assignFiguresToPredecessorZValue(int,int));
        (.assignFiguresToSuccessorZValue(int,int));
        (.basicMoveBy(int,int));
        (.bringToFront(CH.ifa.draw.framework.Figure));
        (.draw(java.awt.Graphics));
        (.draw(java.awt.Graphics,
            CH.ifa.draw.framework.FigureEnumeration));
        (.fFigures);
        (.figureChanged(CH.ifa.draw.framework.FigureChangeEvent));
        (.figureCount());
        (.figureInvalidated(CH.ifa.draw.framework.FigureChangeEvent));
        (.figureRemoved(CH.ifa.draw.framework.FigureChangeEvent));
```

```
        (.figureRequestRemove(CH.ifa.draw.framework.FigureChangeEvent));
        (.figureRequestUpdate(CH.ifa.draw.framework.FigureChangeEvent));
        (.figures());
        (.figures(java.awt.Rectangle));
        (.findFigureInside(int,int));
        (.findFigureInsideWithout(int,int,CH.ifa.draw.framework.Figure));
        (.getFigureFromLayer(int));
        (.getLayer(CH.ifa.draw.framework.Figure));
        (.includes(CH.ifa.draw.framework.Figure));
        (.orphan(CH.ifa.draw.framework.Figure));
        (.orphanAll(CH.ifa.draw.framework.FigureEnumeration));
        (.orphanAll(java.util.Vector));
        (.read(CH.ifa.draw.util.StorableInput));
        (.readObject(java.io.ObjectInputStream));
        (.release());
        (.remove(CH.ifa.draw.framework.Figure));
        (.removeAll());
        (.removeAll(CH.ifa.draw.framework.FigureEnumeration));
        (.removeAll(java.util.Vector));
        (.replace(CH.ifa.draw.framework.Figure,CH.ifa.draw.framework.Figure));
        (.sendToBack(CH.ifa.draw.framework.Figure));
        (.sendToLayer(CH.ifa.draw.framework.Figure,int));
        (.serialVersionUID);
        (.write(CH.ifa.draw.util.StorableOutput))},
        {(.figuresReverse());
        (.findFigure(int,int));
        (.findFigure(java.awt.Rectangle));
        (.findFigure(java.awt.Rectangle,CH.ifa.draw.framework.Figure));
        (.findFigureWithout(int,int,CH.ifa.draw.framework.Figure))}

90) MMS (CH.ifa.draw.util.ColorMap.color(int),
        CH.ifa.draw.util.ColorEntry)  CH.ifa.draw.util.ColorMap ->
        CH.ifa.draw.util.ColorEntry

92) MMP0 (CH.ifa.draw.figures.FigureAttributes.read(
        CH.ifa.draw.util.StorableInput), CH.ifa.draw.util.StorableInput)
        CH.ifa.draw.figures.FigureAttributes ->
        CH.ifa.draw.util.StorableInput

96) SS (CH.ifa.draw.util: [CH.ifa.draw.util.Undoable,
        CH.ifa.draw.util.RedoCommand, CH.ifa.draw.util.UndoManager,
        CH.ifa.draw.util.UndoCommand, CH.ifa.draw.util.UndoableTool,
        CH.ifa.draw.util.UndoableAdapter,
        CH.ifa.draw.util.UndoableCommand, CH.ifa.draw.util.Command,
```

```
        CH.ifa.draw.util.CommandMenu, CH.ifa.draw.util.CommandListener,
        CH.ifa.draw.util.CommandButton, CH.ifa.draw.util.CommandChoice,
        CH.ifa.draw.util.UndoRedoActivity, CH.ifa.draw.util.ColorMap,
        CH.ifa.draw.util.ColorEntry, CH.ifa.draw.util.Iconkit,
        CH.ifa.draw.util.UndoableTool, CH.ifa.draw.util.PaletteIcon,
        CH.ifa.draw.util.UndoRedoActivity, CH.ifa.draw.util.CommandChoice,
        CH.ifa.draw.util.Bounds, CH.ifa.draw.util.PaletteListener,
        CH.ifa.draw.util.PaletteButton, CH.ifa.draw.util.Filler,
        CH.ifa.draw.util.VersionManagement,
        CH.ifa.draw.util.StandardStorageFormat,
        CH.ifa.draw.util.StorageFormat,
        CH.ifa.draw.util.StorageFormatManager,
        CH.ifa.draw.util.StorableOutput, CH.ifa.draw.util.Storable,
        CH.ifa.draw.util.StorableInput,
        CH.ifa.draw.util.SerializationStorageFormat,
        CH.ifa.draw.util.StorableOutput, CH.ifa.draw.util.CommandMenu,
        CH.ifa.draw.util.StorageFormat, CH.ifa.draw.util.GraphLayout,
        CH.ifa.draw.util.GraphNode, CH.ifa.draw.util.PaletteButton] ) ->
        CH.NewSubsystem2811

100) MMP1Inv
    (CH.ifa.draw.util.StorableOutput.writeBoolean(boolean),
    CH.ifa.draw.figures.TextFigure)  CH.ifa.draw.util.StorableOutput
    -> CH.ifa.draw.figures.TextFigure
```

Anhang C.

Konfigurationsdatei

```
##### Konfigurations #####

# Rekombinationswahrscheinlichkeit crossoverProbability = 0.02

# Turniergröße
tournamentSize = 4

#Populationsgröße
populationSize = 10

# Anzahl Evolutionsschritte
generations = 10000

# Eingesetzte Transformationen

# Verschiebe Methode =  mmfull: mm,mmp1,mmp1inv,mma1,mma1inv

# Verschiebe Attribut innerhalb Vererbungshierarchie = pa: pua,pda
# Verschiebe Methode innerhalb Vererbungshierarchie = pm: pum,pdm
# Extrahiere/Integriere Klasse = sp: spc,jcs

# Verschiebe Klasse = MC

# Extrahiere/Integriere Teilsystem = SS,MS


transformations =MC,SS,MS,mmfull,pa,pm,sp

# Verwendete Metriken inklusive Gewichtung

# K-Komplexität (WMC) = WMCWOI
```

```
# K-Komplexität (NOM) = NOMWOI

# K-Kopplung (RFC) = RFC

# K-Kopplung (ICP) = ICP

# K-Stabilität = STB

# K-Kohäsion (TCC) = TCC

# K-Kohäsion (LCOM) = LCOMWOI

# K-Schnittstellenumgehungen = IFBP

# TS-Komplexität (TNOC) = SNOC

# TS-Kopplung = SubsystemCoupling

# TS-Zyklen = SubsystemSCCs

# TS-Flaschenhälse = SubsystemBottlenecks

# TS-Stabilität = SubsystemStability

# TS-Kohäsion = SubsystemCohesiohn


fitness.metrics = SNOC:4,SubsystemCoupling:1.5
,SubsystemCohesion:2,SubsystemBottlenecks:1
,SubsystemCohesion2:2,SubsystemSCCs:0.75,SubsystemStability:0.5
,WMCWOI:3 ,NOMWOI:3,LCOM5WOI:1,LCOM5WI:1,TCC:2,STB:2,ICH:2,ICP:3
,RFC:3,IFBP:0.5

##### algo-runner #####

# Anzahl Läufe

runs = 5

# Ausgabeverzeichnis

outputdir = res/casestudies/jhotdraw/auswertung

##### import #####
```

```
# Modell

import.srf    = res/casestudies/jhotdraw/daten/jhotdraw.srf

# Zu bearbeitende Teilsyteme

import.prefix = CH

# Datei mit Klassifikationsergebnissen

import.dpxml  =
res/casestudies/jhotdraw/daten/ManuellUeberarbeitet.xml
```

Literaturverzeichnis

[AG96] Ken Arnold und James Gosling: *The Java Programming Language*. AddisonWesley, 1996.

[And07] Christoph Andriessens: *Erkennung von Strukturproblemen in modellgetrieben entwickelten Systemen*. Doktorarbeit, Universität Karlsruhe, 2007.

[Bau05] Markus Bauer: *Strukturuntersuchung großer Softwaresysteme*. Doktorarbeit, Universität Karlsruhe, 2005.

[BBC+99] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, T. Richner, M. Rieger, C. Riva, A.-M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar und J. Weisbrod: *The FAMOOS Object-Oriented Reengineering Handbook*. Technischer Bericht, Forschungszentrum Informatik, Karlsruhe, 1999.

[BD02] J. Bansiya und C.G. Davis: *A Hierarchical Model for Object-Oriented Design Quality Assessment*. IEEE Transactions on Software Engineering, 28:4–17, January 2002.

[BDW99] Lionel C. Briand, John W. Daly und Jürgen K. Wüst: *A Unified Framework for Coupling Measurement in Object-Oriented Systems*. IEEE Trans. Softw. Eng., 25(1):91–121, 1999.

[Bec99] Kent Beck: *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[BM03] Bart Du Bois und Tom Mens: *Describing the Impact of Refactoring on Internal Program Quality*. In: *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications*, Seiten 37–48. Vrije Universiteit Brussel, 2003.

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert Keller und Frank Francone: *Genetic Programming - An Introduction*. Morgan Kaufmann Publishers, Inc., 1998.

[BT04] M. Bauer und M. Trifu: *Architecture-Aware Adaptive Clustering of OO Systems*. In: *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, Seiten 3–14. IEEE Computer Society, 2004.

[Cal88] Frank W. Calliss: *Problems with automatic restructurers.* SIGPLAN Not., 23(3):13–21, 1988.

[Ciu01] Oliver Ciupke: *Problemidentifikation in Objektorientierten Softwarestrukturen.* Doktorarbeit, Universität Karlsruhe, 2001.

[CK94] S.R. Chidamber und C.F. Kemerer: *A Metrics Suite for Object Oriented Design.* IEEE Transactions on Software-Engineering, 20(6), 1994.

[DK75] F. DeRemer und H. Kron: *Programming-in-the-Large Versus Programming-in-the-Small.* In: *Proceedings of the international conference on Reliable software.* IEEE, 1975.

[DMM99] D. Doval, S. Mancoridis und B. S. Mitchell: *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)*, 1999.

[DW02] T. Dudzikan und Jan Wlodka: *Tool-Supported Discovery and Refactoring of Structural Weaknesses.* Diplomarbeit, TU Berlin, 2002.

[eAPS00] Fernando Brito e Abreu, Gonalo Pereira und Pedro Sousa: *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).* IEEE Computer Society, 2000.

[FOW66] L. Fogel, A. Owens und M. Walsh: *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons, 1966.

[Fow99] Martin Fowler: *Refactoring - Improving the Design of Existing Code.* Addison Wesley, 1999.

[FP97] Norman E. Fenton und Shari L. Pfeeger: *Software Metrics A Rigorous and Practical Approach.* PWS Publishing, 1997.

[Gen04] Thomas Genssler: *Werkzeuggestützte Adaption Objektorientierter Programme.* Doktorarbeit, Universität Karlsruhe, 2004.

[GHJV94] E. Gamma, R. Helm, R. Johnson und J. Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[Goo96] Gerhard Goos: *Vorlesungen über Informatik - Band 2 - Objektorientiertes Programmieren und Algorithmen.* Springer, 1996.

[HHP02] M. Harman, R. Hierons und M. Proctor: *A New Representation and Crossover Operator for Search-Based Optimization of Software Modularization.* In: *Proceedings of the Genetic and Evolutionary Computation Conference*, 2002.

[HML03] Dirk Heuzeroth, Stefan Mandel und Welf Löwe: *Generating Design Pattern Detectors from Pattern Specifications.* In: *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada.* IEEE Computer Society, 2003.

[Hol92] J. Holland: *Adaptation in Natural and Artifical Systems.* MIT Press, 1992.

[IDC01] IDC: *Software Strategies and Investment Survey*, 2001.

[iee93] *IEEE Std. 1219: Standard for Software Maintenance.* IEEE Computer Society Press, 1993.

[Koz89] J.R. Koza: *Hierarchical Genetic Algorithms Operating on Populations of Computer Programs.* In: *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, Seiten 768–774. Morgan Kaufmann, 1989.

[Kut02] Volker Kuttruff: *Ein Modell für invasive Softwareadaption.* Diplomarbeit, Universität Karlsruhe, 2002.

[LB85] M.M. Lehman und L.A. Belady: *Program Evolution, - Processes of Software Change.* Academic Press Professional, Inc, 1985.

[Lie95] Karl Lieberherr: *Adaptive Object-Oriented Software - The Demeter Method - With Propagation Patterns.* PWS Publishing Company, 1995.

[LS80] B.P. Lientz und E.B. Swanson: *Software Maintenance Management.* Addison Wesley, 1980.

[Lut01] R. Lutz: *Evolving Good Hierarchical Decompositions of Complex Systems.* Journal of systems architecture, 2001.

[Mar01] Radu Marinescu: *Detecting Design Flaws via Metrics in Object-Oriented Systems.* In: *Proceedings of Technology of Object-Oriented Languages and Systems - Tools 39*, 2001.

[Mar02] Robert Martin: *Agile Software Development, Principles, Patterns, and Practices.* Prentice Hall, 2002.

[Mey88] Bertrand Meyer: *Object-Oriented-Software-Construction.* Prentice Hall, 1988.

[Mit02] B.S. Mitchell: *A Heuristic Search Approach to Solving the Software Clustering Problem.* Doktorarbeit, Drexel University, Philadelphia, 2002.

[MMR98] Spiros Mancoridis, Brian Mitchell und C. Rorres: *Using Automatic Clustering to Produce High-Level System Organisations of Source Code.* In: *Proceedings of the 6th International Workshop on Program Comprehension (IWPC).* IEEE Computer Society, 1998.

[Moo96] I. Moore: *Automatic Inheritance Hierarchy Restructuring and Method Refactoring.* In: *Proceedings of the Conference on Object Oriented Programming, Systems, Languages and Applications*, 1996.

[MSS05] Thomas Mohaupt, Olaf Seng und Frank Simon: *Bidirectional Quality Models.* In: *Proceedings of the Software and Systems Quality Conferences 2005.* SQS, apr 2005.

[Mül05] Georg Müller: *Werkzeuggestützte Quelltexttransformation zur Behebung von Strukturproblemen.* Diplomarbeit, Universität Karlsruhe, 2005.

[OC04] Mark O'Keeffe und Mel Ò Cinnèide: *Towards Automated Design Improvement Through Combinatorial Optimisation.* In: *Proceedings of the Workshop on Directions in Software Engineering Environments*, 2004.

[OC06] Mark O'Keeffe und Mel Ò Cinnèide: *Search-Based Software Maintenance.* In: *Proceedings of the Conference on Software Maintenance and Reengineering.* IEEE Computer Society, 2006.

[Opd92] William F. Opdyke: *Refactoring Object-Oriented Frameworks.* Doktorarbeit, University of Illinois at Urbana-Champaign, 1992.

[Pac05] Gert Pache: *Ein metrikbasiertes Suchverfahren zur Automatischen Strukturverbesserung.* Diplomarbeit, Universität Karlsruhe, 2005.

[Par72] D. Parnas: *On the Criteria to be Used in Decomposing Systems into Modules*, 1972.

[Rec93] I. Rechenberg: *Evolutionsstrategie '93.* Frommann Verlag, 1993.

[RRHK00] Derek Rayside, Steve Reuss, Erik Hedges und Kostas Kontogiannis: *The Effect of Call Graph Construction Algorithms for Object-Oriented Programs on Clustering.* In: *Proceedings of the 8th International Workshop on Program Comprehension (IWPC).* ACM, 2000.

[Sah00] H.A. Saharaoui: *Can Metrics Help Bridge the Gap Between the Improvement of OO Design Quality and its Automation.* In: *Proceedings of the International Conference on Software Maintenance*, 2000.

[SBBP05] Olaf Seng, Markus Bauer, Matthias Biehl und Gert Pache: *Search-based Improvement of Subsystem Decompositions.* In: *Proceedings of the Genetic and Evolutionary Computation Conference*, 2005.

[Sch95] H.-P. Schwefel: *Evolution and Optimum Seeking.* Sixth-Generation Computer Technology Series, 1995.

[Sed92] Robert Sedgewick: *Algorithms in C++.* AddisonWesley, 1992.

[Som96] Ian Sommerville: *Software Engineering.* Addison Wesley, 1996.

[SP04] Olaf Seng und Gert Pache: *Search-based Structure Improvement.* In: *Proceedings of the 1st International Workshop on Software Evolution Transformations (SET 2004).* Queens University, Kingston, Ontario, Canada, 2004.

[SS03] M. Streckenbach und G. Snelting: *Refactoring Class Hierarchies with KABA.* In: *Proceedings of the 19th conference on object-oriented programming, systems, languages and applications*, 2003.

[SSB06] Olaf Seng, Johannes Stammel und David Burkhart: *Search-based Determination of Refactorings for improving the Class Structure of object-oriented Systems.* In: *Proceedings of the eigth Annual Genetic and Evolutionary Computation Conference*, Seiten 1909–1916. ACM Press, jul 2006.

[SSM06] Frank Simon, Olaf Seng und Thomas Mohaupt: *Code-Quality-Management.* dpunkt.verlag, 2006.

[STT06] Olaf Seng, Adrian Trifu und Mircea Trifu: *QBench-Faktenextraktion*, 2006.

[TK03] L. Tahvildari und K. Kontogiannis: *A Metric Based Approach to Enhance Design Quality through Meta-Pattern Transformations.* In: *Proceedings of the seventh European Conference on Software Maintenance and Reengineering*, 2003.

[TKS05] Mircea Trifu, Volker Kuttruff und Peter Szulman: *QBench-Systemmetamodell*, 2005.

[TM03] Tom Tourwe und Tom Mens: *Identifying Refactoring Opportunities Using Logic Meta Programming.* In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, 2003.

[TM05] Adrian Trifu und Radu Marinescu: *Diagnosing Design Problems in Object-Oriented Systems.* In: *Proceedings of the Working Conference on Reverse Engineering, WCRE 2005, Pittsburgh USA.* IEEE Computer Society, November 2005.

[Tri01] Adrian Trifu: *Using Cluster Analysis in the Architecture Recovery of Object-Oriented Systems.* Diplomarbeit, Universität Karlsruhe, 2001.

[TSG04] Adrian Trifu, Olaf Seng und Thomas Genssler: *Automated Design Flaw Correction in Object-Oriented Systems.* In: *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR 2004)*, Seiten 174–183. IEEE Computer Society, März 2004.

[Wig97] Theo Wiggerts: *Using Clustering Algorithms in Legacy Systems Remodularization.* In: *Proceedings of the 4th Working Conference on Reverese Engineering (WCRE).* IEEE Computer Society, 1997.

ISBN: 978-3-86644-213-9

www.uvka.de