Automatisierte Auflösung zyklischer Abhängigkeiten in Softwaresystemen

Leo Savernik

19. September 2008

.

Kurzfassung

Zyklische Abhängigkeiten zwischen Komponenten erschweren die Wartung und Weiterentwicklung von Softwaresystemen. So kann man eine in zyklischer Abhängigkeit stehende Komponente nicht für sich alleine betrachten, sondern nur im Kontext aller Komponenten, die sich in zyklischer Abhängigkeit zueinander befinden. Änderungen an einer Komponente wirken sich potentiell auf alle Komponenten in zyklischer Abhängigkeit aus. Je mehr Komponenten sich in Zyklen befinden, desto schwerer wird die Entwicklung eines Softwaresystems handhabbar.

In dieser Arbeit präsentieren wir ein Verfahren und ein Werkzeug, das dem Entwickler eines Softwaresystems nicht nur die zyklischen Abhängigkeiten in Softwaresystemen zeigt, sondern darüber hinaus Auflösungstechniken anbietet, die eine automatisierte Auflösung der zyklischen Abhängigkeiten ermöglicht. Wir untersuchten Effektivität und Auswirkungen der automatisierten Zyklenauflösung sowohl statistisch als auch exemplarisch an realen Softwaresystemen und beurteilten die Ergebnisse anhand von Rückmeldungen, die von Entwicklern der untersuchten Softwaresysteme stammen.

Abstract

Cyclic dependencies between components make maintenance and development of software systems difficult. A component in cyclic dependencies cannot be examined in isolation, but must be regarded within the context of all other components in mutual cyclic dependencies. If such a component is modified, changes potentially impact all components in cyclic dependencies. The more components are entangled in cyclic dependencies, the more difficult the development of a software system becomes.

In this work we present a method and a tool not only showing to the developer cyclic dependencies of a software system, but also providing resolution techniques which enable the developer to resolve cyclic dependencies in an automated fashion. We examined effectiveness and impact of automated resolution of cyclic dependencies on real software systems both statistically and by examples, and we assessed the results by feedback gathered from developers of the examined software systems.

Inhaltsverzeichnis

Kapitel 1

Einführung

Der Schlüssel zu zeitgerechter Fertigstellung und adäquater Implementierung von Anforderungen in Softwaresystemen liegt in der Fähigkeit, Softwaresysteme leicht erweitern, einfach warten und effektiv testen zu können. Die Implementierung zusätzlicher Anforderungen oder auch nur die Anpassung existierender Funktionalität an neue Anforderungen erfordert ein Verständnis des Systems und ein Zeitkontingent zur Einfindung in die vorherrschende Architektur. Erst auf den gewonnenen Erkenntnissen aufbauend lässt sich die Integration dieser Anforderungen durchführen. Die eingeführten Änderungen müssen auch getestet werden. Somit wächst mit der Größe des Softwaresystems der Testaufwand. Maßnahmen zur Verringerung des Testaufwands wirken sich ebenfalls positiv auf das Zeitkontingent zur Entwicklung eines Softwaresystems aus.

Nun besitzen viele Softwaresysteme eine Eigenschaft, die das Programmverständnis sowie die Erweiterbarkeit, Wartbarkeit und Testbarkeit stört: zyklische Abhängigkeiten [56].

Diese zyklischen Abhängigkeiten, auch kurz Zyklen genannt, erschweren

- das Programmverständnis, da der Wissbegierige eine mit anderen Komponenten in zyklischer Abhängigkeit stehende Komponente nicht separat betrachten und begreifen kann,
- die Erweiterbarkeit, da sich eine Komponente nicht erweitern lässt, ohne dass sie über den Zyklus eine Rückkopplung erfährt,
- die Wartbarkeit, da Änderungen in einer Komponente potentielle Nebenwirkungen in allen anderen Komponenten des Zyklus zur Folge haben, und
- die Testbarkeit, da sich die Komponenten über die Zyklen gegenseitig beeinflussen und unabhängige Tests einzelner Komponenten mühsam und zeitaufwendig machen.

Ein erstrebenswertes Ziel zur Verbesserung von Softwaresystemen ist daher die Eliminierung oder zumindest starke Reduktion von zyklischen Abhängigkeiten.

1.1 Motivation

Die Motivation hinter der automatisierten Zyklenauflösung liegt in der Tatsache, dass Zyklen erstens ein Softwaresystem schwerer wart- und erweiterbar machen und zweitens die manuelle Auflösung von Zyklen in realen Softwaresystemen ab 1000 Klassen viel Zeit beansprucht.

1.1.1 Hintergrund

Im Rahmen dieser Forschung konnte sich der Autor von »Softwareentwicklung im Felde« ein gutes Bild machen. Er erkannte, dass Entwickler die Qualität nicht etwa maßgeblich durch Irrtum oder Unwissenheit beeinträchtigen, sondern in der Regel mit Vorsatz, damit sie jene Zeitpläne einhalten können, die direkt zum finanziellen Erfolg des Unternehmens beitragen.

Aus betriebswirtschaftlicher Sicht ist dieses Verhalten mittelfristig sinnvoll, ja sogar notwendig, um am Markt Bestand zu haben. Verfahren und Werkzeugen zur Hebung und langfristigen Erhaltung der Softwarequalität in immer kürzer werdenden Zyklen des Marktes kommt daher eine besondere Bedeutung zu.

1.1.2 Zum Verfahren

Ein Verfahren zur automatisierten Auflösung von Zyklen hilft der langfristigen Hebung der Softwarequalität hinsichtlich Erweiterbarkeit, Wartbarkeit und Testbarkeit, ohne ungebührlich viel Zeit zu verbrauchen, die dann der Erfüllung der unternehmenswichtigen Kurzfristziele fehlt.

Betrachten wir zunächst die Alternativen:

Händische Auflösung Die händische Auflösung von Zyklen verbraucht für Softwaresysteme realistischer Größe eine große Menge an Zeit und ist personalintensiv. Als Interessensvertreter sind nicht nur der Softwarearchitekt, der die Übereinstimmung der Änderungen mit der Sollarchitektur überwacht, sowie der Softwareentwickler, dem die effektive Auflösung der Zyklen in seinen jeweiligen Modulen obliegt, eingebunden, sondern auch die Softwaretester, welche die durchgeführten Änderungen auf eingebrachte Fehler untersuchen müssen.

Werkzeuggestützte Auflösung Eine Anzahl von Werkzeugen unterstützt das Auffinden und teilweise sogar das Auflösen von Zyklen. Obwohl diese Werkzeuge die Arbeit des Softwarearchitekten als auch des Softwareentwicklers erleichtern, erfolgt die tatsächliche Zyklenauflösung immer noch händisch. Dadurch wird das Einschleppen neuer Fehler nicht reduziert – der gesamte Testaufwand ist damit fällig.

Eine vollständig automatische Zyklenauflösung transformiert das vorliegende Softwaresystem so, dass idealerweise keiner der drei Interessensvertreter zu einem weiteren Eingriff genötigt ist. Dieses Ziel ist allerdings utopisch.

Jedoch genügt es bereits, wenn Softwarearchitekt oder Entwickler die Transformation großer Teile des Systems anstoßen können, deren Änderungen sich ohne oder mit geringen manuellen Nachänderungen übernehmen lassen. Dadurch reduziert sich der Testaufwand auf den Bruchteil der tatsächlich notwendigen manuellen Änderungen, während die automatisiert umgebauten Teile als korrekt transformiert anzusehen sind und nur noch oberflächliche Tests erfordern.

Am meisten profitiert der Softwareentwickler, da er durch die automatische Transformation wertvolle Zeit gewinnt, sich den Kurzfristzielen zu widmen.

Durch die Transformation legt das Verfahren ein umfangreiches *Änderungsprotokoll* an, das erstens die gefundenen Probleme und zweitens die durchgeführten Transformationen beschreibt. Die Begutachtung der vorgenommenen Änderungen bleibt dem Personal freilich nicht erspart, da im Endeffekt nur ein Mensch entscheiden kann, ob eine Änderung dem Geiste der Architektur entspricht.

1.2 Ziele und Eigenleistung

Die Forschungsarbeit widmet sich dem Ziel der *automatisierten Auflösung zyklischer Abhängigkeiten in Softwaresystemen*. Als Resultat entsteht ein Verfahren, welches dem Benutzer nicht nur ermöglicht, zyklische Abhängigkeiten in Softwaresystemen zu finden, sondern das Softwaresystem überdies automatisiert so zu transformieren, dass die Anzahl an Artefakten in zyklischen Abhängigkeiten deutlich verringert werden können.

1.2.1 Ziele

Im Einzelnen soll das zu entwickelnde Verfahren

- die Anzahl der Klassen in zyklischen Abhängigkeiten reduzieren,
- automatische Programmtransformationen vornehmen können, um zyklische Abhängigkeiten aufzulösen,
- das Verhalten des transformierten Programms im Vergleich zum nichttransformierten nicht beeinträchtigen und
- die Transformationen schonend vornehmen, sodass der Softwareentwickler das Softwaresystem auch nach der Transformation noch erkennt und akzeptiert.

1.2.2 Eigenleistung

Betrachten wir bereits existierende Lösungen, so stellen wir fest, dass zwar eine Reihe an Verfahren und Werkzeugen zur Entdeckung von Zyklen existieren ([81, 57, 58, 37, 75], ByeCycle, JDepend, Classycle), die entweder ausschließlich Zyklen aufzeigen oder sich auf das Vorschlagen von möglichen Umbauaktionen

beschränken, den letzten Schritt, nämlich den Umbau des Softwaresystems selbst, jedoch nicht durchführen.

Die Eigenleistung dieser Forschungsarbeit liegt daher in der Entwicklung eines Verfahrens und dessen Implementierung in einem Werkzeug, das neben der Auffindung von zyklischen Abhängigkeiten *zusätzlich* die automatisierte Auflösung von zyklischen Abhängigkeiten über ein gesamtes Softwaresystem ermöglicht.

1.3 Kapitelübersicht

In Kapitel 2 werden Grundbegriffe definiert, auf die sich die folgenden Kapitel beziehen.

Kapitel 3 führt verschiedene Erkenntnisse und Verfahren zum Stand der Technik der Zyklenerkennung, Zyklenauflösung und Umbauoperationen auf und vergleicht den hier vorgestellten Ansatz mit existierenden Ansätzen.

Kapitel 4 beschreibt das Modell, das die untersuchten Softwaresysteme in einer für die Zyklenauflösung passenden Art und Weise abbildet.

Kapitel 5 spezifiziert die in der Literatur gefundenen Auflösungstechniken und schätzt grob den Aufwand für ihre Implementierung.

Kapitel 6 stellt den Entwurf des Zyklenauflösungswerkzeugs dar, mit dem die Zyklenauflösung in echten Softwaresystemen bewerkstelligt wird.

Kapitel 7 legt dar, wie wir die implementierten Auflösungstechniken vollautomatisch auf eine statistisch signifikante Anzahl an Softwaresystemen anwenden und die Effektivität der Auflösungstechniken sowie die Auswirkungen auf durch bekannte Systemmetriken beschriebene Sachverhalte messen.

Kapitel 8 beschreibt eine von Menschenhand geleitete, halbautomatische Reduktion zyklischer Abhängigkeiten an ausgewählten realen Softwaresystemen, zeigt statistische Auswertungen sowie die Auswertung von Fragebögen und beschreibt gemachte Erfahrungen.

Zuletzt fasst Kapitel 9 die Erkenntnisse dieser Forschungsarbeit zusammen und präsentiert einen Ausblick für weiterführende Betätigungen.

Kapitel 2

Definitionen

Zunächst definieren wir einige Begriffe, auf die wir uns in der gesamten Arbeit beziehen und die in der Informatik keine eindeutige Bedeutung besitzen. Diese Begriffe stellen Sachverhalte auf allgemeiner Ebene dar, ohne sich auf implementierungs- oder programmiersprachenspezifische Details beschränken zu müssen.

2.1 Begriffe

Artefakt Ein *Artefakt* ist ein physisches oder abstrahiertes Element eines Softwaresystems wie zum Beispiel ein Symbol, eine Funktion, eine Klasse, ein Paket, ein Subsystem oder eine Architekturschicht.

Primärartefakt Primärartefakte sind jene Artefakte, auf deren Abhängigkeiten untereinander das Hauptaugenmerk unserer Untersuchungen liegt.

In objektorientierten Systemen drückt eine Klasse eine vorsätzlich gewählte kohäsive Einheit von Daten und Algorithmen aus. Da wir objektorientierte Systeme untersuchen, stellt die *Klasse* das Primärartefakt dar.

Verhaltensartefakt Ein *Verhaltensartefakt* [18] ist ein Artefakt, das unmittelbar ausführbaren Code enthält und somit das Verhalten des Programms bestimmt. Verhaltensartefakte werden je nach Programmiersprache Prozeduren, Funktionen oder Methoden genannt.

Strukturartefakt Ein *Strukturartefakt* [18] ist ein Artefakt, das unmittelbar Speicherplatz belegt und somit zur Repräsentation des Programmzustands beiträgt. Strukturartefakte sind globale und lokale Variablen, Formalparameter, Klassenvariablen und Attribute (auch Instanzvariablen oder Felder genannt).

Alle anderen Artefakte lassen sich über folgende Artefakte generalisieren.

Ein *Typartefakt* repräsentiert weder Programmcode noch belegt es Speicherplatz, sondern stellt einen Datentyp dar. Typartefakte umfassen Klassen und Klassenschablonen.

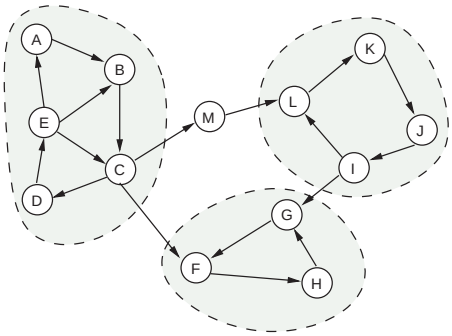


Abbildung 2.1: Knoten, Kanten und Zyklengruppen

Ein *Sammelartefakt* bezeichnet jene Artefakte, die vornehmlich als Behälter weiterer Artefakte dienen. Zu den Sammelartefakten zählen Pakete und Namensräume.

Zur Beschreibung eines Softwaresystems sind neben der Kenntnis seiner Artefakte auch die Abhängigkeiten zwischen den Artefakten notwendig. Also definieren wir zunächst den Grundbegriff der Abhängigkeit zwischen zwei Artefakten und danach darauf aufbauende Begriffe.

Abhängigkeit Ein Artefakt A hängt von Artefakt B ab, wenn Artefakt A Artefakt B zu seiner korrekten Funktionsweise benötigt.

Eine Abhängigkeit ist *mittelbar* beziehungsweise *transitiv*, wenn ein Artefakt A keine direkte Abhängigkeit zu einem Artefakt B unterhält, sondern indirekt über ein drittes Artefakt. Beispielsweise hängt in Abbildung 2.1 M von L ab und C von M. C hängt damit transitiv auch von L ab.

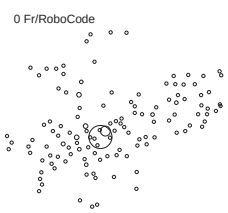
Zyklische Abhängigkeit Eine Abhängigkeit ist *zyklisch*, wenn zwischen zwei Artefakten A und B sowohl A von B (transitiv) abhängt als auch B von A.

Ein Beispiel zyklischer Abhängigkeiten zeigt Abbildung 2.1 mit den Knoten I, J, K und L.

Starke Komponente Eine starke Komponente ist eine in einem Graph größtmögliche Menge von Artefakten, in der zwischen jedem Paar von Artefakten eine zyklische Abhängigkeit besteht.

Zyklengruppe Eine Zyklengruppe ist eine starke Komponente mit mehr als einem Artefakt.

Der Graph in Abbildung 2.1 enthält vier starke Komponenten ($\overline{ABCDE}$, $\overline{FGH}$, $\overline{IJKL}$ und $\overline{M}$). Die drei eingerahmten Komponenten sind Zyklengruppen.



```
1  class A extends B {  // Vererbungsbez. zu B ✪
2    A a;               // Attribut
3    B b;               // Attribut
4    void m() {
5      a = b.a;         // Lesezugriff auf b
6                       // Lesezugriff auf B.a ✪
7                       // Schreibzugriff auf a
8      b.a = a;         // Lesezugriff auf a
9                       // Lesezugriff auf b
10                      // Schreibzugriff auf B.a ✪
11     b.f ();          // Lesezugriff auf b
12                      // Aufrufbeziehung zu B.f ✪
13     m();             // Aufrufbeziehung zu A.m
14     new B(b);        // Aufrufbez. B-Konstruktor ✪
15   }
16 }
```

(a) Klasse A

```
17 class B {
18
19   A a;
20
21   void f() {
22     B v =          // Lokale Variable
23     new B(a);  // Aufrufbez. zu B-Konstruktor
24   }
25
26   B(Object o) {
27     ((B)o).f ();    // Typumwandlung von
28                     // Object in B
29   }
30
31 }
32
```

(b) Klasse B

Abbildung 2.2: Beispiele für grundlegende Abhängigkeitstypen

2.2 Abhängigkeitstypen

Nicht jede Abhängigkeit ist gleich. Daher definieren wir verschiedene Typen von Abhängigkeiten und beschreiben ihre Rolle in der Betrachtung eines Softwaresystems.

Die folgenden drei Typen stellen grundlegende Abhängigkeitstypen dar, die sich direkt aus der Betrachtung des Quelltexts ergeben. Als gemeinsame Eigenschaft drücken sie jeweils eine Implementierungsabhängigkeit aus, wonach Änderungen an der Implementierung des Zielartefakts Änderungen an davon abhängigen Artefakten nach sich ziehen können. Wir nennen solche Abhängigkeiten zur Unterscheidung von allgemeinen Abhängigkeiten auch *Beziehungen*.

Grundlegende Abhängigkeitstypen bestehen immer nur zwischen zwei Verhaltensartefakten, zwischen Verhaltens- und Strukturartefakten oder (als Vererbungsbeziehung) zwischen Typartefakten. Abbildung 2.2 enthält Beispiele zu den Beziehungen, auf die die einzelnen Definitionen verweisen. Unterlegte und mit ✪ markierte Zeilen weisen auf primärartefaktübergreifende Beziehungen hin.

Aufrufbeziehung Ein Verhaltensartefakt V besitzt eine Aufrufbeziehung zu Verhaltensartefakt W, wenn der Quelltext von V einen Aufrufbefehl von W enthält. Nur direkt dem Quelltext entnehmbare Aufrufe zählen als Aufrufbeziehung. Aufrufe indirekter Art, wie zum Beispiel durch Polymorphismus verursacht, zählen nicht als Aufrufbeziehung.

In Abbildung 2.2(a) enthalten die Zeilen 11, 13 und 14, in Abbildung 2.2(b) die Zeile 23 Aufrufbeziehungen.

Zugriffsbeziehung Ein Verhaltensartefakt V besitzt eine Zugriffsbeziehung zu

Strukturartefakt W, wenn der Quelltext von V W ausliest oder beschreibt. Die Zugriffsbeziehung ist eine Vereinigung von Lesezugriff und Schreibzugriff.

Abbildung 2.2(a) enthält Beispiele für Zugriffsbeziehungen in den Zeilen 5 bis 11 und Zeile 13. Letztere repräsentiert eine Zugriffsbeziehung auf die Klasse selbst über den impliziten `this`-Zeiger.

Vererbungsbeziehung Eine Klasse A besitzt eine Vererbungsbeziehung zu einer Klasse B, wenn A von B abgeleitet ist (Beispiel siehe Abb. 2.2(a), Zl. 1). Implementiert Klasse B Schnittstelle S, so zählt dies ebenfalls als Vererbungsbeziehung.

Ein nicht als zyklenerzeugend betrachteter Abhängigkeitstyp ist die Namensabhängigkeit.

Namensabhängigkeit Ein Artefakt A besitzt eine Namensabhängigkeit zu einem Artefakt B, wenn es sich im Quelltext auf den Namen des Artefakts B bezieht.

Als wesentliches Merkmal stellt die Namensabhängigkeit keine Abhängigkeit von einer Implementierung dar. A hängt also nicht von der Implementierung von B ab, womit sich Änderungen an der Implementierung von B nicht auf A auswirken. Typischerweise geht eine Namensabhängigkeit mit einer Beziehung einher, da das Artefakt A in der Regel mit dem referenzierten Artefakt B etwas tun möchte.

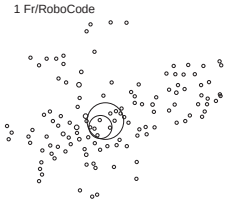
Namensabhängigkeiten finden sich in Formalparameterdefinitionen (`T x`), Variablendefinitionen (`T x`), Objekterzeugungen (`new T()`), Typumwandlungen (`(T)x`) und Importen von Paketen (`import T`).

Werden Formalparameter dereferenziert, tritt unweigerlich eine Zugriffs- oder Aufrufbeziehung auf, welche die Namensabhängigkeit redundant macht. Andernfalls ist der Parameter unbenutzt (weil zum Beispiel von einer Methodenüberschreibung übernommen) und kann zu keinen zyklischen Abhängigkeiten führen. Obige Ausführung gilt analog für Variablendefinitionen.

Objekterzeugungen implizieren bereits den Aufruf des Konstruktors und verursachen immer eine Aufrufbeziehung. Die Namensabhängigkeit ist immer redundant. Die Typumwandlung ist dann redundant, wenn der Ausdruck des umgewandelten Typs eine Zugriffs- oder Aufrufbeziehung enthält (`((T)x).y`), andernfalls bleibt eine reine Namensbeziehung übrig.

Importe sind reine Namensbeziehungen, die niemals von potentiell zyklenerzeugenden Abhängigkeitstypen begleitet sein können.

Zuletzt fehlt noch die Definition der aggregierten Abhängigkeit.



Aggregierte Abhängigkeit Ein Artefakt A unterhält eine aggregierte Abhängigkeit zu einem Artefakt B, wenn Unterartefakte von A Abhängigkeiten zu Unterartefakten von B unterhalten. Das *Gewicht* einer aggregierten Beziehung ist die Summe der Gewichte aller Beziehungen der Unterartefakte zwischen A und B.

Im Beispiel in Abbildung 2.2 besteht eine aggregierte Abhängigkeit von der Klasse A zur Klasse B mit dem Gewicht 5 (zwei Zugriffsbeziehungen, zwei Aufrufbeziehungen, eine Vererbungsbeziehung).

Die Zyklenauflösung findet maßgeblich über aggregierte Abhängigkeiten zwischen Primärartefakten, sodass wir den expliziten Ausdruck »aggregierte Abhängigkeit« in der Regel nicht mehr erwähnen. Sobald von Abhängigkeiten zwischen Klassen die Rede ist, beziehen sich diese auf aggregierte Abhängigkeiten.

2.3 Weitere Definitionen

Fragmentierer Ein Fragmentierer ist ein Artefakt einer Zyklengruppe. Wird ein Fragmentierer aus einer Zyklengruppe entfernt, so schrumpft die übriggebliebene Zyklengruppe oder zerfällt in kleinere starke Komponenten.

Die *Effektivität* eines Fragmentierers bezeichnet die Differenz zwischen der Größe der ursprünglichen Zyklengruppe und der Größe der größten starken Komponente, die übrig bleibt, wenn der Fragmentierer aus der Zyklengruppe entfernt wurde. Wir geben die Effektivität als Paar $n \rightarrow m$ (lies: n auf m) an, wobei n die Anzahl der Artefakte in der ursprünglichen Zyklengruppe und m die Anzahl der Artefakte in der größten starken Komponente nach Entfernung des Fragmentierers darstellt.

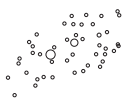
In Abbildung 2.1 besitzen die Fragmentierer der Zyklengruppe $\overline{ABCDE}$ folgende Effektivität: A mit 5 $\rightarrow$ 4, B mit 5 $\rightarrow$ 3, C, D und E mit jeweils 5 $\rightarrow$ 1.

Bester Fragmentierer In einer Zyklengruppe heißen jene Knoten beste Fragmentierer (auch »größte Fragmentierer«) deren Effektivität innerhalb der Zyklengruppe maximal ist.

In Abbildung 2.1 sind in Zyklengruppe $\overline{ABCDE}$ die Knoten C, D und E die besten Fragmentierer mit 5 $\rightarrow$ 1. Wird etwa E entfernt, zerfällt die Zyklengruppe in vier starke Komponenten $\overline{A}$, $\overline{B}$, $\overline{C}$, $\overline{D}$ mit jeweils der Größe eins.

Bei Entfernung von B verbleiben hingegen die starken Komponenten $\overline{A}$ und $\overline{CDE}$. Letztere enthält drei Elemente und ist somit größer als die größte starke Komponente nach der Entfernung von E. B ist daher kein bester Fragmentierer.

Als Konvention für die Darstellung von Artefakten in Diagrammen gilt, dass die Pfeilrichtung die Verwendung (also die Richtung des Zugriffs, des Aufrufs oder der Vererbung widerspiegelt). Bei grafischer Darstellung der Abhängigkeit A→B gilt also durch die Pfeilrichtung, dass A B verwendet, also A von B abhängig ist.



Kapitel 3

Stand der Technik

Zyklen in Softwaresystemen sind ein altbekanntes Phänomen. Bereits in den 1970er Jahren stellten Forscher fest, dass bei fortschreitender Entwicklung von Softwaresystemen zyklische Abhängigkeiten zwischen Modulen entstanden. Diese machten die Entwicklung einzelner Module unabhängig voneinander schwierig bis hin zu dem Punkt, an dem kein einziger Teil des Systems mehr übersetzt und ausgeführt werden konnte, bevor nicht sämtliche Teile ausprogrammiert waren [70].

Die meisten Softwaresysteme werden als Summe kleiner, weitgehend unabhängiger Einheiten entworfen und entwickelt [91]. Bedingt durch die lange Einsatzdauer von Softwaresystemen [82, Kap. 26] wachsen jedoch mit der Größe des Systems nicht nur die Komplexität [45], sondern auch der Anteil der Artefakte in zyklischen Abhängigkeiten [79].

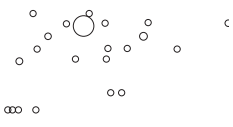
Je mehr Klassen nun von einer Klasse C abhängen (Exportkopplung), umso größer ist die Auswirkung einer Änderung an C auf andere Klassen, und von je mehr Klassen eine Klasse C abhängt (Importkopplung), umso größer sind die Auswirkungen einer Änderung an anderen Klassen auf C selbst [11].

Während bei hierarchischer Dekomposition nur die Auswirkungen der Importkopplung eine Änderung an C nach sich ziehen können, führen zyklische Abhängigkeiten dazu, dass sich auch Änderung der Import- und Exportkopplung über Rückkopplung auf die Klasse C selbst auswirkt. Dies gilt für jede Klasse in einer Zyklengruppe. Je mehr Klassen sich in Zyklengruppen befinden, auf desto größere Teile des Softwaresystems wirken sich auch die Änderungen aus.

3.1 Zyklische Abhängigkeiten

In einem guten Softwareentwurf können Subsysteme unabhängig voneinander entwickelt werden, und das Verständnis eines Subsystems bedingt nur geringe Kenntnisse von anderen Subsystemen [62]. Zyklische Abhängigkeiten stören diese Eigenschaften, daher finden sich in der Literatur seit Jahren Gebote zur Vermeidung zyklischer Abhängigkeiten [42, 52, 26].

Für die existierenden Systeme kommen diese Gebote allerdings zu spät, wenn



sie nicht von Anfang an verfolgt wurden. Es besteht daher der Bedarf, in existierenden Softwaresystemen zyklische Abhängigkeiten nachträglich aufzulösen. Dazu stehen bereits einige Verfahren und Werkzeuge zur Verfügung, die wir nachfolgend genauer betrachten.

3.1.1 Zyklische Abhängigkeiten in Java-Applikationen

Die neuesten Untersuchungen zu zyklischen Abhängigkeiten wurden anhand von Java-Applikationen durchgeführt. Zuallererst sei die Zyklenstudie [59] genannt, die in einem großangelegten Aufbau 78 repräsentative Java-Applikationen auf zyklische Abhängigkeiten untersucht. Dabei stellt die Studie fest, dass Zyklen selbst in nach dem heutigen Stand der Softwareentwicklungstechnik entworfenen Softwaresystemen der *Regelfall* sind, nicht die Ausnahme.

Zur Feststellung und Behandlung zyklischer Abhängigkeiten existieren bereits einige Verfahren und Werkzeuge.

ByeCycle Das Werkzeug *ByeCycle*[1] – eine Eclipse-Einsteckkomponente – visualisiert die Beziehungen zwischen Klassen als Verbindungsdiagramm. Zyklen werden dabei gesondert hervorgehoben. Der Benutzer kann eine solche Verbindung anwählen und wird zu dem verursachenden Artefakt in der Quelltextsicht geführt.

ByeCycle erleichtert die Entdeckung von zyklischen Abhängigkeiten, überlässt deren Auflösung allerdings vollständig dem Benutzer.

Classycle Das Werkzeug *Classycle*[2] erkennt Zyklen sowohl auf Klassen- als auch Paketebene und ordnet die übrigen Artefakte Schichten zu. Als Resultat eines Durchlaufs liefert das Werkzeug die Erkenntnisse als XML-Dokument. Die Auflösung der Zyklen bleibt dem Benutzer überlassen.

JDepend Das Werkzeug *JDepend*[3] berechnet Metriken und Beziehungen eines Softwaresytems auf Paketebene und präsentiert diese dem Benutzer. Die Metriken belaufen sich nach [51] auf afferente/efferente Kopplungen, Instabilität, Abstraktheit und Distanz von der Hauptreihe.

Weiters erkennt das Werkzeug zyklische Abhängigkeiten zwischen Paketen und zeigt die an Zyklen beteiligten Klassen an. Das Auflösen der Abhängigkeiten bleibt dem Benutzer überlassen.

Jepends Das Werkzeug *Jepends* [57] dient der Auffindung von Zyklen in Java-Softwaresystemen, die im Quelltext vorliegen. Jepends erkennt zyklische Abhängigkeiten zwischen Übersetzungseinheiten auf Basis einzelner Zyklen (anstatt Zyklengruppen) und liefert eine Liste der gefundenen Zyklen

[1] http://byecycle.sourceforge.net/ (17. Aug. 2006)
[2] http://classycle.sourceforge.net/ (18. Aug. 2006)
[3] http://www.clarkware.com/software/JDepend.html (17. Aug. 2006)

zurück. Da für das Problem der vollständigen Zyklenerkennung keine effizienten Algorithmen bekannt sind, werden nur eine gewisse Mindestanzahl Zyklen erkannt, an denen eine Übersetzungseinheit beteiligt ist.

Jepends wird benutzt, um Zyklen vor und nach dem Umbau eines Programms zu finden und die Unterschiede zu vergleichen. Der Umbau selbst erfolgt jedoch rein manuell.

JooJ Das Werkzeug *JooJ* [58] setzt auf Vorbeugung statt Reparatur. Es ist als Eclipse-Einsteckkomponente realisiert und prüft den Quelltext auf zyklische Abhängigkeiten während der Eingabe durch den Entwickler. Fügt der Entwickler eine zyklische Abhängigkeit ein, markiert das Werkzeug den inkriminierenden Quelltextteil. Weiters gibt das Werkzeug Vorschläge zum Umbau des Quelltexts, Details über Art und Weise ließen sich dem Artikel jedoch nicht entnehmen.

Das zugrundeliegende Verfahren zur Ermittlung der Zyklen liegt in der Berechnung der *minimalen Rückwärtskantenmenge*, also der Minimalanzahl Kanten, die von einem Graphen zu entfernen sind, damit ein gerichteter azyklischer Graph entsteht. Die durch die minimale Rückwärtskantenmenge gelieferten Kanten markiert das Werkzeug dann im Quelltext als zyklenbildend.

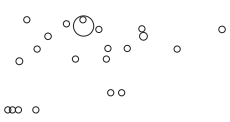
Lattix LDM Das Werkzeug *Lattix LDM* [75] visualisiert die Abhängigkeiten zwischen Artefakten mittels sogenannter Abhängigkeitsstrukturmatrix. In dieser Matrix repräsentiert jede Zeile und jede Spalte je ein Artefakt. Ein Schnittpunkt von Zeile und Spalte gibt eine gerichtete Abhängigkeit zwischen zwei Artefakten an, wobei die Spalte von der Zeile abhängt. Ein zyklenfreies System wird so dargestellt, dass es keine Abhängigkeiten oberhalb der Nebendiagonale enthält. Zyklische Abhängigkeiten sind daher einfach auszumachen.

Auch dieses Werkzeug bietet lediglich Unterstützung im Finden, aber nicht im Auflösen von Zyklen.

PASTA Das Werkzeug *PASTA* [37] visualisiert Abhängigkeiten auf Java-Paketebene nach Schichten und schlägt eine automatische Aufteilung stark verzykelter Pakete (die sich per definitionem in einer Schicht befinden) auf mehrere Schichten vor. PASTA verwendet hierzu eine Heuristik mit der Bezeichnung *intelligente Schichtung*.

Die intelligente Schichtung berechnet zuerst für jede Kante zwischen zwei Artefakten ein Gewicht, das den Aufwand widerspiegelt, um diese Kante aufzulösen. Anschließend sucht das Verfahren eine Menge an Kanten, für die der Gesamtaufwand zu deren Auflösung es als minimal einschätzt, blendet diese Kanten aus und weist die Knoten des nun azyklischen Graphen zur zugehörigen Schicht zu.

Ein tatsächlicher Umbau des Softwaresystems findet nicht statt.



Die aufgeführten Werkzeuge können zwar Zyklen finden, teils verhindern (JooJ) und teils Hinweise zur Auflösung geben (PASTA), doch sie vermögen nicht das unterliegende Softwaresystem entsprechend umzubauen.

3.1.2 Weitere Erkenntnisse zu zyklischen Abhängigkeiten

Der Artikel »Granularität« [53] beschreibt die Nachteile hinsichtlich Wart- und Testbarkeit durch verzykelte Softwaresysteme. Darauf aufbauend definiert er das sogenannte *azyklische Abhängigkeitsprinzip*: Die Abhängigkeitsstruktur zwischen Paketen soll ein gerichteter azyklischer Graph sein.

Die Beachtung des azyklischen Abhängigkeitsprinzips bringe somit eine Softwareverbesserung, die eine konsequente Vermeidung von Zyklen von vornherein rechtfertige.

Parnas [70] vermeidet Zyklen durch die Einführung einer *Benutzt*-Beziehung zwischen Unterprogrammen und ordnet sie in eine Hierarchie ein, in der der längste Pfad eines Unterprogramms zu einem Unterprogramm, das keine weiteren Unterprogramme benutzt, der Ebene des Unterprogramms entspricht. Benutzt zum Beispiel `B` `A`, `C` `B` und `C` `A`, `A` aber nichts, so befindet sich `A` auf Ebene 0, `B` auf Ebene 1 und `C` auf Ebene 2 (der längste Pfad zu `A` geht über `B`).

Ein Unterprogramm niedrigerer Ebene darf keine Unterprogramme höherer Ebene benutzen. Die konsequente Einordnung der Unterprogramme in eine solche Hierarchie gewährleistet ein zyklenfreies System.

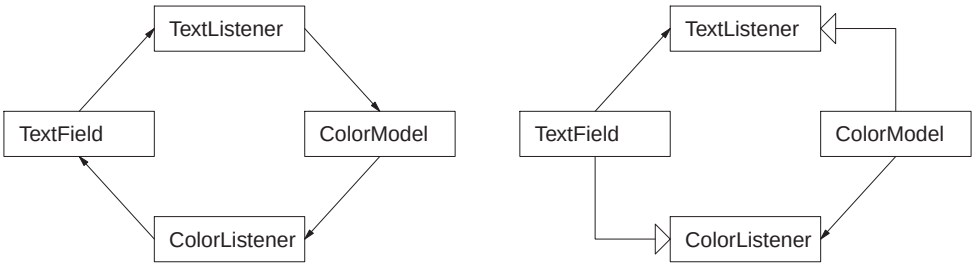
Bislang haben wir lediglich statische Zyklen betrachtet. Als Kontrapunkt dazu existieren auch dynamische Zyklen, das heißt zur Laufzeit auftretende. Diese verursachen in der Regel ein unmittelbares Fehlverhalten des Programms durch unendliche Rekursion.

Beispielsweise beschreibt [34] die unbeabsichtigte Einführung eines Laufzeitzyklus anhand unvorsichtiger Anwendung des Beobachtermusters [28, S. 293] in einen Java-Texteditor.

Abbildung 3.1(a) zeigt einen Aufrufzyklus über vier Java-Klassen, ausgelöst durch ein Änderungsereignis aus `java.awt.Textfield.setText`. `Color-Model` registriert das Ereignis und sendet selbst ein Farbänderungsereignis aus. Dieses registriert `Textfield` und sendet wieder ein Änderungsereignis aus, selbst wenn der Inhalt nicht geändert wurde. Die Struktur des Systems weist hingegen keinerlei statische Zyklen auf (siehe Abb. 3.1(b)).

Dieser Exkurs in die Welt der Laufzeit soll lediglich aufzeigen, dass statische Zyklenfreiheit keine dynamische Zyklenfreiheit nach sich zieht.

Die Analyse des Programmablaufs ist ein eigenes Forschungsgebiet und sprengt den Rahmen dieser Arbeit. Wir beschränken uns in sämtlichen Untersuchungen auf statische zyklische Abhängigkeiten, die aus Quelltext herleitbar sind.



(a) Dynamisches Aufrufdiagramm (b) Strukturdiagramm

Abbildung 3.1: Dynamische Zyklen ohne direkte Repräsentation in statischer Struktur

Quelle von (a): [34]

3.2 Zyklenerkennung

Um zyklische Abhängigkeiten entfernen zu können, müssen sie erst gefunden werden. Da wir die automatisierte Auflösung von Zyklen anstreben, sind natürlich Verfahren zur automatischen Erkennung von Zyklen eine notwendige Voraussetzung.

In keiner anderen Disziplin als der Graphentheorie lassen sich zyklische Abhängigkeiten besser beschreiben, veranschaulichen, algorithmisch erkennen und algorithmisch transformieren. Die Graphentheorie bietet sich daher als primäre Fundgrube von Algorithmen zur automatischen Erkennung von Zyklen an.

Als Ausgangspunkt zur Erkennung dienen *Zyklengruppen*, auch als *starke Komponenten* [14, S. 256] bekannt. Ein Graph aus starken Komponenten ist zyklenfrei. Eine Zyklengruppe entspricht einer starken Komponente mit mehr als einem Element, da einelementige starke Komponenten nicht Teil eines Zyklus sein können und daher zur Zyklenauflösung nicht beitragen.

Zur Erkennung von Zyklengruppen existieren mehrere Algorithmen. Die ältesten Zyklengruppenerkennungsalgorithmen sind von Tarjan und Kosaraju [65]. Beide basieren auf Tiefensuche, mit der sie einen Spannbaum konstruieren und die über übriggebliebene Rückwärtskanten zusammenhängende Knoten zu entsprechenden starken Komponenten zusammenfassen.

Der Zyklengruppenerkennungsalgorithmus von Nuutila-Soisalon-Soininen [67] verbessert Tarjans Algorithmus dahingehend, die Benutzung von Kellerspeicher zu minimieren und die Zyklengruppenberechnung für bereits weitgehend azyklische Graphen zu beschleunigen.

Der *inkrementelle Zyklengruppenerkennungsalgorithmus* [95] liefert aus einem Graphen unter Verwendung binärer Entscheidungsdiagramme, kombiniert mit Erreichbarkeitsanalyse alle Zyklengruppen. Gemäß [95] benötigt der Algorithmus selbst bei riesigen Graphen (mehrere Millionen Knoten) lediglich wenige Sekunden, solange der Graph nur wenige Zyklengruppen enthält. Bei einer großen Anzahl Zyklengruppen steigt die Verarbeitungsdauer stark an.

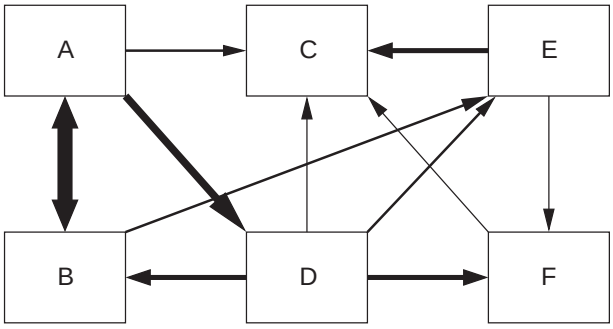


Abbildung 3.2: Artefakte hoher Ebene mit teilweise zyklischen Abhängigkeiten.

Der *lineare symbolische Schrittalgorithmus* [30] berechnet die Zyklengruppen eines Graphen, der durch ein geordnetes binäres Entscheidungsdiagramm repräsentiert wird. Die Verwendung von Entscheidungsdiagrammen im Gegensatz zu herkömmlicher expliziter Graphrepräsentationen wie Adjazenzmatrizen und -listen liegt in der Platzersparnis bei riesigen Graphen. Der Algorithmus besitzt im Gegensatz zu älteren Algorithmen lineare Laufzeitkomplexität.

Visuelle Erfassung von Zyklen bieten viele Werkzeuge an. Ihnen allen ist die Darstellung des Softwaresystems über Verbindungsdiagramme gemein, sodass sich Zyklen insbesondere bei Betrachtung von Artefakten auf hoher Ebene mit wenig Aufwand erkennen lassen. Abbildung 3.2 veranschaulicht ein solches Verbindungsdiagramm, aus dem ersichtlich ist, dass die Komponenten A, B und D einem Zyklus angehören, während die restlichen azyklisch sind. Nebenbei gibt die Stärke der Linie die kumulierte Menge der einzelnen Beziehungen wieder.

Beispiele für die Unterstützung visueller Entdeckung von Zyklen sind der *Software Tomograph* [81], *Moose* [66], *ArchView* [72] und *Rigi* [94], um nur einige zu nennen.

Die visuelle Zyklenerkennung funktioniert nur bei einer verhältnismäßig geringen Menge an angezeigten Knoten und Kanten. Für komplexere Darstellungen, wie sie häufig bei realen Softwaresystemen auftreten, ist eine explizite Softwareunterstützung zum Auffinden von Zyklen unabdingbar.

3.3 Zyklenauflösung

Die Möglichkeiten zur Erkennung von Zyklen wurden nun behandelt. Wie lassen sich jedoch jene Kanten finden, die als Kandidaten für eine Auflösung in Frage kommen?

In der Graphentheorie gibt es ein Standardverfahren zur Auflösung von Zyklen mit der Bezeichnung minimale Rückwärtskantenmenge [80]. Eine *minimale Rückwärtskantenmenge* ist die kleinste Menge aller Kanten, die aus einem gerichteten Graphen zu entfernen sind, um ihn in einen gerichteten azyklischen Graphen zu

überführen. Der exakte Algorithmus ist allerdings NP-schwierig [39] und schließt daher einen Einsatz für Graphrepräsentationen realer Softwaresysteme mit Millionen von Knoten und Kanten aus.

Auch der Bereich des Softwaretestens profitiert von zyklenfreien Systemen zur Minimierung des Testaufwands. Bekannt unter dem Problem »Klassenintegration und Testreihenfolge« ist eine Reihenfolge der Klassen zu bestimmen, in der diese getestet werden können. In einem Zyklus befindliche Klassen lassen sich allerdings nur in ihrer Gesamtheit testen, was den Testaufwand beträchtlich erhöht. Als Ausweg aus dieser Situation führt der Tester eine Scheinklasse ein, die eine Klasse im Zyklus funktional ersetzt und zu keiner anderen Klasse Abhängigkeiten erzeugt. Der Zyklus ist damit aufgebrochen – aber nur für den Testprozess, dem Softwaresystem selbst bleibt der Zyklus erhalten.

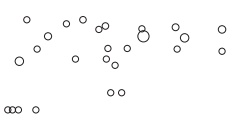
Die Testreihenfolge lässt sich nicht eindeutig bestimmen, wenn mehr als eine einzige Reihenfolge existiert, Klassen durch Scheinklassen zu ersetzen, die zum Aufbruch des Zyklus führt. Zur Ermittlung der *günstigsten* Testreihenfolge existiert deshalb ein Verfahren [68], welches die Schwierigkeit der Scheinklassenerstellung in die Festlegung auf die tatsächlich zu ersetzenden Klassen miteinbezieht.

Hierbei betrachtet das Verfahren die Klassenbeziehungen als Graph und versieht zunächst die Kanten mit Gewichten, die die Kosten der Ersetzung ihrer jeweiligen Ausgangsklassen durch Scheinklassen widerspiegeln. Dann durchläuft ein Algorithmus sämtliche Zyklengruppen und löst die enthaltenen Zyklen durch Entfernung der Kanten mit dem geringsten Gewicht auf. Der Startknoten der jeweils aufgelösten Kante ist ein Kandidat für eine einfache Ersetzung durch eine Scheinklasse.

Das Verfahren nimmt Rücksicht auf besonders schwer durch Scheinklassen aufzulösende Beziehungen wie zum Beispiel Vererbung oder Komposition, indem es diesen ein besonders hohes Gewicht zuweist.

Eades u. a. [20] beschreiben eine effektive Heuristik zur Berechnung des größten azyklischen Graphen, genannt GR-Algorithmus (ab hier GR-550 genannt). Der Algorithmus ermittelt einen »guten« (wenn auch nicht optimalen) azyklischen Graphen zu einem gegebenen Graphen mit einer Laufzeitkomplexität von O(m), wobei m die Anzahl der Kanten repräsentiert. Die Größe der Rückwärtsmenge beträgt für alle Graphen höchstens $\frac{m}{2} - \frac{n}{6}$ mit n gleich der Anzahl der Knoten.

Demetrescu und Finocci [16] stellen zwei Algorithmen zur Berechnung des größten azyklischen Graphen auf Kanten- und Knotenebene vor. Da wir nur an der Kantenebene interessiert sind, betrachten wir lediglich diesen Algorithmus (ab hier FAS-552 genannt) dafür. Der Algorithmus ist kombinatorisch und erreicht eine Annäherungsrate, die durch die Länge (in Kanten) des längsten einfachen Kreises im Graphen beschränkt ist. Die Laufzeitkomplexität beträgt O($m \times n$), wobei m die Anzahl der Kanten und n die Anzahl der Knoten darstellt. Der Algorithmus arbeitet auf gewichteten Kanten und versucht einen Kompromiss zwischen der Entfernung vieler leichter Kanten und der Entfernung weniger schwerer Kanten, die zu vielen



Zyklen gehören, zu finden. Gemäß Experimenten arbeitet der Algorithmus besonders gut für dichte Graphen mit vielen kleinen Zyklen.

Es ist nicht unmittelbar ersichtlich, welcher Algorithmus besser zur Entdeckung auflösbarer Kanten geeignet ist. Freilich besitzt ein linearer Algorithmus wie GR-550 Vorteile – insbesondere bei riesigen Zyklengruppen mit über tausend Knoten und tausenden Kanten –, jedoch ist eine geringere Genauigkeit des Ergebnisses als bei kombinatorischen Algorithmen wie FAS-552 zu erwarten.

3.4 Umbau

Unter *Umbau* (refactoring) versteht sich die Umstrukturierung von Quelltextartefakten zur Verbesserung des Programmverständnisses oder der Wartbarkeit, ohne das nach außen beobachtbare Verhalten eines Programms zu verändern [25, Kap. 2].

Beispiele für Umbauoperationen sind das Umbenennen von Klassen/Funktionen, das Herausheben von Quelltext aus Funktionen in neue Funktionen, das Aufspalten von Klassen/Funktionen und das Verschieben von Funktionen in andere Klassen/Pakete.

Um Zyklen tatsächlich auflösen zu können, müssen wir uns unausweichlich einiger Umbauoperationen [25] bedienen müssen. Wir stellen daher einige Erkenntnisse zum Stand der Technik des Umbaus dar.

Die grundlegenden Verfahrensweisen zum Umbau definiert [69]. Dieses Werk beschreibt 26 atomare Umbauoperationen, die sich verhaltenskonservierend auf objektorientierten Quelltext anwenden lassen und deren Verhaltenskonservierung erwiesen ist [69, S. 29].

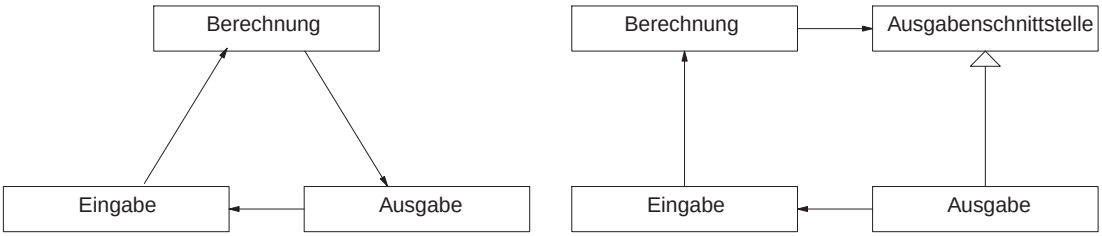
Fowler [25] definiert 74 Umbauoperationen, die sich teilweise mit den in [69] genannten überschneiden. Die Beschreibungen der Umbauoperationen erfolgt unter Berücksichtigung der Verständlichkeit und verzichtet auf Formalismen. Als interessante Umbauoperationen für die Zyklenauflösung erscheinen »bidirektionale Beziehung in unidirektionale umwandeln« [25, S. 200], »Methode verschieben« [25, S. 142], »Klasse extrahieren« [25, S. 149].

Werkzeuge zum Umbau sind zum Beispiel *Jrbx* [54] auf Java-Basis, ein flexibles, erweiterbares Umbauwerkzeug, sowie *Guru* [64], welches automatisch Methodenverdoppelung durch Herausheben gemeinsamer Funktionalität in Basisklassen behebt (Smalltalk-Basis).

Speziell auf die Berücksichtigung von Präprozessoranweisungen in Verbindung mit C/C++ nehmen die Verfahren [83, 29] beim Umbau Bedacht.

3.5 Auflösungstechniken

Wir suchten in der verfügbaren Literatur nach Techniken, die sich zur Auflösung zyklischer Abhängigkeiten heranziehen lassen. Dabei kommt jeder Auflösungs-



(a) Direkte Abhängigkeit (b) Umgekehrte Abhängigkeit

Abbildung 3.3: Beispiel für Abhängigkeitsumkehr

technik die Aufgabe zu, eine bestimmte Abhängigkeit zwischen zwei Artefakten zu entfernen. Diese Abhängigkeit kann eine Rückwärtskante sein, die von einem Algorithmus in Kapitel 3.3 gefunden wurde, oder sie kann extern von einem Menschen festgestellt worden sein, der die Abhängigkeit nun auflösen möchte.

Wir konzentrieren uns in diesem Kapitel auf jene Techniken, die eine Auflösung von Abhängigkeiten des Typs Zugriff oder Aufruf (siehe Kap. 2.2) zulassen. Techniken, die ausschließlich der Auflösung von Namensabhängigkeiten dienen (*verdeckende Zeiger* [42, 5.4], *einfache Daten* [42, 5.5]) beziehungsweise überhaupt nicht zur Zyklenauflösung beitragen (*Kapselungshebung* [42, 5.10]), finden keine weitergehende Erwähnung.

3.5.1 Abhängigkeitsumkehr

Das *Abhängigkeitsumkehrprinzip* besagt, dass konkrete Artefakte von abstrakten Artefakten abhängen sollen und nicht umgekehrt [52]. Oftmals hängen jedoch abstrakte Artefakte von konkreten ab und führen so zu Zyklen.

Handelt es sich bei den Artefakten um Klassen, lässt sich in objektorientierter Programmierung dieses Problem durch das Einziehen einer Schnittstelle lösen, die von der konkreten Klasse implementiert wird. Wenn nun die abstrakte Klasse die neue Schnittstelle statt der konkreten Klasse benutzt, hängt die abstrakte Klasse nicht mehr von der konkreten Klasse ab, sondern die konkrete Klasse nur noch von der (ebenfalls abstrakten) Schnittstelle. Die Abhängigkeit wurde dadurch »umgekehrt« [52].

Nehmen wir als Beispiel das in Abbildung 3.3(a) gezeigte Softwaresystem, in dem `Eingabe` `Berechnung` verwendet, `Berechnung` auf `Ausgabe` zugreift und `Ausgabe` wieder `Eingabe` aufruft, um einen erneuten Rechenvorgang vorzunehmen. Diese drei Klassen stehen in einer zyklischen Abhängigkeit, die wir mittels Abhängigkeitsumkehr lösen wollen.

Wir führen eine Schnittstellenklasse `Ausgabenschnittstelle` ein, lassen `Ausgabe` diese Schnittstelle implementieren und leiten die Zugriffe von `Berechnung` auf ursprünglich `Ausgabe` auf die neue Schnittstelle um. Damit ist

(a) Zyklische Abhängigkeit (b) Nach der Spaltung

Abbildung 3.4: Beispiel für gerichtete Knotenspaltung

die zyklische Abhängigkeit aufgehoben, wie aus Abbildung 3.3(b) hervorgeht. Am Laufzeitverhalten des Programms ändert sich dadurch nichts.

3.5.2 Gerichtete Knotenspaltung

Die Methode der *gerichteten Knotenspaltung* [4] stellt eine simple Methode der Zyklenauflösung durch Aufspaltung von Knoten vor. Dabei wird ein an einem Zyklus beteiligter Knoten in zwei aufgespalten, von denen einer lediglich ausgehende Verbindungen zum Restkreis und der andere eingehende vom Restkreis unterhält. Das Spalten genügend vieler Knoten führt letztlich zu einer Auflösung des Zyklus.

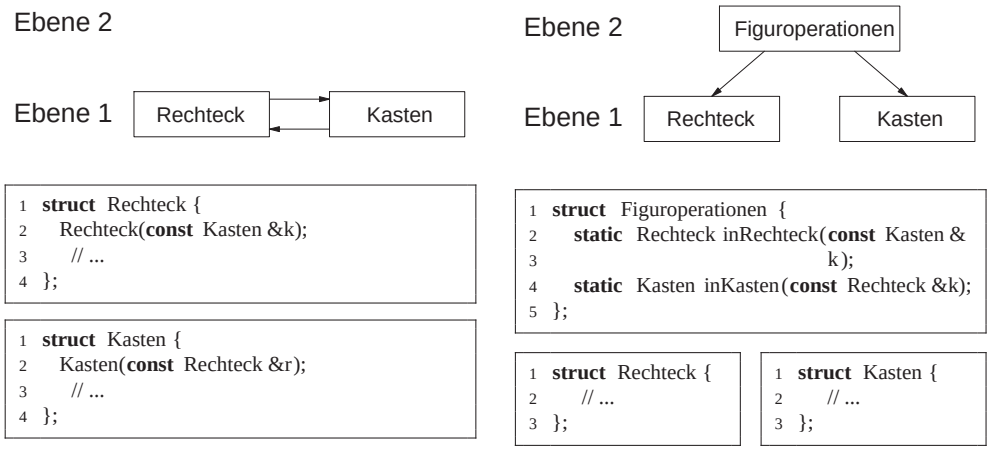
Abbildung 3.4(a) zeigt einen kleinen Zyklus zwischen zwei Knoten A und B. In Abbildung 3.4(b) wurde B so gespalten, dass die ausgehenden Verbindungen in B' verbleiben, während die eingehenden Verbindungen auf den neuen Knoten B'' übergingen. Der Zyklus ist damit aufgelöst.

Knotenspaltung kann auf mehreren Abstraktionsebenen durchgeführt werden. Während sich Pakete ohne großen Zusatzaufwand spalten lassen, indem die enthaltenen Klassen auf die gespalteten Pakete aufgeteilt werden, ist bei der Spaltung einzelner Klassen beträchtlicher Umbauaufwand des Quelltextes zu erwarten.

3.5.3 Hebung

Für große C++-Softwaresysteme stellt [42] eine Reihe von Techniken zur Auflösung zyklischer Abhängigkeiten vor, die – obwohl C++-spezifisch – sich weitgehend programmiersprachenunabhängig auf beliebige Softwaresysteme anwenden lassen. Wir sehen uns alle Techniken im einzelnen an.

Die *Hebung* [42, 5.2] hebt jene Unterartefakte von zwei verzykelten Artefakten auf eine höhere Ebene, die die zyklische Abhängigkeit verursachen. Abbildung 3.5(a) zeigt zwei zyklisch voneinander abhängige Klassen nebst Quelltexten. In Abbildung 3.5(b) wurde eine zusätzliche Klasse `Figuroperationen` eingeführt, die die zyklenverursachenden Methoden aus den beiden anderen Klassen heraushebt und somit den Zyklus auflöst.



(a) Zyklische Abhängigkeit

(b) Nach der Hebung

Abbildung 3.5: Beispiel für eine Hebung

Die Klasse `Figuroperationen` befindet sich auf einer höheren Ebene, da sie von den beiden unterliegenden Klassen abhängig ist, jedoch die unterliegenden Klassen nicht von ihr.

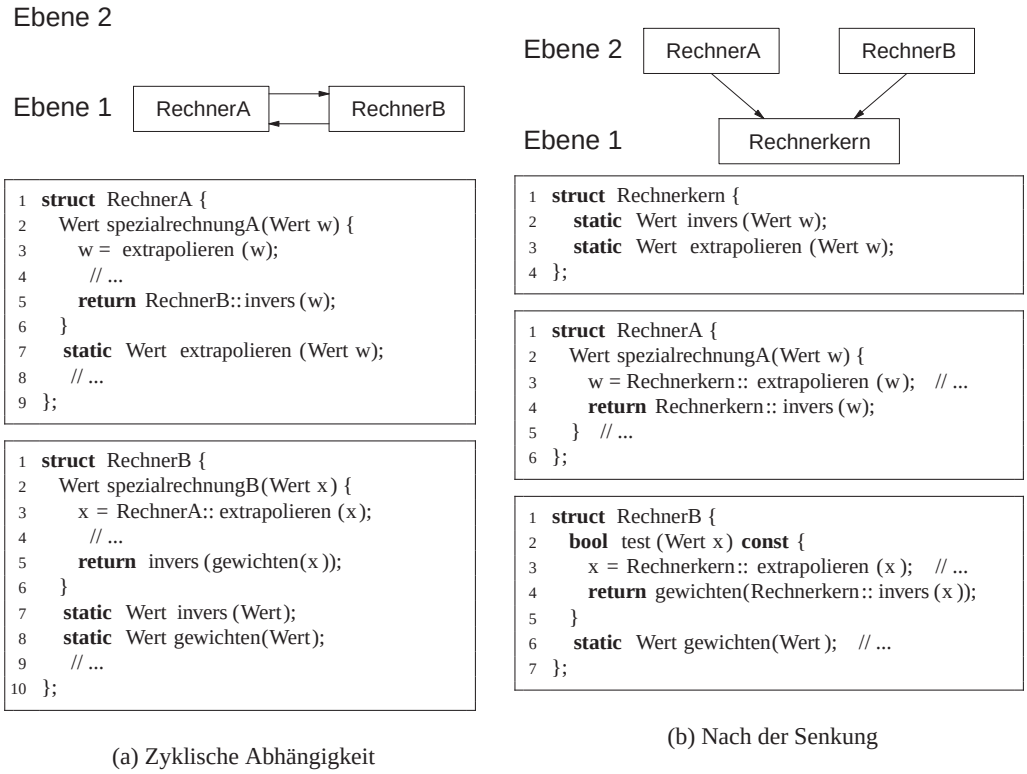
3.5.4 Senkung

Die *Senkung* [42, 5.3] – das Gegenstück zur oben erwähnten Hebung – senkt zyklenverursachende Unterartefakte von Artefakten in zyklischer Abhängigkeit in ein gemeinsames Artefakt auf niedrigerer Ebene. Abbildung 3.6(a) zeigt zwei verzykelte Klassen, die ein gewachsenes System für Berechnungen repräsentieren. `RechnerA` und `RechnerB` führen jeweils Spezialoperationen durch und bedienen sich Hilfsfunktionen. Diese Hilfsfunktionen wurden zunächst in den entsprechenden Rechnerklassen implementiert und mit dem Wachstum des Systems von der anderen Rechnerklasse verwendet, sodass eine zyklische Abhängigkeit entstand. Um dieser Entwurfsdegeneration abzuhelfen, wird in Abbildung 3.6(b) eine dritte Klasse `Rechnerkern` eingeführt, welche die ehemals zyklenverursachenden Methoden auf eine niedrigere Ebene heraushebt.

3.5.5 Redundanz

Redundanz [42, 5.6] macht einige Formen der Wiederverwendung vorsätzlich rückgängig. Dadurch ist eine Auflösung von zyklischen Abhängigkeiten möglich.

Abbildung 3.7(a) zeigt ein Beispiel, in dem die Klasse `Aktivierung` einen Effekt aktiviert. Der Effekt kann aus einem Arbeitsablauf `Ablauf` oder direkt aus der Konsole `Konsole` aktiviert werden. Die Optionen des Effekts benötigen jedoch Werte aus dem Arbeitsablauf, wodurch eine zyklische Abhängigkeit zwischen




```
1  struct RechnerA {
2    Wert spezialrechnungA(Wert w) {
3      w = extrapolieren (w);
4        // ...
5      return RechnerB::invers (w);
6    }
7    static Wert extrapolieren (Wert w);
8     // ...
9  };
```

```
1  struct RechnerB {
2    Wert spezialrechnungB(Wert x) {
3      x = RechnerA:: extrapolieren (x);
4        // ...
5      return invers (gewichten(x));
6    }
7    static Wert invers (Wert);
8    static Wert gewichten(Wert);
9     // ...
10 };
```

(a) Zyklische Abhängigkeit

```
1  struct Rechnerkern {
2    static Wert invers (Wert w);
3    static Wert extrapolieren (Wert w);
4  };
```

```
1  struct RechnerA {
2    Wert spezialrechnungA(Wert w) {
3      w = Rechnerkern:: extrapolieren (w);   // ...
4      return Rechnerkern:: invers (w);
5    }   // ...
6  };
```

```
1  struct RechnerB {
2    bool test (Wert x) const {
3      x = Rechnerkern:: extrapolieren (x);   // ...
4      return gewichten(Rechnerkern:: invers (x));
5    }
6    static Wert gewichten(Wert);   // ...
7  };
```

(b) Nach der Senkung

Abbildung 3.6: Beispiel für eine Senkung

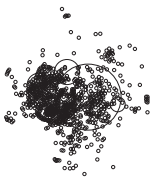
`Aktivierung` und `Ablauf` zustande kommt.

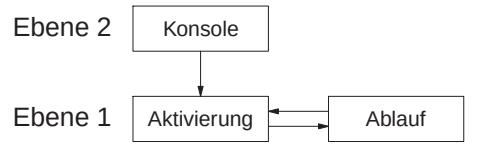
Nach Einführung der Redundanz (Abb. 3.7(b)) wurde die Aktivierung sowohl in den Arbeitsablauf als auch in die Konsole einkopiert. Die Aktivierung verschwindet und damit auch die zyklische Abhängigkeit, da `Ablauf` keine Abhängigkeit zu `Konsole` besitzt und die existierende zyklische Abhängigkeit durch das Einkopieren aufgelöst wurde. Die Verringerung der Kopplung wurde hier durch Einführung von Redundanz bezahlt.

3.5.6 Rückrufe

Rückrufe [42, 5.7] dienen der Auflösung von Abhängigkeiten, indem anstatt eines direkten Aufrufs einer Funktion ein Aufruf über einen Funktionszeiger stattfindet. Der Funktionszeiger ist dabei an anderer Stelle explizit mit der konkreten Funktionsadresse zu initialisieren.

Das Beispiel in Abbildung 3.8 beschreibt einen Wahlautomaten, in dem `Automat` eine Abstimmung über `Stimme` anstößt, welche wiederum die Stimme mittels `Zaehler` zählt, welcher `Automat` zur Entgegennahme eine neuen Stimme bringt. Wir erhalten dadurch eine zyklische Abhängigkeit (s. Abb. 3.8(a)).



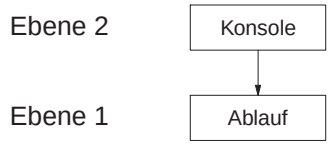



```
1  struct Aktivierung {
2    Ablauf *ablauf;
3    Effekt * effekt;
4    Aktivierung(Ablauf *a, Effekt *e)
5    : ablauf(a), effekt(e) {}
6     // ...
7    void aktivieren () {
8       effekt->option1Setzen(ablauf->x);
9       effekt->option2Setzen(ablauf->y);
10      effekt->option3Setzen (...);
11      effekt->aktivieren ();
12   }
13 };
```

```
1  struct Ablauf {
2    Effekt * effekt;
3     // ...
4    void ausfuehren () {
5       Aktivierung akt(this, effekt);
6       akt.aktivieren ();
7    }
8  };
```

```
1  struct Konsole {
2    Ablauf *ablauf;
3    Effekt * effekt;
4     // ...
5    void effektAktivieren () {
6       Aktivierung akt(ablauf, effekt);
7       akt.aktivieren ();
8    }
9  };
```

(a) Zusammengefasster Quelltext




```
1  struct Ablauf {
2    Effekt * effekt;
3     // ...
4    void aktivieren () {
5       effekt->option1Setzen(this->x);
6       effekt->option2Setzen(this->y);
7       effekt->option3Setzen (...);
8       effekt->aktivieren ();
9    }
10   void ausfuehren () {
11      aktivieren ();
12   }
13 };
```

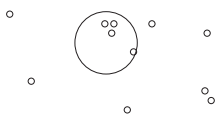
```
1  struct Konsole {
2    Ablauf *ablauf;
3    Effekt * effekt;
4     // ...
5    void aktivieren () {
6       effekt->option1Setzen(ablauf->x);
7       effekt->option2Setzen(ablauf->y);
8       effekt->option3Setzen (...);
9       effekt->aktivieren ();
10   }
11   void effektAktivieren () {
12      aktivieren ();
13   }
14 };
```

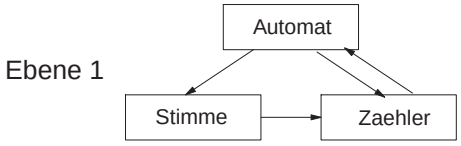
(b) Nach Einführung der Redundanz

Abbildung 3.7: Beispiel für Auflösung durch Redundanz

Durch Einführung eines Rückrufs wird die Abhängigkeit des `Zaehler`s von `Automat` aufgelöst (s. Abb. 3.8(b)). Die Initialisierung des Rückrufs erfolgt in `Automat`, wodurch `Zaehler` von der Kenntnis der Klasse `Automat` gänzlich befreit wird.

Rückrufe lassen sich in der objektorientierten Programmierung oftmals über polymorphe Aufrufe realisieren, wodurch die explizite Kodierung eines Rückrufs meist überflüssig ist.



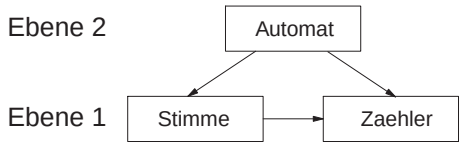



```
1 struct Stimme {
2   int index;
3   Zaehler * zaehler;
4    // ...
5   void abstimmen() {
6       // ...
7     zaehler->hinzu(index);
8   }
9 };
```

```
1 struct Zaehler {
2   int *stimmen;
3    // ...
4   void hinzu(int index) {
5     stimmen[index]++;
6     Automat::stimmeEntgegennehmen(index);
7   }
8 };
```

```
1 struct Automat {
2   static Zaehler zaehler;
3    // ...
4   static void stimmeEntgegennehmen() {
5     Stimme stimme(&zaehler);
6     stimme.abstimmen();
7   }
8 };
```

(a) Direkter Aufruf




```
1 struct Stimme {
2   int index;
3   Zaehler * zaehler;
4    // ...
5   void abstimmen() {
6       // ...
7     zaehler->hinzu(index);
8   }
9 };
```

```
1 struct Zaehler {
2   typedef void Hinzugefuegt(int index);
3   Hinzugefuegt * hinzugefuegt;
4   int *stimmen;
5    // ...
6   void hinzu(int index) {
7     stimmen[index]++;
8     hinzugefuegt(index);
9   }
10 };
```

```
1 struct Automat {
2   static Zaehler zaehler;
3   static void init () {  zaehler.hinzugefuegt
4                 = &stimmeEntgegennehmen; }
5    // ...
6   static void stimmeEntgegennehmen() {
7     Stimme stimme(&zaehler);
8     stimme.abstimmen();
9   }
10 };
```

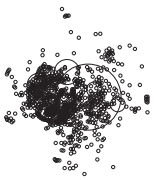
(b) Nach Einführung eines Rückrufs

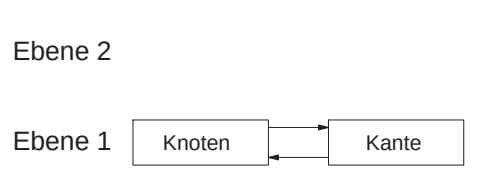
Abbildung 3.8: Beispiel für Auflösung durch Rückrufe

3.5.7 Verwalterklassen

Über Verwalterklassen [42, 5.8] kann Funktionalität aus gegenseitig abhängigen Klassen herausgehoben werden, die zur Verwaltung der Abhängigkeitsverhältnisse dienen. Die gegenseitigen Abhängigkeiten lassen sich so auflösen.

In Abbildung 3.9(a) besitzen die Artefakte `Knoten` und `Kante` eine beiderseitige, unsaubere Aufteilung der Verwaltung der Graphstruktur und sind dadurch zyklisch voneinander abhängig. Durch Einführung einer Verwalterklasse `Graph` in Abbildung 3.9(b) wurde die Verwaltung der Kantenliste in die Verwalterklasse gehoben und die zyklische Abhängigkeit aufgelöst.

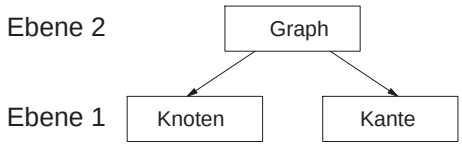





```
1 class Knoten {
2   set<Kante> kanten;
3 public:
4   void kanteHinzu(const Kante &k) {
5     kanten.insert(k);
6   }
7 };
```

```
1 struct Kante {
2   Knoten *anfang;
3   Knoten *ende;
4   Kante(Knoten *von, Knoten *bis)
5   : anfang(von), ende(bis) {
6     von->kanteHinzu(*this);
7     bis->kanteHinzu(*this);
8   }
9 };
```

(a) Ohne Verwalterklassen




```
1 class Graph {
2   set<Kante> kanten;
3 public:
4   void kanteHinzu(const Kante &k) {
5     kanten.insert(k);
6   }
7 };
```

```
1 struct Knoten {
2   // ...
3 };
```

```
1 struct Kante {
2   Knoten *anfang;
3   Knoten *ende;
4   Kante(Knoten *von, Knoten *bis)
5   : anfang(von), ende(bis) {
6   }
7 };
```

(b) Nach Einführung der Verwalterklasse

Abbildung 3.9: Beispiel für Auflösung durch Verwalterklasse

3.5.8 Herausheben

Durch *Herausheben* [42, 5.9] lassen sich kohäsive Programmteile in eine niedrigere Ebene verschieben, wo sie unabhängig wiederverwendet und getestet werden können. Beim Herausheben handelt es sich um eine Technik, die nicht mehr auf Klassenebene greift, sondern auf Komponentenebene (wobei eine Komponente aus einem oder mehreren Paketen beziehungsweise Namensräumen bestehen kann). Das Ziel des Heraushebens liegt vor allem in der Auflösung zyklischer Abhängigkeiten *zwischen* Komponenten, indem zyklische Abhängigkeiten über mehrere Komponenten in eine einzige Komponente verschoben werden.

Das Beispiel in Abbildung 3.10(a) zeigt ein System, in dem jede Klasse `Graph`, `Knoten` und `Kante` jeweils die Merkmale eines Graphen, eines Knoten oder einer Kante aufweist (ohne jedoch speziell als generische Graphverwaltung konzipiert worden zu sein). Jede Klasse ist in einer jeweils eigenen Komponente `Kn` untergebracht und weist zusätzlichen fachspezifischen Quelltext auf (durch `// ...` markiert). Jede Klasse ist von jeder anderen Klasse abhängig.

Durch das Herausheben wird der komponentenübergreifende Zyklus in einen komponentenbeschränkten Zyklus überführt, wie aus Abbildung 3.10(b) ersicht-

1 ZG/ArgoUML

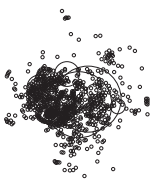
lich ist. Dabei landet der graphstrukturunabhängige Quelltext in den Klassen `Basisknoten` und `Basiskante`, die jeweils voneinander unabhängig sind und sich auf niedrigerer Ebene als die Graphkomponente befinden. Die zyklischen Abhängigkeiten verbleiben in den Klassen `Knoten` und `Kante`, die nun – befreit von graphstrukturunabhängiger Implementierungen – in die Komponente von `Graph` verschoben werden können.

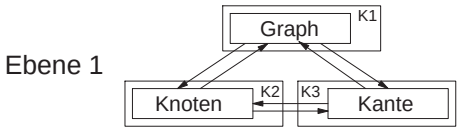
3.5.9 Aspektorientierte Zyklenauflösung

Einen unkonventionellen aber praxisorientierten Weg zur Zyklenauflösung verfolgt die Strategie zur Implementierung [35] von Entwurfsmustern [28] durch Aspekte [40]. Die Auflösung von Zyklen kommt vor allem durch Abhängigkeitsumkehr analog zu [52] zustande. Durch die Zusammenfassung systemweiter Belange in eigene Aspekte verschwinden die Beziehungen auf die konkreten Klassen. Dadurch verursachte Zyklen werden somit aufgelöst.

Ein Beispiel für die Zyklenauflösung durch Aspekte ist der Mediator [28, S. 273]. Im herkömmlichen objektorientierten Entwurf (siehe Abb. 3.11(a)) benachrichtigen die Kollegen den Mediator, der wiederum jeden Kollegen zu dessen Aktualisierung benachrichtigt. Dadurch entstehen zyklische Abhängigkeiten. Durch Einführung des Aspekts *Mediatorprotokoll* als abstrakte Klasse (siehe Abb. 3.11(b)) wird die Benachrichtigung des Mediators aus den jeweiligen Kollegen herausgehoben und zentral über das Mediatorprotokoll implementiert [35]. Dadurch verschwinden die zyklischen Abhängigkeiten zwischen Mediator und Kollegen.

Der Ausflug in die aspektorientierte Programmierung sei der Vollständigkeit halber erwähnt, da diese bis heute in der Softwareentwicklung keine große Verbreitung fand. Erstens gilt die aspektorientierte Programmierung nach wie vor noch nicht als einsatzreif [93], zweitens erscheint ein automatischer Umbau eines Produktivsoftwaresystems in die aspektorientierte Technik wenig brauchbar, da die heute verfügbaren Softwareentwickler in der Regel aspektorientierte Programmierung nicht beherrschen und eine derartige Umwandlung auf wenig Akzeptanz stieße.



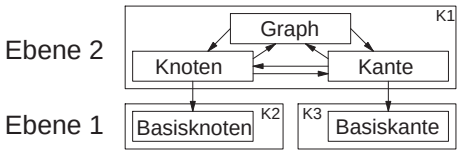



```
1  struct Knoten {
2    Graph *graph;
3    set<Kanten *> kanten;
4    Knoten(Graph *g) : graph(g)
5      { g->knoten.insert( this ); }
6    Kante *verbinden(Knoten *k, int gew)
7      { return new Kante(this, k, gew); }
8    int kantengewichtssumme() { int summe = 0;
9      set<Kanten *>:: const_iterator it;
10     for ( it = kanten.begin (); it != kanten
11       .end(); ++ it ) summe += (*it)->gewicht;
12     return summe;
13   }
14    // ...
15 };
```

```
1  struct Kante {
2    Graph *graph;
3    Knoten *anfang;
4    Knoten *ende;
5    Kante(Knoten *von, Knoten *bis, int gew)
6    : graph(von->graph()), anfang(von)
7    , ende(bis ), gewicht(gew) {
8      graph->kanten.insert( this );
9      von->kanten.insert( this );
10     bis->kanten.insert( this );
11   }
12    // ...
13   int gewicht;
14 };
```

```
1  struct Graph {
2    set<Knoten *> knoten;
3    set<Kanten *> kanten;
4    Graph::~Graph() {
5      for ( set<Kanten *>:: const_iterator it
6        = kanten.begin (); it != kanten.end();
7        ++it ) delete * it;
8      for ( set<Kanten *>:: const_iterator it
9        = knoten.begin (); it != knoten.end();
10       ++it ) delete * it;
11   }
12    // ...
13 };
```

(a) Vor dem Herausheben




```
1  struct Knoten : public Basisknoten {
2    Graph *graph;
3    set<Kanten *> kanten;
4    Knoten(Graph *g) : graph(g)
5      { g->knoten.insert( this ); }
6    Kante *verbinden(Knoten *k, int gew) {
7      { return new Kante(this, k, gew); }
8    int kantengewichtssumme() { int summe = 0;
9      set<Kanten *>:: const_iterator it;
10     for ( it = kanten.begin (); it != kanten
11       .end(); ++ it ) summe += (*it)->gewicht;
12     return summe;
13   }
14 };
```

```
1  struct Kante : public Basiskante {
2    Graph *graph;
3    Knoten *anfang;
4    Knoten *ende;
5    Kante(Knoten *von, Knoten *bis, int gew)
6    : graph(von->graph()), anfang(von)
7    , ende(bis ), gewicht(gew) {
8      graph->kanten.insert( this );
9      von->kanten.insert( this );
10     bis->kanten.insert( this );
11   }
12 };
```

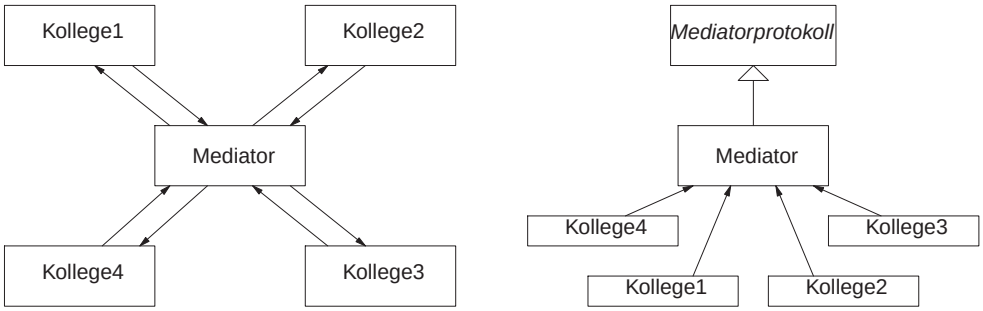
```
1  struct Graph {
2    set<Knoten *> knoten;
3    set<Kanten *> kanten;
4    Graph::~Graph() {
5      for ( set<Kanten *>:: const_iterator it
6        = kanten.begin (); it != kanten.end();
7        ++it ) delete * it;
8      for ( set<Kanten *>:: const_iterator it
9        = knoten.begin (); it != knoten.end();
10       ++it ) delete * it;
11   }
12    // ...
13 };
```

```
1  struct Basisknoten {
2    // ...
3  };
```

```
1  struct Basiskante {
2    int gewicht;
3    // ...
4  };
```

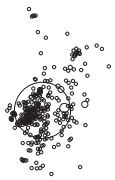
(b) Nach dem Herausheben

Abbildung 3.10: Beispiel für Auflösung durch Herausheben



(a) Zyklische Abhängigkeiten (b) Auflösung durch Aspekte

Abbildung 3.11: Auflösung durch Aspekte anhand des Mediatormusters (Quelle: [35]).



Kapitel 4

Das FAMIX+-Modell

Um zyklische Abhängigkeiten in einem Softwaresystem darstellen und in weiterer Folge auflösen zu können, benötigten wir eine adäquate Repräsentation des Systems in einem Modell. Dieses Modell musste sowohl alle relevanten Artefakte eines objektorientierten Softwaresystems darstellen, nämlich Datentypen, Pakete/ Namensräume, Funktionen, Variablen und Parameter, als auch die Abhängigkeiten zwischen diesen Artefakten.

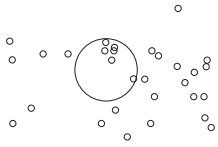
Dabei durfte das Modell nicht auf eine spezifische Programmiersprache beschränkt sein, es sollte gerade so viele Daten beherbergen, wie wir zur Durchführung unserer Forschung brauchten, und doch musste es vielseitig genug sein, um noch nicht absehbare Anforderungen ohne großen Aufwand unterstützen zu können.

4.1 Anforderungen an die Repräsentation

Beschreiben wir zuerst sämtliche Anforderungen, die an eine Repräsentation für ein Modell eines Softwaresystems gestellt werden. Als Maßstab für Objektorientierung dienen die Sprachen C++ und Java. Die Beispiele in Klammern sind jeweils in Java-Syntax aufgeführt, es sei denn, es handelt sich um reine C++-Bestandteile.

An Artefakten muss das Modell

- Klassen (`class X { }`),
- Methoden (`class X { void m() {} }`),
- Attribute (`class X { int a; }`),
- Funktionen (`int main(int argc, char **argv) {}`),
- Formalparameter (`void m(int fp)`),
- lokale Variablen (`void m() { int l; }`),
- globale Variablen (`int gVar;`),



- Pakete/Namensräume (`package P;`),
- Enumerationen (`enum E { ··· }`) und
- entsprechende Schablonen von Klassen, Methoden und Funktionen (`template<typename X> class A {};`)

unterstützen.

Zwischen Artefakten sind folgende Abhängigkeiten abzubilden. Die Beispiele beziehen sich auf Abbildung 2.2 auf Seite 7.

- Lesezugriffe: Beschreibt ein zu lesendes Attribut oder eine zu lesende globale Variable (z. B.: A.m → B.a mit Quelle `b.a` in Z 5).
- Schreibzugriffe: Beschreibt ein zu schreibendes Attribut oder eine zu schreibende globale Variable (z. B.: A.m → B.a mit Quelle `b.a` in Z 8).
- Methodenaufrufe: Beschreibt einen Methodenaufruf (z. B.: A.m → B.f mit Quelle `b.f()` in Z 11).
- Funktionsaufrufe: Beschreibt einen Funktionsaufruf.
- Objekterzeugungen: Beschreibt die Erzeugung eines neuen Objekts aus einer konkreten Klasse (z. B.: A.m → B.B mit Quelle `new B()` in Z 14).
- Typumwandlungen: Beschreibt eine explizite Typumwandlung eines Ausdrucks (z. B.: Object → B mit Quelle `(B)o` in Z 27).
- Vererbungsbeziehungen: Beschreibt die direkte Vererbungsbeziehung zwischen zwei Klassen (z. B.: A → B mit Quelle `A extends B` in Z 1).

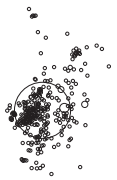
Für die Quelle einer Beziehung (das ist jener Ort, an dem die Beziehung hergestellt wird, also der Methodenaufruf, der Attributzugriff oder die Vererbung stattfindet) muss genügend Kontextinformation bereitstehen, um den Quelltext der Beziehungsquelle finden und gegebenenfalls automatisiert bearbeiten zu können.

Weiters müssen Enthaltenseinsbeziehungen festlegen, welches Artefakt in welchem anderen Artefakt enthalten ist. Pakete enthalten Pakete, Klassen, Enumerationen, Funktionen und globale Variablen, Klassen enthalten Klassen, Enumerationen, Methoden und Attribute. Methoden und Funktionen enthalten Formalparameter und lokale Variablen.

4.2 Begriffsdefinitionen

Präzisieren wir zuerst den Unterschied zwischen Modell und Metamodell.

Modell Ein Modell repräsentiert Artefakte und Abhängigkeiten eines bestimmtes Softwaresystems, die für die vorzunehmenden Untersuchungen, Fragestellungen oder Änderungen nötig sind.



Metamodell Ein Metamodell definiert Elemente, aus denen Modelle für eine bestimmte Problemklasse erstellt werden können. In objektorientierter Terminologie stellt ein Metamodell ein Klassendiagramm mit Implementierungs- und Enthaltenseinsbeziehungen dar, deren konkrete Instanzen jeweils ein Modell repräsentieren.

Wir suchten also ein Metamodell, mit dem wir objektorientierte Softwaresysteme in Modelle überführen konnten, auf denen sich Zyklenentdeckung und Zyklenauflösung durchführen ließen.

4.3 UML

Als allererstes Metamodell fällt dem Informatiker die vereinheitlichte Modellierungssprache UML [92] ein. UML gilt als Standard für die umfassende architekturelle Softwarebeschreibung und wird von nahezu jedem Modellierungswerkzeug unterstützt [38, Kap. 1].

Grundsätzlich können in UML alle Artefakte eines Softwaresystems als auch die Abhängigkeiten zwischen diesen abgebildet werden. Das UML-Klassendiagramm entspricht in etwa dem Detaillierungsgrad, auf dem wir zyklische Abhängigkeiten betrachten. Es bildet Primärartefakte ab und spiegelt Abhängigkeiten, Enthaltenseinsbeziehungen (in UML »Aggregationen« genannt) und Vererbungsbeziehungen wider.

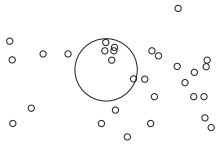
Allerdings traten bereits hier Probleme auf. Wir benötigten nicht nur die Klassen sowie deren aggregierte Abhängigkeiten, sondern auch eine explizite Darstellung feinerer Artefakte wie Attribute und Methoden und deren Beziehungen. Im UML-Klassendiagramm können diese feineren Artefakte nicht adäquat dargestellt werden, und eine Verschmelzung verschiedener Struktur- und Verhaltensdiagrammen erwies sich zur Umsetzung der Anforderungen als zu zeitaufwendig.

Ähnliche Schwierigkeiten mit UML wurden bereits in der Literatur festgestellt. Insbesondere eignet sich UML schlecht zur Rekonstruktion eines Modells aus Programmcode und lässt sich nur mühsam an eigene Bedürfnisse anpassen [17].

UML 2.0 [92] unterstützt zwar Erweiterungen über Profile, doch die Komplexität der Repräsentation nahm im Vergleich zu UML 1 weiter zu. Eine regelkonforme Erstellung eines UML-Profils nimmt nicht nur an sich viel Zeit in Anspruch, sondern bedingt zusätzlich dessen Implementierung im Rahmen eines Forschungsprototypen. Bevor wir jedoch eine komplexe Anpassung eines komplexen Metamodells vornehmen, ist zu prüfen, ob nicht einfachere Metamodelle gefunden werden können.

Weiter gegen UML spricht, dass kein einheitliches Format zur Ablage von UML-Modellen existiert. Selbst eines der meistverwendeten Formate, XMI [61], wird je nach eingesetztem Werkzeug inkonsistent interpretiert wird.

Wir suchten also weiter.



4.4 Das FAMIX-Metamodell

Als in Frage kommendes Metamodell stießen wir auf das FAMOOS-Information-Exchange-Metamodell (kurz: FAMIX) [18]. FAMIX wurde an der Universität Bern zur Repräsentation von Softwaresystemen in der MOOSE-Reengineering-Umgebung [66] entwickelt und dokumentiert. Laut [18, Kap. 2]

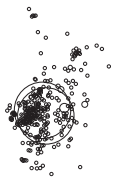
- bietet FAMIX eine einfache Erweiterbarkeit des Datenmodells,
- stellt FAMIX eine hinreichende Basis für Metriken, Heuristiken, Zusammenfassungen und Umbauoperationen zur Verfügung,
- erlaubt FAMIX die einfache Extraktion aus Quelltext,
- eine einfache Erzeugung von Modellen aus verfügbaren Syntaxanalysetechnologien,
- eine einfache Verarbeitbarkeit der Repräsentation mit Standardwerkzeugen,
- eine einfache Abfrage von Artefakten und Abhängigkeiten,
- bietet FAMIX die einfache Kombination mit Informationen aus anderen Quellen und
- unterstützt Industrienormen (CDIF [24], XMI [61]).

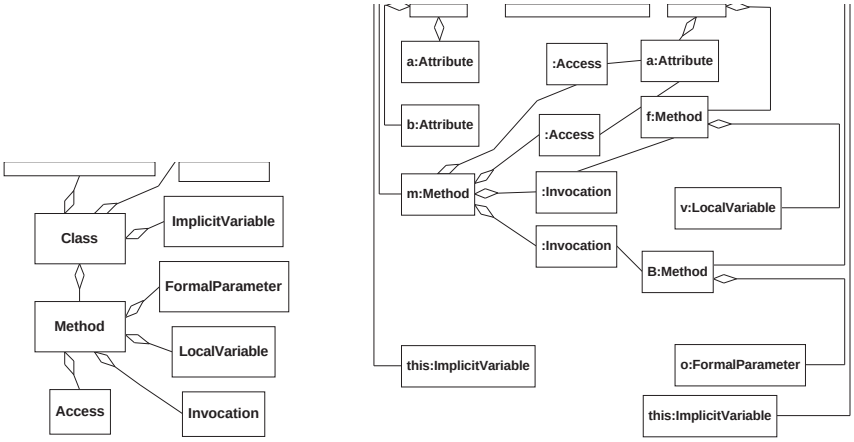
4.4.1 Beispiel

Anhand des Beispiels in Abbildung 2.2 auf Seite 7 soll die Zusammenwirkung des FAMIX-Metamodells mit dem FAMIX-Modell dargestellt werden. Abbildung 4.1 zeigt links ein Klassendiagramm, welches nur die zur Modellierung des Beispiels benötigten Schnittstellen enthält, sowie rechts ein Objektdiagramm, welches das tatsächliche FAMIX-Modell des Beispiels repräsentiert.

Das FAMIX-Metamodell zur Linken beschreibt die Schnittstellen, die die Artefakte und Beziehungen eines Softwaresystems repräsentieren. Zur Rechten sehen wir das tatsächliche FAMIX-Modell, das das Beispiel in Abbildung 2.2 widerspiegelt. Die Aggregationen in Abbildung 4.1(b) zeigen an, welches Artefakt zu welchem Artefakt gehört, beispielsweise ist `a` ein Attribut der Klasse `A` (links) und ein anderes `a` Attribut der Klasse `B` (rechts). Ebenso lassen sich Zugehörigkeiten von Formalparametern und lokalen Variablen feststellen.

Eine Beziehung wird über drei FAMIX-Objekte dargestellt, nämlich Quellartefakt – Beziehung – Zielartefakt. Das Quellartefakt, in Abbildung 4.1(b) `m`, enthält die Beziehungsobjekte, die wiederum jeweils eine Assoziation zum Zielartefakt unterhalten. Für jede einzelne Beziehung wird ein eigenes Beziehungsobjekt angelegt. Im Beispiel existieren zwei Zugriffsbeziehungen von `A.m` auf `B.a` (Zeile 5 und 8), daher gibt es auch zwei `Access`-Objekte, die diese Beziehungen modellieren.





(a) FAMIX-Metamodell (b) FAMIX-Modell

Abbildung 4.1: Beispiel für ein FAMIX-Modell und sein Metamodell

Im gezeigten Modell wurden nur die mit ✪ markierten Beziehungen abgebildet, das FAMIX-Metamodell würde grundsätzlich auch die Abbildung anderer Beziehungen (zum Beispiel innerhalb einer Klasse) erlauben.

4.4.2 Klassendiagramm

Die Diagramme in Abbildung 4.2 zeigen das FAMIX-Metamodell inklusive seiner C++- und Java-Erweiterungen (grau hinterlegt). Als Basis aller Schnittstellen dient die Schnittstelle `Object`, von der die Schnittstellen `Entity` und `Association` abgeleitet sind.

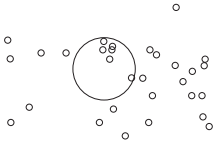
`Entity` repräsentiert Artefakte, `Association` Beziehungen. Als Artefakte unterstützt werden Pakete (`Package`), Klassen (`Class`), Verhaltensartefakte (`BehaviouralEntity`) und Strukturartefakte (`StructuralEntity`).

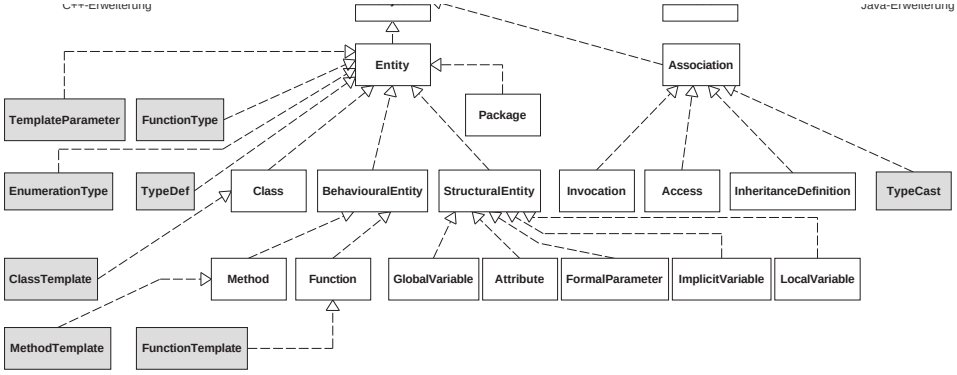
Die C++-Erweiterung [10] enthält weiters Funktionstypen (`FunctionType`), Aufzählungstypen (`EnumerationType`), Schablonenparameter (`TemplateParameter`), Klassenschablonen (`ClassTemplate`), Methodenschablonen (`MethodTemplate`) und Funktionsschablonen (`FunctionTemplate`).

`Association` repräsentiert die drei Beziehungstypen Aufruf (`Invocation`), Zugriff (`Access`) und Vererbung (`InheritanceDefinition`).

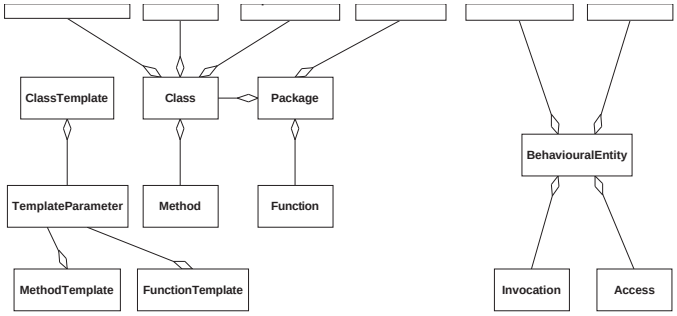
Sowohl die C++- als auch die Java-Erweiterung [84, 90] stellen als zusätzliche Abhängigkeit Typumwandlungen (`TypeCast`) zur Verfügung.

Pakete enthalten als Artefakte Pakete, globale Variablen, Funktionen und Klassen. Klassen enthalten als Artefakte Klassen, Methoden, Attribute und implizite Variablen (`this`-Zeiger) sowie Vererbungen als Beziehungen. Verhaltensartefakte enthalten als Artefakte formale Parameter und lokale Variablen sowie als Be-





(a) FAMIX-Vererbungshierarchie



(b) FAMIX-Aggregationen

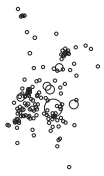
Abbildung 4.2: Klassendiagramm des FAMIX-Metamodells

ziehungen Zugriffe und Aufrufe. Klassen-, Funktions- und Methodenschablonen enthalten als Artefakte Schablonenparameter.

4.4.3 Gründe für FAMIX

Das FAMOOS-Information-Exchange-Modells FAMIX 2.0 [18] eignet sich als Metamodell aufgrund

- der Verwendung in ähnlich gelagerten wissenschaftlichen Forschungsarbeiten zur Softwareentwicklung [43, 19, 33, 44, 72],
- der Sprachunabhängigkeit, insbesondere in der Unterstützung von C++ und Java,
- des verhältnismäßig einfachen Aufbaus,



- der Verständlichkeit und
- der einfachen Erweiterbarkeit.

4.4.4 Gründe gegen FAMIX

Für FAMIX existieren bereits Erweiterungen für Java [84, 90] und C++ [10]. Während die Java-Erweiterung auf der Sprachversion 1.2 verharrt, bietet die C++-Erweiterung bereits die im C++-Standard [12] definierten Sprachfunktionen.

Seit der Definition von FAMIX 2.0 entwickelte sich Java [31] signifikant weiter und wird nicht mehr adäquat von der FAMIX-Erweiterung repräsentiert.

Ansonsten entsprach FAMIX weitgehend unseren in Kapitel 4.1 beschriebenen Anforderungen. Es war daher zweckmäßig, FAMIX als Basis zu verwenden und lediglich um jene Teile zu erweitern, die wir für unsere Forschung benötigten.

4.5 Das FAMIX+-Metamodell

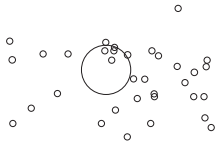
Wir konzipierten unser eigenes Metamodell FAMIX+. Es baut auf FAMIX 2.0 [18], inkorporiert die Java- [84, 90] und C++-Erweiterungen [10] und passt das Metamodell an den Stand der Technik beider Sprachen an. Für Java bedeutet dies insbesondere die Berücksichtigung generischer Klassen.

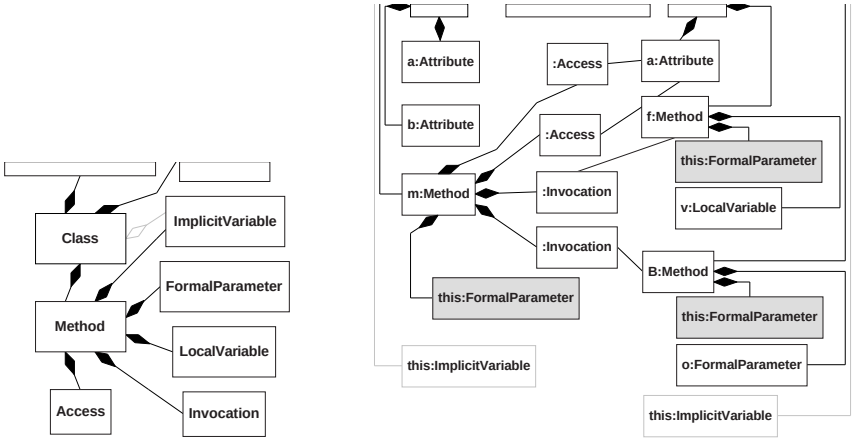
Damit umfasst FAMIX+ einen Großteil des Grundmetamodells von FAMIX sowie seiner Erweiterungen in einem einzigen Metamodell. Damit lassen sich mehrere Sprachen einfach unterstützen, ohne einen expliziten Erweiterungsmechanismus anbieten zu müssen oder grundlegende Algorithmen zur Programmanalyse von Eigenschaften einzelner Sprachmodelle abhängig zu machen.

Im Gegensatz zu FAMIX liegt bei FAMIX+ der Fokus nicht auf einfacher Austauschbarkeit zwischen Werkzeugen und standardisierter Repräsentation, sondern auf flexibler Anpassbarkeit an neue Erfordernisse sowie direkten programmatischen Veränderungen des Metamodells aus einer prototypischen Implementierung heraus.

FAMIX+ übernimmt sämtliche Klassen aus FAMIX und deren Erweiterungen, soweit sie für unsere Zwecke adäquate Unterstützung bieten. Während FAMIX vor allem das relationale Zusammenspiel der Klassen des Metamodells hervorhebt (insbesondere da das für FAMIX [89] gewählte Standardübertragungsformat CDIF [24] ähnlich zu XML-DTDs Zusammenhänge nur über Relationen erlaubt), legt FAMIX+ den Fokus auf eine objektorientierte Repräsentation des Metamodells, die sich auch in den angebotenen Schnittstellen niederschlägt.

FAMIX [18, Kap. 4.5] unterscheidet zwischen fünf Extrahierungsstufen, die die Granularität der aus dem ursprünglichen Softwaresystem extrahierten Informationen widerspiegeln. FAMIX+-Modelle besitzen diese Unterscheidung nicht und entsprechen stets der in FAMIX angegebenen höchsten Extrahierungsstufe 4, also





(a) FAMIX+-Metamodell

(b) FAMIX+-Modell

Abbildung 4.3: Beispiel für ein FAMIX+-Modell und sein Metamodell

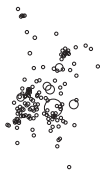
bis auf die Ebene von Methoden, Attribute, Formalparameter, lokalen Variablen und Beziehungen herab.

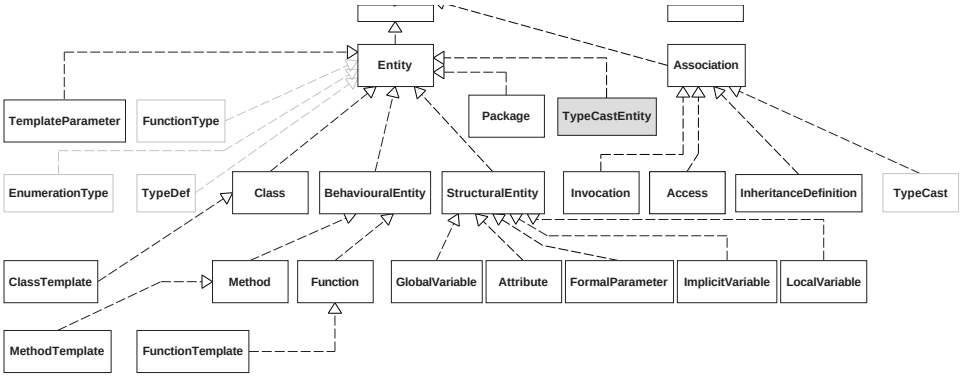
4.5.1 Beispiel

Anhand des Beispiels in Abbildung 2.2 soll die Zusammenwirkung des FAMIX+-Metamodells mit dem FAMIX+-Modell dargestellt werden. Abbildung 4.3 zeigt links ein Klassendiagramm mit den für die Modellierung des Beispiels notwendigen Schnittstellen sowie rechts ein Objektdiagramm, welches das tatsächliche FAMIX+-Modell des Beispiels repräsentiert. Gegenüber FAMIX (Abb. 4.1) sind grau hinterlegte Elemente hinzugefügt, ausgegraute Elemente weggefallen.

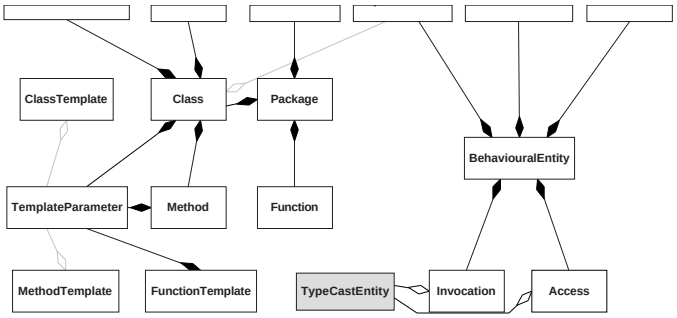
Dem Beispiel ist zu entnehmen, dass FAMIX+ implizite Variablen nicht mehr wie FAMIX auf Klassenebene abbildet, sondern als Formalparameter auf Methodenebene. Die anderen Artefakte und Beziehungen dieses Beispiels bleiben gleich.

Das FAMIX+-Metamodell bildet auch klasseninterne Beziehungen über ein Tripel Quellartefakt – Beziehung – Zielartefakt ab (in Abbildung 4.3(b) nicht explizit dargestellt). Sie spielen zwar unmittelbar für die Betrachtung zyklischer Abhängigkeiten keine Rolle, werden aber für manche Auflösungstechniken als Kontextinformation benötigt. Ein Beispiel für eine klasseninterne Beziehung ist die Aufrufbeziehung von `B.f` → `B.B` in Zeile 23 von Abb. 2.2(b).





(a) FAMIX+-Vererbungshierarchie



(b) FAMIX+-Aggregationen

Abbildung 4.4: Klassendiagramm des FAMIX+-Metamodells

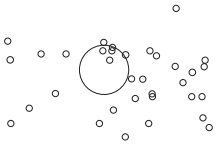
4.5.2 Klassendiagramm

Das Klassendiagramm in Abbildung 4.4 zeigt das FAMIX+-Metamodell. Als Basis aller Schnittstellen dient die Schnittstelle `Object`, von der die Schnittstellen `Entity` und `Association` abgeleitet sind.

`Entity` repräsentiert Artefakte, `Association` Beziehungen. Als Artefakte unterstützt werden Pakete (`Package`), Klassen (`Class`), Schablonenparameter (`TemplateParameter`), Verhaltensartefakte (`BehaviouralEntity`), Strukturartefakte (`StructuralEntity`) und Typumwandlungsartefakte (`TypeCastEntity`).

Das Artefakt `TypeCastEntity` wurde gegenüber FAMIX anstatt der Beziehung `TypeCast` eingeführt, um Typumwandlungen als angebundenen Ausdruck (s. S. 39) darstellen zu können.

Nicht abgebildet wurden die Artefakte `FunctionType`, `Enumeration-`



`Type` und `TypeDef` der C++-Erweiterung [10], da sie zur Repräsentation der untersuchten Beziehungen keinen Beitrag liefern.

`Association` repräsentiert die drei Beziehungstypen Aufruf (`Invocation`), Zugriff (`Access`) und Vererbung (`InheritanceDefinition`).

Im Gegensatz zu FAMIX macht FAMIX+ starken Gebrauch von Kompositionen, sodass für jedes Element ein entsprechendes verantwortliches Element ausgemacht werden kann. Wir sprechen daher nicht mehr bloß von verantwortlichen Elementen, sondern von *Behältern*. Behälter enthalten nicht nur Artefakte, sondern können auch den Ausgangspunkt von Beziehungen darstellen. Behälter sind Pakete, Klassen, Verhaltensartefakte, Funktionen, Methoden sowie Funktions- und Methodenschablonen.

Pakete enthalten als Artefakte Pakete, globale Variablen, Funktionen und Klassen. Klassen enthalten als Artefakte Klassen, Methoden, Attribute und Schablonenparameter sowie Vererbungen als Beziehungen. Während die C++-Erweiterung Schablonenparameter nur für Klassenschablonen zulässt, musste zur Unterstützung generischer Java-Klassen [31] die Schablonenparameterinformation bereits in der Basisklasse gespeichert werden, da der Schablonenansatz unter Java ohne explizite Klassenschablonen auskommt.

Verhaltensartefakte enthalten als Artefakte formale Parameter, lokale Variablen und implizite Variablen sowie als Beziehungen Zugriffe und Aufrufe. Im Gegensatz zu FAMIX, welches implizite Variablen auf Klassenebene ablegt, repräsentiert FAMIX+ implizite Variablen als Formalparameter (`this`-Zeiger).

Funktionsschablonen und Methoden enthalten als Artefakte Schablonenparameter. Auch für Methoden müssen Schablonenparameter wegen der Java-Spezifikation [31] bereits auf Methodenebene gespeichert werden, da Java keiner expliziten Methodenschablonen bedarf.

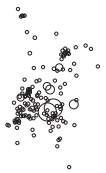
4.5.3 Bestandteile

Ein FAMIX+-Modell setzt sich aus Artefakten zusammen, die über Beziehungen in Abhängigkeit zueinander gesetzt werden.

Aus den Beziehungen lassen sich beliebige Abhängigkeiten zwischen Artefakten eruieren. Während derartige Abhängigkeiten nicht direkt im FAMIX+-Modell repräsentiert werden, können sie nach Bedarf aggregiert werden und stehen danach für Analysen zur Verfügung. Insbesondere Abhängigkeiten zwischen Primärartefakten werden aus Beziehungen aggregiert und über diese die zyklischen Abhängigkeiten ermittelt.

Neben Artefakten und Beziehungen kennt das FAMIX+-Metamodell noch weitere Eigenschaften, die nachfolgend beschrieben werden.

Unveränderliche Artefakte Ein Artefakt ist unveränderlich, wenn es im Zuge einer Umbauoperation nicht geändert werden darf. Unveränderlichkeit des enthaltenden Artefakts (z. B. Klasse) impliziert keine Unveränderlichkeit des enthaltenen Artefakts (z. B. Methode).



Der `this`-Zeiger einer Methode ist zum Beispiel ein unveränderliches Artefakt, da weder sein Name noch sein Typ explizit geändert werden können.

Unvollständige Artefakte Ein Artefakt ist unvollständig, wenn seine vollständige Definition nicht bekannt ist.

Beim Aufbau eines Modells werden nicht aufgelöste Referenzen zunächst als unvollständige Artefakte angelegt, um entsprechende Abhängigkeiten eintragen zu können. Wird die vollständige Definition im Laufe der Extrahierung bekannt, wird der Status des Artefakts auf vollständig gesetzt.

Ein Artefakt bleibt über den gesamten Analysezeitraum unvollständig, wenn es für die Analyse irrelevant ist (zum Beispiel eine Klasse einer externen Bibliothek) und deshalb nur seine Schnittstelle, aber nicht seine Definition gelesen wurde.

Behältereindeutiger Name Ein Name ist behältereindeutig, wenn der Name im Kontext eines Behälters ein darin enthaltenes Artefakt eindeutig beschreibt.

Der behältereindeutigen Name ist weiters im Kontext des gesuchten Artefakttyps abzufragen, da manche Programmiersprachen wie Java Artefakte gleichen Namens in Behältern erlauben (Methode und Attribut dürfen gleich benannt sein: `class X { int a; void a() {}}` [31, § 6.8.5]).

Der behältereindeutige Name eines Artefakts setzt sich aus dem Artefakttyp (Klasse, Methode, Variable, . . .), seinem einfachen Namen und gegebenenfalls seiner Signatur (bei Methoden und Funktionen) zusammen.

Angebundene Ausdrücke Als angebundener Ausdruck ist jener Teil eines Ausdrucks zu verstehen, aus dem sich der Typ einer Klasse ergibt, auf deren Methoden oder Attribute zugegriffen werden soll.

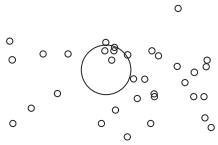
Der angebundene Ausdruck stellt immer den linken Teil eines Selektors **`a.`**$\cdots$**`.b.`**`x` (in diesem Falle **`a.`**$\cdots$**`.b`**) dar, dessen Typ bestimmt, welches `x` welcher Klasse gemeint ist. Ein angebundener Ausdruck kann, muss aber keine Variable sein wie z. B. in **`new S().`**`c` oder **`(a + b).`**`c`.

Ein angebundener Ausdruck **`a.`**`x` repräsentiert jedes beliebige in der Programmiersprache des Softwaresystems gültige Konstrukt, aus dem jene Klasse hervorgeht, die als Attribut `x` enthält. `x` kann auch eine Methode sein, **`a`** ein zusammengesetzter Ausdruck, eine Variable oder ein Typumwandlungsausdruck sein.

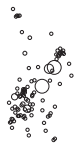
In Abbildung 2.2(a) stellt in der Zugriffsbeziehung `b.a` in Zeile 5 `b` den angebundenen Ausdruck dar, ebenso wie in der Aufrufbeziehung `b.f()` in Zeile 11.

Besteht ein angebundener Ausdruck lediglich aus einem Strukturartefakt, so wird dieser auch *angebundene Variable* genannt. Ist ein angebundener Ausdruck eine Typumwandlung, so wird dieser auch *angebundene Typumwandlung* genannt.

FAMIX+ bildet als angebundene Ausdrücke Typumwandlungen und Variablen ab. Diese erwiesen sich zur Durchführung einfacher Umbauoperationenals ausreichend. Die angebundenen Ausdrücke sind dabei Eigenschaft der jeweiligen Beziehung.



9 Fr/ArgoUML



Kapitel 5

Auflösungtechniken

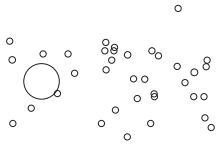
Es gibt eine Reihe von Techniken, die sich zur Auflösung von Abhängigkeiten in einem Softwaresystem eignen. In Kapitel 3.5 beschrieben wir diese Auflösungstechniken informell. In diesem Kapitel spezifizieren wir Auflösungstechniken als Hierarchie von Umbauoperationen, die sich als Algorithmus implementieren lassen. Dazu geben wir für jede Umbauoperation entsprechende Einschränkungen und Annahmen an, die für eine automatisierte Auflösung notwendig sind. Weiters schätzen wir den potentiellen Aufwand einer Implementierung mit einem einfachen Punktesystem.

5.1 Aufwandsschätzung

Eine *Aufwandsschätzung* dient der Ermittlung des voraussichtlichen Aufwands, um ein bestimmtes Ziel zu erreichen. Sie erfolgt entweder experten- oder modellbasiert, wobei bei ersterer der Aufwand durch Fachleute geschätzt wird, bei letzterer durch Anwendung von Regeln und Rechenschritten ein Schätzwert resultiert [60]. Die expertenbasierte Schätzung überwiegt [71, 63], gilt jedoch als weniger stabil hinsichtlich der geschätzten Aufwände im Vergleich zu algorithmischen Schätzverfahren [32].

Zur Aufwandsschätzung der Implementierung von Umbauoperationen wurde daher die Verwendung eines modellbasierten Verfahrens ins Auge gefasst. Von den Schätzverfahren stellen COCOMO [8], Funktionspunktanalyse [5] und die COSMIC-Vollfunktionspunktanalyse [1] die gebräuchlichsten dar. COCOMO schätzt direkt die Projektkosten und ist somit für unseren Fall nicht anwendbar. Die Funktionspunktanalyse sowie die Vollfunktionspunktanalyse liefern beide als Rohwert ein Größenmaß, aus welchem sich der projektspezifische Aufwand durch Anwendung weiterer Vorschriften ermitteln lässt. Jener Rohwert repräsentiert die »intrinsische Größe der Aufgabe« [86], nämlich die Schätzung ohne projektspezifische äußere Einflüsse wie Termine, Wissen oder Budgets. Ein solcher Rohwert genügte unseren Zwecken bereits.

Die Funktionspunktanalyse liefert jedoch nur für Geschäftsanwendungen hin-



reichend genaue Schätzungen [41], die Vollfunktionspunktanalyse unterstützt zusätzlich Echtzeitanwendungen [9]. Die hier aufgeführten Spezifikationen repräsentieren jedoch algorithmenlastige Verfahren, die explizit nicht unterstützt werden [1, Kap. 4.1.5].

Selbst solche Varianten der Funktionspunktanalysen, die bereits bei der Anforderungsanalyse ansetzen [76, 77] und zu unseren Spezifikationen tendenziell besser passen, lassen aufgrund ihrer Abstammung von der Funktionspunktanalyse ebenfalls keine zuverlässige Unterstützung zur Abschätzung algorithmenlastiger Verfahren erkennen.

Weiters scheint es weder eine beste Methode der Aufwandsschätzung zu geben, noch lässt sich a priori feststellen, welches Schätzverfahren in welcher Situation das beste Ergebnis liefert [50]. Da erstens die Anpassung eines existierenden Schätzverfahrens an unsere Bedürfnisse schwierig ist, und zweitens einfache Verfahren ebenso genaue Schätzwerte liefern wie komplexe [73], wurde ein äußerst einfaches, der Funktionspunktanalyse ähnliches Schätzverfahren konzipiert.

5.1.1 Punktevergabe

Das hier vorgestellte Schätzverfahren ermittelt den Aufwand zur Implementierung einer Auflösungstechnik wie folgt: Der Gesamtaufwand setzt sich aus der Summe der Aufwände seiner Bestandteile zusammen. Die Bestandteile einer Auflösungstechnik setzen sich aus Metamodellanforderungen und Durchführungsschritten zusammen, wovon jeder Bestandteil mit einem Punktwert gewichtet ist und deren Punktesumme den Gesamtaufwand widerspiegelt. Hierbei ist zu beachten, dass jeder verwendete Bestandteil genau einmal in die Summe eingerechnet wird, da er nur einmal implementiert werden muss.

Metamodellanforderungen Auflösungstechniken und Umbauoperationen benötigen ein Modell des Softwaresystems, aus dem sie ihre Informationen beziehen und auf dem sie Veränderungen durchführen. Damit die Operationen universell einsetzbar implementiert werden können, benötigen wir ein Metamodell. Dieses Metamodell soll gerade soviele Informationen über ein Softwaresystem enthalten, wie nötig ist, um die Operationen zu implementieren. Die Metamodellanforderungen stellen insofern einen Bestandteil der Aufwandsschätzung dar, als dass sie den Aufwand zur Implementierung eines derartigen Metamodells widerspiegeln. Sollte ein adäquates Metamodell oder adäquate Teile bereits zur Verfügung stehen, so verringert sich der Aufwand um die entsprechende Punktezahl.

Die Umsetzung jeder Metamodellanforderung schlägt sich mit einem gegebenen Punktwert in der Aufwandsschätzung nieder (siehe Tab. 5.2). Der Punktwert je Anforderung repräsentiert den Aufwand allerdings nicht ausreichend, da sich ähnliche Anforderungen dieselbe Infrastruktur teilen und nur die Implementierung der ersten Metamodellanforderung die Implementierung der Infrastruktur erfordert.

Um diesem Umstand Rechnung zu tragen, werden ähnliche Anforderungen zu *Infrastrukturklassen* zusammengefasst. Der Punktwert einer Infrastrukturklas-

K	Pkte	Beschreibung
S	20	Hierarchische Repräsentation struktureller Artefakte ohne Beziehungen
B	20	Repräsentation von Beziehungen zwischen Primärartefakten
F	20	Repräsentation von Artefaktnamen
C	20	Repräsentation von angebundenen Ausdrücken
I	20	Repräsentation von aggregierten Abhängigkeiten innerhalb von Primärartefakten

Tabelle 5.1: Infrastrukturklassen der Metamodellanforderungen

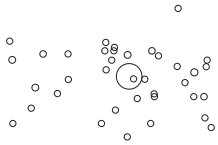
se stellt jenen allgemeinen einmaligen Aufwand dar, der für eine Klasse ähnlicher Anforderungen als bereitzustellende Infrastruktur anfällt. Der Punktewert einer Infrastrukturklasse ist somit als Aufwand miteinzubeziehen, sobald eine Operation mindestens eine Metamodellanforderung besitzt, die in diese Infrastrukturklasse fällt. Tabelle 5.1 listet die Infrastrukturklassen (K) nebst Punktewert und Erklärung auf. Das Kürzel einer Anforderung in Tabelle 5.2 setzt sich dabei aus dem Kürzel einer Infrastrukturklasse sowie einer laufenden Nummer zusammen.

Der Punktewert jeder Infrastrukturklasse leitet sich aus der Überlegung ab, dass die Implementierung der jeweils benötigten Infrastruktur schwerer wiegt als ein einzelner Durchführungsschritt oder die Implementierung einer einzelnen Metamodellanforderung, da die Infrastrukturklasse die größtmöglichen Gemeinsamkeiten einer jeden Metamodellanforderung darstellt. Die genauen Gewichte hängen aber vom Projektumfeld ab und können damit nicht spezifiziert werden. Deswegen wurde für jede Infrastrukturklasse ein Aufwand angenommen, der ähnlich hoch liegt wie der Aufwand für die Implementierung der Durchführungsschritte einer durchschnittlichen Auflösungstechnik.

Durchführungsschritte Als zweiten Bestandteil der Aufwandsschätzung bewerten und summieren wir die Aufwände aller Durchführungsschritte, die zur Umsetzung einer Auflösungstechnik nötig sind. Ein Durchführungsschritt wird höchstens einmal in jeder Berechnung des Gesamtaufwands miteinbezogen.

Der Gesamtaufwand zur Implementierung einer Auflösungstechnik ergibt sich aus der Summe der Aufwände der Bestandteile, nämlich Infrastrukturklassen, Metamodellanforderungen und Durchführungsschritte. Der Gesamtaufwand zur Implementierung mehrerer Auflösungstechniken ergibt sich analog. Hier ist nur zu beachten, dass jede Infrastrukturklasse, jede Metamodellanforderung und jeder Durchführungsschritt nur einmal gezählt werden, auch wenn mehrere Operationen sie benutzen.

Sei U die Menge aller Umbauoperationen, D die Menge aller Durchführungsschritte, A die Menge aller Metamodellanforderungen und I die Menge aller Infrastrukturklassen. Repräsentiere $\mathrm{P_D}(d)$ den Punktewert des Durchführungsschritts $d \in D$, $\mathrm{P_A}(a)$ den Punktewert einer Metamodellanforderung $a \in A$ (Tab. 5.2) sowie $\mathrm{P_I}(i)$ den Punktewert einer Infrastrukturklasse $i \in I$ (Tab. 5.1). Außerdem sei $\mathcal{P}(X)$ die Potenzmenge der Menge X.



AE	Pkte	Beschreibung
S1	1	Das Modell repräsentiert Klassen.
S2	1	Das Modell bildet Methoden auf die zugehörige Klasse ab.
S3	1	Das Modell bildet Attribute auf die zugehörige Klasse ab.
S4	1	Das Modell repräsentiert Attribute.
S5	1	Das Modell repräsentiert Methoden.
S6	1	Das Modell bildet Vererbungsbeziehungen ab.
S7	1	Das Modell repräsentiert globale Variablen.
S8	1	Das Modell repräsentiert Funktionen.
S9	1	Das Modell repräsentiert Pakete.
S10	1	Das Modell bildet Klassen auf die zugehörigen Pakete ab.
S11	1	Jede Klasse gehört zu genau einem Paket.
B1	1	Das Modell bildet Zugriffsbeziehungen auf Attribute ab.
B2	1	Das Modell bildet Beziehungen von Methoden zu anderen Methoden oder Attributen ab.
B3	1	Das Modell bildet Beziehungen von Methoden zu Funktionen oder globalen Variablen ab.
B4	1	Das Modell bildet Beziehungen von Funktionen zu Methoden
F1	1	Das Modell bildet Namen von Methoden ab.
F2	1	Das Modell bildet Namen von Attributen ab.
C1	1	Das Modell bildet die Definition der an einen Methodenaufruf oder Attributzugriff angebundenen Variablen ab.
C2	1	Das Modell bildet die strukturelle Definition der an einen Methodenaufruf oder Attributzugriff angebundenen Typumwandlung ab.
I1	1	Aggregierte Methodenabhängigkeiten innerhalb einer Klasse werden repräsentiert.
I2	1	Aggregierte Attributabhängigkeiten innerhalb einer Klasse werden repräsentiert.

Tabelle 5.2: Alle Metamodellanforderungen

Die Funktion Anf: $U \rightarrow \mathcal{P}(A)$ bilde jede Umbauoperation auf die Menge der von ihr unmittelbar benötigten Metamodellanforderungen ab, Durchf: $U \rightarrow \mathcal{P}(D)$ auf die Menge der unmittelbar die Operation repräsentierenden Durchführungsschritte, sowie Abh: $U \rightarrow \mathcal{P}(U)$ auf die Menge jener Umbauoperationen, von denen diese Umbauoperation mittelbar abhängt. Ferner bilde die Funktion IK: $\mathcal{P}(A) \rightarrow \mathcal{P}(I)$ jede Menge and Metamodellanforderung auf die zugehörige Menge an Infrastrukturklassen ab (der erste Buchstabe von AE in Tab. 5.2 entspricht der Infrastrukturklasse in Tab. 5.1).

Die Mengen aller transitiven Abhängigkeiten ergeben sich aus

$$\text{Abh}_\text{T} : \mathcal{P}(U) \rightarrow \mathcal{P}(D) \times \mathcal{P}(A) \times \mathcal{P}(I),$$
$$\forall_{V \in \mathcal{P}(U)} \forall_{u \in V}\ u \mapsto \left(\bigcup_{v \in \text{Abh}(u)} \text{Durchf}(v), \bigcup_{v \in \text{Abh}(u)} \text{Anf}(v), \bigcup_{v \in \text{Abh}(u)} \text{IK}(\text{Anf}(v)) \right)$$

woraus der Aufwand wie folgt berechnet wird:

$$\text{Aufwand} : U \rightarrow \mathbb{N},\ \forall_{u \in U}\ u \mapsto \sum_{e_d \in d} \text{P}_\text{D}(e_d) + \sum_{e_a \in a} \text{P}_\text{A}(e_a) + \sum_{e_i \in i} \text{P}_\text{I}(e_i)$$
$$\text{mit } (d \in \mathcal{P}(D), a \in \mathcal{P}(A), i \in \mathcal{P}(I)) \mapsto \text{Abh}_\text{T}(\{u\})$$

Der Gesamtaufwand über mehrere Umbauoperationen ergibt sich aus

$$\text{GesAufwand} : \mathcal{P}(U) \rightarrow \mathbb{N},\ \forall_{u \in \mathcal{P}(U)}\ u \mapsto \sum_{e_d \in d} \text{P}_\text{D}(e_d) + \sum_{e_a \in a} \text{P}_\text{A}(e_a) + \sum_{e_i \in i} \text{P}_\text{I}(e_i)$$
$$\text{mit } (d \in \mathcal{P}(D), a \in \mathcal{P}(A), i \in \mathcal{P}(I)) \mapsto \text{Abh}_\text{T}(u)$$

Ein Beispiel findet sich in Kapitel 5.4 auf Seite 68.

5.2 Umbauoperationen

In Kapitel 3.5 wurden die Auflösungstechniken informell und beispielhaft beschrieben. Um eine automatische Auflösung von Zyklen vornehmen zu können, benötigen wir eine detaillierte Beschreibung, mittels derer jede Auflösungstechnik in einem Algorithmus umgesetzt werden kann.

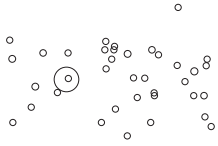
Die hier genannten Umbauoperationen beziehen sich auf Fowler [25], sind jedoch soweit formalisiert beschrieben, dass sie sich in eine Implementierung gießen lassen. Wo Fowler unspezifisch ist, werden entsprechende Annahmen getroffen.

Eine Umbauoperation heißt *Hilfsumbauoperation*, wenn sie sich nicht in der Literatur [25, 69] findet.

Schema Alle Umbauoperationen sind nach folgendem Schema aufgebaut.

Beschreibung Hier wird in wenigen Sätzen das Wesen der Operation beschrieben.

Eingabeparameter Dieser Block führt die Eingabeparameter in einer kommagetrennten Liste auf, welche in nachfolgenden Blöcken referenziert werden.



Durchführung Die Durchführung besteht aus einer Liste von Schritten, die zur Ausführung der Umbauoperation nötig sind. Allfällige Aufrufe von anderen Umbauoperationen sind textuell aufgeführt – inklusive einer Aktualparameterliste und einem Verweis auf das entsprechende Kapitel. Ein Durchführungsschritt entspricht der Form

p. Beschreibung [*Aufgerufene Umbauop*(*param1*, *param2*, . . .) (Verweis)]. (m + n)

und enthält eine fortlaufende Nummer *p*, eine Beschreibung, optional einen Aufruf einer untergeordneten Umbauoperation und den Aufwand im Format (m + n), der zu seiner Implementierung nötig ist. m beschreibt den Aufwand, den dieser Schritt unmittelbar verursacht, während n den mittelbaren Aufwand zur Implementierung der aufgerufenen Umbauoperation enthält (wobei n m nicht enthält).

Der mittelbare Aufwand entspricht der Summe der unmittelbaren Aufwände der Durchführungsschritte der aufgerufenen Operation und deren Unteroperationen und wird nur für den ersten Aufruf eingerechnet, da die aufgerufene Funktion nur einmal implementiert werden muss. Enthält der Durchführungsschritt keinen Aufruf oder wurde der Aufwand der aufgerufenen Operation bereits in einem vorhergehenden Durchführungsschritt eingerechnet, beträgt n null.

Gründe für den Fehlschlag der Operation Dieser Abschnitt enthält eine Liste von Gründen in Form von Bedingungen, die eine ordnungsgemäße Ausführung der Umbauoperation verhindern. Trifft eine einzige Bedingung zu, so schlägt die Umbauoperation als Ganzes fehl. Die Liste enthält ebenfalls sämtliche von aufgerufenen Umbauoperationen gegebenen Gründe nebst deren Herkunft in eckigen Klammern.

Strukturelle Garantien Diese Liste beschreibt Invarianten, die durch die Ausführung der Umbauoperation nicht verletzt werden.

Variante n Eine Umbauoperation kann keine bis mehrere Varianten besitzen, die als Differenz der Durchführungsliste gegeben sind. Jeder Eintrag entspricht einem Schritt der Durchführungsliste und ersetzt oder entfernt ihn (Bezeichnung: (entfällt)).

Der unmittelbare Aufwand entspricht dem tatsächlichen Aufwand der Ersetzung (0 für entfallene Schritte). Der mittelbare Aufwand stellt die Differenz dar, die zum entsprechenden Aufwand der Durchführungsliste addiert werden muss, um den Aufwand der Variante korrekt widerzuspiegeln.

Ausgabe Manche Operationen liefern Variablen zurück, die hier in einer kommagetrennten Liste spezifiziert sind.

Die Metamodellanforderungen werden nicht für jede einzelne Umbauoperation angeführt. Sie sind in Tabelle 5.2 aufgeführt und werden in Tabelle 5.3 auf Seite 69 auf die entsprechenden Umbauoperationen abgebildet.

5.2.1 Umbauoperation: Schnittstelle extrahieren

Beschreibung Die Extrahierung einer Schnittstelle [25, S. 341] aus einer Klasse umfasst die Erstellung einer Schnittstelle mit denselben Methoden und -signaturen wie in denen der Ausgangsklasse.

Eingabeparameter *Klasse, NeuerName*.

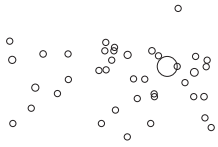
Durchführung

1. Eine neue Schnittstelle n mit dem Namen *NeuerName* erzeugen. (1 + 0)
2. Menge von Klassen- k und Instanzattributen i der Ausgangsklasse *Klasse* ermitteln, auf die von außerhalb der Klasse zugegriffen wird. (1 + 0)
3. Für jedes Instanzattribut $a \in i$ jeweils eine Lese- und Schreibmethode in *Klasse* erzeugen (Zugriffsmethoden einsetzen(a) ☞5.2.2), sofern es diese Methode nicht bereits gibt. (1 + 6)
4. Für jede Methode $m \in$ *Klasse* eine Methode desselben Namens in der Schnittstelle n erzeugen. (1 + 0)
5. Jedes Klassenattribut $a \in k$ in die Schnittstelle heben (Attribut heben(a, n) ☞5.2.5), sofern die Sprache das zulässt. (1 + 2)
6. Jede Klassenmethode m aus *Klasse* in die Schnittstelle heben (Methode heben(m, n) ☞5.2.6), sofern die Sprache das zulässt. (1 + 2)
7. Die Ausgangsklasse *Klasse* von der geschaffenen Schnittstelle n ableiten. (1 + 0)

Können die Schritte 5 und 6 nicht ausgeführt werden, schlägt die Operation nicht fehl. Allerdings führt die unvollständige Extrahierung zur unvollständigen Auflösung von Beziehungen, weil sich diese Beziehungen dann nicht auf die Schnittstelle umleiten lassen.

Gründe für den Fehlschlag der Operation

- Die Klasse von a ist als unveränderlich markiert [aus ☞5.2.5, ☞5.2.2].
- Die Klasse von m ist als unveränderlich markiert [aus ☞5.2.6].
- a ist als unveränderlich markiert [aus ☞5.2.5].



- m greift auf Instanzattribute zu, die nicht über n erreichbar sind [aus ☞5.2.6].
- m ist als unveränderlich markiert [aus ☞5.2.6].
- n ist als unveränderlich markiert [aus ☞5.2.5, ☞5.2.6].
- n ist keine Basisklasse der Klasse von a [aus ☞5.2.5].
- n ist keine Basisklasse der Klasse von m [aus ☞5.2.6].
- *Klasse* ist als unveränderlich markiert.

Strukturelle Garantien

- Die entworfene Klassenhierarchie bleibt unverändert.
- Das Verhalten bleibt unverändert.

Die Klassenhierarchie wird durch die Ableitung von der Schnittstelle geändert. Die Vererbungshierarchie bleibt unverändert.

Das Verhalten bleibt unverändert, da die Schnittstelle neu erzeugt wurde und die Unterklasse die Schnittstelle implementiert. Beziehungen, die auf die Schnittstelle zeigen, führen weiterhin denselben Code derselben Implementierung aus.

Ausgabe n.

5.2.2 Umbauoperation: Zugriffsmethoden einsetzen

Beschreibung Das Einsetzen von Zugriffsmethoden zum Zugriff auf ein Attribut [25, S. 171] umfasst die Erstellung einer Lese- und Schreibmethode sowie die systemweite Ersetzung aller Direktzugriffe durch Aufrufe der entsprechenden Zugriffsmethoden.

Eingabeparameter *Attribut*.

Durchführung

1. Eine Lesemethode l mit behältereindeutigem Namen und derselben Sichtbarkeit wie *Attribut* in der Klasse von *Attribut* erzeugen. (1 + 0)
2. Eine Schreibmethode s mit behältereindeutigem Namen und denselben Zugriffsrechten wie *Attribut* in der Klasse von *Attribut* erzeugen. (1 + 0)
3. Alle Zugriffsverbindungen auf *Attribut* finden. (1 + 0)

3a Jeden Lesezugriff durch einen Aufruf von l ersetzen. (1 + 0)

3b Jeden Schreibzugriff durch einen Aufruf von s ersetzen. (1 + 0)

4. Sichtbarkeit von *Attribut* auf privat setzen. (1 + 0)

Gründe für den Fehlschlag der Operation

- Die Klasse von *Attribut* ist als unveränderlich markiert.

Strukturelle Garantien

- Die Klassenhierarchie bleibt unverändert.
- Existierende Methoden bleiben unverändert.
- Nicht betroffene Attribute bleiben unverändert.

Variante 1 Alternativ kann die Veränderung der Sichtbarkeit des Attributs unterbleiben. Hierzu sind folgende Regeländerungen erforderlich:

4. (entfällt) (0 – 1)

5.2.3 Hilfsumbauoperation: Beziehungen in Basisklasse heben

Beschreibung Diese Operation hebt modellweit Beziehungen auf eine abgeleitete Klasse auf deren Basisklasse. Dabei werden Variablendefinitionstypen der abgeleiteten Klasse durch die Basisklasse ersetzt, sofern keine weiteren Beziehungen auf die abgeleitete Klasse übrigbleiben.

Eingabeparameter *Klasse*, *Basisklasse*.

Durchführung

1. Sei l eine Menge aller Beziehungen b, für die gilt: Entweder ist die angebundene Variable s vom Typ *Klasse*, oder es handelt sich um eine Beziehung zu einem Klassenartefakt (im Gegensatz zu einem Instanzartefakt) oder um eine Typumwandlung.

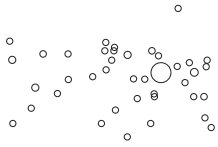
   Sei weiters S die Menge aller angebundenen Variablen s aller Beziehungen in l. (1 + 0)

2. Für jedes b in l prüfen, ob das Zielartefakt in *Basisklasse* (behältereindeutiger Namensvergleich) oder eine entsprechende Zugriffsmethode vorhanden ist. (1 + 0)

3. Wenn nein, s zur Menge *unersetzbar* hinzufügen. (1 + 0)

4. Typ von jedem s in S und nicht in *unersetzbar* durch *Basisklasse* ersetzen. (1 + 0)

5. Jede Beziehung b in l, für die gilt, dass b mit einem $s \notin$ *unersetzbar* verbunden ist, auf den Typ von s umbiegen. (1 + 0)



Ist eine Beziehung oder eine angebundene Variable als unveränderlich markiert, so führt dies lediglich dazu, dass die entsprechende Variable s in die Menge *unersetzbar* aufgenommen wird. Das Ergebnis wird dadurch schlechter, aber nicht verunmöglicht.

Gründe für den Fehlschlag der Operation

- *Klasse* ist als unveränderlich markiert.
- *Klasse* ist unvollständig.

Strukturelle Garantien

- Die Klassenhierarchie bleibt unverändert.

5.2.4 Hilfsumbauoperation: Beziehungen verschieben

Beschreibung Diese Operation verschiebt modellweit Beziehungen, die sich auf eine alte Klasse beziehen, in eine neue Klasse. Dabei werden Variablendefinitionstypen der alten Klasse durch die der neuen Klasse ersetzt, sofern keine weiteren Beziehungen auf die alte Klasse übrigbleiben.

Eingabeparameter *AlteKlasse*, *NeueKlasse*.
Die Funktion Ziel(b) liefert das Zielartefakt der Beziehung b.

Durchführung

1. Sei Angeb(v) eine Funktion, die die Variable v auf eine Menge aller Beziehungen abbildet, die v als angebundene Variable enthalten. (1 + 0)
2. Sei l eine Menge aller Beziehungen B, deren jeweiliges Zielartefakt in *AlteKlasse* oder *NeueKlasse* liegt: $l = \{b \in B | \text{Ziel } (b) \in \textit{AlteKlasse} \cup \textit{NeueKlasse}\}$. (1 + 0)
3. Sei V die Menge aller angebundenen Variablen aller Beziehungen in l. (1 + 0)
4. Menge P der angebundenen Variablen ermitteln, deren Typ durch *NeueKlasse* ersetzt werden kann: $P = \{p \text{ ist eine Variable} | \forall_{b \in \text{Angeb}(p)} (\text{Ziel}(b) \in \textit{NeueKlasse} \wedge \text{Ziel}(b) \notin \textit{AlteKlasse})\}$. (1 + 0)
5. Für jede Beziehung b in l, wobei *NeueKlasse* jeweils das Zielartefakt von b ist: (1 + 0)

5a Wenn $z \in$ *NeueKlasse* und *NeueKlasse* ein Klassenartefakt (kein Instanzartefakt) ist, Zugriffsqualifizierer auf *NeueKlasse* umbiegen. (1 + 0)

5b Wenn $z \in$ *NeueKlasse* und der angebundene Ausdruck von b eine Typumwandlung ist, Typ des angebundenen Ausdrucks auf *NeueKlasse* setzen. (1 + 0)

5c Wenn $z \in$ *NeueKlasse* und der angebundene Ausdruck von b eine angebundene Variable v und $v \in P$ ist, Typ von v auf *NeueKlasse* setzen. (1 + 0)

5d Sonst *Widerspruch* $\leftarrow$ *Widerspruch* $\cup\, b$. (1 + 0)

Ein Widerspruch tritt auf, wenn die konservative Verschiebung von Beziehungen in *NeueKlasse* nicht durchgeführt werden kann, da entweder kein entsprechender angebundener Ausdruck zur Verfügung steht und somit dessen Typ nicht angepasst werden kann oder der angebundene Typ eine angebundene Variable darstellt, die *sowohl* für Zugriffe auf *AlteKlasse* als auch auf *NeueKlasse* herangezogen wird und somit einer konservativen Verschiebung im Wege steht.

Gründe für den Fehlschlag der Operation

- *NeueKlasse* ist als unveränderlich markiert.
- *NeueKlasse* ist unvollständig.
- *Widerspruch* ist nicht leer.

Strukturelle Garantien

- Die Klassenhierarchie bleibt unverändert.

5.2.5 Hilfsumbauoperation: Attribut heben

Beschreibung Diese Operation hebt ein Attribut einer Klasse in eine seiner Basisklassen. Sie repräsentiert eine leichtgewichtige Variante von [25, S. 320].

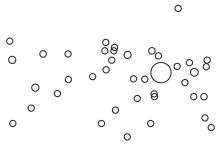
Eingabeparameter *Attribut*, *Basisklasse*.

Durchführung

1. Das Attribut *Attribut* in die Klasse *Basisklasse* verschieben. (1 + 0)
2. Sichtbarkeit von *Attribut* auf höher als privat setzen. (1 + 0)

Gründe für den Fehlschlag der Operation

- Die Klasse von *Attribut* ist als unveränderlich markiert.
- *Attribut* ist als unveränderlich markiert.
- *Basisklasse* ist als unveränderlich markiert.
- *Basisklasse* ist keine Basisklasse der Klasse von *Attribut*.



Strukturelle Garantien

- Die Klassenhierarchie bleibt unverändert.
- Existierende Methoden bleiben unverändert.
- Nicht betroffene Attribute bleiben unverändert.

Variante 1 Alternativ kann die Veränderung der Sichtbarkeit des Attributs unterbleiben. Hierzu sind folgende Regeländerungen erforderlich:

2. (entfällt) (0 – 1)

5.2.6 Hilfsumbauoperation: Methode heben

Beschreibung Diese Operation hebt eine Methode einer Klasse in eine seiner Basisklassen. Sie repräsentiert eine leichtgewichtige Variante von [25, S. 322].

Eingabeparameter *Methode*, *Basisklasse*.

Durchführung

1. Die Methode *Methode* in die Klasse *Basisklasse* verschieben. (1 + 0)
2. Sichtbarkeit von *Methode* auf höher als privat setzen. (1 + 0)

Gründe für den Fehlschlag der Operation

- Die Klasse von *Methode* ist als unveränderlich markiert.
- *Basisklasse* ist als unveränderlich markiert.
- *Basisklasse* ist keine Basisklasse der Klasse von *Methode*.
- *Methode* greift auf Instanzattribute zu, die nicht über *Basisklasse* erreichbar sind.
- *Methode* ist als unveränderlich markiert.

Strukturelle Garantien

- Die Klassenhierarchie bleibt unverändert.
- Existierende Attribute bleiben unverändert.
- Nicht betroffene Methoden bleiben unverändert.

5.2.7 Hilfsumbauoperation: Methode verschieben

Beschreibung Diese Operation verschiebt eine Methode einer Klasse in eine andere Klasse. Sie repräsentiert eine leichtgewichtige Variante von [25, S. 142].

Eingabeparameter *Methode*, *Klasse*.

Durchführung

1. Die Methode *Methode* in die Klasse *Klasse* verschieben. (1 + 0)

Gründe für den Fehlschlag der Operation

- Die Klasse von *Methode* ist als unveränderlich markiert.
- *Klasse* ist als unveränderlich markiert.
- *Methode* ist als unveränderlich markiert.
- *Methode* wird von Methoden in abgeleiteten Klassen überschrieben.

Strukturelle Garantien

- Die Klassenhierarchie bleibt unverändert.
- Existierende Attribute bleiben unverändert.
- Nicht betroffene Methoden bleiben unverändert.

5.2.8 Hilfsumbauoperation: Attribut verschieben

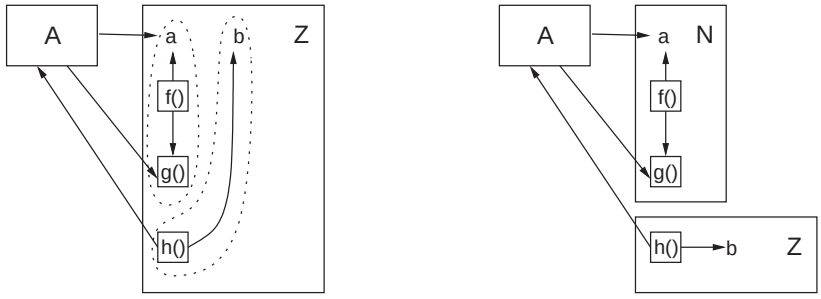
Beschreibung Diese Operation verschiebt ein Attribut einer Klasse in eine andere Klasse. Sie repräsentiert eine leichtgewichtige Variante von [25, S. 146].

Eingabeparameter *Attribut*, *Klasse*.

Durchführung

1. Das Attribut *Attribut* in die Klasse *Klasse* verschieben. (1 + 0)

Diese Hilfsoperation nimmt keine Rücksicht auf die Konsistenz des Modells, sodass unerlaubte Verschiebungen möglich sind. Die Prüfung beziehungsweise Wiederherstellung der Konsistenz obliegt dem Aufrufer.



(a) Vor der Aufteilung (b) Nach der Aufteilung

Abbildung 5.1: Beispiel für Aufteilung von Artefakten

Gründe für den Fehlschlag der Operation

- Die Klasse von *Attribut* ist als unveränderlich markiert.
- *Attribut* ist als unveränderlich markiert.
- *Klasse* ist als unveränderlich markiert.

Strukturelle Garantien

- Die Klassenhierarchie bleibt unverändert.
- Existierende Attribute bleiben unverändert.
- Nicht betroffene Methoden bleiben unverändert.

5.2.9 Hilfsumbauoperation: Klassenartefakte aufteilen

Beschreibung Diese Operation verschiebt alle transitiv zu einer Ausgangsklasse abhängigkeitsverursachenden Artefakte einer Klasse in eine dritte Klasse.

Abbildung 5.1 illustriert die Möglichkeit einer Durchführung. `a` und `b` sind Attribute, `f()`, `g()` und `h()` Methoden. Eine Aufteilung ist nur möglich, wenn sich je zwei voneinander unabhängige Artefaktmengen finden lassen (in der Abbildung strichliert eingekreist), von denen die eine nur zu A ausgehende, die andere nur von A eingehende Beziehungen unterhält.

Eingabeparameter A, Z, N.

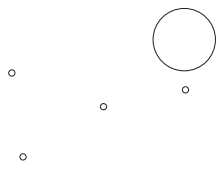
Die Funktion Methoden(K) liefert eine Menge aller Methoden der Klasse K, die Funktion Attribute(K) eine Menge aller Attribute der Klasse K zurück.

Durchführung

1. Sei $\text{Abh}_\text{M}(m)$ eine Funktion, die die Methode m auf eine Menge aller Methoden abbildet, von welchen diese Methode transitiv abhängt und die in derselben Klasse wie m liegen. (1 + 0)
2. Sei $\text{Abh}_\text{A}(m)$ eine Funktion, die die Methode m auf eine Menge aller Attribute abbildet, von welchen diese Methode transitiv abhängt und die in derselben Klasse wie m liegen. (1 + 0)
3. Menge P aller Methoden in Z ermitteln, von denen A direkt abhängig ist. (1 + 0)
4. Menge Q aller Attribute in Z ermitteln, von denen A direkt abhängig ist. (1 + 0)
5. P mit transitiver Hülle der Methodenabhängigkeiten aller Methoden in P zu P' vereinigen: $P' \leftarrow P \cup \{m \in \text{Methoden}(Z) | m \in \bigcup_{n \in P} \text{Abh}_\text{M}(n)\}$. (1 + 0)
6. Q mit transitiver Hülle der Attributabhängigkeiten aller Methoden in P zu Q' vereinigen: $Q' \leftarrow Q \cup \{a \in \text{Attribute}(Z) | a \in \bigcup_{n \in P} \text{Abh}_\text{A}(n)\}$. (1 + 0)
7. Jede Methode $m \in P'$ nach N verschieben (Methode verschieben(m, N) ☞5.2.7). (1 + 1)
8. Jedes Attribut $a \in Q'$ nach N verschieben (Attribut verschieben(a, N) ☞5.2.8). (1 + 1)
9. Beziehungen aller nach N verschobenen Artefakte nachziehen (Beziehungen verschieben(Z, N) ☞5.2.4). (1 + 9)

Gründe für den Fehlschlag der Operation

- Die Klasse von a ist als unveränderlich markiert [aus ☞5.2.8].
- Die Klasse von m ist als unveränderlich markiert [aus ☞5.2.7].
- Eine Methode einer von Z abgeleiteten Klasse greift auf ein Attribut in Q' zu.
- Eine Methode einer von Z abgeleiteten Klasse ruft eine Methode in P' auf.
- Eine Methode in P' greift auf ein Attribut einer Basisklasse von Z zu.
- Eine Methode in P' ist in einer Unterklasse von Z überschrieben.
- Eine Methode in P' ruft eine Methode einer Basisklasse von Z auf.
- Eine Methode in P' überschreibt eine Methode einer Basisklasse von Z.



- A ist als unveränderlich markiert.
- N enthält bereits zu verschiebende Artefakte desselben Primärnamens.
- N ist als unveränderlich markiert [aus ☞5.2.8, ☞5.2.4, ☞5.2.7].
- N ist unvollständig [aus ☞5.2.4].
- Z ist als unveränderlich markiert.
- Z ist nicht separierbar: $P' = \text{Methoden}(Z) \wedge Q' = \text{Attribute}(Z)$.
- a ist als unveränderlich markiert [aus ☞5.2.8].
- m ist als unveränderlich markiert [aus ☞5.2.7].
- m wird von Methoden in abgeleiteten Klassen überschrieben [aus ☞5.2.7].
- *Widerspruch* ist nicht leer [aus ☞5.2.4].

Strukturelle Garantien

- Die Klassenhierarchie bleibt unverändert.

Variante 1 Die Abhängigkeit zur Ausgangsklasse ist umgekehrt. Verschoben werden jene Klassen, die transitiv von der Ausgangsklasse A abhängen. Hierzu sind folgende Regeländerungen erforderlich:

3. Menge P aller Methoden in Z ermitteln, die direkt von A abhängen. (1 – 1)

4. (entfällt) (0 – 1)

5.2.10 Hilfsumbauoperation: Rückruf vorbereiten

Beschreibung Bereitet die Umwandlung von Aufrufbeziehungen zu einer Methode in einen Rückruf vor, indem eine Variable und eine entsprechende Adapterfunktion angelegt werden.

Eingabeparameter *Methode*.
Die Funktion Ziel(b) liefert das Zielartefakt der Beziehung b.

Durchführung

1. Sei $s = \langle s_1, s_2, \ldots, s_n \rangle$ die Signatur der Methode *Methode*, dann sei $t = \langle k, s_1, s_2, \ldots, s_n \rangle$, wobei k einen Referenztyp auf die Klasse von *Methode* darstellt. (1 + 0)
2. Globale Funktion f mit behältereindeutigem Namen und Signatur t erzeugen. (1 + 0)
3. Globale Variable g mit behältereindeutigem Namen und Typ f erzeugen. (1 + 0)
4. Eine Aufrufbeziehung b von $f \rightarrow$ *Methode* erzeugen. (1 + 0)
5. g mit f in einem globalen Konstruktor initialisieren. (1 + 0)

Gründe für den Fehlschlag der Operation

- *Methode* ist privat oder anderweitig von f nicht aufrufbar.

Strukturelle Garantien

- Existierende Artefakte und Beziehungen werden nicht verändert.

Ausgabe g.

5.3 Auflösungstechniken

Dieses Kapitel führt die Spezifikationen aller untersuchten Auflösungstechniken auf. Abbildung 5.2 zeigt die Aufrufabhängigkeiten der Auflösungstechniken von den primitiven Umbauoperationen. Die doppelt umrandeten Rechtecke stellen Auflösungstechniken dar, die einfach umrandeten Umbauoperationen.

Die Auflösungstechniken dienen an sich nicht der Zyklenauflösung, sondern lediglich dem Aufbrechen einzelner Kanten, sodass Artefakte aus einer Zyklengruppe herausgelöst werden können. Für die nachfolgenden Beschreibungen der Auflösungstechniken wird daher als Eingabe stets das Anfangsprimärartefakt A und das Endprimärartefakt Z angegeben, zwischen welchen die gerichtete aggregierte Abhängigkeit $A \rightarrow Z$ aufzulösen ist.

Das vollständige Aufbrechen einer gesamten Zyklengruppe verlangt somit das wiederholte Anwenden von Auflösungstechniken auf verschiedene Kanten, und selbst dies vermag Zyklengruppen in der Regel nur zu verkleinern oder aufzuspalten, sie jedoch nicht restlos zu beseitigen.

Das verwendete Schema entspricht dem Schema der Umbauoperationen in Kapitel 5.2, da die Auflösungstechniken ebenfalls über Umbauoperationen spezifiziert werden.

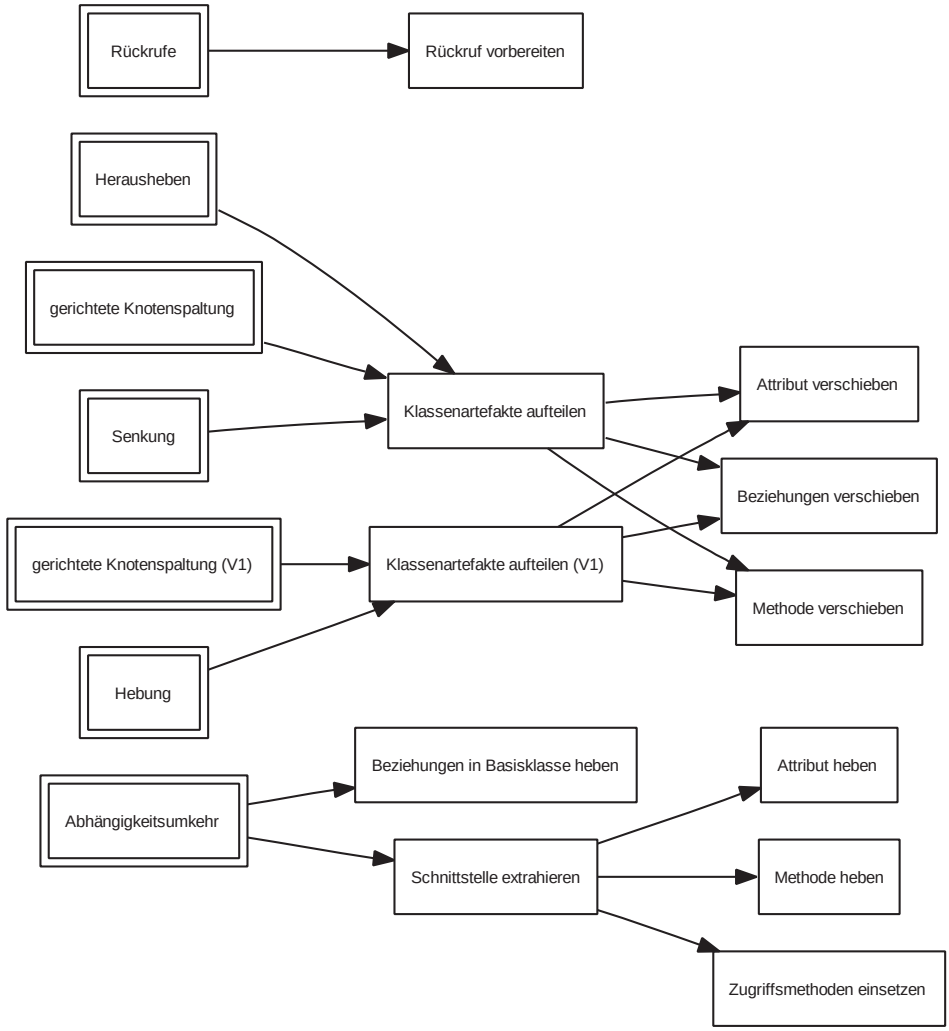


Abbildung 5.2: Aufrufabhängigkeiten der Auflösungsverfahren

5.3.1 Spezifikation der Abhängigkeitsumkehr

Eingabeparameter A, Z.

Durchführung

1. Schnittstelle extrahieren(Z, `I`Z) (☞5.2.1) $\rightarrow S$. (1 + 17)
2. Beziehungen in Basisklasse heben(Z, S) (☞5.2.3) (1 + 5)

Gründe für den Fehlschlag der Operation

- Die Klasse von a ist als unveränderlich markiert [aus ☞5.2.5, ☞5.2.2].

- Die Klasse von m ist als unveränderlich markiert [aus ☞5.2.6].
- Z ist als unveränderlich markiert [aus ☞5.2.3, ☞5.2.1].
- Z ist unvollständig [aus ☞5.2.3].
- a ist als unveränderlich markiert [aus ☞5.2.5].
- m greift auf Instanzattribute zu, die nicht über n erreichbar sind [aus ☞5.2.6].
- m ist als unveränderlich markiert [aus ☞5.2.6].
- n ist als unveränderlich markiert [aus ☞5.2.5, ☞5.2.6].
- n ist keine Basisklasse der Klasse von a [aus ☞5.2.5].
- n ist keine Basisklasse der Klasse von m [aus ☞5.2.6].

Strukturelle Garantien

- Die entworfene Klassenhierarchie bleibt unverändert.
- Das Verhalten bleibt unverändert.

Das Verhalten bleibt unverändert, da »Schnittstelle extrahieren« keine Verhaltensänderungen bewirkt und durch das Heben möglicher Beziehungen in die neu erzeugte Schnittstelle S, deren einzige Implementierung Z ist, nach wie vor der ursprüngliche Code ausgeführt wird.

5.3.2 Spezifikation der gerichteten Knotenspaltung

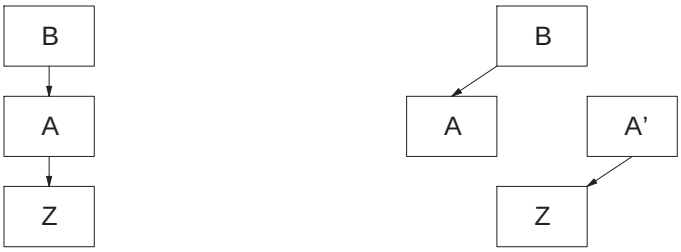
Eine allgemeine Lösung der gerichteten Knotenspaltung ist, wie bereits in Kapitel 3.5.2 erwähnt, auf Klassenebene zu komplex und nicht ohne menschliches Zutun durchführbar. Wir beschränken uns daher auf die Spezifizierung einer einfacheren Variante, die eine Spaltung nur dann vornimmt, wenn die zu spaltende Klasse *ohne* Zerteilung einer Methode einer Spaltung zugeführt werden kann.

Abbildung 5.3 illustriert die spezifizierte Durchführung der Knotenspaltung. Die Auflösung einer zyklischen Abhängigkeit funktioniert dann, wenn durch die Spaltung von A im Kreis $Z \rightarrow Y \rightarrow \cdots \rightarrow B \rightarrow A \rightarrow Z\ B$ auf A' umgebogen wurde.

Eingabeparameter A, Z.

Durchführung

1. Sei B eine Klasse, die von A abhängig ist und in derselben Zyklengruppe wie Z liegt. (1 + 0)
2. Neue Klasse A' erzeugen. (1 + 0)
3. Klassenartefakte aufteilen(B, A, A') (☞5.2.9). (1 + 20)



(a) Vor der Spaltung (b) Nach der Spaltung

Abbildung 5.3: Beispiel zur Spezifikation der gerichteten Knotenspaltung

Gründe für den Fehlschlag der Operation

- Die Klasse von a ist als unveränderlich markiert [aus ☞5.2.8].
- Die Klasse von m ist als unveränderlich markiert [aus ☞5.2.7].
- Eine Methode einer von A abgeleiteten Klasse greift auf ein Attribut in Q' zu [aus ☞5.2.9].
- Eine Methode einer von A abgeleiteten Klasse ruft eine Methode in P' auf [aus ☞5.2.9].
- Eine Methode in P' greift auf ein Attribut einer Basisklasse von A zu [aus ☞5.2.9].
- Eine Methode in P' ist in einer Unterklasse von A überschrieben [aus ☞5.2.9].
- Eine Methode in P' ruft eine Methode einer Basisklasse von A auf [aus ☞5.2.9].
- Eine Methode in P' überschreibt eine Methode einer Basisklasse von A [aus ☞5.2.9].
- A ist als unveränderlich markiert [aus ☞5.2.9].
- A ist nicht separierbar: $P' = \text{Methoden}(A) \wedge Q' = \text{Attribute}(A)$ [aus ☞5.2.9].
- A' enthält bereits zu verschiebende Artefakte desselben Primärnamens [aus ☞5.2.9].
- A' ist als unveränderlich markiert [aus ☞5.2.8, ☞5.2.4, ☞5.2.7].
- A' ist unvollständig [aus ☞5.2.4].
- B ist als unveränderlich markiert [aus ☞5.2.9].

(a) Vor der Hebung

(b) Nach der Hebung

Abbildung 5.4: Beispiel zur Spezifikation der Hebung

- a ist als unveränderlich markiert [aus ☞5.2.8].
- m ist als unveränderlich markiert [aus ☞5.2.7].
- m wird von Methoden in abgeleiteten Klassen überschrieben [aus ☞5.2.7].
- *Widerspruch* ist nicht leer [aus ☞5.2.4].

Strukturelle Garantien

- Die entworfene Klassenhierarchie bleibt unverändert.
- Das Verhalten bleibt unverändert.

Das Verhalten bleibt durch die Erzeugung der neuen Klasse A', in die die aufgeteilten Artefakte verschoben werden, und durch die Garantie unverändert, dass für eine erfolgreiche Durchführung der Operation tatsächlich die Typen aller angebundenen Ausdrücke von sich auf in der neuen Klasse befindlichen Artefakten mit dem Typ der neuen Klasse ersetzen haben lassen.

Variante 1 Die Abhängigkeitsrichtung zur Bestimmung der zu verschiebenden Artefakte ist hier umgekehrt. Hierzu sind folgende Regeländerungen erforderlich:

3. Klassenartefakte aufteilen(B, A, A') nach Variante 1 (☞5.2.9). (1 – 2)

5.3.3 Spezifikation der Hebung

Eine allgemeine Lösung der Hebung (Kap. 3.5.3) erfordert die sinnvolle Benennung neuer Klassen und ist nicht ohne menschliches Zutun durchführbar. Wir beschränken uns daher auf die Spezifizierung einer einfacheren Variante, die eine Hebung nur dann vornimmt, wenn sie *ohne* Zerteilung von Methoden möglich ist.

Aus Abbildung 5.4 ist die Auflösung der Abhängigkeit $A \rightarrow Z$ durch die Hebung ersichtlich.

Eingabeparameter A, Z.

Durchführung

1. Neue Klasse Z' erzeugen. (1 + 0)
2. Klassenartefakte aufteilen(A, Z, Z') nach Variante 1 (☞5.2.9). (1 + 19)
3. Neue Klasse A' erzeugen. (1 + 0)
4. Klassenartefakte aufteilen(Z, A, A') nach Variante 1 (☞5.2.9) (1 + 0)

Gründe für den Fehlschlag der Operation

- Die Klasse von a ist als unveränderlich markiert [aus ☞5.2.8].
- Die Klasse von m ist als unveränderlich markiert [aus ☞5.2.7].
- A' ist als unveränderlich markiert [aus ☞5.2.8, ☞5.2.4, ☞5.2.7].
- A' ist unvollständig [aus ☞5.2.4].
- a ist als unveränderlich markiert [aus ☞5.2.8].
- m ist als unveränderlich markiert [aus ☞5.2.7].
- m wird von Methoden in abgeleiteten Klassen überschrieben [aus ☞5.2.7].
- *Widerspruch* ist nicht leer [aus ☞5.2.4].

Strukturelle Garantien

- Die entworfene Klassenhierarchie bleibt unverändert.
- Das Verhalten bleibt unverändert.

Das Verhalten bleibt durch die Erzeugung neuer Klassen A' und Z' unverändert, in die die aufgeteilten Artefakte verschoben werden und durch die Garantie, dass sich für eine erfolgreiche Durchführung der Operation tatsächlich alle angebundenen Ausdrücke von auf in die neue Klasse 'Z verschobenen Artefakte zeigenden Beziehungen ersetzen haben lassen.

5.3.4 Spezifikation der Senkung

Eine allgemeine Lösung der Senkung (Kap. 3.5.4) erfordert die sinnvolle Benennung neuer Klassen und ist nicht ohne menschliches Zutun durchführbar. Wir beschränken uns daher auf die Spezifizierung einer einfacheren Variante, die eine Senkung nur dann vornimmt, wenn sie *ohne* Zerteilung von Methoden möglich ist.

Aus Abbildung 5.5 ist die Auflösung der Abhängigkeit $A \rightarrow Z$ durch die Senkung ersichtlich.

(a) Vor der Senkung

(b) Nach der Senkung

Abbildung 5.5: Beispiel zur Spezifikation der Senkung

Eingabeparameter A, Z.

Durchführung

1. Neue Klasse Z' erzeugen. (1 + 0)
2. Klassenartefakte aufteilen(A, Z, Z') (☞5.2.9). (1 + 20)
3. Neue Klasse A' erzeugen. (1 + 0)
4. Klassenartefakte aufteilen(Z, A, A') (☞5.2.9) (1 + 0)

Gründe für den Fehlschlag der Operation

- Die Klasse von a ist als unveränderlich markiert [aus ☞5.2.8].
- Die Klasse von m ist als unveränderlich markiert [aus ☞5.2.7].
- Eine Methode einer von A abgeleiteten Klasse greift auf ein Attribut in Q' zu [aus ☞5.2.9].
- Eine Methode einer von A abgeleiteten Klasse ruft eine Methode in P' auf [aus ☞5.2.9].
- Eine Methode in P' greift auf ein Attribut einer Basisklasse von A zu [aus ☞5.2.9].
- Eine Methode in P' ist in einer Unterklasse von A überschrieben [aus ☞5.2.9].
- Eine Methode in P' ruft eine Methode einer Basisklasse von A auf [aus ☞5.2.9].
- Eine Methode in P' überschreibt eine Methode einer Basisklasse von A [aus ☞5.2.9].
- A ist als unveränderlich markiert [aus ☞5.2.9].

(a) Ohne Rückruf (b) Mit Rückruf

Abbildung 5.6: Beispiel zur Spezifikation der Rückrufe

- A ist nicht separierbar: $P' = \text{Methoden}(A) \wedge Q' = \text{Attribute}(A)$ [aus ☞5.2.9].
- A' enthält bereits zu verschiebende Artefakte desselben Primärnamens [aus ☞5.2.9].
- A' ist als unveränderlich markiert [aus ☞5.2.8, ☞5.2.4, ☞5.2.7].
- A' ist unvollständig [aus ☞5.2.4].
- a ist als unveränderlich markiert [aus ☞5.2.8].
- m ist als unveränderlich markiert [aus ☞5.2.7].
- m wird von Methoden in abgeleiteten Klassen überschrieben [aus ☞5.2.7].
- *Widerspruch* ist nicht leer [aus ☞5.2.4].

Strukturelle Garantien

- Die entworfene Klassenhierarchie bleibt unverändert.
- Das Verhalten bleibt unverändert.

Das Verhalten bleibt durch die Erzeugung neuer Klassen A' und Z' unverändert, in die die aufgeteilten Artefakte verschoben werden und durch die Garantie, dass sich für eine erfolgreiche Durchführung der Operation tatsächlich alle angebundenen Ausdrücke von auf in die neue Klasse 'Z verschobenen Artefakte zeigenden Beziehungen ersetzen haben lassen.

5.3.5 Spezifikation der Rückrufe

Die Auflösung von Abhängigkeiten durch Rückrufe (Kap. 3.5.6) erfolgt durch das Ersetzen von direkten Aufrufbeziehungen durch reine Namensbeziehungen. Die Kopplung zwischen zwei Artefakte wird dadurch aufgehoben.

Abbildung 5.6 zeigt die Auflösung der Abhängigkeit $A \rightarrow Z$ durch Rückrufe. Die Abhängigkeit verschwindet, da `f()` nicht statisch an `g` gekoppelt ist.

Eingabeparameter A, Z.

Durchführung

1. Für jede Methode $m \in$ Methoden(Z): (1 + 0)

1a Rückruf vorbereiten(m) (☞5.2.10) $\rightarrow g$. (1 + 5)

1b Sei l die Menge aller Aufrufbeziehungen zu m, deren jeweiliges Quellartefakt sich in A befindet. (1 + 0)

1c Jede Beziehung $b \in l$ von m auf indirekten Aufruf von g umbiegen. (1 + 0)

Gründe für den Fehlschlag der Operation

- A greift auf Attribute von Z zu.
- m ist privat oder anderweitig von f nicht aufrufbar [aus ☞5.2.10].

Wenn l unveränderliche Beziehungen enthält oder die den Beziehungen zugrundeliegenden Quellmethoden unveränderlich sind, werden diese von der Umlenkung auf den Rückruf ausgenommen. Dies führt nicht zum Fehlschlag der Gesamtoperation.

Strukturelle Garantien

- Die Klassenhierarchie wird nicht verändert.
- Das Verhalten wird nicht geändert.

5.3.6 Spezifikation des Heraushebens

Zur Auflösung zyklischer Abhängigkeiten mittels Heraushebens (Kap. 3.5.8) wird eine vereinfachte, automatisierbare Variante vorgestellt. Das Zielartefakt wird von einer neuen Klasse abgeleitet, in die sämtliche Methoden und Attribute verschoben werden, die nicht transitiv vom Quellartefakt abhängen. Das Herausheben verhält sich analog zur gerichteten Knotenspaltung (☞5.3.2) mit der Ergänzung, dass Paketgrenzen in die Auflösungstechnik einbezogen werden.

Aus Abbildung 5.7 ist die Auflösung der Abhängigkeit $A \rightarrow Z$ durch Herausheben ersichtlich.

Eingabeparameter A, Z.

Die Funktion Paket(x) bildet das Artefakt x auf das x enthaltende Paket (bzw. Namensraum) ab.

(a) Vor dem Herausheben (b) Nach dem Herausheben

Abbildung 5.7: Beispiel zur Spezifikation des Heraushebens

Durchführung

1. Neue Klasse Z' in Paket Paket(Z) erstellen. (1 + 0)
2. Klassenartefakte aufteilen(A, Z, Z') (☞5.2.9). (1 + 20)
3. Z von Z' ableiten. (1 + 0)
4. Z in Paket Paket(A) verschieben. (1 + 0)

Gründe für den Fehlschlag der Operation

- Paket$(Z) =$ Paket(A).
- Die Klasse von a ist als unveränderlich markiert [aus ☞5.2.8].
- Die Klasse von m ist als unveränderlich markiert [aus ☞5.2.7].
- Eine Methode einer von Z abgeleiteten Klasse greift auf ein Attribut in Q' zu [aus ☞5.2.9].
- Eine Methode einer von Z abgeleiteten Klasse ruft eine Methode in P' auf [aus ☞5.2.9].
- Eine Methode in P' greift auf ein Attribut einer Basisklasse von Z zu [aus ☞5.2.9].
- Eine Methode in P' ist in einer Unterklasse von Z überschrieben [aus ☞5.2.9].
- Eine Methode in P' ruft eine Methode einer Basisklasse von Z auf [aus ☞5.2.9].
- Eine Methode in P' überschreibt eine Methode einer Basisklasse von Z [aus ☞5.2.9].
- A ist als unveränderlich markiert [aus ☞5.2.9].
- Z ist als unveränderlich markiert [aus ☞5.2.9].

- Z ist bereits von einer anderen Klasse (die keine Schnittstelle ist) abgeleitet.
- Z ist nicht separierbar: $P' = \text{Methoden}(Z) \wedge Q' = \text{Attribute}(Z)$ [aus ☞5.2.9].
- Z' enthält bereits zu verschiebende Artefakte desselben Primärnamens [aus ☞5.2.9].
- Z' ist als unveränderlich markiert [aus ☞5.2.8, ☞5.2.4, ☞5.2.7].
- Z' ist unvollständig [aus ☞5.2.4].
- a ist als unveränderlich markiert [aus ☞5.2.8].
- m ist als unveränderlich markiert [aus ☞5.2.7].
- m wird von Methoden in abgeleiteten Klassen überschrieben [aus ☞5.2.7].
- *Widerspruch* ist nicht leer [aus ☞5.2.4].

Strukturelle Garantien

- Das Verhalten wird nicht verändert.

Das Verhalten wird nicht verändert, da die abgespaltenen Artefakte in 'Z durch Vererbung in Z wieder zur Verfügung stehen.

5.3.7 Nicht spezifizierte Techniken

Die Auflösungstechniken Verwalterklassen (Kap. 3.5.7), Redundanz (Kap. 3.5.5) und der aspektorientierten Zyklenauflösung (Kap. 3.5.9) werden aus folgenden Gründen nicht spezifiziert.

Die Verwalterklassen stellen lediglich einen Spezialfall der Hebung (Kap. 3.5.3) dar, in dem die Bezeichnung *Verwalterklasse* die rein semantische Bedeutung besitzt, eine Klasse zu verwalten. Ob eine Beziehung eine »Verwaltungsbeziehung« darstellt, lässt sich nicht automatisch feststellen.

Andernfalls verhält sich die Erstellung einer Verwalterklasse analog zur Hebung, weswegen auf eine Spezifikation verzichtet werden kann.

Bei der Redundanz kann nicht garantiert werden, dass das Einkopieren von Artefakten einer Klasse in eine andere das Programmverhalten nicht ändert. Eine automatisierte Implementierung der Redundanz ist zwar möglich, verlangt aber in jedem Fall eine manuelle Nachprüfung. Sie ist damit als automatisch anwendbare Auflösungstechnik ungeeignet.

Für die aspektorientierte Zyklenauflösung müssen zuerst die Aspekte, unter welchen ein Softwaresystem betrachtet wird, gefunden und spezifiziert werden. Da die aspektorientierte Softwareentwicklung wie bereits in Kapitel 3.5.9 erwähnt noch keine Breitenwirkung erzielte und außerdem den Rahmen dieser Arbeit sprengt, werden Aspekte nicht weiter verfolgt.

5.4 Durchführung und Interpretation der Aufwandsschätzung

Nachdem alle Umbauoperationen spezifiziert, die Metamodellanforderungen eruiert und die Durchführungsschritte aufgeführt sind, kann nun der Aufwand zur Implementierung der Auflösungstechniken geschätzt werden. Tabelle 5.3 listet sämtliche Umbauoperationen, deren unmittelbaren und mittelbaren Aufwand, deren unmittelbare Abhängigkeiten von anderen Umbauoperationen sowie die mittelbar notwendigen Metamodellanforderungen auf. Weiters enthält die Tabelle den Gesamtaufwand interessanter Kombinationen von Auflösungstechniken.

Beispiel Als Beispiel einer Berechnung wollen wir den Aufwand in Punkten für die Implementierung der Umbauoperationen KNO und HER Schritt für Schritt ermitteln. Wir benutzen die dafür vorgesehene Funktion GesAufwand({KNO, HER}).

Der Gesamtaufwand setzt sich aus der Summe der Punkte aller verwendeten Durchführungsschritte, aller verwendeten Metamodellanforderungen sowie aller verwendeten Infrastrukturklassen zusammen. Daher müssen wir zuerst die Menge aller von {KNO, HER} verwendeten Durchführungsschritte, Metamodellanforderungen und Infrastrukturklassen ermitteln, indem wir die Funktion $(d, a, i) \mapsto \text{Abh}_\text{T}(\{\text{KNO}, \text{HER}\})$ aufrufen und ein entsprechendes Tupel mit den jeweiligen Mengen enthalten.

Zuerst ermitteln wir die transitive Hülle V aller für {KNO, HER} benötigten Umbauoperation. Sie ist Tabelle 5.3 zu entnehmen und lautet V = {KNO, HER, AA, AV, MV, BV}.

Dann berechnen wir die Aufwände für die Durchführungsschritte einer jeden Umbauoperation in V. Die Durchführungsmenge d ist für die Berechnung unpraktisch, da sie 27 Durchführungsschritte umfasst und wir nur am Aufwand der Durchführungsschritte für jede Operation interessiert sind. Stattdessen ermitteln wir direkt eine Menge d_W, die für jede Umbauoperation in V den Aufwand der unmittelbaren Durchführungsschritte enthält. Dazu fügen wir von jeder Umbauoperation in V die entsprechende unmittelbare Punktezahl aus Spalte **Pkte** (die Zahl außerhalb der Klammer) der Tabelle 5.3 zu d_W hinzu (effektive Anwendung der Funktion P_D) und erhalten die Menge d_W = {3, 4, 9, 1, 1, 9}.

Danach stellen wir die Menge aller verwendeten Metamodellanforderungen für alle Umbauoperationen in V zusammen. Tabelle 5.3 enthält bereits die transitiven Hüllen der Anforderungen je Umbauoperation, sodass wir nur noch die Mengen von KNO und HER vereinigen müssen: a = {S1, S2, S3, S4, S5, S9, S10, S11, B2, C1, C2, I1, I2}.

Zuletzt ermitteln wir noch die Menge der involvierten Infrastrukturklassen. Das erste Zeichen eines Kürzels in a repräsentiert jeweils die Infrastrukturklasse gemäß Tabelle 5.1. Damit können wir die transitive Hülle aller benötigten Infrastrukturklassen als i = {S, B, C, I} ablesen.

Kenn	Bezeichnung	Pkte	Benötigt	Metamodellanforderungen
AH	Attribut heben	2 (2)	–	S1 S4 S6
AHa	Attribut heben (V1)	1 (1)	–	S1 S4 S6
AV	Attribut verschieben	1 (1)	–	S1 S4
BH	Beziehungen in Basisklasse heben	5 (5)	–	S1 S2 S3 B2 C1 C2
BV	Beziehungen verschieben	9 (9)	–	S1 S2 S3 B2 C1 C2
MH	Methode heben	2 (2)	–	S1 S5 S6
MV	Methode verschieben	1 (1)	–	S1 S5
RV	Rückruf vorbereiten	5 (5)	–	S5 S7 S8 B2 B3 B4 C1 C2
ZE	Zugriffsmethoden einsetzen	6 (6)	–	S1 S2 S3 S4 S5 B1
ZEa	Zugriffsmethoden einsetzen (V1)	5 (5)	–	S1 S2 S3 S4 S5 B1
AA	Klassenartefakte aufteilen	9 (20)	AV BV MV	S1 S2 S3 S4 S5 B2 C1 C2 I1 I2
AAa	Klassenartefakte aufteilen (V1)	8 (19)	AV BV MV	S1 S2 S3 S4 S5 B2 C1 C2 I1 I2
SE	Schnittstelle extrahieren	7 (17)	AH MH ZE	S1 S2 S3 S4 S5 S6 B1 F1 F2
HEB	Hebung	4 (23)	AAa	S1 S2 S3 S4 S5 B2 C1 C2 I1 I2
HER	Herausheben	4 (24)	AA	S1 S2 S3 S4 S5 S9 S10 S11 B2 C1 C2 I1 I2
KNO	gerichtete Knotenspaltung	3 (23)	AA	S1 S2 S3 S4 S5 B2 C1 C2 I1 I2
KNOa	gerichtete Knotenspaltung (V1)	3 (22)	AAa	S1 S2 S3 S4 S5 B2 C1 C2 I1 I2
RUE	Rückrufe	4 (9)	RV	S5 S7 S8 B2 B3 B4 C1 C2
SEN	Senkung	4 (24)	AA	S1 S2 S3 S4 S5 B2 C1 C2 I1 I2
ABH	Abhängigkeitsumkehr	2 (24)	BH SE	S1 S2 S3 S4 S5 S6 B1 B2 F1 F2 C1 C2

Tabelle 5.3: Übersicht über alle Umbauoperationen
Kenn Kennung der Umbauoperation, **Bezeichnung** Name der Umbauoperation, **Pkte** Punktsumme der Durchführungsliste (Gesamtpunktsumme aller abhängigen Durchführungslisten), **Benötigt** Kennungen von Umbauoperationen, von welchen die Operation abhängt, **Metamodellanforderungen** Liste von Metamodellanforderungskürzeln gemäß Tab. 5.2.

Nun bilden wir für die Menge a eine Menge a_W, die die Punkte für jede involvierte Metamodellanforderung enthält, indem wir für jede Anforderung x in a die Funktion $P_A(x)$ aufrufen (d. h. die Aufwände aus Tabelle 5.2 ablesen) und erhalten folgendes Ergebnis a_W = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}.

Analog bilden wir i_W zur Menge der Infrastrukturklassen i, indem wir die Funktion P_I bemühen (d. h. die Aufwände Tabelle 5.1 entnehmen): i_W = {20, 20,

Beschreibung	Kombination	Punkte
Rückrufe	RUE	72
gerichtete Knotenspaltung	KNO	93
gerichtete Knotenspaltung (V1)	KNOa	93
Abhängigkeitsumkehr	ABH	94
Hebung	HEB	94
Senkung	SEN	94
Herausheben	HER	97
Gesamtaufwand	ABH-HEB-HER-KNO-KNOa-RUE-SEN	143
Dreierkombination mit geringstem Aufwand	HEB-KNO-KNOa	98
Dreierkombination mit höchstem Aufwand	ABH-HER-RUE	131

Tabelle 5.4: Zusammenfassung der Aufwandsberechnungen

20, 20}.

Damit können wir den Gesamtaufwand berechnen. Wir summieren sämtliche Punktwerte der Mengen a_W, d_W und i_W auf und erhalten als Gesamtsumme den geschätzten Gesamtaufwand der Implementierung der Auflösungstechniken {KNO, HER}:

$$\text{GesAufwand}(\{\text{KNO}, \text{HER}\}) = \sum a_W + \sum d_W + \sum i_W = 13 + 27 + 80 = 120$$

Interpretation Die Aufwände für die Implementierung wurden sowohl für jede einzelne Auflösungstechnik für sich berechnet als auch für alle Auflösungstechniken sowie alle Kombinationen von je drei Auflösungstechniken. Die Reihung der Aufwände wurde nur für die einzelnen Auflösungstechniken vorgenommen, von den Aufwänden kombinierter Techniken sind lediglich jene Kombinationen mit geringstem und höchstem Aufwand aufgeführt.

Von den einzelnen Auflösungstechniken schlägt »Rückrufe« mit 72 Punkten am günstigsten zu Buche, »Herausheben« mit 97 am ungünstigsten. Für die Mehrheit der Auflösungstechniken liegen die Aufwände nicht mehr als 4 Punkte auseinander, womit deren Implementierungsaufwand als praktisch gleich betrachtet werden kann. Der Gesamtaufwand beträgt 143 Punkte, die Dreierkombination mit dem günstigsten Aufwand von 98 ist HEB-KNO-KNOa, die mit dem höchsten Aufwand mit 131 ist ABH-HER-RUE. Tabelle 5.4 fasst die obigen Aussagen noch einmal übersichtlich zusammen.

Aus den gegebenen Zahlen lässt sich ablesen, dass die Implementierung aller Auflösungstechniken im Vergleich zur Implementierung der einfachsten Technik fast den doppelten Aufwand bedeutet (72+71=143: 98% zusätzlich). Ausgehend von der schwierigsten Auflösungstechnik beträgt der Zusatzaufwand nur noch 47% (97+46=143), da der größte Teil auf gemeinsamer Infrastruktur basiert und sich

weitere Auflösungstechniken ohne wesentlichen Zusatzaufwand implementieren lassen. Die ermittelten Punktwerte sind somit plausibel.

Es sei jedoch noch einmal darauf hingewiesen, dass die Punktwerte Rohwerte darstellen, die das projektspezifische Umfeld nicht miteinbeziehen. Dadurch können die Punktwerte nicht unmodifiziert als Spiegel des tatsächlichen Implementierungsaufwands betrachtet werden, sondern als einfache Richtlinie.

5.5 Auswahl zu implementierender Techniken

Wir spezifizierten in diesem Kapitel sechs Auflösungstechniken, die sich für die automatisierte Auflösung von Abhängigkeiten eignen. Nun ist es angebracht, auserwählte Auflösungstechniken zu implementieren und einige empirische Untersuchungen durchzuführen. Doch welche Auflösungstechniken kommen für eine Implementierung in Frage?

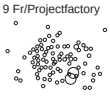
Es ist nicht zweckmäßig, alle Auflösungstechniken zu implementieren, insbesondere solche, die sich stark ähneln. Stattdessen ist es sinnvoller, sich auf *repräsentative* Techniken zu konzentrieren, die jeweils wenig Gemeinsamkeiten aufweisen.

Betrachten wir Abbildung 5.2 auf Seite 58, so fällt uns auf, dass sich die Auflösungstechniken nach gemeinsam verwendeten Umbauoperationen in Äquivalenzklassen einteilen lassen. Diese Äquivalenzklassen sind ABH, KNO-KNOa-HEB-SEN-HER und RUE.

Da ähnliche Effekte von Auflösungstechniken aus einer gemeinsamen Äquivalenzklasse zu erwarten sind, sollte aus jeder Äquivalenzklasse eine Auflösungstechnik exemplarisch implementiert und auf reale Softwaresysteme angewandt werden.

Die Äquivalenzklassen ABH und RUE enthalten nur je eine Auflösungstechnik, deswegen wählten wir sie direkt für eine Implementierung aus.

Für die Äquivalenzklasse KNO-KNOa-HEB-SEN-HER verwendeten wir die Aufwandsschätzungen aus Tabelle 5.4. Die gerichtete Knotenspaltung mit 93 Punkten versprach den geringsten Aufwand. Wir entschieden uns daher für die Implementierung der gerichteten Knotenspaltung als dritte Auflösungstechnik.



Kapitel 6

Entwurf

Für die empirische Forschung entwarfen wir ein prototypisches Werkzeug, das drei Auflösungstechniken zur Anwendung auf Java-Softwaresystemen implementiert und eine Eclipse-Komponente anbietet, die dem Softwareentwickler eine Schnittstelle zur Benutzung des Werkzeugs zur Verfügung stellt. Dieses Kapitel beschreibt den Entwurf des Werkzeugs.

Der Entwurf umfasst eine Beschreibung der Komponenten sowie deren Interaktion in verschiedenen Detailstufen. Zuerst beschreiben wir den Grobentwurf, dann die Strukturen und Interaktionen der verschiedenen Komponenten.

6.1 Grobentwurf

Das prototypische Werkzeug – interner Codename *Meinges* – besteht aus zwei Komponenten, dem Kern und der Eclipse-Integration (siehe Abb. 6.1). Der Kern ist von der Integration unabhängig.

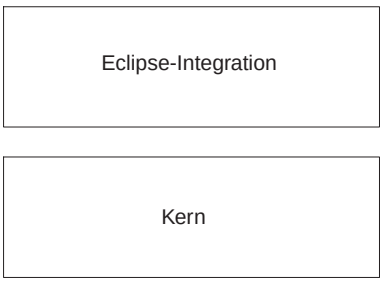


Abbildung 6.1: Grobentwurf des prototypischen Werkzeugs

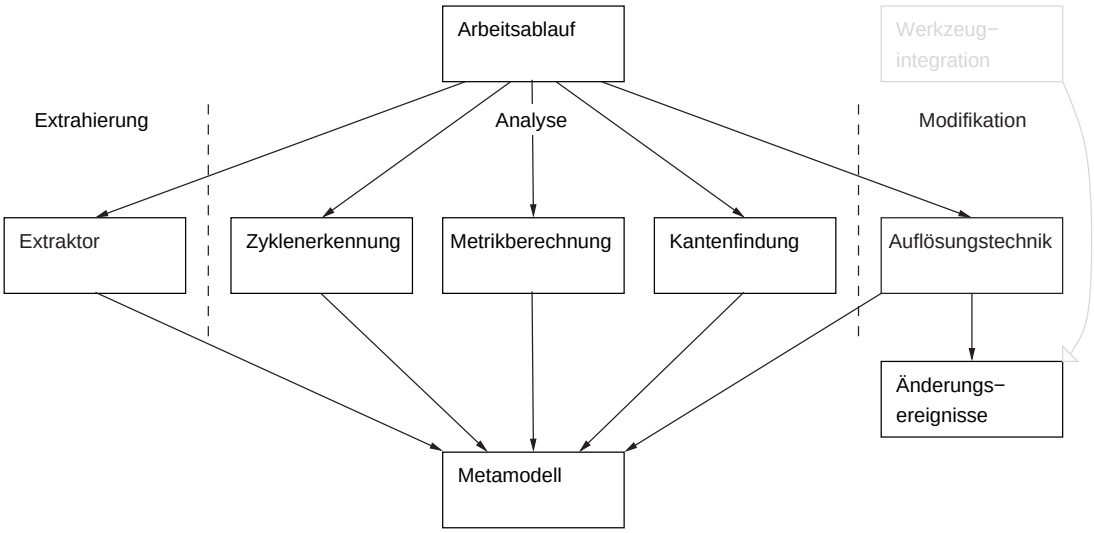


Abbildung 6.2: Kernkomponenten und ihre Abhängigkeiten

6.2 Komponentenstruktur

Softwareanalysewerkzeuge lassen sich den drei Bereichen Extrahierung, Analyse und Präsentation zuordnen [85]. Die Extrahierung überführt Informationen aus Quelltext und anderen Begleittexten in ein Modell, auf dem die weiterführenden Bereiche aufsetzen. Die Analyse wertet die Informationen im Modell aus. Die Präsentation stellt die Ergebnisse der Analyse dar und umfasst auch Interaktion. Ein Werkzeug kann einen oder mehrere Bereiche abdecken.

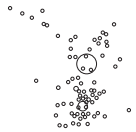
Der Kern besteht aus 8 Komponenten, deren Abhängigkeiten aus Abbildung 6.2 ersichtlich sind. Die verarbeitenden Komponenten gehören jeweils einem der Bereiche Extrahierung oder Analyse an. Der Bereich Modifikation wurde speziell für diese Untersuchung aus der Analyse abgespalten, da Analyseoperationen als nicht verändernde Beobachtungen aufgefasst werden.

Arbeitsablauf Steuert die automatische Ausführung von Meinges, um statistische Daten zu erlangen.

Metamodell Repräsentiert die Implementierung des FAMIX+-Metamodells zur Darstellung des zu untersuchenden Softwaresystems.

Extraktor Enthält einen Syntaxanalysator und semantische Aktionen zur Extrahierung von FAMIX+-Modellen aus Softwaresystemen. Die Extraktorschnittstelle ist erweiterbar konzipiert. Momentan existieren zwei Extraktoren, einer für Java-Quelltext und einer für Java-Bytecode.

Zyklenerkennung Enthält Algorithmen zur Ermittlung von zyklischen Abhängigkeiten in einem Softwaresystem.



	Signal
Elementare Umbauoperationen	Klasse erzeugen
	Von Klasse erben
	Methode erzeugen
	Attribut erzeugen
	Methode verschieben
	Attribut verschieben
	Methode extrahieren
	Attribut in Basisklasse heben
	Angebundenen Ausdruck ändern
	Typ angebundener Variable ändern
	Methoden-/Konstruktoraufruf einfügen
	Methoden-/Konstruktoraufruf ersetzen
	Fehler aufgetreten

Tabelle 6.1: Von Komponente Änderungsereignisse signalisierte Ereignisse.

Metrikberechnung Dient der Berechnung von Systemmetriken.

Kantenfindung Enthält Algorithmen zum Auffinden von Kanten, um Zyklengruppen aufzulösen.

Auflösungstechnik Umfasst die implementierten Auflösungstechniken (siehe Kap. 5.5), die tatsächliche Änderungen auf dem FAMIX+-Modell durchführen.

Änderungsereignisse Wenn eine Auflösungstechnik das FAMIX+-Modell ändert, teilt sie Art und Umfang der Änderung über die Schnittstelle Änderungsereignisse mit. Die Werkzeugintegration kann diese Schnittstelle implementieren (siehe die ausgegraute Komponente in Abb. 6.2) und die Änderungen auf den Quelltext übertragen. Tabelle 6.1 listet alle Signale auf, die die Änderungsereigniskomponente aussendet.

Die Änderungsereignisse dienen wesentlich der sauberen Trennung zwischen Kern und Werkzeugintegration.

6.3 Komponenteninteraktion

Das untersuchte Softwaresystem wird als FAMIX+-Modell (Kap. 4) repräsentiert. Sämtliche Operationen des Werkzeugs basieren auf der FAMIX+-Repräsentation. Abbildung 6.3 illustriert mögliche Ausführungsreihenfolgen der Operationen.

Die Operationen werden nun wie folgt ausgeführt.

1. Extrahieren (implementiert durch Komponente *Extraktor*) nimmt die Abbildung eines Softwaresystems auf ein FAMIX+-Modell vor und schreibt die Modelldaten.

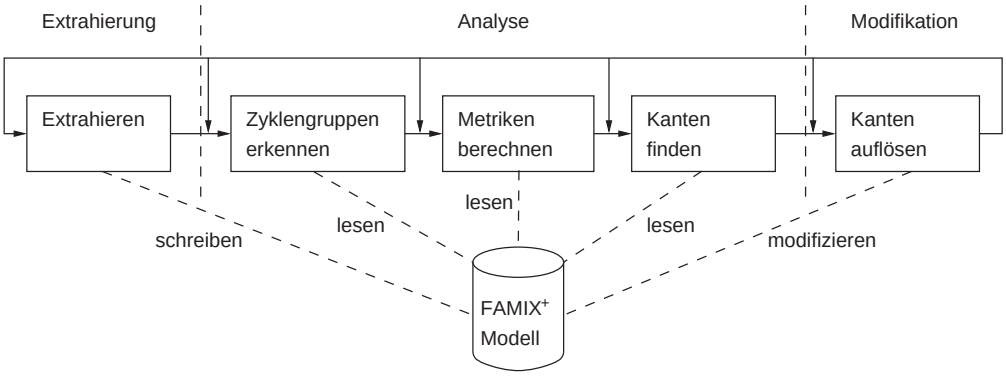


Abbildung 6.3: Reihung und Kopplung der verschiedenen Operationen im Werkzeug

2. Danach erkennt die Komponente *Zyklenerkennung* die vorhandenen Zyklengruppen.

3. Die Komponente *Metrikenberechnung* berechnet zum unterliegenden FAMIX+-Modell Systemmetriken. Sie dienen der statistischen Auswertung und können als Rückmeldung durchgeführter Änderungen herangezogen werden.

4. Die Komponente *Kantenfindung* sucht in den ermittelten Zyklengruppen eines FAMIX+-Modells nach Kanten, deren vollständige Auflösung zu einem Wegfall der jeweiligen Zyklengruppe führte.

5. Zuletzt werden in der Komponente Auflösung Kanten durch Anwendung entsprechender Auflösungstechniken aufgelöst, um die Größe der Zyklengruppen zu reduzieren. Die dazu notwendigen Änderungen werden in das Modell zurückgeschrieben.

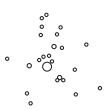
Rückkopplungsschleifen erlaubt das Werkzeug nach der letzten Operation zu jeder Operation.

Die standardmäßige Verfahrensweise ist die lineare Ausführung sämtlicher Operationen 1, 2, 3, 4, 5 mit einer Rückkopplungsschleife von 5 auf sich selbst (Kanten auflösen, bis keine weiteren Auflösungen möglich sind) sowie von 5 auf 3, um die Metriken nach der Änderung zu berechnen.

Das Werkzeug hält jede Änderung in einem Protokoll fest, ebenso jede Fehlerursache, die eine Änderung verhindert.

6.4 Werkzeugintegration in Entwicklungsumgebung

Meinges bietet eine Komponente *Änderungsereignisse* mit Schnittstellen an, die einen Beobachter über vorgenommene Umbauoperationen am Modell benachrich-



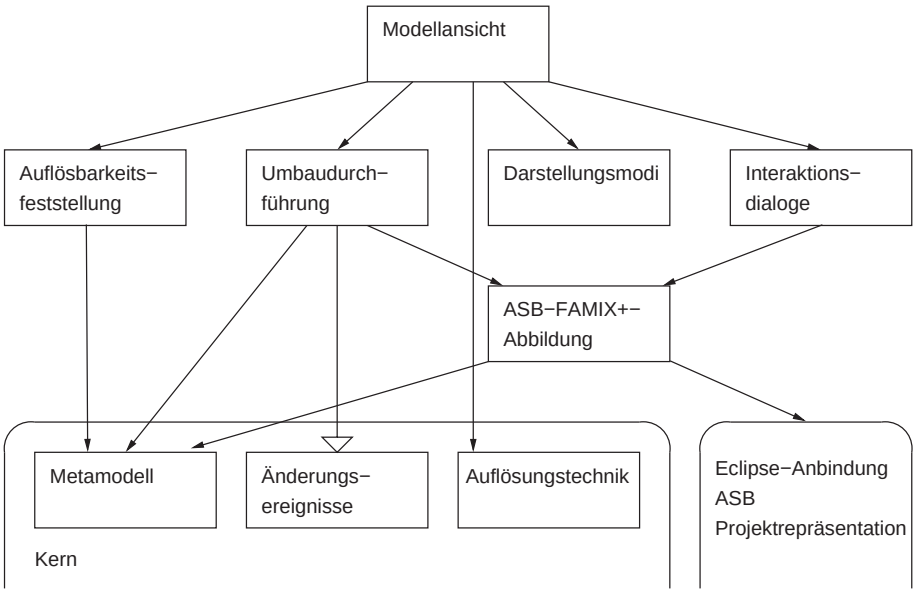


Abbildung 6.4: Komponenten und Abhängigkeiten der Eclipse-Integration

tigen. Über diese Schnittstelle wurde eine Integrationskomponente für die Interaktion mit der Eclipse-Entwicklungsumgebung konzipiert, um dem Anwender eine interaktive, automationsgestützte Zyklenauflösung mit entsprechenden Rückmeldungen zur Verfügung zu stellen. Die Eclipse-Integration übernimmt damit den von [85] definierten Bereich der Präsentation.

Abbildung 6.4 zeigt die Komponenten der Werkzeugintegration und Abhängigkeiten von auserwählten Komponenten des Kerns sowie von Eclipse.

Modellansicht Zeigt die Komponente in Eclipse an, behandelt Eingaben und verwaltet Ausgaben. Über die Modellansicht interagiert der Anwender mit der Integrationskomponente.

Darstellungsmodi Diese Komponente stellt ein Softwaresystem unter verschiedenen Blickwinkeln grafisch dar. Jeder solcher Blickwinkel wird von einem Darstellungsmodus repräsentiert. Darstellungsmodi zeigen nicht nur das System im Ganzen, sondern dienen auch zur Eingrenzung der relevanten Artefakte eines Softwaresystems.

Auflösbarkeitsfeststellung Probiert für jede Kante einer Zyklengruppe, ob sie sich mit einer Auflösungstechnik automatisch auflösen lässt. Im Gegensatz zur Kantenfindung im Kern (s. Abb. 6.2) testet diese Komponente alle Kanten einer Zyklengruppe, während die Kantenfindung nur die Rückwärtskanten im System findet (die dann die Komponente »Kanten auflösen« zu lösen versucht).

FAMIX+-Modellelement	Eclipse-ASB-Klasse	JDT-Klasse
`Class`	`AbstractTypeDeclaration`	`IType`
`Attribute`	`VariableDeclarationFragment`	`IField`
`Method`	`MethodDeclaration` oder `Initializer`	`IMethod`
`LocalVariable`	`VariableDeclarationFragment`	–
`FormalParameter`	`SingleVariableDeclaration`	–
`Access`	`FieldAccess` oder `QualifiedName` oder `SimpleName`	`IBinding`
`Invocation`	`ClassInstanceCreation` oder `MethodInvocation`	`IBinding`
`InheritanceDefinition`	`Type`	`IBinding`
`Package`	–	`IPackageFragment`

Tabelle 6.2: Abbildung von FAMIX+-Elementen auf Eclipse-Syntaxbaum- und -JDT-Klassen

Die Auflösbarkeitsfeststellung beschränkt sich auf das FAMIX+-Modell und erfordert keinen Durchgriff auf die Eclipse-Projektrepräsentation und den abstrakten Syntaxbaum. Sie kann damit auch keine Änderungen am Quelltext durchführen.

Interaktionsdialoge Repräsentiert eine Anzahl von Dialogen zur Darstellung von Informationen über das Softwaresystem sowie zur qualifizierten Auswahl von Knoten und Kanten im Softwaresystemgraph. Diese Dialoge stellen einen essentiellen Bestandteil der automatisierten Zyklenauflösung dar.

Umbaudurchführung Implementiert die Kernschnittstelle *Änderungsereignisse* und überträgt die am FAMIX+-Modell von der jeweils angewandten Auflösungstechnik vorgenommenen Änderungen auf den Quelltext des Softwaresystems.

ASB-FAMIX+-Abbildung Diese Komponente bildet FAMIX+-Modellelemente auf entsprechende Artefakte des abstrakten Syntaxbaums von Eclipse sowie der Eclipse-Projektrepräsentation ab. Tabelle 6.2 zeigt die Abbildung von FAMIX+-Metamodellklassen auf Klassen des Eclipse-Java-Syntaxbaums und der Eclipse-Java-Development-Tools (JDT).

6.5 Darstellungsmodi

Die Werkzeugintegration besitzt drei Darstellungsmodi (auch Ansichten genannt). Zwei davon kann der Benutzer direkt einstellen, eine dritte ergibt sich aus dem Verwendungskontext. Alle Darstellungsmodi sind graphbasiert, das heißt, sie enthal-

ten Knoten, die Artefakte darstellen, und Kanten, die die Abhängigkeiten zwischen den Artefakten anzeigen.

Die Richtung der Abhängigkeit weist dabei immer vom Verwender auf den Verwendeten. Als Kanten werden sämtliche in Kapitel 2.2 aufgeführten Beziehungen sowie die aggregierten Abhängigkeiten repräsentiert, die sich aus diesen Beziehungen ergeben.

Klassenansicht Der Modus zeigt sämtliche Klassen eines Softwaresystems (siehe Abb. 6.5(a)). Ein Knoten repräsentiert eine Klasse, eine Kante sämtliche Beziehungen zwischen den beiden Klassen.

Die Darstellung der Klassen erfolgt nach einem physikalischen Prinzip, das je zwei Klassen mit einer hohen Anzahl an Beziehungen zwischen einander als näher zusammenliegend darstellt als zwei Klassen mit einer niedrigen Anzahl an Beziehungen.

Dunkle Knoten gehören dabei Zyklengruppen an, helle nicht.

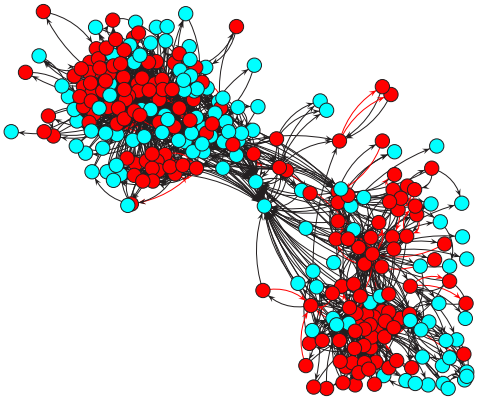
Zyklengruppenansicht In diesem Modus entsprechen die Knoten nicht einzelnen Klassen, sondern Zyklengruppen. Außerhalb von Zyklengruppen liegende Klassen werden als kleine, helle Knoten dargestellt (siehe Abb. 6.5(b)).

Der Graph ist azyklisch und spiegelt die Abhängigkeitshierarchie der Klassen im betrachteten Softwaresystem wider. Je größer ein Knoten, desto größer ist die von ihm symbolisierte Zyklengruppe.

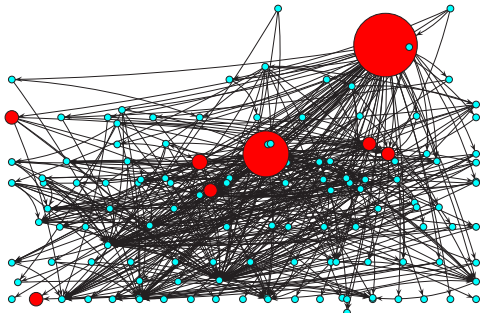
Fragmentiereransicht Die Fragmentiereransicht repräsentiert eine gefilterte Klassenansicht, die ausschließlich die Klassen einer Zyklengruppe anzeigt (siehe Abb. 6.5(c), hier sind die Klassen der größten Zyklengruppe aus Abb. 6.5(b) dargestellt). Der Modus lässt sich auch nur aus der Zyklengruppenansicht heraus aktivieren.

Die Knoten stellen Klassen dar, und die Position der Klassen entspricht exakt jener in der Klassenansicht. Je größer ein Knoten, desto besser ist diese Klasse als Fragmentierer der Zyklengruppe geeignet.

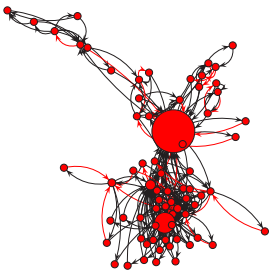
Mit Meinges stand uns nun ein Werkzeug zur Verfügung, mit dem wir Auflösungstechniken an realen Softwaresystemen testen konnten. Die Untersuchungen und Ergebnisse der nächsten Kapitel wären ohne dieses Werkzeug nicht möglich gewesen.



(a) Klassenansicht

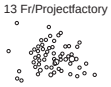


(b) Zyklengruppenansicht



(c) Fragmentiereransicht

Abbildung 6.5: Darstellungsmodi



Kapitel 7

Auswirkungen der Auflösungstechniken

Wir stellten in Kapitel 6 ein Werkzeug vor, das die drei Auflösungstechniken Abhängigkeitsumkehr, gerichtete Knotenspaltung und Rückrufe implementiert und sowohl eine vollautomatische Auflösung zyklischer Abhängigkeiten mittels Stapelverarbeitung als auch eine semiautomatische mittels grafischer Benutzeroberfläche ermöglicht. Nun waren die Auflösungstechniken zwar implementiert, aber es fehlte an Wissen über deren Auswirkung.

Wir wandten daher jeweils die drei Auflösungstechniken über Stapelverarbeitung auf 6091 Softwaresysteme an, um die Effektivität der Zyklenauflösung sowie deren Auswirkung auf systemweite Metriken zu messen und statistische Aussagen treffen zu können. Dieses Kapitel beschreibt die Vorgehensweise und die erlangten Erkenntnisse.

7.1 Beschreibung der Verfahren und Algorithmen

Die Frage nach der Effektivität von Auflösungstechniken wird anhand von drei Kriterien bewertet.

- Welche Auflösungstechnik kann wieviele Rückwärtskanten auflösen?
- Welche Auflösungstechnik löst wieviele Artefakte aus Zyklengruppen heraus?
- Wie beeinflussen die verschiedenen Auflösungstechniken die Systemmetriken?

Die Auswirkungen auf die Systemmetriken werden als statistische Rangkorrelation berechnet (siehe Kap. 7.1.2). Die Nullhypothese lautet für jede untersuchte Metrik: »Die Änderung dieser Metrik im Vergleich zur Änderung der Anzahl der Artefakte in Zyklengruppen ist zufällig.«

Aufgrund der Untersuchungsanordnung lassen sich aus einer Korrelation auch kausale Zusammenhänge erschließen, da eine Metrikänderung immer durch eine Auflösungstechnik verursacht wurde.

7.1.1 Metriken zur Komplexitätsmessung

Aussagen über die Komplexität eines Softwaresystems werden über die Messung von systemweiten Metriken vorgenommen. Da es keine allgemein akzeptierten Messverfahren für Softwarekomplexität gibt [23, 55], verwenden wir eine Mischung verschiedener Metriken, die Aussagen über Softwaresysteme hinsichtlich ihrer Größe sowie Kohäsion und Kopplung zulassen.

Wir beschränkten uns dabei auf interne Metriken (nur auf das Softwaresystem selbst bezogen), da sich diese im Gegensatz zu externen Metriken [15], die auch die Umgebung miteinbeziehen, direkt aus dem Quelltext eines Softwaresystems eruieren lassen.

Zur Messung der Komplexität beschränkt sich unsere Untersuchung auf systemweite Metriken, also auf solche, die eine Maßzahl über ein gesamtes Softwaresystem liefern (im Gegensatz zu Paketmetriken, Klassenmetriken, Funktionsmetriken usw.) und folgende Bedingungen erfüllen.

- Die Messungen sind vollständig auf dem vorliegenden Modell des Softwaresystems durchführbar.
- Die Messungen lassen sich ohne menschliches Zutun vornehmen.

Damit fallen externe Metriken [27] aus dem Raster.

Die in unserer Untersuchung in Frage kommenden Systeme sind zudem alle objektorientiert, sodass eine Beschränkung der Metriken auf *objektorientierte Metriken* angebracht ist [87].

Von der großen Anzahl objektorientierter Metriken [6] fallen alle jene weg, die nicht auf Systemebene gemessen werden [13, 15, 46, 21, 51].

Damit bleibt eine Anzahl von Systemmetriken übrig, die sich auf objektorientierte Softwaresysteme anwenden lassen. Tabelle 7.1 zeigt die für diese Untersuchung ausgewählten Metriken sowie deren bisher in der Literatur vermerkte Validierung (sofern erfolgt) nach dem jeweiligen Gesichtspunkt.

Die Systemebenenmetriken Die Systemebenenmetriken von Tegarden [88, Tab. 8] stellen einfach zu ermittelnde objektorientierte Metriken dar, die laut [88] die Komplexität eines Softwaresystems durch die Aspekte der Kohäsion und Kopplung ausdrücken. Die Validierung erfolgte ebenfalls von [88] durch die Befragung von Nutzern objektorientierter Techniken.

In die Untersuchung wurden die Metriken AAK, AKK, MVT, MVB, VB, AEM und AM einbezogen.

Kürzel	Quelle	Validierung	Beschreibung
AK	[88]	Komplexität [88]	Anzahl der Klassen
AKK			Anzahl konkreter Klassen
AAK			Anzahl abstrakter Klassen
MVT			Maximale Tiefe der Vererbungshierarchie
MVB			Maximale Breite der Vererbungshierarchie
VB			Anzahl der Vererbungsbeziehungen
AEM			Anzahl eindeutiger Methodenaufrufe
AM			Anzahl der Methoden
PF	[2]	Fehlerdichte [3], Kapselung/Vererbung/ Kopplung/dyn. Bindung [36], Komplexität [88]	Polymorphismusfaktor
CF			Kopplungsfaktor
MHF			Methodenkapselungsfaktor
AHF			Attributkapselungsfaktor
MIF			Methodenvererbungsfaktor
AIF			Attributvererbungsfaktor
NKKA	[42]	–	Normalisierte kumulierte Komponentenabhängigkeit
VK	[49]	–	Verbreitungskosten
BL	[48]	–	Beschreibungslänge in Bits

Tabelle 7.1: Liste der untersuchten Systemmetriken

Die MOOD-Metriken Die MOOD-Metriksuite von Abreu und Carapuça [2] enthält eine Reihe von Systemmetriken, von denen sechs in der Untersuchung von [88] validiert wurden.

Die Eigenschaften dieser sechs Systemmetriken seien im Folgenden beschrieben. C ist die Menge aller Klassen im System, C_i die i-te Klasse. $M_d(C_i)$ liefert die Anzahl aller in C_i definierten Methoden, $A_d(C_i)$ die Anzahl aller in C_i definierten Attribute. $M_i(C_i)$ liefert die Anzahl aller von C_i geerbten, aber nicht überschriebenen Methoden, $A_i(C_i)$ die Anzahl aller von C_i geerbten Attribute. $M_n(C_i)$ liefert die Anzahl aller in C_i definierten, nicht überschriebenen Methoden, $M_o(C_i)$ die Anzahl der überschriebenen. M_{ij} repräsentiert die j-te Methode in der i-ten Klasse, A_{ij} das j-te Attribut. $DC(C_i)$ liefert die Anzahl aller von C_i transitiv abgeleiteten Klassen. $V(u)$ liefert das Verhältnis der Anzahl Klassen, in denen u sichtbar ist, zur Gesamtanzahl der Klassen $|C|$.

Der *Polymorphismusfaktor* (PF) beschreibt das Verhältnis der tatsächlichen polymorphen Situationen (d. h. der tatsächlichen Anzahl von Methodenüberschreibungen in abgeleiteten Klassen) zu den maximal möglichen polymorphen Situationen (d. h. der Anzahl der Überschreibungen, die sich ergäbe, wenn jede Methode in jeder abgeleiteten Klasse überschrieben worden wäre).

$$\text{PF} = \frac{\sum_{i=1}^{|C|} M_o(C_i)}{\sum_{i=1}^{|C|} (M_n(C_i) \cdot DC(C_i))}$$

Der *Kopplungsfaktor* (CF) bezeichnet das Verhältnis der Anzahl tatsächlicher Abhängigkeiten zwischen Klassen (ohne Vererbungsabhängigkeiten) zur größtmöglichen Anzahl von Abhängigkeiten. benutzt(C_c, C_s) ist 1, wenn C_c von C_s abhängig ist, sonst 0.

$$\text{CF} = \frac{\sum_{i=1}^{|C|} \sum_{j=1}^{|C|} \text{benutzt}(C_i, C_j)}{|C|^2 - |C|}$$

Für den *Methodenkapselungsfaktor* (MHF) wird je Methode das Verhältnis der Anzahl Klassen, in denen die Methode nicht sichtbar ist, zur Gesamtanzahl der Klassen berechnet. Die Summe dieser Verhältnisse über alle Methoden durch die Gesamtanzahl aller Methoden stellt den Methodenkapselungsfaktor dar.

$$\text{MHF} = \frac{\sum_{i=1}^{|C|} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{im}))}{\sum_{i=1}^{|C|} M_d(C_i)}$$

Der *Methodenvererbungsfaktor* (MIF) repräsentiert das Verhältnis der Anzahl geerbter aber nicht überschriebener Methoden einer Klasse zur Anzahl aller (durch Vererbung oder Definition) verfügbaren Methoden in der Klasse, summiert über alle Klassen des Systems.

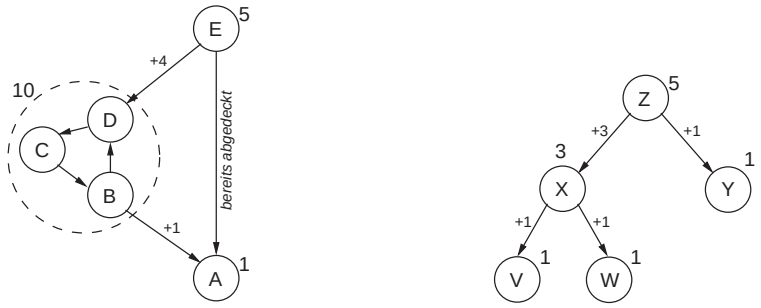
$$\text{MIF} = \frac{\sum_{i=1}^{|C|} M_i(C_i)}{\sum_{i=1}^{|C|} (M_d(C_i) + M_i(C_i))}$$

Der *Attributkapselungsfaktor* (AHF) entspricht dem Methodenkapselungsfaktor, jedoch auf Attribute von Klassen gemünzt.

$$\text{AHF} = \frac{\sum_{i=1}^{|C|} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{im}))}{\sum_{i=1}^{|C|} A_d(C_i)}$$

Der *Attributvererbungsfaktor* (AIF) entspricht dem Methodenvererbungsfaktor, jedoch auf Attribute von Klassen gemünzt.

$$\text{AIF} = \frac{\sum_{i=1}^{|C|} A_i(C_i)}{\sum_{i=1}^{|C|} (A_d(C_i) + A_i(C_i))}$$



(a) KKA (b) BKA

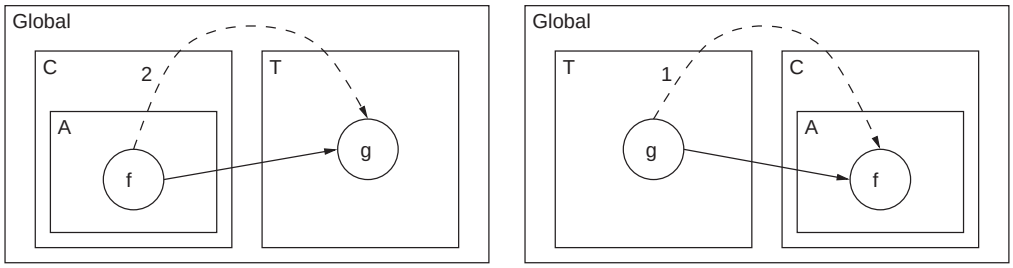
Abbildung 7.1: Beispiele zur kumulierten Komponentenabhängigkeit

Die normalisierte kumulierte Komponentenabhängigkeit Lakos beschreibt in seinem Werk [42] über C++-Softwareentwicklung die Problematik zyklischer Abhängigkeiten und nennt eine Vielzahl von Auflösungmöglichkeiten. Weiters stellt er darin eine Reihe von Metriken vor, die die Komplexität des Softwaresystems aus Sicht des Testaufwands beschreiben.

Die *kumulierte Komponentenabhängigkeit* (KKA) ist eine Zahl, welche die Summe der Abhängigkeiten aller Komponenten von jeweils jenen Komponenten repräsentiert, von welchen sie abhängen. Dabei werden Komponenten in Zyklengruppen mit dem Quadrat der Anzahl Komponenten in der Zyklengruppe gewichtet, da in einer Zyklengruppe jede Komponente von jeder anderen transitiv abhängig ist.

In Abbildung 7.1(a) ergibt sich die KKA wie folgt: A ist unabhängig und trägt den Wert 1. B, C und D befinden sich in einer Zyklengruppe und tragen gemeinsam den Wert 10 (3^2 durch die Zyklengruppe plus 1 aufgrund der Abhängigkeit zu A). E ist von der Zyklengruppe und transitiv von A abhängig und trägt den Wert 5 (1 von A, 3 von der Zyklengruppe und 1 für E selbst). Die Abhängigkeit Es von A zählt nicht, da A bereits über die anderen Abhängigkeiten abgedeckt wird. Die Quadrierung innerhalb der Zyklengruppe wirkt sich auf E nicht aus, da E von B, C und D nur jeweils einfach abhängig ist. Die KKA ergibt sich aus der Summe der kumulierten Abhängigkeiten und beträgt 1 + 10 + 5 = 16.

Die KKA ist stark von der Größe des Softwaresystems abhängig. Um vergleichbare Werte zwischen Softwaresystemen zu erzielen, existiert die dimensionslose und größenunabhängige *normalisierte kumulierte Komponentenabhängigkeit* (NKKA). Diese setzt die KKA eines Softwaresystems in Proportion zur KKA eines binären Baums derselben Anzahl von Komponenten, der die »ideale« Komponentenabhängigkeit eines Softwaresystems dieser Größe widerspiegelt. Die KKA des »idealen« Softwaresystems heißt



(a) Kurzer Pfad (b) Langer Pfad

Abbildung 7.2: Beispiele zur Beschreibungslänge

balancierte Komponentenabhängigkeit (BKA). NKKA = $\frac{\text{KKA}}{\text{BKA}}$.

Die »ideale« Komponentenabhängigkeit für Softwaresysteme mit fünf Komponenten ist in Abbildung 7.1(b) dargestellt. Deren KKA beträgt 11 (V 1, W 1, X 3, Y 1 und Z 5), die NKKA des Beispiels daher $\frac{16}{11}$ = 1,45.

Stärkere zyklische Abhängigkeiten spiegeln einen höheren Testaufwand wider. Da Testaufwände mit der Testbarkeit zusammenhängen, wurde diese Metrik ebenfalls in die Untersuchung aufgenommen.

Eine Validierung der Metrik ist uns nicht bekannt.

Die Verbreitungskosten MacCormack u. a. [49] stellen die Metrik *Verbreitungskosten* (VK) zur Berechnung der Modularität eines Softwaresystems vor, ausgedrückt durch den Grad der systemweiten Kopplung der Module. Sie ermöglicht den Vergleich der Modularität zwischen Softwaresystemen oder zwischen Revisionen eines Softwaresystems.

Die Verbreitungskosten setzen sich aus dem Verhältnis der Summe der Anzahl von jeder Komponente abhängigen Komponenten zur größtmöglichen Anzahl Abhängigkeiten zusammen. Die Metrik ähnelt damit der NKKA-Metrik und wird deshalb in die Untersuchung aufgenommen. Eine Validierung dieser Metrik ist nicht bekannt.

Die Beschreibungslänge Als besondere Metrik zur strukturellen Komplexität von Softwaresystemen dient die *Beschreibungslänge* (BL) von Lutz [48, 47]. Die Beschreibungslänge repräsentiert die informationstheoretische Anzahl an Bits, die für die strukturelle Beschreibung aller Knoten und Komponenten eines Softwaresystems, deren Hierarchie und deren Abhängigkeiten nötig sind.

Beziehungen zwischen Knoten (z. B. Klassen) werden über relative Pfade ausgedrückt. Abbildung 7.2(a) besteht aus den Komponenten C, A, T und den Knoten f und g. f bezieht sich auf g (durchgehender Pfeil), wird aber

indirekt über 2 T/g beschrieben (strichlierter Pfeil), sprich, zwei Komponenten aufwärts bis zur nächsten gemeinsam umschließenden Komponente, dann in Komponente T abgestiegen und dort g referenziert.

Die Adressierung von Knoten und Komponenten erfolgt immer relativ zu ihrer unmittelbaren Elternkomponente, daher wird die informationstheoretisch kürzeste Bitanzahl zur Adressierung der unmittelbaren Kindknoten verwendet. Je mehr Elemente eine Komponente enthält, desto mehr Bits benötigt die Beschreibung, um alle Elemente der Komponente adressieren zu können.

Die Beschreibungslänge steigt, je länger die Pfade zur Beschreibung von Beziehungen sind beziehungsweise je höher die Anzahl der Elemente in einer Komponente ist. Das System in Abbildung 7.2(a) benötigt weniger Bits zu seiner Beschreibung als das System in Abbildung 7.2(b), da der Pfad 2 T/g kürzer zu beschreiben ist als der Pfad 1 C/A/f.

Uns ist keine Validierung dieser Metrik bekannt. Die Metrik in die Untersuchung aufzunehmen dient vor allem der Frage, ob mit Zyklenreduktionen auch Reduktionen der Beschreibungslänge einhergehen.

Tabelle 7.2 beschreibt die Interpretation von Metrikwerten hinsichtlich ihrer Beziehung für die Softwarekomplexität. Die Grenzwerte sowie die Richtung der Wertänderung bei wachsender Komplexität dienen in weiterer Linie der statistischen Auswertung in Kapitel 7.3.

7.1.2 Algorithmen und statistische Verfahren

Zur Erkennung von Zyklengruppen wurde der Algorithmus von Nuutila und Soisalon-Soininen [67] (siehe auch Kap. 3.2 auf Seite 15) eingesetzt.

Zur Berechnung von Rückwärtskanten wurde der Algorithmus GR-550 [20] (siehe auch Kap. 3.3 auf Seite 17) herangezogen.

Die statistische Verteilung der Metrikwerte über die gezogene Stichprobe von Softwaresystemen ist unbekannt. Daher wurde zur Ähnlichkeitsmessung zwischen zwei Messreihen die Rangkorrelation nach Spearman [22, Kap. 3.4.2] gemäß der Formel

$$r_S = \frac{\sum_{i=1}^{n}(r_i - \bar{r}) \cdot (s_i - \bar{s})}{\sqrt{\sum_{i=1}^{n}(r_i - \bar{r}) \cdot \sum_{i=1}^{n}(s_i - \bar{s})}} \tag{7.1}$$

berechnet, wobei r und s die Ränge der Messreihen und n die Anzahl der Messwerte in einer Reihe darstellen.

Bei der Rangkorrelation werden von zwei Verteilungen mit derselben Anzahl Elemente Ränge gebildet, das heißt, das kleinste Element durch 1, das zweitkleinste durch 2 usw. ersetzt.

Die Korrelation berechnet nun die lineare Ähnlichkeit zweier Verteilungen und liefert als Maß einen Koeffizienten r_S im Bereich von -1 bis +1 (inklusive). Negative Korrelationen bedeuten einen gegenläufigen Zusammenhang (r nimmt ab, s

Kürzel	Art	Min	Max	Richtung
AK	S	0	∞	>
AKK	S	0	AK	>
AAK	S	0	AK	>
MVT	S	0	AK	>
MVB	S	0	AK	>
VB	D	0	$\frac{\mathrm{AK}\cdot(\mathrm{AK}+1)}{2}$	>
AEM	D	0	∞	>
AM	S	0	∞	>
PF	D	0	1	>
CF	D	0	1	>
MHF	S	0	1	>
AHF	S	0	1	>
MIF	S	0	1	>
AIF	S	0	1	>
NKKA	D	0	$\frac{\mathrm{AK}^2}{(\mathrm{AK}+1)\cdot(\mathrm{ld}(\mathrm{AK}+1)-1)+1}$	>
VK	D	0	1	>
BL	S	0	∞	>

Tabelle 7.2: Interpretationsanleitung für die Systemmetrikwerte
Kürzel Metrikabkürzung gemäß Tabelle 7.1 **Art** Art der Komplexität: S – Strukturkomplexität, D – Datenkomplexität **Min** Minimalwert **Max** Maximalwert **Richtung** Verhältnis von Wertzuwachs zur Komplexitätssteigerung: > – proportional, < – reziprok

nimmt zu), positive einen gleichläufigen Zusammenhang (r nimmt zu und s nimmt zu).

Die Korrelation gilt bei $1 \leq |r_S| < 0{,}8$ als sehr stark, bei $0{,}8 \leq |r_S| < 0{,}6$ stark, bei $0{,}6 \leq |r_S| < 0{,}4$ als mittel, bei $0{,}4 \leq |r_S| < 0{,}2$ als schwach und bei $0{,}2 \leq |r_S| \leq 0$ als sehr schwach beziehungsweise nicht gegeben.

Die Überschreitungswahrscheinlichkeit wurde mittels AS 89 [7] berechnet. Alle Berechnungen erfolgten im Statistikprogramm R-2.5.1 [74].

7.2 Vorgehensweise

Dieser Abschnitt beschreibt, wie wir die Durchführung durchführten und welche Werkzeuge und Methoden wir dabei anwandten.

7.2.1 Werkzeug zur Messung

Das Werkzeug ist das in Kapitel 6 vorgestellte Meinges. Wir betreiben es in der Stapelverarbeitung, in der es Zyklengruppen erkennt, die in Kapitel 7.1.1 angeführten

Metriken berechnet, Auflösungstechniken anwendet und die Metriken nach Ablauf noch einmal berechnet.

Meinges schreibt sämtliche Schritte und Analyseergebnisse in maschinenlesbare Protokolle, die als Grundlage für die statistische Auswertung dienen.

7.2.2 Anordnung

Um zu einer statistisch signifikanten Aussage zu gelangen, musste eine hinreichend große Anzahl an Softwaresystemen untersucht werden.

Daher wurde die Auswertung einer der größten im Netz befindlichen Java-Softwaresammlung bei sourceforge.net in Betracht gezogen. Die Softwarepalette bietet von Kleinstprogrammen bis hin zu riesigen Softwaresystemen eine mit anderen Mitteln nur unverhältnismäßig schwer zu erreichende Heterogenität der zu untersuchenden Systeme.

Zum Stichtag 1. August 2007 beherbergte sourceforge.net 12115 Java-Softwaresysteme. Davon besaßen 6024 keine Revisionsgeschichte und kamen für eine automatische Auswertung nicht in Frage. Von den übrigen 6091 Softwaresystemen ließen sich 279 nicht syntaxanalysieren. Für die Analyse verblieben damit 5812 Softwaresysteme.

Als Testgerät stand der Entwicklerrechner zur Verfügung. Da keine zeitlichen Messungen vorgenommen werden mussten, war eine sich während des Testlaufs ändernde Systemlast ohne Belang. Weiters arbeitet der zum Einsatz gelangte Zyklenfindungsalgorithmus [67] deterministisch und idempotent, sodass bereits ein einziger Analyselauf je Softwaresystem zum Auffinden aller zyklischen Abhängigkeiten, zur Anwendung aller Auflösungstechniken sowie zur Berechnung aller Metriken genügte.

7.2.3 Durchführung

Für jedes analysierte Softwaresystem wurden folgende Schritte ausgeführt.

1. Die neueste Revision aus der Versionsverwaltung extrahieren.

2. Für jede Auflösungstechnik t der drei untersuchten Auflösungstechniken die Schritte 3 bis 6 ausführen.

3. Die Zyklengruppen mit Meinges identifizieren und die Metriken des Kapitels 7.1.1 berechnen. Weiters die Trivialmetriken *Anzahl der Klassen* (AK), Anzahl der Zyklengruppen (ZG) sowie Anzahl der Klassen in Zyklengruppen (AZG) berechnen und alle berechneten Metriken für nachgelagerte Auswertungen persistieren (Vorhermetriken).

4. Die Menge der Rückwärtskanten jeder Zyklengruppe ist ermitteln.

5. Auf jede Rückwärtskante die aktuelle Auflösungstechnik t anwenden. Jeden Versuch sowie jede erfolgreiche Auflösung einer Kante akkumulieren und für weitere Auswertungen persistieren, sodass nach der Verarbeitung aller Kanten die Summe der verarbeiteten Kanten zur Verfügung steht sowie Summe der erfolgreich aufgelösten Kanten.

6. Die Metriken nach der Anwendung der Auflösungstechnik wie in Schritt 3 berechnen und unter einer von Schritt 3 eindeutig zu unterscheidenden Weise persistieren (Nachhermetriken).

Als nachgelagerter Schritt erfolgte die statistische Auswertung.

7. Für jede Auflösungstechnik t:

   Für die Analyse der Auswirkung der Zyklenreduktionen auf (S. 81) zuerst sämtliche durch t verursachte Zyklenreduktionen in allen AZG-Verteilungen finden. Eine Zyklenreduktion liegt dann vor, wenn $AZG_{Vorhermetriken} > AZG_{Nachhermetriken}$ gilt.

   In der so gewonnenen Verteilung über alle Zyklenreduktionen aller Softwaresysteme AZG mit jeweils AK, AKK, AAK, MVT, MVB, VB, AEM, AM, PF, CF, MHF, AHF, MIF, AIF, NKKA, VK und BL korrelieren.

8. Die Anzahl der erfolgreich angewandten Auflösungstechniken über alle Softwaresysteme nach Techniken getrennt summieren und ausweisen.

9. Die Zunahme von Klassen je Anwendung von Auflösungstechniken nach Techniken getrennt summieren und ausweisen.

10. Die Anzahl der durch AZG-Reduktion betroffenen Softwaresysteme je Anwendung von Auflösungstechniken nach Techniken getrennt summieren und ausweisen.

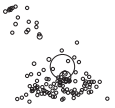
11. Die Anzahl der aus Zyklengruppen herausgelösten Klassen je Anwendung von Auflösungstechniken nach Techniken getrennt summieren und ausweisen.

7.3 Auswertung

Wir untersuchten die Effektivität von drei Auflösungstechniken anhand von 6091 Softwaresystemen. Ein untersuchtes System bestand im Durchschnitt aus 143 ($\pm$300,8) Klassen. Davon befanden sich 22,8% ($\pm$21,1) in Zyklengruppen (AZG%). Die Anzahl der Zyklengruppen (ZG) belief sich im Durchschnitt auf 6 ($\pm$12,9). Die normalisierte kumulierte Komponentenabhängigkeit (NKKA) betrug 2,3 ($\pm$4,2), der Median der Beschreibungslänge (BL) 8614,5 Bits.

Tabelle 7.3 enthält Kennzahlen zu allen Metriken für alle analysierten Systeme zusammen, Tabelle 7.4 für alle Systeme ab 100 Klassen.

0 Fr/imp-a



Eigenschaft	μ (σ)	Min	$Q_{25\%}$	Med	$Q_{75\%}$	Max
AK	142,8 (±300,8)	1	21	55	140	8156
AKK	121,1 (±251,1)	0	19	48	118	6914
AAK	21,7 (±57,9)	0	1	6	19	1600
MVT	2,1 (±1,7)	0	1	2	3	21
MVB	16,0 (±40,5)	1	2	5	14	1328
VB	79,4 (±215,9)	0	3	18	64	5053
AEM	1774,6 (±5492,0)	0	114	404	1374	164800
AM	1163,5 (±2856,0)	1	149	398	1046	72726
PF	0,32 (±0,27)	0,00	0,12	0,26	0,45	1,00
CF	0,07 (±0,09)	0,00	0,02	0,04	0,08	1,00
MHF	0,12 (±0,08)	0,00	0,06	0,11	0,16	0,61
AHF	0,64 (±0,22)	0,00	0,54	0,69	0,80	0,99
MIF	0,20 (±0,19)	0,00	0,02	0,15	0,31	0,95
AIF	0,17 (±0,20)	0,00	0,00	0,09	0,26	0,99
NKKA	2,3 (±4,2)	0,22	0,84	1,18	2,08	113,53
VK	0,25 (±0,20)	0,01	0,11	0,20	0,34	1,00
BL	203695,6 (±1707935,4)	5	1739	8615	46507	69164865
ZG	5,5 (±12,9)	0	1	2	5	416
AZG	35,8 (±93,6)	0	2	10	32	3005
AZG%	22,8 (±21,1)	0,0	4,5	18,5	35,0	100,0

Tabelle 7.3: Übersicht über die Auswertung

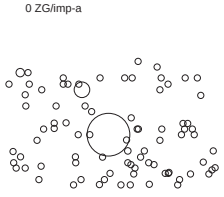
Bei Softwaresystemen ab 100 Klassen beträgt die durchschnittliche Größe 356 Klassen (±449,6) beziehungsweise die Länge (BL) im Median 98069,1 Bits. Die normalisierte kumulierte Komponentenabhängigkeit (NKKA) beläuft sich auf 4,0 (±6,9).

Die Anzahl der Artefakte in Zyklengruppen liegt mit 22,8% (±21,1) nur wenige Prozentpunkte höher als bei der Betrachtung aller Systeme in Tabelle 7.3. Die Anzahl der Artefakte in Zyklengruppen scheinen prozentuell nicht mit der Größe der Softwaresysteme zu wachsen.

Abbildung 7.3 untermauert diese Vermutung. Hier werden die untersuchten Applikationen nach ihrer Größe in Klassen zusammengefasst und die Mediane der Anzahl Artefakte in Zyklengruppen als Balken dargestellt. Während wie zu erwarten die absolute Anzahl Klassen in Zyklengruppen (AZG) mit der Klassenanzahl steigt (Abb. 7.3(a)), so pendelt sich der prozentuelle Anteil (AZG%) ab der Größe 10 bei einem Median von ungefähr 20% ein (Abb. 7.3(b)).

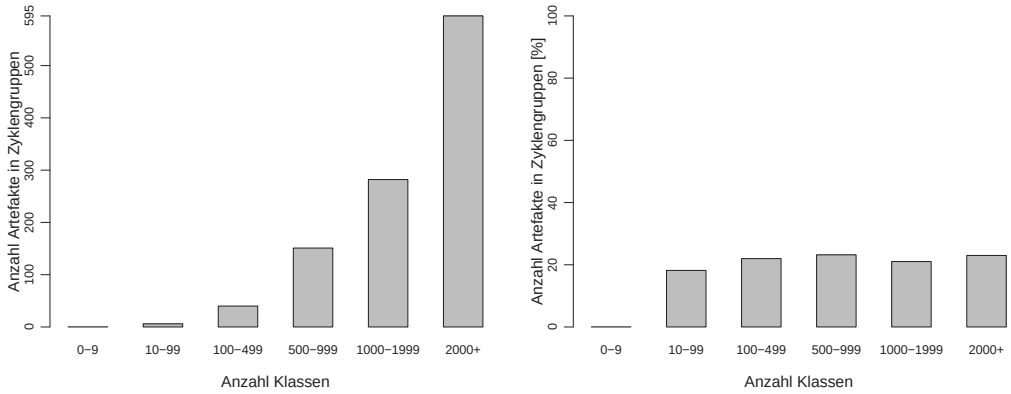
Der prozentuelle Anteil der Artefakte in Zyklengruppen nimmt also mit der Systemgröße nicht in erkennbarem Ausmaß zu.

Betrachten wir noch die Verteilung der untersuchten Softwaresysteme nach dem Anteil der Artefakte in Zyklengruppen (AZG%) (Abbildung 7.4). Die relative Mehrheit aller Systeme besitzt einen AZG%-Anteil unter 10%. Der Median liegt



Eigenschaft	μ (σ)	Min	$Q_{25\%}$	Med	$Q_{75\%}$	Max
AK	356,0 (±449,6)	100	140	210	384	8156
AKK	299,6 (±374,8)	30	118	180	327	6914
AAK	56,4 (±90,5)	0	16	30	61	1600
MVT	3,6 (±1,7)	0	2	3	4	21
MVB	38,4 (±64,1)	1	11	20	40	1328
VB	212,8 (±335,8)	0	60	110	230	5053
AEM	4676,5 (±8804,8)	41	1194	2180	4733	164800
AM	2940,9 (±4431,9)	281	1034	1659	3097	72726
PF	0,29 (±0,16)	0,00	0,18	0,26	0,38	1,00
CF	0,01 (±0,01)	0,00	0,01	0,01	0,02	0,06
MHF	0,13 (±0,06)	0,00	0,09	0,13	0,17	0,43
AHF	0,70 (±0,16)	0,01	0,62	0,72	0,81	0,98
MIF	0,32 (±0,19)	0,00	0,18	0,30	0,44	0,95
AIF	0,27 (±0,22)	0,00	0,10	0,22	0,39	0,99
NKKA	4,0 (±6,9)	0,22	1,01	1,87	3,86	113,53
VK	0,16 (±0,14)	0,01	0,06	0,11	0,21	0,98
BL	597336,4 (±2920180,0)	3123	42180	98069	297488	69164865
ZG	13,2 (±20,0)	0	4	7	15	416
AZG	90,6 (±146,8)	0	22	51	106	3005
AZG%	25,3 (±18,0)	0,0	10,9	22,1	36,3	98,0

Tabelle 7.4: Übersicht über die Auswertung der Systeme ab 100 Klassen



(a) Absolute Häufigkeiten

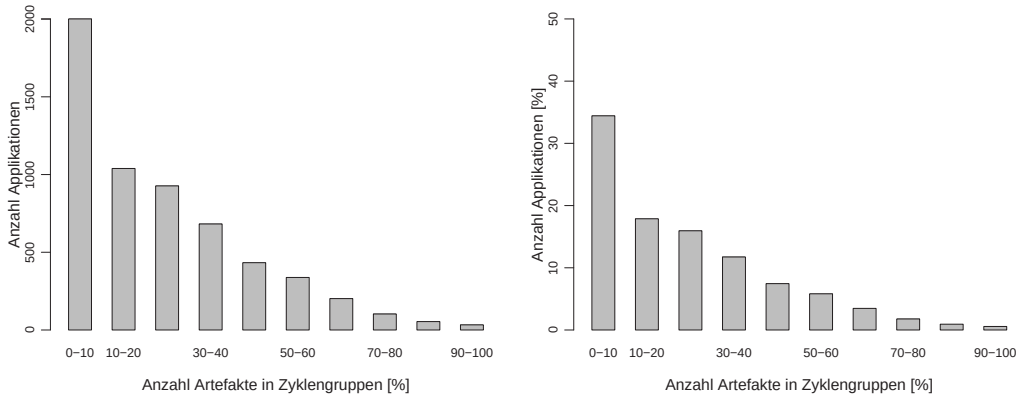
(b) Relative Häufigkeiten

Abbildung 7.3: Mediane von AZG und AZG% nach Klassenanzahl

in der Äquivalenzklasse von 10 bis 20%. Nur 12,6% aller Applikationen besitzen einen AZG%-Anteil ab 50%.

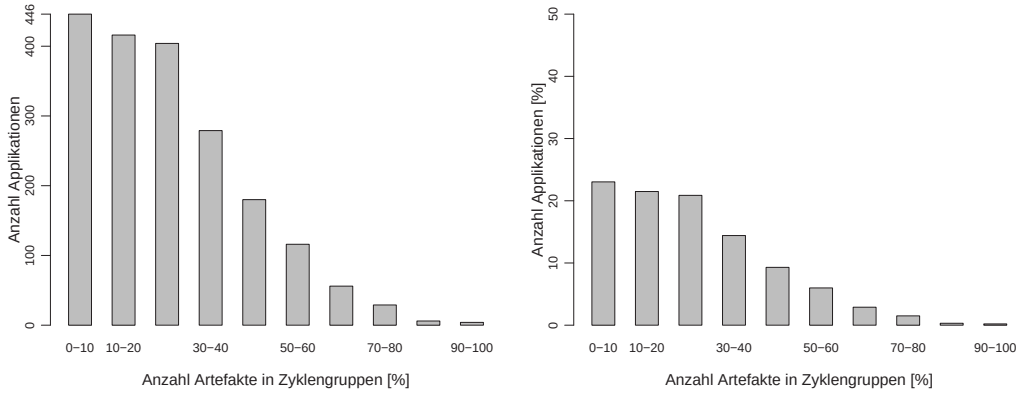
Bei Softwaresystemen ab 100 Klassen liegen gut 60% aller Systeme in den

1 Fr/imp-a



(a) Absolute Häufigkeiten (b) Relative Häufigkeiten

Abbildung 7.4: Anzahl Applikationen bei ansteigendem AZG%



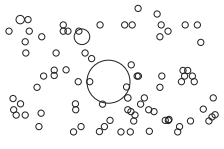
(a) Absolute Häufigkeiten (b) Relative Häufigkeiten

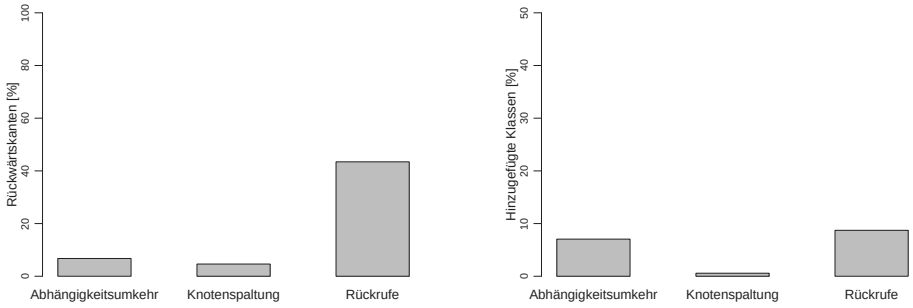
Abbildung 7.5: Anzahl Applikationen ab 100 Klassen bei ansteigendem AZG%

ersten drei Äquivalenzklassen bis 30% AZG% (Abbildung 7.5). Der Median liegt hier in der Äquivalenzklasse von 20 bis 30%. Nur 10,9% aller Applikationen besitzen einen AZG%-Anteil ab 50%.

7.3.1 Effektivität

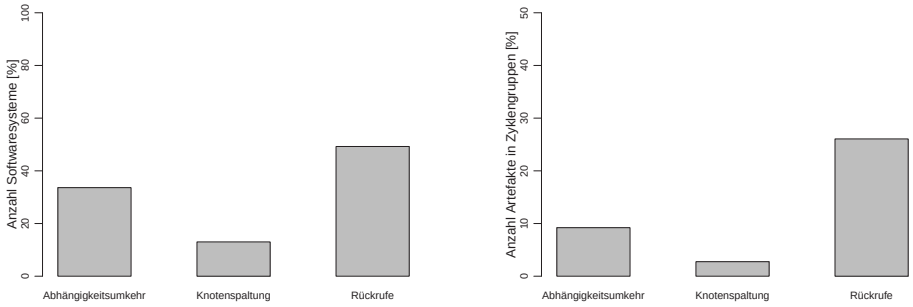
Die untersuchten Zyklengruppen enthielten in Summe 101987 Rückwärtskanten. Davon konnten durch die vollautomatische Anwendung von Abhängigkeitsumkehr 6882, von Knotenspaltung 4731 und von Rückrufen 44298 Kanten aufgelöst werden. In Prozent beläuft sich die Effektivität je Auflösungstechnik auf 6,7%, 4,6%





(a) Effektivität (b) Zuwachs an Klassen

Abbildung 7.6: Effektivität der einzelnen Auflösungstechniken



(a) nach Applikationen (b) nach Gesamtanzahl AZG

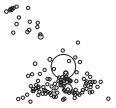
Abbildung 7.7: Reduktion der Artefakte in Zyklengruppen

und 43% (siehe Abbildung 7.6(a)).

Auf Ebene ganzer Softwaresysteme vermochte die Abhängigkeitsumkehr in 1954, die Knotenspaltung in 757 und die Rückrufe in 2862 von insgesamt 5812 Softwaresystemen die Anzahl der Artefakte in Zyklengruppen (AZG) verringern. In Prozent betragen die Reduktionen jeweils 34%, 13% und 49% (siehe Abb. 7.7(a)).

Insgesamt befanden sich in allen untersuchten Applikationen 207861 Artefakte in Zyklengruppen. Davon lösten die Abhängigkeitsumkehr 19139 Artefakte aus Zyklengruppen heraus (9,2%), die Knotenspaltung 5730 (2,8%) und die Rückrufe 54166 (26%). Abbildung 7.7(b) stellt den prozentuellen Sachverhalt grafisch dar.

In allen Fällen stellen die Rückrufe die effektivste Auflösungstechnik dar, die ineffektivste die Knotenspaltung. Hierbei sei jedoch in Erinnerung gerufen, dass sich die Effektivität auf die rein vollautomatische Anwendbarkeit der Auflösungstechnik bezieht. In Kapitel 8 lösen wir die zielgerichtet zyklische Abhängigkeiten



auf, wobei der automatische Umbau lediglich als Hilfsmittel dient, aber nicht alleinig zur Änderung des Softwaresystems beiträgt. Die dort ermittelten Statistiken weichen erwartungsgemäß von den hiesigen ab.

Nun fordert der Einsatz einer Auflösungstechnik einen Tribut, und zwar in der Anzahl der zusätzlich erzeugten Klassen, die durch die Anwendung einer Auflösungstechnik angelegt werden. Abbildung 7.6(b) zeigt die prozentuelle Zunahme der Klassenanzahl für jede Auflösungstechnik auf.

Die Rückrufe verursachten eine Zunahme der Klassen um 8,7%, die Abhängigkeitsumkehr um 7,0%, die Knotenspaltung um lediglich 0,6%.

Für jede aufgelöste Kante nahm die Anzahl an Klassen für die Abhängigkeitsumkehr 8,5, für die Knotenspaltung 1,0 und für die Rückrufe 1,6 zu. Die zuwachsschonenste Auflösungstechnik ist hier die Knotenspaltung, die zuwachsstärkste die Abhängigkeitsumkehr.

7.3.2 Exemplarische Metrikveränderungen

Beantworten wir nun die Frage, inwiefern sich die Reduktion von Artefakten in Zyklengruppen auf die Systemmetriken auswirkt. Dazu betrachten wir exemplarisch die Metrikveränderungen an den zehn größten untersuchten Softwaresystemen für jede angewandte Auflösungstechnik. Tabelle 7.5 führt die Metrikwerte für jede Applikation und jede Auflösungstechnik an. Fettgedruckte Zahlen bedeuten einen verringerten Metrikwert gegenüber dem Ausgangszustand, normal gedruckte Zahlen einen erhöhten und graue Zahlen keine Änderung.

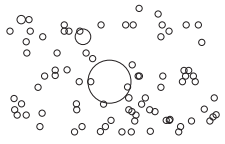
Alle Systemebenenmetriken steigen oder bleiben gleich, wie aus Anwendung der entsprechenden Auflösungstechnik zu erwarten ist.

Unter den MOOD-Metriken ist nur beim Kopplungsfaktor (CF) eine Abnahme über alle Auflösungstechniken feststellbar. Bereits beim Methodenkapselungsfaktor (MHF) führt die Anwendung der Knotenspaltung in den meisten Fällen zur Erhöhung. Beim Attributkapselungsfaktor (AHF) treten sporadisch sogar bei mehr als einer Auflösungstechnik keine Verbesserungen auf. Der Methodenvererbungsfaktor (MIF) erfährt in der Regel eine Reduktion, der Attributvererbungsfaktor (MHF) und der Polymorphismusfaktor (PF) hingegen nicht.

Lediglich für CF lässt sich daher die Vermutung aussprechen, dass er bei einer Senkung der Artefakte in Zyklengruppen sinkt. Die restlichen MOOD-Metriken zeigen keine eindeutige Entwicklung.

Die normalisierte kumulierte Komponentenabhängigkeit (NKKA) sinkt bei der Abhängigkeitsumkehr abgesehen von einem Fall immer, Knotenspaltung und Rückrufen verursachen hingegen mehrheitlich einen Anstieg. Insbesondere bei den Rückrufen fällt bei manchen Applikationen (jfs-vfs, x4l-reload) eine bis zu zehnfache Zunahme auf, während sich die Zunahme bei den anderen Applikationen in einem moderaten Rahmen bewegt.

Der Grund liegt in der Einführung einer allgemeinen Fabriksklassenimplementierung, die auf oberster Schicht Objekte aus Klassen erzeugt und damit deren



System	T	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
azureus	0	3476	2531	945	5	153	2293	52458	29423	0,27	0,00	0,17	0,77	0,44	0,38	56,93	0,33	19510845	71	1588	45,7
	A	3869	2531	1338	5	153	2686	52809	38797	**0,26**	**0,00**	**0,10**	**0,77**	**0,38**	0,39	**49,41**	**0,29**	19769107	76	**1524**	**39,4**
	K	3533	2588	945	5	153	2293	52458	29423	**0,27**	**0,00**	**0,17**	**0,77**	0,44	**0,38**	**54,65**	**0,33**	19679013	72	**1587**	**44,9**
	R	3768	2533	1235	5	153	2583	53025	30338	0,28	**0,00**	**0,17**	**0,77**	**0,43**	0,38	**39,92**	**0,26**	20132818	90	**1230**	**32,6**
cayenne	0	2979	2655	324	9	173	2299	40384	22835	0,14	0,00	0,15	0,70	0,54	0,51	62,29	0,12	6676469	72	803	27,0
	A	3219	2655	564	9	173	2539	40621	26842	0,20	**0,00**	**0,11**	**0,70**	**0,51**	0,52	**44,00**	**0,09**	**6661188**	87	**653**	**20,3**
	K	2987	2663	324	9	173	2299	40384	22835	0,14	**0,00**	0,15	0,70	**0,54**	0,51	64,10	**0,12**	**6667865**	**71**	**792**	**26,5**
	R	3195	2657	538	9	173	2513	40768	23445	0,15	**0,00**	**0,15**	**0,70**	**0,53**	0,51	**17,50**	**0,09**	7081775	95	**469**	**14,7**
comxe	0	2674	2554	120	7	490	1912	74317	35788	0,06	0,00	0,10	0,58	0,77	0,55	25,45	0,08	43578619	65	580	21,7
	A	2849	2554	295	7	490	2087	75950	39410	0,06	**0,00**	**0,09**	0,58	**0,77**	0,55	**23,57**	**0,08**	43876640	**64**	**546**	**19,2**
	K	2699	2579	120	7	490	1912	74317	35788	0,06	**0,00**	0,10	0,58	**0,77**	0,55	26,50	**0,08**	**42556221**	66	**571**	**21,2**
	R	2832	2556	276	7	490	2068	74683	36285	0,06	**0,00**	**0,10**	0,58	**0,77**	0,55	47,10	0,09	43780872	75	**458**	**16,2**
grouppac	0	6098	4498	1600	6	763	5053	109422	50545	0,35	0,00	0,07	0,59	0,42	0,27	2,70	0,01	69164865	193	935	15,3
	A	6260	4498	1762	6	763	5215	109885	54911	**0,34**	**0,00**	**0,06**	**0,59**	**0,41**	0,27	**2,49**	**0,01**	69222144	193	**896**	**14,3**
	K	6110	4510	1600	6	763	5053	109422	50545	**0,35**	**0,00**	**0,07**	**0,59**	0,42	0,27	**2,69**	**0,01**	69213038	**192**	**923**	**15,1**
	R	6294	4500	1794	6	763	5247	109805	51142	0,35	**0,00**	**0,07**	**0,59**	**0,41**	0,27	3,67	0,02	69486208	**192**	**717**	**11,4**
hibernate	0	4170	3508	662	7	131	2345	71977	38629	0,21	0,00	0,14	0,85	0,48	0,19	12,42	0,07	5182331	78	885	21,2
	A	4315	3508	807	7	131	2490	72086	42674	0,23	**0,00**	**0,11**	0,85	**0,46**	0,20	**11,21**	**0,06**	5236544	83	**730**	**16,9**
	K	4206	3544	662	7	131	2345	71977	38629	0,21	**0,00**	0,14	0,85	0,48	**0,19**	14,52	**0,06**	5204964	88	**735**	**17,5**
	R	4402	3510	892	7	131	2575	72429	39202	0,21	**0,00**	**0,13**	**0,85**	**0,48**	0,19	40,27	**0,06**	5769582	90	**469**	**10,7**
jvs-vfs	0	8156	6914	1242	6	199	5033	120661	72726	0,16	0,00	0,19	0,71	0,54	0,43	13,08	0,02	27541919	416	3005	36,8
	A	8923	6914	2009	6	199	5800	122320	91533	0,22	**0,00**	**0,14**	**0,71**	**0,50**	0,43	**11,04**	**0,02**	27768033	434	**2622**	**29,4**
	K	8228	6986	1242	6	199	5033	120661	72726	**0,16**	**0,00**	**0,19**	**0,71**	**0,54**	**0,42**	13,37	**0,02**	27635578	418	**2836**	**34,5**
	R	8809	6916	1893	6	199	5684	121984	74606	0,17	**0,00**	**0,18**	**0,71**	**0,53**	0,43	113,30	0,05	29511065	457	**2360**	**26,8**
omnigene	0	3104	2857	247	3	80	791	16425	35209	0,48	0,00	0,05	0,94	0,06	0,06	0,77	0,01	2238251	83	355	11,4
	A	3192	2857	335	3	80	879	16972	37177	0,55	**0,00**	**0,05**	**0,94**	0,06	0,06	**0,68**	**0,01**	2279991	84	**330**	**10,3**
	K	3107	2860	247	3	80	791	16425	35209	0,48	**0,00**	0,05	**0,94**	0,06	0,06	**0,75**	**0,01**	2242822	84	**351**	**11,3**
	R	3186	2859	327	3	80	871	16661	35578	0,49	**0,00**	**0,05**	**0,94**	**0,06**	0,06	**0,69**	**0,01**	2283451	**82**	**252**	**7,9**
sapia	0	3148	2668	480	5	130	1952	26343	17324	0,44	0,00	0,11	0,83	0,22	0,23	1,02	0,01	1930692	102	393	12,5
	A	3218	2668	550	5	130	2022	26372	18025	0,45	**0,00**	**0,10**	**0,83**	**0,21**	0,24	**0,94**	**0,01**	1975349	**100**	**356**	**11,1**
	K	3154	2674	480	5	130	1952	26343	17324	0,44	**0,00**	**0,11**	**0,83**	0,22	0,23	1,03	0,01	1933784	102	**385**	**12,2**
	R	3187	2670	517	5	130	1989	26453	17490	0,44	**0,00**	**0,11**	**0,83**	**0,22**	0,23	**0,94**	0,01	1988118	**94**	**301**	**9,4**
squirrel-sql	0	2850	2258	592	4	197	1963	28468	17924	0,28	0,00	0,18	0,71	0,35	0,11	20,92	0,17	17474262	186	1160	40,7
	A	3126	2258	868	4	197	2239	28863	20981	0,33	**0,00**	**0,15**	0,71	**0,33**	0,12	21,58	**0,16**	17546902	187	**1104**	**35,3**
	K	2880	2288	592	4	197	1963	28468	17924	0,28	**0,00**	0,18	0,71	0,35	**0,11**	21,23	**0,17**	17490602	191	**1134**	**39,4**
	R	3032	2260	772	4	197	2143	28842	18427	0,29	**0,00**	**0,18**	0,71	**0,35**	0,11	34,90	**0,13**	17994024	213	**872**	**28,8**
x4l-reload	0	3980	3271	709	7	104	3531	54306	40347	0,21	0,00	0,11	0,46	0,45	0,77	11,66	0,06	8059099	96	1703	42,8
	A	4348	3271	1077	7	104	3899	54557	49267	0,26	**0,00**	**0,08**	0,46	**0,42**	0,77	**9,40**	**0,06**	8083606	109	**1607**	**37,0**
	K	3999	3290	709	7	104	3531	54306	40347	**0,21**	**0,00**	0,11	0,46	0,45	**0,77**	11,72	**0,06**	**8046373**	96	**1686**	**42,2**
	R	4213	3273	940	7	104	3762	54985	41192	0,22	**0,00**	**0,11**	0,46	**0,44**	0,77	165,04	**0,06**	8659909	159	**1245**	**29,6**

Tabelle 7.5: Exemplarische Metrikänderungen bei den zehn größten untersuchten Systemen
Metrikwerte im **0** Ausgangszustand und nach **A** Abhängigkeitsumkehr, **K** Knotenspaltung, **R** Rückrufen

kumulierte Abhängigkeiten trägt. Bei exzessiver Anwendung führt dies zu einer beträchtlichen Zunahme der kumulierten Komponentenabhängigkeit und führt die Reduktion der Klassen in Zyklengruppen ad absurdum. Der Einsatz der Rückrufe hat daher mit Bedacht zu erfolgen.

Die Verbreitungskosten (VB) entwickeln sich in sechs der zehn Applikationen für alle Auflösungstechniken positiv, auch unter den restlichen vier findet für mindestens eine Technik eine Reduktion statt. Die systemweite Kopplung scheint durch die Anwendung von Auflösungstechniken abzunehmen.

		validierte Metriken														nicht validiert			
Zl	**T**	**ΔAK**	**ΔAKK**	**ΔAAK**	**ΔMVT**	**ΔMVB**	**ΔVB**	**ΔAEM**	**ΔAM**	**ΔPF**	**ΔCF**	**ΔMHF**	**ΔAHF**	**ΔMIF**	**ΔAIF**	**ΔNKKA**	**ΔVK**	**ΔBL**	**ΔZG**
A	ρ	**-0,41**	–	**-0,41**	**0,07**	–	**-0,41**	**-0,31**	**-0,39**	0,06	**-0,22**	**0,14**	**0,13**	**0,15**	**-0,10**	**0,24**	**0,15**	**-0,23**	**-0,10**
	p	**0,00**	–	**0,00**	**0,00**	–	**0,00**	**0,00**	**0,00**	0,01	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**
K	ρ	**-0,54**	**-0,54**	–	–	–	–	–	–	-0,03	**-0,16**	-0,00	0,02	**0,14**	**0,17**	**0,26**	**0,28**	-0,06	**-0,14**
	p	**0,00**	**0,00**	–	–	–	–	–	–	0,39	**0,00**	0,99	0,56	**0,00**	**0,00**	**0,00**	**0,00**	0,10	**0,00**
R	ρ	**-0,69**	**-0,31**	**-0,69**	**0,13**	–	**-0,69**	**-0,78**	**-0,78**	**-0,06**	**-0,24**	**0,42**	**0,35**	**0,45**	–	0,05	**0,13**	**-0,76**	**-0,12**
	p	**0,00**	**0,00**	**0,00**	**0,00**	–	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	**0,00**	–	0,01	**0,00**	**0,00**	**0,00**

Tabelle 7.6: Korrelation von AZG mit allen Systemmetriken bei α=0,01
A Abhängigkeitsumkehr **K** Knotenspaltung **R** Rückrufe **ρ** Korrelationskoeffizient
p Überschreitungswahrscheinlichkeit

Die Beschreibungslänge (BL) nimmt im Durchschnitt zu. Lediglich bei cayenne nimmt sie mehrheitlich ab, sowie bei vereinzelten anderen Anwendungen bei der Knotenspaltung. Normalerweise ist bei einer Zunahme der Artefakte eine Zunahme der Beschreibungslänge zu erwarten. In Einzelfällen rücken durch Zyklenauflösungen Artefakte »näher zusammen«, wodurch die Anzahl der Bits, um ihre Abhängigkeiten zu beschreiben, abnimmt.

Die Anzahl der Zyklengruppen (ZG) nehmen in der Regel zu. Lediglich wenn Zyklengruppen vollständig aufgelöst werden, nimmt ihre Anzahl ab.

Die Metriken AZG und AZG% dienten zur Fortschrittsmessung und sanken daher in jedem Fall. Im Durchschnitt reduzierte die Abhängigkeitsumkehr AZG% um 4,2 Prozentpunkte ($\pm$2,5), die Knotenspaltung um 1,0 Prozentpunkte ($\pm$1,2) und die Rückrufe um 8,7 Prozentpunkte ($\pm$4,2).

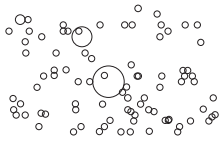
7.3.3 Systematische Metrikveränderungen

Nach der exemplarischen Betrachtung der Metrikveränderungen einiger Anwendungen aus der Stichprobe wenden wir uns einer statistisch aussagekräftigen Analyse zu, um die Beobachtungen zu bekräftigen oder zu verwerfen.

Tabelle 7.6 führt die Korrelation jeder untersuchten Systemmetrik mit der Metrik AZG auf, wobei fettgedruckte Werte hochsignifikant sind (mit α=0,01). Die leeren Zellen (–) repräsentieren Verteilungen, die durch die Auflösungstechniken nicht verändert wurden und somit keinen Korrelationskoeffizienten besitzen. Die ersten zwei Zeilen spiegeln Korrelationen bei Anwendung der Abhängigkeitsumkehr wider, die mittleren zwei Zeilen bei Anwendung der Knotenspaltung und die letzten bei Anwendung der Rückrufe.

Für die Abhängigkeitsumkehr sind alle Korrelationen mit Ausnahme von ΔPF hochsignifikant. Die Systemebenenmetriken ΔAK, ΔAAK, ΔVB, ΔAEM und ΔAM weisen eine schwache negative Korrelation auf, nehmen also zu, wenn AZG abnimmt. Dies entspricht den Erwartungen. ΔMVT korreliert nicht mit AZG, da sich die Einführung von Schnittstellen nur selten auf die maximale Vererbungstiefe auswirkt.

Die MOOD-Metriken korrelieren mit der Abnahme von AZG nicht beziehungs-



weise nur sehr schwach (ΔMHF, ΔAHF, ΔMIF, ΔAIF). Lediglich der Kopplungsfaktor (CF) zeigt eine leichte Tendenz zum Anstieg, wenn AZG absinkt. ΔPF ist nicht signifikant.

Von den nichtvalidierten Metriken korrelieren ΔNKKA schwach positiv und ΔBL schwach negativ mit der Abnahme von AZG. Dies entspricht den Erwartungen. Die sehr schwache Korrelation von ΔZG weist darauf hin, dass die Anzahl der Zyklengruppen durch die Anwendung der Auflösungstechniken nur sehr gering ansteigt.

Die Knotenspaltung führt bei den Systemebenenmetriken eine schwache negative Korrelation mit ΔAK und ΔAKK herbei. Diese entspricht den Erwartungen, da die Knotenspaltung eine geringe Zunahme der Anzahl konkreter Klassen verursacht.

Bei den MOOD-Metriken lassen sich nur bei ΔCF, ΔMIF und ΔAIF signifikante Korrelationen feststellen, die sehr schwach ausfallen. Knotenspaltungen wirken sich auf die MOOD-Metriken daher nicht in ausreichendem Maße aus.

ΔNKKA und ΔVK korrelieren schwach positiv mit der Abnahme von AZG. Dies entspricht den Erwartungen. Die sehr schwache positive Korrelation mit ΔBL weist darauf hin, dass die Beschreibungslänge im Zuge der Knotenspaltung abzunehmen pflegt, wenn AZG abnimmt. Dies entspricht auch den exemplarischen Beobachtungen von BL.

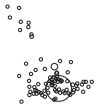
Für die Rückrufe sind alle Systemmetrikkorrelationen hochsignifikant. Dabei zeigen sich starke negative Korrelationen bei ΔAK, ΔAAK, ΔVM, ΔAEM und ΔAM. Diese starken Steigerungen rühren von der Erzeugung und Benutzung von Einzelmethodenschnittstellen her und entsprechen den Erwartungen. ΔMVT korreliert nicht mit AZG, da sich die Einführung von Schnittstellen nur selten auf die maximale Vererbungstiefe auswirkt.

Bei den MOOD-Metriken treten bei ΔPF und ΔCF je eine schwache negative sowie bei ΔMHF, ΔAHF und ΔMIF je eine schwache positive Korrelation auf. Die positiven Korrelationen (die Metrik sinkt, wenn AZG sinkt) erklären sich mit der Hinzufügung von Klassen und Methoden, sodass eine insgesamt größere Menge von Artefakten zu einer Schrumpfung der verschiedenen Faktoren führt. Der Anstieg des Kopplungsfaktors ΔCF bei sinkendem AZG lässt sich nicht unmittelbar erklären, da die Anzahl der Klassen nicht sinkt. Es ist daher anzunehmen, dass die Anwendung der Rückrufe zu einer höheren Kopplung im System führt.

ΔNKKA wird überhaupt nicht von AZG-Änderungen beeinflusst, ΔVK und ΔZG nur minimal.

Die fehlende Signifikanz von ΔNKKA verallgemeinert die Erkenntnis in der exemplarischen Auswertung, dass die NKKA-Werte je nach Softwaresystem extrem vom Ausgangswert abweichen, also jede Position zwischen eindeutiger NKKA-Verringerung bis hin zu einer vielfachen Erhöhung einnehmen können. Die Rückrufe führen daher keinesfalls immer zu einer Verringerung der kumulierten Komponentenabhängigkeit.

Damit zusammenhängend lässt sich analog die starke negative Korrelation mit

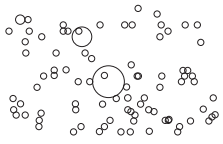


der Beschreibungslänge ΔBL erklären. Durch die spezielle Implementierung der Rückrufe über eine globale Verteilerklasse erfolgen zusätzliche Methodenaufrufe über Paketgrenzen hinweg, sodass die Beschreibung dieser Methodenaufrufe eine größere Anzahl Bits benötigt.

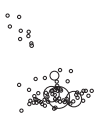
Im allgemeinen entspricht die Korrelation der Systemebenenmetriken und der nicht validierten Metriken den Erwartungen. Die MOOD-Metriken korrelieren zu schwach oder überhaupt nicht, um eine Auswirkung der Zyklenauflösung auf irgendwelche durch die MOOD-Metriken beschriebenen Eigenschaften feststellen zu können.

Die Abhängigkeitsumkehr und die Knotenspaltung wirken sich positiv auf die kumulierte Komponentenabhängigkeit aus, was einem verringerten Testaufwand entspricht. Alle Auflösungstechniken verringern die über VK ausgedrückte systemweite Kopplung.

Wir kennen nun die statistischen Auswirkungen von Auflösungstechniken auf Softwaresysteme. Der Versuchsaufbau entspricht jedoch nicht annähernd einer Vorgangsweise, die ein Mensch an den Tag legen würde, wenn er zyklische Abhängigkeiten eines Softwaresystems auflösen wollte. Im nächsten Kapitel untersuchen wir daher die Anwendung von Auflösungstechniken unter diesem Gesichtspunkt.



5 Fr/imp-a



Kapitel 8

Anwendung

Im letzten Kapitel versuchten wir, vollautomatisch Rückbezugskanten in Softwaresystemen aufzulösen und werteten die Ergebnisse statistisch aus. Dadurch konnten wir zwar die Charakteristika der einzelnen Auflösungstechniken testen, wussten aber nicht, wie sich die Auflösungstechniken in der Praxis anwenden lassen, um zyklische Abhängigkeiten in Softwaresystemen zu reduzieren.

Wir wandten in diesem Kapitel daher die Auflösungstechniken *Abhängigkeitsumkehr*, *gerichtete Knotenspaltung* sowie *Rückrufe* an fünf realen Java-Applikationen an, indem wir schrittweise nach zyklischen Abhängigkeiten suchten und dann lohnende Abhängigkeiten automatisiert auflösten, soweit dies möglich war.

Dazu benutzten wir die Eclipse-Integration von Meinges (siehe Kap. 6) und prüften die einhergehende Veränderung von Systemmetriken, maßen den Zeitaufwand für die Durchführung der Auflösung und ließen Beteiligte an den untersuchten Softwaresystemen zu unseren Änderungen Stellung nehmen.

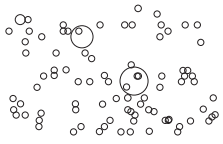
8.1 Vorgehensweise

Die Anwendung der Auflösungstechniken erfolgte nach bestimmten Kriterien, die hier aufgeführt und erklärt werden. Als Konvention heißt jene Person *Aktor*, die das Werkzeug zur Zyklenauflösung bedient.

8.1.1 Auswahl eines geeigneten Softwaresystems

Ein Softwaresystem kommt für die Untersuchung in Frage, wenn es bestimmte Eigenschaften besitzt, die im Interesse einer Untersuchung liegen. Dabei müssen folgende Bedingungen erfüllt sein.

1. Das Softwaresystem weist Zyklengruppen auf.
2. Die Minimalgröße des Systems liegt über 100 Klassen.
3. Das System weist Produktqualität auf.



4. Das System befindet oder befand sich in tatsächlichem Einsatz.

Für alle untersuchten Systeme gelten folgende Bedingungen.

1. In mindestens einem Softwaresystem befinden sich mehr als 80% der Klassen in Zyklengruppen.
2. In mindestens einem Softwaresystem befinden sich weniger als 20% der Klassen in Zyklengruppen, aber mehr als 0%.
3. In mindestens einem Softwaresystem befinden soviele Klassen in Zyklengruppen, sodass sie nicht mehr als $\pm$10% vom Median des Anteils an Klassen in Zyklengruppen abweichen. (Der Median wurde in Kapitel 7.3 bestimmt und beträgt 18,5%.
4. Keine zwei Softwaresysteme gehören demselben Gebiet an.

8.1.2 Vorbereitung

Das Softwaresystem ist so anzupassen, dass es in Eclipse 3.1 lädt und übersetzt.

Der Ausgangszustand ist zu sichern.

8.1.3 Durchführung der Zyklenauflösung

Die Zyklenauflösung erfolgt in Schritten. In jedem Schritt wendet der Aktor eine Auflösungstechnik an. Diese Schritte heißen *Auflösungsschritte*.

Für die Zyklenauflösung selbst werden keine festen Vorschriften gegeben, sondern lediglich Richtlinien, da verschiedene Softwaresysteme verschiedene Lösungsansätze erfordern. Allerdings ist es zweckmäßig, mit dem besten Fragmentierer zu beginnen, da sein Herauslösen aus der Zyklengruppe den größten Fortschritt bringt.

Wurde eine entsprechende aufzulösende Abhängigkeit gefunden, so ist eine geeignete Auflösungstechnik anzuwenden. Ist dies nicht möglich, kann der Aktor die Anwendung einer Auflösungstechnik erzwingen oder eine händische Anpassung des Quelltextes vornehmen.

Nach jeder Anwendung einer Auflösungstechnik ist der Zustand des Softwaresystems auf Quelltextebene zu sichern.

Der Schritt ist damit abgeschlossen, und ein neuer Schritt beginnt.

Die Zyklenauflösung kann beendet werden, wenn keine Zyklengruppen mit mehr als 15 Klassen existieren, die NKKA-Metrik 1.0 erreicht oder keine Auflösungstechnik mehr angewandt werden kann. Harmlose Zyklengruppen [78] mit über 15 Klassen fallen nicht ins Gewicht.

Dieser Zustand heißt *Endzustand* des untersuchten Softwaresystems.

8.1.4 Beurteilung des Endzustands

Im Endzustand werden die erhobenen Metrikwerte mit den Metrikwerten des Ausgangszustands verglichen und interpretiert.

8.1.5 Phasen und Zeitmessung

Jeder Auflösungsschritt ist zwecks besserer Vergleichbarkeit in Phasen unterteilt, nämlich Analyse, Aufbereitung, automatischen Umbau, Fehlerbehebung und manuellen Umbau.

Analyse In der Analysephase untersucht der Aktor das System, um die nächste aufzulösende Abhängigkeit zu finden.

Aufbereitung In der Aufbereitung passt der Aktor den Quelltext so an, dass ihn Meinges verarbeiten kann. Diese Phase ist optional.

Automatischer Umbau In dieser Phase stößt der Aktor die automatische Durchführung einer Auflösungstechnik an.

Fehlerbehebung Sollte die automatische Durchführung Fehler im Quelltext nach sich ziehen, was durch Fehler in Meinges möglich ist, kann der Aktor die Fehler in dieser Phase beheben. Diese Phase ist optional.

Manueller Umbau Nicht alle Beziehungen lassen sich automatisch auflösen. Daher kann der Aktor Beziehungen in dieser Phase manuell auflösen. Diese Phase ist optional.

Für statistische Zwecke ist die Zeitdauer einzelner Phasen für die Untersuchung eines Systems festzuhalten. Die Einheit beträgt Mannstunden.

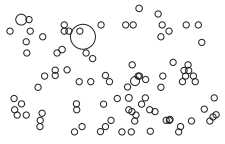
Das Flussdiagramm in Abbildung 8.1 visualisiert das Zusammenspiel von Schritten und Phasen. Außer den Phasen Analyse und automatischer Umbau sind alle Phasen optional.

8.1.6 Manuelle Schätzung

In der Phase des automatischen Umbaus führt Meinges eine Reihe elementarer Umbauoperationen durch, um die entsprechende Abhängigkeit aufzulösen. Die Reihenfolge und Art der elementaren Umbauoperationen wird in ein Änderungsprotokoll geschrieben. Aus diesem Änderungsprotokoll schätzen wir die Zeitdauer, die ein Mensch benötigt hätte, wenn er die Abhängigkeit manuell hätte auflösen wollen.

Tabelle 8.1 listet sämtliche elementaren Umbauoperationen der drei implementierten Auflösungstechniken auf nebst ihrer geschätzten Dauer in Minuten bei manueller Durchführung. *Elementare Umbauoperationen* sind Änderungen, die direkt am Quelltext vorgenommen werden. Die Dauer ist jeweils konservativ geschätzt und berücksichtigt den Umstand, dass ein Entwickler nicht durchgehend wie eine Maschine Umbauoperationen durchführen kann.

Die geschätzte Zeit mag immer noch niedrig erscheinen. Dabei ist zu berücksichtigen, dass der mentale Aufwand (Ist die Änderung zweckmäßig? Welche Teile sind zu ändern?) weitgehend der Analysephase zuzuschlagen ist.



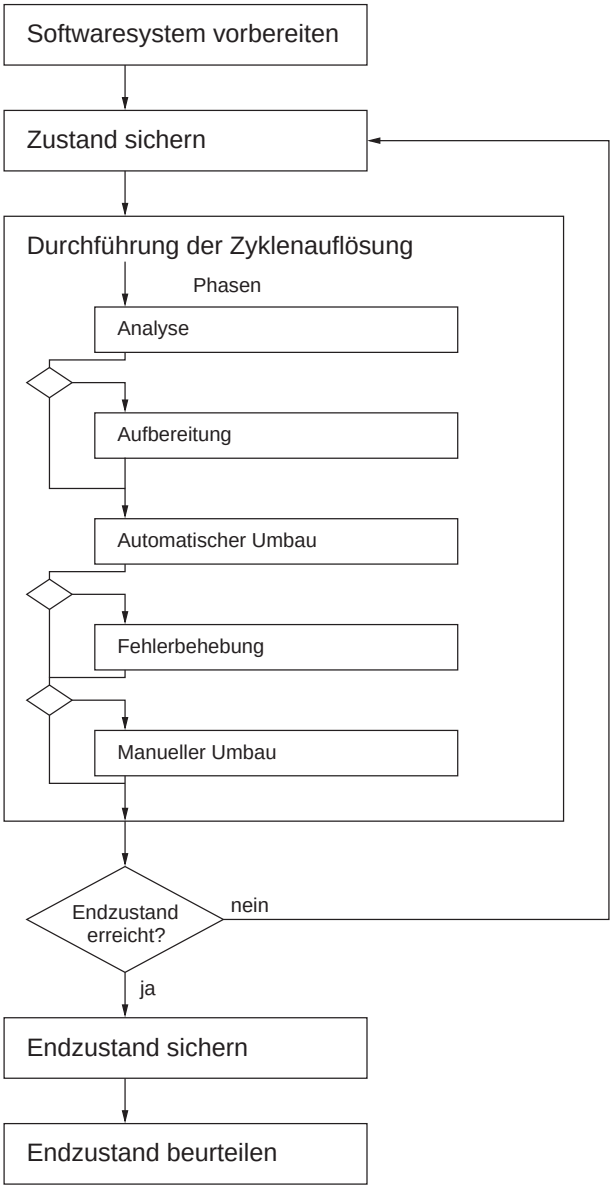


Abbildung 8.1: Ablauf der Anwendung von Auflösungstechniken auf ein Softwaresystem

Die gesamte manuelle Dauer ergibt sich aus der Addition aller angewandten elementaren Umbauoperationen gemäß Änderungsprotokoll jeweils mit der in Tabelle 8.1 aufgeführten Dauer.

Die in der Tabelle aufgeführten elementaren Umbauoperationen unterscheiden sich von den Umbauoperationen in Kapitel 5.2 insofern, als dass erstere sich auf die Quelltextebene, letztere sich auf die Ebene des FAMIX+-Modells beziehen.

Umbauoperation	Dauer
Klasse erzeugen	5 min
Von Klasse erben	0,5 min
Methode erzeugen	1 min
Attribut erzeugen	1 min
Methode verschieben	2 min
Attribut verschieben	2 min
Methode extrahieren	1 min
Attribut heben	1 min
Angebundenen Ausdruck ändern	0,5 min
Typ angebundener Variable ändern	1 min
Methoden-/Konstruktoraufruf einfügen	0,5 min
Methoden-/Konstruktoraufruf ersetzen	1 min

Tabelle 8.1: Schätzung der Dauer einer manuell durchgeführten elementaren Umbauoperation

8.1.7 Akzeptanztest

Die vorgestellten Auflösungstechniken nehmen Änderungen am Quelltext des untersuchten Softwaresystems vor. Wir prüfen daher bei allen Systemen die Akzeptanz der automatisch vorgenommenen Änderungen durch die ursprünglich an der Entwicklung des Softwaresystems Beteiligten.

Der Akzeptanztest ist die Befragung eines Entwicklers zur Identifikation mit dem geänderten Softwaresystem und liefert eine qualitative Aussage über die Bereitschaft, mit einem automatisiert veränderten Softwaresystem weiterarbeiten zu wollen.

Die Zeit der zur Verfügung stehenden Entwickler ist allerdings begrenzt. Deswegen befragen wir die Entwickler nicht zu jedem einzelnen Auflösungsschritt, sondern unterbreiten ihnen den Quelltext im Endzustand und ein textuelles Delta zwischen Ausgangszustand und Endzustand im Format `diff -u` sowie eine Kurzübersicht über die durchgeführten Änderungen.

Der Fragebogen lautet wie folgt:

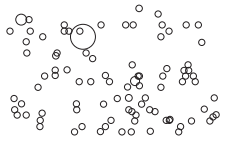
Angaben zur Person

Ihre Rolle in der Erstellung des Softwaresystems (z. B. Programmierer, SW-Ingenieur, Systemarchitekt)

Fragen

1. Erkennen Sie das Softwaresystem wieder (ja/nein)? Wenn nein, warum nicht?

2. Halten Sie die Änderungen für sinnvoll bzw. für eine Verbesserung (ja/nein)? Wenn nein, warum nicht?



Anwendung	Version	Gebiet	Beschreibung
RoboCode	1.5.4	Künstliche Intelligenz	Programmierung von Kampfrobotern
ArgoUML	0.24	SW-Entwurf/-Architektur	UML-Erstellungswerkzeug
Projectfactory	0.6	Planung	Aufgaben- und Projektplanung
imp-a	–	Gestaltung	Gestaltung von Benutzeroberflächen
imp-b	–	Verwaltung	Verwaltet Konfigurationen von Softwaresystemen

Tabelle 8.2: Liste der untersuchten Anwendungen

3. Würden Sie an diesem automatisiert umgebauten Softwaresystem weiterarbeiten können (ja/nein)? Wenn nein, warum nicht?

4. Würden Sie an diesem automatisiert umgebauten Softwaresystem weiterarbeiten wollen (ja/nein)? Wenn nein, warum nicht?

Je höher die Anzahl der positiven Antworten, desto positiver ist die Akzeptanz automatisch durchgeführter Änderungen zu werten.

Eine rein statistische Auswertung der binären Antworten führt allerdings nicht zum Ziel, da die Befragten durch die offene Struktur des Fragebogens (freie Texteingabe) beliebige Kommentare und Anmerkungen vornehmen können.

Diese Entscheidung wurde bewusst gewählt, um die Gedanken und Eindrücke der Befragten besser nachvollziehen zu können. Die Anmerkungen fließen über die binäre Entscheidung (ja/nein) maßgeblich in die Interpretation mit ein.

8.2 Anwendung der Auflösungstechniken

Wir suchten fünf Java-Applikationen aus, die den Vorgaben in Kapitel 8.1 entsprachen und reduzierten die darin enthaltenen zyklischen Abhängigkeiten. Die Applikationen sind aus Tabelle 8.2 zu entnehmen.

Die Beschreibung der Zyklenauflösung ist für jede Applikation ist in einem eigenen Unterkapitel mit ähnlichem Aufbau zusammengefasst. Die Diagramme zeigen sowohl die Zyklengruppen als auch die Fragmentierer der jeweils größten Zyklengruppe im Ausgangs- und im Endzustand an. Weitere Diagramme repräsentieren den Gesamtzeitaufwand je Phase und die Zyklengruppenreduktion gemessen an der Abnahme der Artefakte in Zyklengruppen (AZG) je Schritt.

In letzterem Diagramm stellt die schwarze Kurve AZG und die graue Kurve die Anzahl der Klassen (AK) dar. Die Zeitdauer der Auflösung ist je Schritt als Balken überlagert angezeigt, um auf einen Blick den mit einem Schritt verbundenen Aufwand erkennen zu können. Die vertikalen grauen Striche markieren jeweils eine erfolgte Senkung von AZG.

Die Applikationen imp-a und imp-b unterliegen Geheimhaltungsbestimmungen. Sämtliche Namen, die sich auf Artefakte aus diesen Applikationen beziehen,

8 Fr/imp-a

Schlüssel	Beschreibung
Name	RoboCode
Version	1.5.4
Anbieter	robocode.sourceforge.net
Lizenz	CPL

Tabelle 8.3: Übersicht über RoboCode

sind in verschlüsselter Form wiedergegeben.

Der Leser wird bereits bemerkt haben, dass sich am unteren Rand dieses Buches Bilder von Ansichten befinden. Sie spiegeln den Fortschritt der Zyklenreduktionen für jede hier behandelte Applikation wider. In jedem Bild ist die Nummer des Auflösungsschrittes gefolgt vom Typ der Ansicht (ZG – Zyklengruppen, Fr – Fragmentierer) und dem Namen des Softwaresystems verzeichnet.

Blättert der Leser das Buch schnell von vorne nach hinten, sieht er auf den ungeraden Seiten, wie sich die Zyklengruppen im Laufe der Zeit verkleinern. Die geraden Seiten zeigen die Fragmentierer der jeweils größten Zyklengruppe. An den Fragmentierern lässt sich verfolgen, wie schnell eine Zyklengruppe schrumpft. Zur besseren Übersichtlichkeit enthalten die Bilder keine Kanten zwischen den Knoten.

Diese unorthodoxe Darstellung ist eine praktische Möglichkeit, den Fortschritt der Untersuchung ohne viele Worte in einem statischen Medium wiederzugeben.

8.2.1 RoboCode

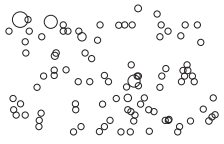
RoboCode ermöglicht die Programmierung von virtuellen Kampfrobotern, die dann auf einem virtuellen Schlachtfeld gegeneinander antreten (siehe auch Tabelle 8.3). Mit jedem Roboter ist ein Steuerungscode verknüpft, der seine Handlungen und Reaktionen auf die Umwelt festlegt.

Die Applikation bestand im Ausgangszustand aus 232 Klassen, von denen sich 142 in Zyklengruppen befanden. Ferner enthielt RoboCode 9 Zyklengruppen, von denen sich die größte aus 124 Klassen zusammensetzte.

In einer Folge von 7 Schritten konnten wir die Anzahl der Klassen in Zyklengruppen um nahezu 50 auf 97 reduzieren. Der prozentuelle Anteil der Klassen in Zyklengruppen verglichen mit der Gesamtanzahl fiel von 61% auf 41%, die größte Zyklengruppe enthielt am Ende nur noch 11 Klassen.

Tabelle 8.4 zeigt die Veränderung sämtlicher Metriken über alle Schritte dieses Zyklenauflösungsprozesses.

Im Ausgangszustand präsentierte sich das System wie aus Abbildung 8.2 ersichtlich. Die Zyklenauflösung erfolgte in drei Abschnitten. Die größte Zyklengruppe wurde so lange reduziert, bis sie in Zyklengruppen mit nicht mehr als 15 Klassen zerfallen war. In den nächsten beiden Abschnitten reduzierten wir bis zur Erfüllung einer Endbedingung die jeweils danach größte Zyklengruppe, bis ein



	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Ausgangszustand	232	219	13	6	14	59	3596	2816	0,12	0,01	0,25	0,88	0,24	0,07	10,5	0,54	111799,3	9	142	61%
Schritt 1	233	219	14	6	14	60	3596	2842	0,14	0,01	0,24	0,88	0,23	0,07	10,4	0,54	111862,2	9	142	61%
Schritt 2	234	219	15	6	14	61	3596	2862	0,16	0,01	0,24	0,88	0,23	0,07	2,8	0,24	112703,0	22	126	54%
Schritt 3	235	219	16	6	14	62	3596	2922	0,20	0,01	0,24	0,88	0,23	0,07	2,7	0,24	113149,1	22	125	53%
Schritt 4	236	219	17	6	14	64	3667	3061	0,29	0,01	0,23	0,88	0,22	0,07	1,9	0,17	113897,9	24	109	46%
Schritt 5	237	219	18	6	14	65	3667	3091	0,31	0,01	0,22	0,88	0,22	0,07	1,5	0,14	114202,0	26	105	44%
Schritt 6	238	219	19	6	14	66	3687	3117	0,32	0,01	0,22	0,88	0,22	0,07	1,5	0,13	114351,1	29	101	42%
Schritt 7	239	219	20	6	14	67	3687	3162	0,34	0,01	0,22	0,88	0,22	0,07	1,4	0,12	114852,0	30	97	41%

Tabelle 8.4: Metrikübersicht über alle Auflösungsschritte von RoboCode

Schritt	Auflösungstechnik	ΔAna	ΔAuf	ΔAut	ΔFeh	ΔMan	ΔSch	#Auf	#Aut	#Feh	#Man	∑	∑(M)
Schritt 0	Abhängigkeitsumkehr	0,25	0,10	0,10	0,00	0,25	0,52	1	28	0	3	0,70	1,12
Schritt 1	Abhängigkeitsumkehr	0,20	0,00	0,10	0,30	0,00	0,82	0	46	26	0	0,60	1,32
Schritt 2	Abhängigkeitsumkehr	0,20	0,00	0,10	0,40	0,00	1,36	0	78	7	0	0,70	1,96
Schritt 3	Abhängigkeitsumkehr	0,25	0,00	0,10	0,50	1,20	3,79	0	226	35	114	2,05	5,74
Schritt 4	Abhängigkeitsumkehr	0,20	0,00	0,10	0,10	0,00	0,69	0	38	3	0	0,40	0,99
Schritt 5	Abhängigkeitsumkehr	0,20	0,00	0,10	0,25	0,20	0,56	0	30	15	9	0,75	1,21
Schritt 6	Abhängigkeitsumkehr	0,25	0,00	0,10	0,10	0,00	0,88	0	49	2	0	0,45	1,23
Schritt 7		0,20	0,00	0,00	0,00	0,00	0,00	0	0	0	0	0,20	0,20

Tabelle 8.5: Aufwandsübersicht über alle Auflösungsschritte von RoboCode
Zeitaufwand in Std. für Δ**Ana** Analyse, Δ**Auf** Aufbereitung, Δ**Aut** automatischen Umbau, Δ**Feh** Fehlerbehebung und Δ**Man** manuellen Umbau. Δ**Sch** Manuelle Schätzung für ΔAut. Anzahl elementarer Umbauoperationen für **#Ana** Analyse, **#Auf** Aufbereitung, **#Aut** automatischen Umbau, **#Feh** Fehlerbehebung und **#Man** manuellen Umbau. $\sum = \Delta\text{Ana} + \Delta\text{Auf} + \Delta\text{Aut} + \Delta\text{Feh} + \Delta\text{Man}$, $\sum(\text{M}) = \Delta\text{Ana} + \Delta\text{Auf} + \Delta\text{Feh} + \Delta\text{Man} + \Delta\text{Sch}$

Endzustand erreicht war (siehe Abb. 8.3). Den Verlauf der Auflösung gibt Abbildung 8.5 wieder.

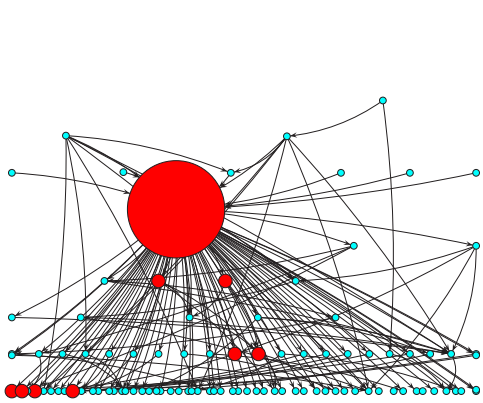
Die Zyklenreduktion von RoboCode ließ sich in relativ wenigen Schritten bewerkstelligen, da die besten Fragmentierer ausschließlich über Instanzmethoden angesprochen wurden und daher ein natürliches Ziel für eine Abhängigkeitsumkehr darstellten. Die Größe der einzelnen Zyklengruppen konnte stark reduziert werden, was sich in einem auf nahezu ein Zehntel geschrumpften NKKA-Wert widerspiegelt. Die Anzahl der Artefakte in Zyklengruppen nahm insgesamt jedoch nur um 20 Prozentpunkte ab.

8.2.2 ArgoUML

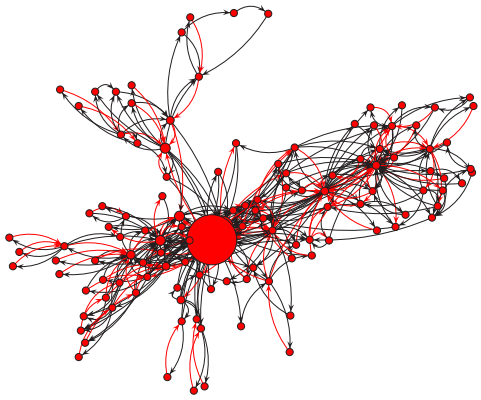
ArgoUML ist ein Werkzeug zur Erstellung und Verwaltung von UML-Diagrammen (siehe Tabelle 8.6).

Die Applikation bestand im Ausgangszustand aus 1618 Klassen, von denen sich 905 in Zyklengruppen befanden. Ferner enthielt ArgoUML 12 Zyklengruppen, von denen die größte aus 871 Klassen bestand.

Die riesige Zyklengruppe schuldet ArgoUML vor allem dem Umstand, dass das Softwaresystem aus drei Singletonklassen besteht, deren Kombination aus Klas-

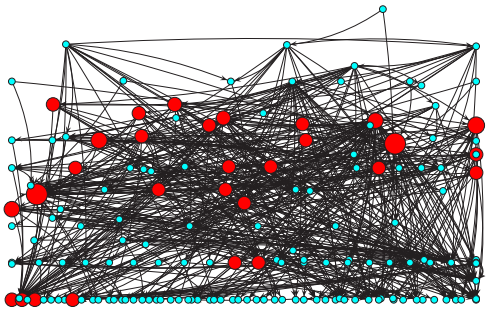


(a) Zyklengruppen in Schritt 0

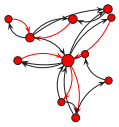


(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.2: RoboCode-Repräsentation im Ausgangszustand



(a) Zyklengruppen in Schritt 6



(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.3: RoboCode-Repräsentation im Endzustand

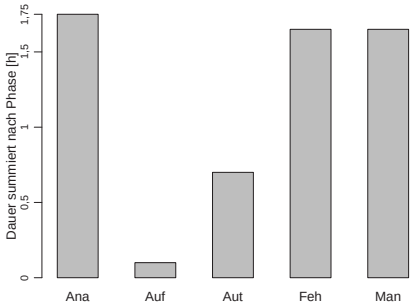


Abbildung 8.4: Gesamtzeitaufwand je Phase von RoboCode

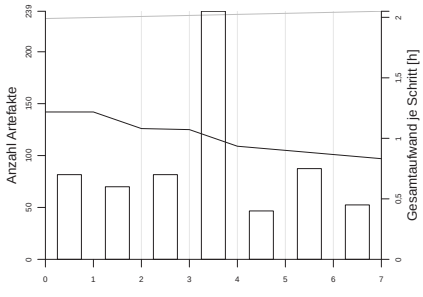
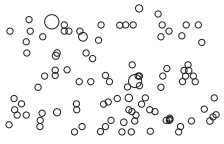


Abbildung 8.5: Zyklengruppenreduktion in RoboCode



Schlüssel	Beschreibung
Name	ArgoUML
Version	0.24
Anbieter	argouml.tigris.org
Lizenz	BSD

Tabelle 8.6: Übersicht über ArgoUML

	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Ausgangszustand	1618	1380	238	9	176	1109	23673	11603	0,11	0,00	0,27	0,81	0,70	0,39	65,1	0,53	14705827,4	12	905	56%
Schritt 1	1619	1380	239	9	176	1110	23677	11644	0,11	0,00	0,27	0,81	0,70	0,39	65,0	0,53	14750250,4	12	905	56%
Schritt 2	1620	1381	239	9	176	1110	23681	11645	0,11	0,00	0,27	0,81	0,70	0,39	65,1	0,53	14749863,5	12	906	56%
Schritt 3	1623	1383	240	9	176	1111	23683	11651	0,11	0,00	0,27	0,81	0,70	0,39	56,3	0,23	14754341,6	31	416	26%
Schritt 4	1624	1383	241	9	176	1112	23683	11657	0,11	0,00	0,27	0,81	0,70	0,39	56,2	0,23	14761784,3	31	416	26%
Schritt 5	1625	1384	241	9	176	1112	23684	11658	0,11	0,00	0,27	0,81	0,70	0,39	56,4	0,23	14761854,8	31	417	26%
Schritt 6	1625	1384	241	9	176	1112	23686	11661	0,11	0,00	0,27	0,81	0,70	0,39	25,3	0,14	14763507,2	35	279	17%
Schritt 7	1626	1384	242	9	176	1113	23686	11682	0,11	0,00	0,27	0,81	0,70	0,39	25,2	0,14	14767993,6	35	279	17%
Schritt 8	1627	1385	242	9	176	1113	23687	11683	0,11	0,00	0,27	0,81	0,70	0,39	25,4	0,14	14768068,8	35	280	17%
Schritt 9	1627	1385	242	9	176	1113	23689	11686	0,11	0,00	0,27	0,81	0,70	0,39	10,6	0,10	14768623,0	37	241	15%
Schritt 10	1628	1385	243	9	176	1114	23689	11707	0,11	0,00	0,27	0,81	0,70	0,39	8,3	0,10	14768693,1	38	221	14%
Schritt 11	1629	1385	244	9	176	1115	23689	11775	0,11	0,00	0,26	0,81	0,70	0,39	8,1	0,09	14768435,0	38	218	13%
Schritt 12	1629	1385	244	9	176	1115	23691	11778	0,11	0,00	0,26	0,81	0,70	0,39	9,1	0,08	14772612,5	40	208	13%
Schritt 13	1630	1385	245	9	176	1116	23692	11802	0,12	0,00	0,26	0,81	0,70	0,39	9,1	0,08	15075475,0	40	208	13%
Schritt 14	1631	1386	245	9	176	1116	23694	11804	0,12	0,00	0,26	0,81	0,70	0,39	9,1	0,09	15070955,0	40	209	13%
Schritt 15	1631	1386	245	9	176	1116	23696	11807	0,12	0,00	0,26	0,81	0,70	0,39	6,5	0,07	15075305,4	41	195	12%

Tabelle 8.7: Metrikübersicht über alle Auflösungsschritte von ArgoUML

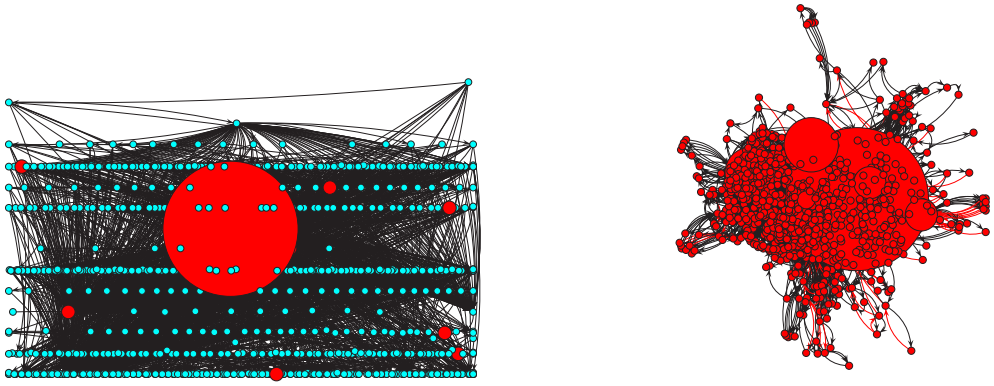
sen- und Instanzmethoden verschiedene Teile des Softwaresystems über Zyklen stark zusammenbindet.

In einer Folge von 15 Schritten konnten wir die Anzahl der Klassen in Zyklengruppen um über 700 auf 195 reduzieren. Der prozentuelle Anteil der Klassen in Zyklengruppen verglichen mit der Gesamtanzahl fiel von 56% auf lediglich 12%. Enthielt im Anfangszustand die größte Zyklengruppe noch 871 Klassen, so bestand die am Ende übriggebliebene größte Zyklengruppe nur noch aus 27 Klassen.

Tabelle 8.7 zeigt die Veränderung sämtlicher Metriken über alle Schritte dieses Zyklenauflösungsprozesses.

Im Ausgangszustand präsentierte sich das System wie aus Abbildung 8.6 ersichtlich. Die Zyklenauflösung erfolgte in drei Abschnitten. Der erste Abschnitt umfasste die starke Reduktion der größten Zyklengruppe nach einem festen Schema in Dreierschritten. Im zweiten Abschnitt war das große Auflösungspotential erschöpft, und wir wandten ad-hoc Maßnahmen an, die der jeweiligen Situation angemessen waren. Im dritten Abschnitt ließ sich noch eine Reduktion nach dem Schema in einem Dreierschritt durchführen, bis wir den Endzustand erreichten (siehe Abb. 8.6). Den Verlauf der Zyklenauflösung gibt Abbildung 8.9 wieder.

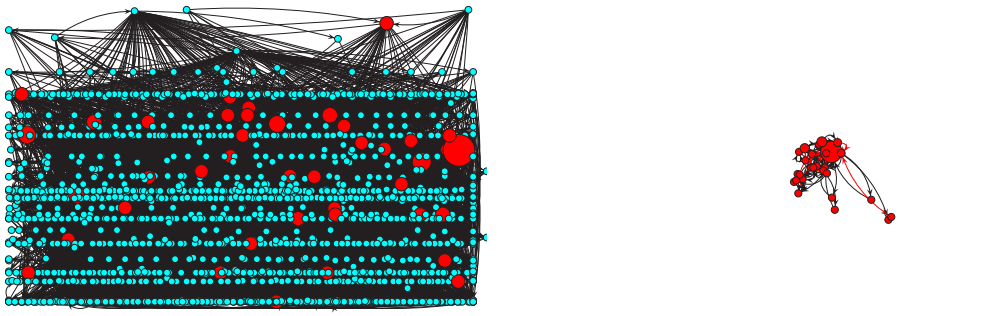
Die erwähnten Dreierschritte sind eine Kombination aller drei Auflösungstechniken hintereinander, um durch Singletonklassen verursachte Zyklen automatisiert auflösen zu können.



(a) Zyklengruppen in Schritt 0

(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.6: ArgoUML-Repräsentation im Ausgangszustand



(a) Zyklengruppen in Schritt 15

(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.7: ArgoUML-Repräsentation im Endzustand

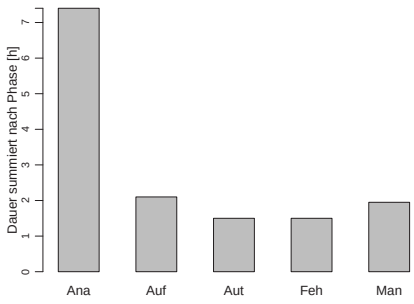


Abbildung 8.8: Gesamtzeitaufwand je Phase von ArgoUML

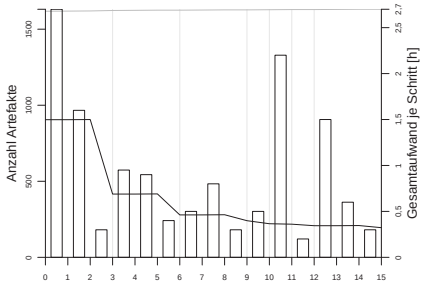
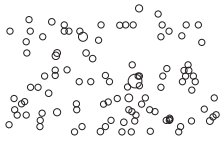


Abbildung 8.9: Zyklengruppenreduktion in ArgoUML



Schritt	Auflösungstechnik	ΔAna	ΔAuf	ΔAut	ΔFeh	ΔMan	ΔSch	#Auf	#Aut	#Feh	#Man	∑	∑(M)
Schritt 0	Abhängigkeitsumkehr	2,20	0,00	0,10	0,00	0,40	0,82	0	46	0	11	2,70	3,42
Schritt 1	Knotenspaltung	0,10	1,10	0,10	0,00	0,30	0,75	6	75	0	11	1,60	2,25
Schritt 2	Rückruf	0,20	0,00	0,10	0,00	0,00	0,36	0	11	0	0	0,30	0,56
Schritt 3	Abhängigkeitsumkehr	0,60	0,00	0,10	0,00	0,25	0,22	0	10	0	2	0,95	1,07
Schritt 4	Knotenspaltung	0,20	0,30	0,10	0,00	0,30	0,28	1	18	0	3	0,90	1,07
Schritt 5	Rückruf	0,20	0,10	0,10	0,00	0,00	0,07	1	5	1	0	0,40	0,37
Schritt 6	Abhängigkeitsumkehr	0,20	0,00	0,10	0,00	0,20	0,51	0	27	0	4	0,50	0,91
Schritt 7	Knotenspaltung	0,10	0,50	0,10	0,00	0,10	0,77	4	65	0	4	0,80	1,47
Schritt 8	Rückruf	0,10	0,00	0,10	0,10	0,00	0,07	0	5	1	0	0,30	0,27
Schritt 9	Abhängigkeitsumkehr	0,20	0,00	0,10	0,00	0,20	0,44	0	23	0	2	0,50	0,84
Schritt 10	Abhängigkeitsumkehr	1,00	0,00	0,10	1,10	0,00	3,41	0	203	63	0	2,20	5,51
Schritt 11	Rückruf	0,10	0,00	0,10	0,00	0,00	0,08	0	6	0	0	0,20	0,18
Schritt 12	Abhängigkeitsumkehr	1,30	0,00	0,10	0,00	0,10	0,53	0	29	0	2	1,50	1,93
Schritt 13	Knotenspaltung	0,10	0,10	0,10	0,20	0,10	2,33	3	264	8	3	0,60	2,83
Schritt 14	Rückruf	0,10	0,00	0,10	0,10	0,00	0,07	0	5	1	0	0,30	0,27
Schritt 15		0,70	0,00	0,00	0,00	0,00	0,00	0	0	0	0	0,70	0,70

Tabelle 8.8: Aufwandsübersicht über alle Auflösungsschritte von ArgoUML Legende siehe Tab. 8.5

Schlüssel	Beschreibung
Name	Projectfactory
Version	0.6
Anbieter	projectfactory.sourceforge.net
Lizenz	GPL v2+

Tabelle 8.9: Übersicht über Projectfactory

8.2.3 Projectfactory

Projectfactory ist ein Werkzeug zur Planung und Verwaltung von Projekten sowie der die Projekte betreuenden Personen (siehe Tabelle 8.9).

Die Applikation bestand im Ausgangszustand aus 279 Klassen, von denen sich 230 (82%) in Zyklengruppen befanden. Ferner enthielt Projectfactory 5 Zyklengruppen, von denen die größte aus 216 Klassen bestand.

Der enorme Anteil der Klassen in Zyklengruppen entstammte der Tatsache, dass Projectfactory Objekte vieler abgeleiteter Klassen in Basisklassen erzeugte. Über die Vererbung als Vorwärtskante entstand durch die Objekterzeugung jeweils eine nur über Rückrufe aufzulösende Rückwärtskante.

Wir brauchten jedoch nicht auf Rückrufe zurückzugreifen, da wir eine bereits existierende Klasse als Fabrik zur Objekterzeugung nutzen konnten. Diese Klasse stand in jeder Methode der Basisklassen, die Objekte abgeleiteter Klassen erzeugten, zur Verfügung. Es genügte daher, entsprechende Fabrikmethoden einzufügen und statt der konkreten Klasse eine Schnittstelle zurückzuliefern, die wir über die Abhängigkeitsumkehr erhielten.

Somit konnten wir in einer Folge von 17 Schritten die Anzahl der Klassen in Zyklengruppen um über 130 auf 93 reduzieren. Der prozentuelle Anteil der Klassen in Zyklengruppen verglichen mit der Gesamtanzahl fiel von 82% auf lediglich

	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Ausgangszustand	279	269	10	6	39	215	7322	2059	0,07	0,02	0,24	0,43	0,83	0,58	23,8	0,78	475056,0	5	230	82%
Schritt 1	280	269	11	6	39	216	7322	2071	0,07	0,02	0,24	0,43	0,83	0,58	23,7	0,78	475170,8	5	230	82%
Schritt 2	281	269	12	6	39	217	7322	2089	0,07	0,02	0,23	0,43	0,83	0,58	13,0	0,54	475042,5	6	230	82%
Schritt 3	282	269	13	6	39	218	7426	2129	0,07	0,02	0,23	0,44	0,82	0,58	9,9	0,47	475774,0	7	207	73%
Schritt 4	283	269	14	7	39	219	7432	2160	0,07	0,02	0,23	0,44	0,82	0,56	9,8	0,47	474910,3	7	206	73%
Schritt 5	284	269	15	7	59	221	7432	2200	0,08	0,02	0,22	0,44	0,82	0,56	7,4	0,44	469611,9	7	190	67%
Schritt 6	285	269	16	7	60	223	7435	2225	0,08	0,02	0,22	0,44	0,82	0,56	7,4	0,44	468319,6	7	190	67%
Schritt 7	286	269	17	7	60	225	7450	2236	0,08	0,02	0,22	0,44	0,82	0,56	7,3	0,44	468647,0	7	189	66%
Schritt 8	287	269	18	7	60	227	7461	2244	0,08	0,02	0,22	0,44	0,82	0,56	7,3	0,44	468990,2	7	188	66%
Schritt 9	288	269	19	7	61	229	7461	2247	0,08	0,02	0,22	0,44	0,82	0,56	7,2	0,44	469313,8	7	188	65%
Schritt 10	289	269	20	7	61	231	7472	2255	0,08	0,02	0,22	0,44	0,82	0,56	5,7	0,39	469672,4	8	169	58%
Schritt 11	290	269	21	7	61	232	7535	2358	0,08	0,02	0,21	0,45	0,81	0,56	4,7	0,21	469582,6	12	137	47%
Schritt 12	291	269	22	7	61	233	7535	2386	0,09	0,02	0,21	0,45	0,81	0,56	3,7	0,21	468347,6	15	136	47%
Schritt 13	292	269	23	7	62	235	7523	2393	0,09	0,02	0,21	0,45	0,81	0,56	3,6	0,20	471235,3	15	131	45%
Schritt 14	293	269	24	7	63	237	7534	2413	0,09	0,02	0,20	0,46	0,81	0,56	3,2	0,19	470414,2	15	124	42%
Schritt 15	294	269	25	7	64	239	7785	2466	0,09	0,02	0,20	0,47	0,81	0,56	2,4	0,17	485651,9	16	108	37%
Schritt 16	294	269	25	7	64	239	7797	2472	0,09	0,02	0,20	0,47	0,81	0,56	2,2	0,16	487924,9	17	103	35%
Schritt 17	294	269	25	7	64	239	7801	2480	0,09	0,02	0,20	0,47	0,81	0,56	2,4	0,15	493096,1	19	93	32%

Tabelle 8.10: Metrikübersicht über alle Auflösungsschritte von Projectfactory

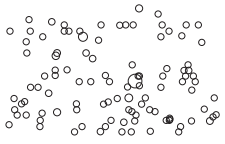
Schritt	Auflösungstechnik	ΔAna	ΔAuf	ΔAut	ΔFeh	ΔMan	ΔSch	#Auf	#Aut	#Feh	#Man	∑	∑(M)
Schritt 0	Abhängigkeitsumkehr	0,50	0,00	0,10	0,00	0,25	0,31	0	15	0	2	0,85	1,06
Schritt 1	Abhängigkeitsumkehr	0,10	0,00	0,10	0,00	0,00	0,44	0	23	4	0	0,20	0,54
Schritt 2	Abhängigkeitsumkehr	0,30	0,80	0,10	0,10	0,00	0,72	110	40	5	0	1,30	1,92
Schritt 3	Abhängigkeitsumkehr	0,90	0,25	0,10	0,20	0,25	0,59	10	32	7	17	1,70	2,19
Schritt 4	Abhängigkeitsumkehr	0,10	0,00	0,10	0,40	0,10	0,81	0	45	5	3	0,70	1,41
Schritt 5	Abhängigkeitsumkehr	1,25	0,00	0,10	0,80	0,00	0,49	0	26	44	0	2,15	2,54
Schritt 6	Abhängigkeitsumkehr	0,10	0,50	0,10	0,00	0,50	0,16	9	6	0	21	1,20	1,26
Schritt 7	Abhängigkeitsumkehr	0,10	0,20	0,10	0,00	0,25	0,16	6	6	0	16	0,65	0,71
Schritt 8	Abhängigkeitsumkehr	0,10	0,00	0,10	0,00	0,20	0,12	0	4	0	3	0,40	0,42
Schritt 9	Abhängigkeitsumkehr	0,10	0,10	0,10	0,00	0,10	0,16	6	6	0	16	0,40	0,46
Schritt 10	Abhängigkeitsumkehr	0,25	0,10	0,10	0,25	0,00	1,53	14	89	19	0	0,70	2,13
Schritt 11	Abhängigkeitsumkehr	0,85	0,00	0,10	0,10	0,20	0,62	0	34	4	1	1,25	1,77
Schritt 12	Abhängigkeitsumkehr	0,20	0,20	0,10	0,00	0,60	0,12	2	4	0	21	1,10	1,12
Schritt 13	Abhängigkeitsumkehr	0,25	0,00	0,10	0,00	0,60	0,21	0	9	0	34	0,95	1,06
Schritt 14	Abhängigkeitsumkehr	0,20	0,20	0,10	0,30	0,90	0,34	10	17	16	79	1,70	1,94
Schritt 15	Abhängigkeitsumkehr	0,20	0,10	0,10	0,00	0,30	0,00	2	0	0	27	0,70	0,60
Schritt 16	Abhängigkeitsumkehr	0,10	0,20	0,10	0,00	0,60	0,00	2	0	0	56	1,00	0,90
Schritt 17		0,10	0,00	0,00	0,00	0,00	0,00	0	0	0	0	0,10	0,10

Tabelle 8.11: Aufwandsübersicht über alle Auflösungsschritte von Projectfactory
Legende siehe Tab. 8.5

32% – eine Reduktion um 50 Prozentpunkte. Enthielt im Ausgangszustand die größte Zyklengruppe noch 216 Klassen, so bestand die am Ende übriggebliebene größte Zyklengruppe nur noch aus 12 Klassen.

Tabelle 8.10 zeigt die Veränderung sämtlicher Metriken über alle Schritte dieses Zyklenauflösungsprozesses.

Im Ausgangszustand präsentierte sich das System wie aus Abbildung 8.10 ersichtlich. Die Zyklenauflösung erfolgte in sechs Abschnitten. Der erste Abschnitt brach die Verzyklung zwischen den Schichten der Applikation auf. Der zweite und dritte Abschnitt reduzierte die größte Zyklengruppe der Serverschicht. Im vierten Abschnitt erfolgte die Reduktion der Zyklengruppe der Klientenschicht. Die rest-



Schlüssel	Beschreibung
Name	imp-a
Version	–
Anbieter	proprietär
Lizenz	proprietär

Tabelle 8.12: Übersicht über imp-a

	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Ausgangszustand	2238	1896	342	6	46	946	26516	18575	0,27	0,00	0,36	0,83	0,29	0,21	10,0	0,09	6383205,8	84	436	19%
Schritt 1	2239	1897	342	6	46	946	26802	18576	0,27	0,00	0,36	0,83	0,29	0,21	10,0	0,09	6394995,0	84	437	20%
Schritt 2	2240	1897	343	6	46	947	26863	18588	0,27	0,00	0,36	0,83	0,29	0,21	10,0	0,09	6424720,4	84	437	20%
Schritt 3	2241	1898	343	6	46	947	26867	18589	0,27	0,00	0,36	0,83	0,29	0,21	7,5	0,08	6373196,5	87	408	18%
Schritt 4	2243	1899	344	6	46	950	26868	18601	0,27	0,00	0,36	0,83	0,29	0,21	7,5	0,08	6374402,9	87	409	18%
Schritt 5	2245	1900	345	6	46	950	26869	18605	0,27	0,00	0,36	0,83	0,29	0,21	6,8	0,08	6375313,7	88	399	18%
Schritt 6	2245	1900	345	6	46	950	26871	18608	0,27	0,00	0,36	0,83	0,29	0,21	9,1	0,07	6376178,9	92	370	16%
Schritt 7	2246	1900	346	6	46	951	26871	18612	0,27	0,00	0,36	0,83	0,29	0,21	9,1	0,07	6376312,6	92	370	16%
Schritt 8	2246	1900	346	6	46	951	26873	18615	0,27	0,00	0,36	0,83	0,29	0,21	8,5	0,07	6377061,2	96	358	16%
Schritt 9	2247	1900	347	6	46	952	26873	18626	0,27	0,00	0,35	0,83	0,29	0,21	6,7	0,07	6377181,7	100	350	16%
Schritt 10	2248	1900	348	6	46	953	26873	18656	0,27	0,00	0,35	0,83	0,29	0,21	7,1	0,07	6376604,3	104	343	15%
Schritt 11	2249	1901	348	6	46	953	26873	18657	0,27	0,00	0,35	0,83	0,29	0,21	7,1	0,06	6371622,5	104	343	15%
Schritt 12	2250	1901	349	6	46	954	26873	18661	0,27	0,00	0,35	0,83	0,29	0,21	7,1	0,06	6372623,1	104	343	15%
Schritt 13	2250	1901	349	6	46	954	26875	18664	0,27	0,00	0,35	0,83	0,29	0,21	5,7	0,06	6373659,9	105	335	15%

Tabelle 8.13: Metrikübersicht über alle Auflösungsschritte von imp-a

lichen Abschnitte widmeten sich wieder der Reduktion der größten Zyklengruppe der Serverschicht, bis ein Endzustand erreicht war (siehe Abb. 8.11). Den Verlauf der Zyklenauflösung gibt Abbildung 8.13 wieder.

8.2.4 imp-a

imp-a ist ein Werkzeug zur Gestaltung von Benutzeroberflächen für eingebettete Geräte (siehe Tabelle 8.12).

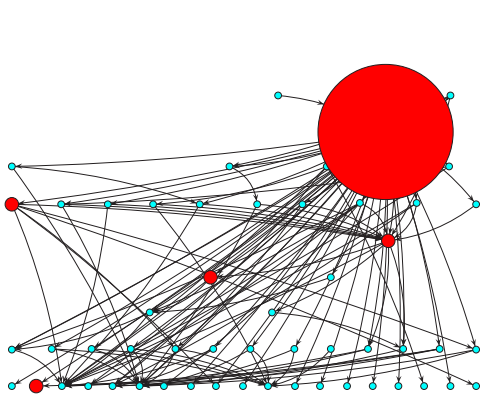
Die Applikation bestand im Ausgangszustand aus 2238 Klassen, von denen sich 436 in Zyklengruppen befanden. Ferner enthielt imp-a 84 Zyklengruppen, von denen die größte aus 117 Klassen bestand.

In einer Folge von 13 Schritten konnten wir die Anzahl der Klassen in Zyklengruppen um über 100 auf 335 reduzieren. Der prozentuelle Anteil der Klassen in Zyklengruppen verglichen mit der Gesamtanzahl fiel von 19% auf 15%. Enthielt im Anfangszustand die größte Zyklengruppe noch 117 Klassen, so bestand die am Ende übriggebliebene größte Zyklengruppe nur noch aus 15 Klassen.

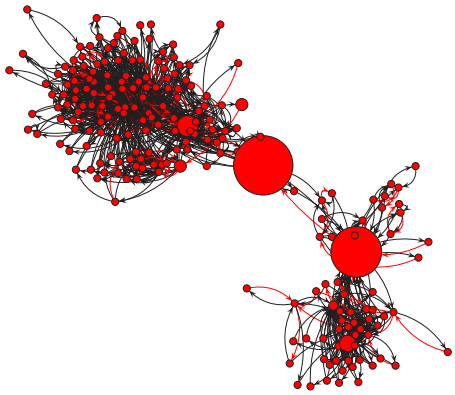
Tabelle 8.13 zeigt die Veränderung sämtlicher Metriken über alle Schritte dieses Zyklenauflösungsprozesses.

Im Ausgangszustand präsentierte sich das System wie in Abbildung 8.14 ersichtlich. Die Zyklenauflösung erfolgte in vier Abschnitten. Ein Abschnitt umfasste jeweils die Verkleinerung einer Zyklengruppe mittels Auflösungstechniken, bis dass diese Zyklengruppe nicht mehr die größte aller Zyklengruppen darstellte.

12 Fr/imp-a

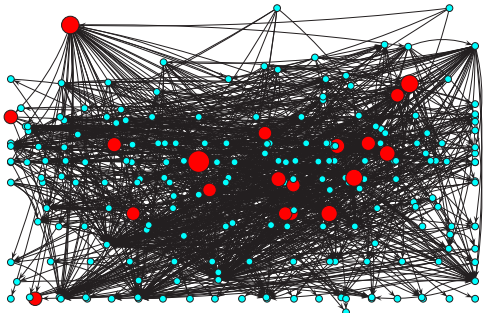


(a) Zyklengruppen in Schritt 0



(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.10: Projectfactory-Repräsentation im Ausgangszustand



(a) Zyklengruppen in Schritt 17

(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.11: Projectfactory-Repräsentation im Endzustand

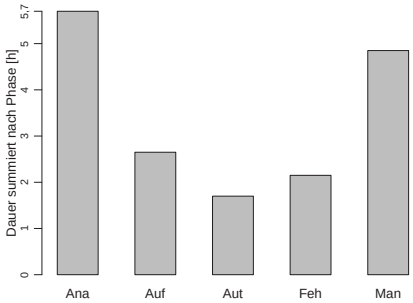


Abbildung 8.12: Gesamtzeitaufwand je Phase von Projectfactory

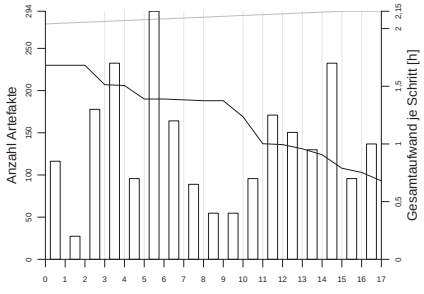
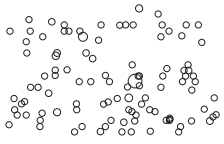


Abbildung 8.13: Zyklengruppenreduktion in Projectfactory

12 ZG/imp-a



Schritt	Auflösungstechnik	ΔAna	ΔAuf	ΔAut	ΔFeh	ΔMan	ΔSch	#Auf	#Aut	#Feh	#Man	∑	∑(M)
Schritt 0	Knotenspaltung	3,25	0,00	0,68	0,10	0,00	2,42	0	237	5	0	4,03	5,78
Schritt 1	Abhängigkeitsumkehr	0,50	0,00	0,10	0,10	1,00	0,28	0	13	1	31	1,70	1,88
Schritt 2	Knotenspaltung	0,25	0,00	0,20	0,75	0,95	1,66	2	184	15	119	2,15	3,61
Schritt 3	Abhängigkeitsumkehr	1,00	0,00	0,10	0,00	0,20	0,29	0	14	0	5	1,30	1,49
Schritt 4	Rückruf	0,20	0,20	0,10	0,20	0,00	0,33	4	9	1	0	0,70	0,93
Schritt 5	Rückruf	0,25	0,00	0,10	0,00	0,00	0,08	0	6	0	0	0,35	0,33
Schritt 6	Abhängigkeitsumkehr	1,60	0,00	0,10	0,00	0,00	0,17	0	7	0	0	1,70	1,77
Schritt 7	Rückruf	0,10	0,00	0,10	0,00	0,00	0,08	0	6	0	0	0,20	0,18
Schritt 8	Abhängigkeitsumkehr	0,30	0,00	0,10	0,20	0,00	0,36	0	18	4	0	0,60	0,86
Schritt 9	Abhängigkeitsumkehr	0,30	0,00	0,10	0,25	0,00	0,66	0	36	1	0	0,65	1,21
Schritt 10	Knotenspaltung	1,00	0,00	0,10	0,20	0,00	0,37	0	23	2	0	1,30	1,57
Schritt 11	Abhängigkeitsumkehr	0,50	0,20	0,10	0,00	0,00	0,26	1	12	0	0	0,80	0,96
Schritt 12	Rückruf	0,10	0,00	0,10	0,10	0,00	0,07	0	5	1	0	0,30	0,27
Schritt 13		0,20	0,00	0,00	0,00	0,00	0,00	0	0	0	0	0,20	0,20

Tabelle 8.14: Aufwandsübersicht über alle Auflösungsschritte von imp-a
Legende siehe Tab. 8.5

Schlüssel	Beschreibung
Name	imp-b
Version	–
Anbieter	proprietär
Lizenz	proprietär

Tabelle 8.15: Übersicht über imp-b

Danach setzten wir mit der jeweils nunmehr größten Zyklengruppe fort, bis ein Endzustand erreicht war (siehe Abb. 8.15) Den Verlauf der Zyklenauflösung gibt Abbildung 8.17 wieder.

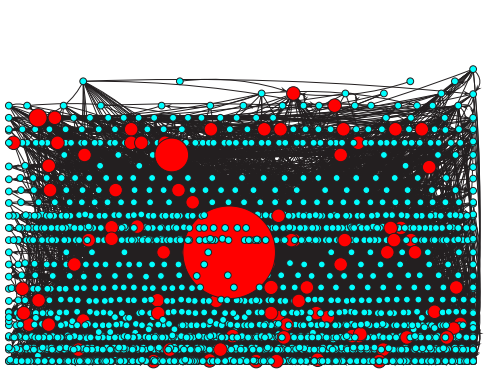
imp-a war von Anfang an nach den Grundsätzen sauberer objektorientierter Entwicklung entworfen worden. Dies machte sich auch in der geringen Anzahl der Klassen in Zyklengruppen bemerkbar, obwohl die Architekten der Software kein Augenmerk auf zyklische Abhängigkeiten legten. Dementsprechend wenig konnten wir die Anzahl der Artefakte in Zyklengruppen reduzieren.

8.2.5 imp-b

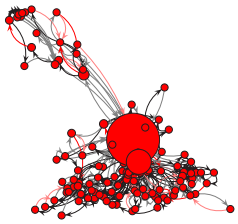
imp-b ist ein Werkzeug zur Konfigurationsverwaltung für eingebettete Geräte (siehe Tabelle 8.15).

Die Applikation bestand im Ausgangszustand aus 720 Klassen, von denen sich 264 in Zyklengruppen befanden (37%). Ferner enthielt imp-b 22 Zyklengruppen, von denen die größte aus 128 Klassen bestand.

In einer Folge von 6 Schritten konnten wir die Anzahl der Klassen in Zyklengruppen um über 40 auf 228 reduzieren. Der prozentuelle Anteil der Klassen in Zyklengruppen verglichen mit der Gesamtanzahl fiel von 37% auf 31%. Enthielt im Anfangszustand die größte Zyklengruppe noch 128 Klassen, so bestand die am Ende übriggebliebene größte Zyklengruppe nur noch aus 15 Klassen.

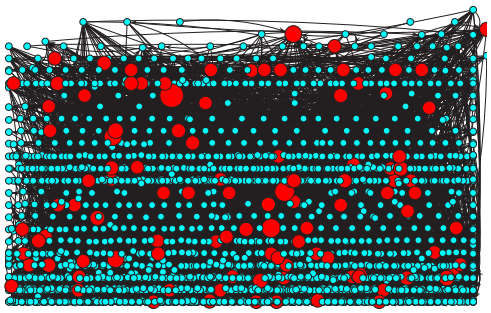


(a) Zyklengruppen in Schritt 0

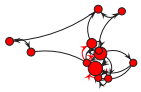


(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.14: imp-a-Repräsentation im Ausgangszustand



(a) Zyklengruppen in Schritt 13



(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.15: imp-a-Repräsentation im Endzustand

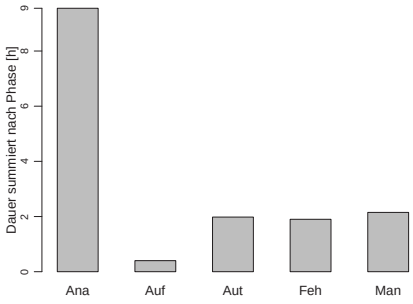


Abbildung 8.16: Gesamtzeitaufwand je Phase von imp-a

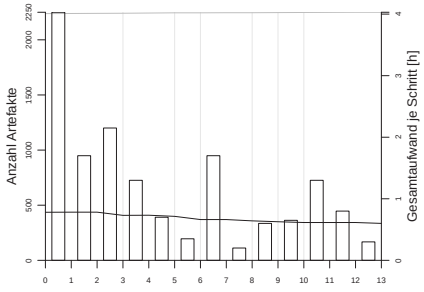


Abbildung 8.17: Zyklengruppenreduktion in imp-a

	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Ausgangszustand	720	643	77	4	100	389	9345	6458	0,47	0,01	0,25	0,61	0,24	0,09	3,7	0,14	833156,1	22	264	37%
Schritt 1	721	643	78	4	100	390	9348	6482	0,47	0,01	0,25	0,61	0,24	0,09	3,7	0,14	828787,0	22	264	37%
Schritt 2	722	644	78	4	100	390	9356	6483	0,47	0,01	0,25	0,61	0,24	0,09	2,2	0,07	844877,5	40	240	33%
Schritt 3	723	644	79	4	100	391	9356	6507	0,47	0,01	0,25	0,61	0,24	0,09	2,0	0,06	844565,3	44	239	33%
Schritt 4	724	644	80	4	100	393	9356	6531	0,47	0,01	0,25	0,61	0,24	0,09	1,5	0,06	844837,3	48	239	33%
Schritt 5	725	645	80	4	100	393	9354	6533	0,47	0,01	0,25	0,61	0,24	0,09	1,4	0,05	839802,2	48	236	33%
Schritt 6	726	645	81	4	100	394	9367	6555	0,47	0,01	0,24	0,61	0,24	0,09	1,7	0,05	842323,8	49	228	31%

Tabelle 8.16: Metrikübersicht über alle Auflösungsschritte von imp-b

Schritt	Auflösungstechnik	ΔAna	ΔAuf	ΔAut	ΔFeh	ΔMan	ΔSch	#Auf	#Aut	#Feh	#Man	∑	∑(M)
Schritt 0	Abhängigkeitsumkehr	1,15	0,00	0,20	0,30	0,00	0,57	0	31	12	0	1,65	2,02
Schritt 1	Knotenspaltung	0,50	0,95	0,10	0,15	0,25	1,17	68	98	1	17	1,95	3,02
Schritt 2	Abhängigkeitsumkehr	0,70	0,00	0,10	0,50	0,00	0,57	0	31	14	0	1,30	1,77
Schritt 3	Abhängigkeitsumkehr	0,25	0,00	0,10	0,00	0,50	0,48	0	26	0	12	0,85	1,23
Schritt 4	Knotenspaltung	1,10	0,30	0,10	0,00	0,00	0,83	4	82	0	0	1,50	2,23
Schritt 5	Abhängigkeitsumkehr	0,20	0,00	0,10	1,60	0,10	0,80	0	45	85	4	2,00	2,70
Schritt 6		0,00	0,00	0,00	0,00	0,00	0,00	0	0	0	0	0,00	0,00

Tabelle 8.17: Aufwandsübersicht über alle Auflösungsschritte von imp-b
Legende siehe Tab. 8.5

Tabelle 8.16 zeigt die Veränderung sämtlicher Metriken über alle Schritte dieses Zyklenauflösungsprozesses.

Im Ausgangszustand präsentierte sich das System wie in Abbildung 8.18 ersichtlich. Die Zyklenauflösung erfolgte in zwei Abschnitten. Der erste Abschnitt umfasste jeweils die Verkleinerung einer Zyklengruppe mittels Auflösungstechniken, bis dass diese Zyklengruppe unter den Schwellwert von 15 Klassen gesunken war. Im zweiten Abschnitt setzten wir mit der jeweils nunmehr größten Zyklengruppe fort, bis wir einen Endzustand erreichten (siehe Abb. 8.19) Den Verlauf der Zyklenauflösung gibt Abbildung 8.21 wieder.

8.3 Auswertung

Zuletzt erfolgt die Auswertung der Daten, die wir durch die Anwendung der Auflösungstechniken auf fünf reale Softwaresysteme erlangten, sowie die Interpretation der Akzeptanztests und der durch die Anwendung gewonnenen Erfahrungen.

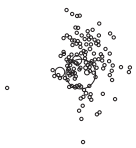
8.3.1 Statistiken

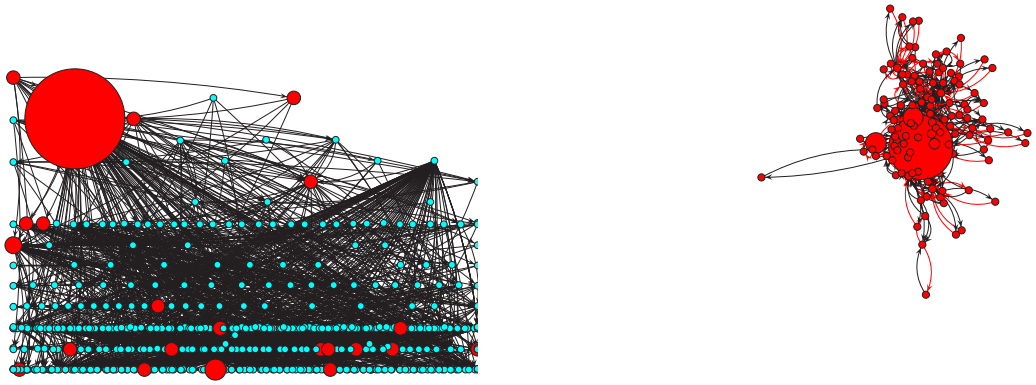
Wir reduzierten in fünf realen Softwaresystemen die Anzahl der Klassen in Zyklengruppen unter Verwendung der automatischen Auflösungstechniken *Abhängigkeitsumkehr*, *gerichtete Knotenspaltung* und *Rückrufe*.

Die Anwendung der Auflösungstechniken fand in Schritten statt, die zusätzlich die gedankliche Leistung der Analyse sowie weitere händische Phasen enthielten.

In allen Softwaresystemen konnte die Größe der auflösbaren Zyklengruppen auf nicht mehr als 15 Elemente reduziert werden.

0 Fr/imp-b

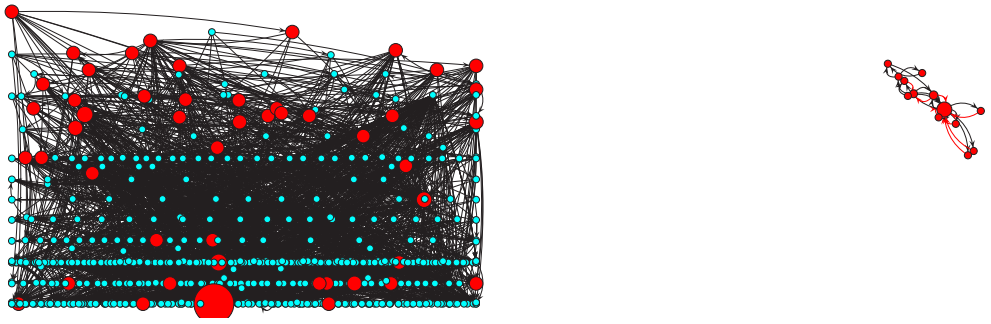




(a) Zyklengruppen in Schritt 0

(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.18: imp-b-Repräsentation im Ausgangszustand



(a) Zyklengruppen in Schritt 6

(b) Fragmentierer der größten Zyklengruppe

Abbildung 8.19: imp-b-Repräsentation im Endzustand

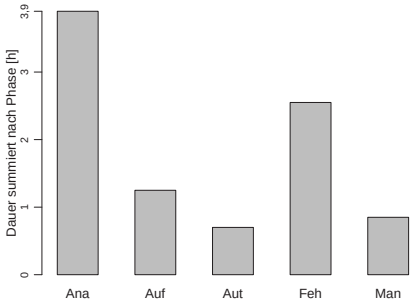


Abbildung 8.20: Gesamtzeitaufwand je Phase von imp-b

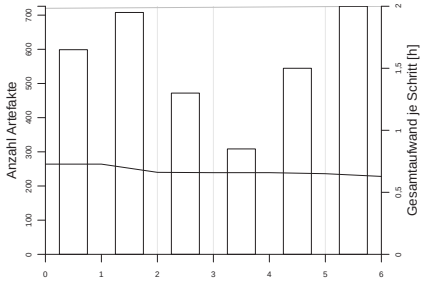
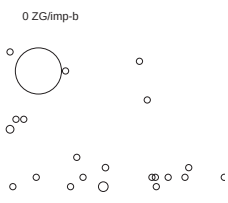
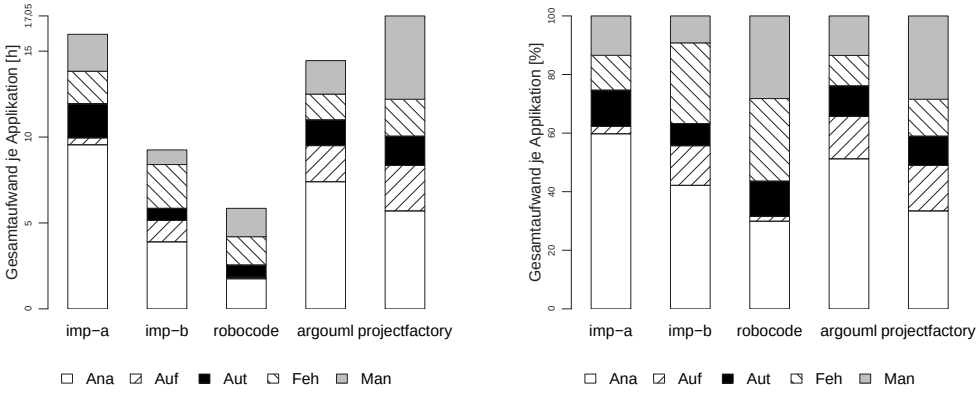


Abbildung 8.21: Zyklengruppenreduktion in imp-b





(a) Absolute Häufigkeiten

(b) Relative Häufigkeiten

Abbildung 8.22: Dauer der Zyklenreduktionen nach Phasen je Applikation

Zeitauswertung Die Gesamtzeit über alle Schritte betrug 62,6 Stunden. Nicht in dieser Zahl enthalten sind Protokollierung, Rüstzeiten und Fehlerbehebungen am Werkzeug selbst.

Projectfactory benötigte mit 17,1 Stunden die längste Zeit, RoboCode mit 5,8 Stunden die kürzeste. Im Mittel belief sich die Dauer auf 12,5 Stunden ($\pm$4,8). Die Zeitdauer der verschiedenen Phasen summiert über die verschiedenen Applikationen lässt sich aus Abbildung 8.22 entnehmen.

Die Phase der automatischen Auflösung nahm wie erwartet nur einen Bruchteil der Gesamtzeit ein. Die Phasen Analyse und Aufbereitung beanspruchten außer bei RoboCode stets mehr als die Hälfte der Zeit. Bei RoboCode wurde die geringe Analysedauer durch entsprechend höhere Aufwände bei Fehlerbehebung und manuellem Umbau »bezahlt«, welche sich absolut gesehen kaum als zeitaufwendiger erwiesen als bei imp-a.

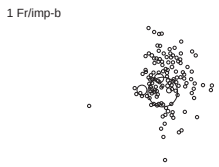
Mit sechs Schritten benötigte imp-b die geringste Anzahl von Schritten, Projectfactory mit 17 die größte.

Die Anzahl der Schritte korreliert jedoch nur schwach mit der aufgewandten Zeit. So bedurfte zwar RoboCode mehr Schritte zur Auflösung als imp-b, schlägt aber imp-b dennoch bezüglich des Gesamtzeitaufwandes um ein Drittel.

Im Durchschnitt entfielen für eine Applikation auf die Zyklenauflösung 12 Schritte ($\pm$4,9). Davon genügten durchschnittlich 5 Schritte ($\pm$3,6), um von den insgesamt aus Zyklengruppen herausgelösten Klassen (ΔAZG) die Hälfte zu bewältigen.

Nach ungefähr 44,3% der Schritte ($\pm$18,0) sind bereits die Hälfte der insgesamt herausgelösten Klassen aus Zyklengruppen herausgelöst.

Tabelle 8.18 gibt die benötigten Schritte sowie den prozentuellen Fortschritt für die Quartile 25% und 75% sowie für den Median (50%) und die tatsächliche



Applikation	**∆AZG**	**∆AZG reduziert um**			
		25%	**50%**	**75%**	**100%**
imp-a	101	3 (23%)	6 (46%)	8 (62%)	13 (100%)
imp-b	36	2 (33%)	2 (33%)	5 (83%)	6 (100%)
RoboCode	45	2 (29%)	4 (57%)	5 (71%)	7 (100%)
ArgoUML	710	3 (20%)	3 (20%)	6 (40%)	15 (100%)
Projectfactory	137	5 (29%)	11 (65%)	14 (82%)	17 (100%)
Gesamt	205,8 (±284,9)	3,0 (±1,2)	5,2 (±3,6)	7,6 (±3,8)	11,6 (±4,9)
(in %)	–	26,9% (±5,3)	44,3% (±18,0)	67,7% (±17,9)	100,0% (±0,0)

Tabelle 8.18: AZG-Differenz und Schritte zum jeweils aufgelösten Anteil

durchgeführte Anzahl Schritte (100%) für jedes Softwaresystem an. Die Gesamtwerte spiegeln die Durchschnitte über die Spalten wider.

Die Reduktion der Artefakte in Zyklengruppen schreitet also weitgehend linear mit der Anzahl der Schritte fort. Weder lässt sich feststellen, dass ein großer Teil der Gesamtreduktion an Artefakten in Zyklengruppen bereits gegen Anfang noch erst gegen Ende erfolgt.

Betrachten wir die Aufschlüsselung des Zeitaufwandes nach Phasen je Applikation und Schritt (siehe Abb. 8.23), können wir folgende Schlüsse ziehen.

- Aus der Verteilung der Zeitaufwände lässt sich ad hoc kein Muster erkennen.

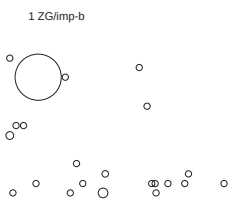
- Bei Anlegen einer Regressionsgerade (die grauen Linien in Abb. 8.23) erkennen wir einen tendenziell sinkenden Zeitaufwand bei fortschreitender Schrittanzahl.

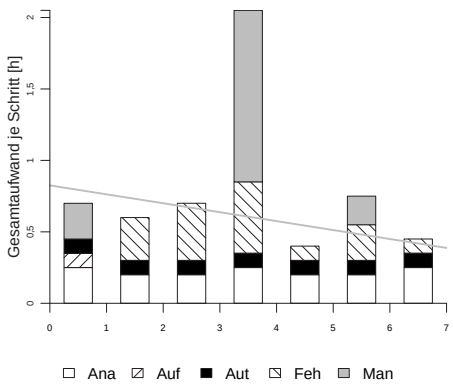
  Die Regressionsgeraden sind in manchen Fällen jedoch so schwach ausgeprägt, dass wir nicht grundsätzlich auf abnehmenden Aufwand bei fortschreitender Zyklenauflösung folgern können.

- Mit Ausnahme von RoboCode ist der Analyseaufwand im Ausgangszustand deutlich höher als die Analyseaufwände unmittelbar nachfolgender Schritte, bei zwei Applikationen sogar am höchsten.

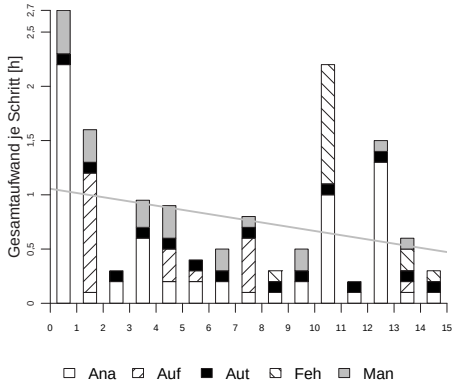
  Diese Steigerung lässt sich durch zwei Faktoren erklären: Erstens muss sich der Aktor am Anfang in die Struktur des Softwaresystems einfinden und zweitens eine adäquate Auflösungsstrategie in einem noch unbearbeiteten und damit maximal komplexen Softwaresystem suchen.

- Langen Analysephasen folgen oftmals Schritte mit sehr kurzen Analysephasen, zum Beispiel in ArgoUML, Schritte 0 bis 3 oder 10 bis 12 (Abb. 8.23(b)), Projectfactory, Schritte 5 bis 10 (Abb. 8.23(c)) oder imp-a, Schritte 6 bis 8 (Abb. 8.23(d)).

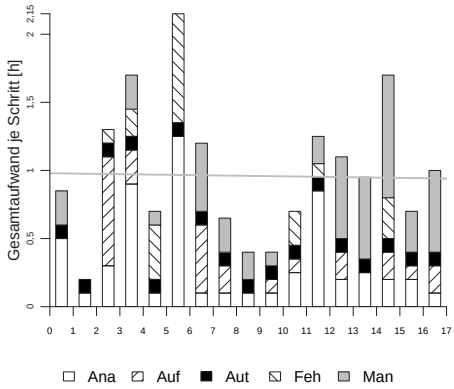




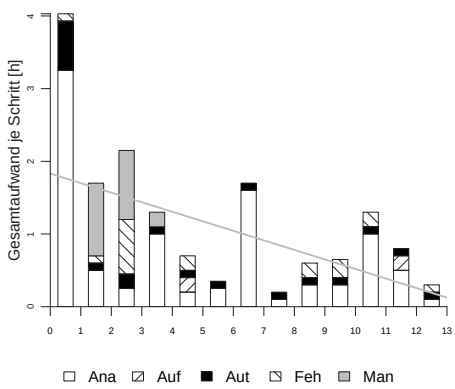
(a) RoboCode



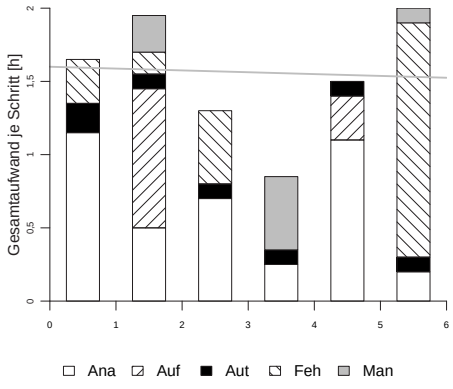
(b) ArgoUML



(c) Projectfactory



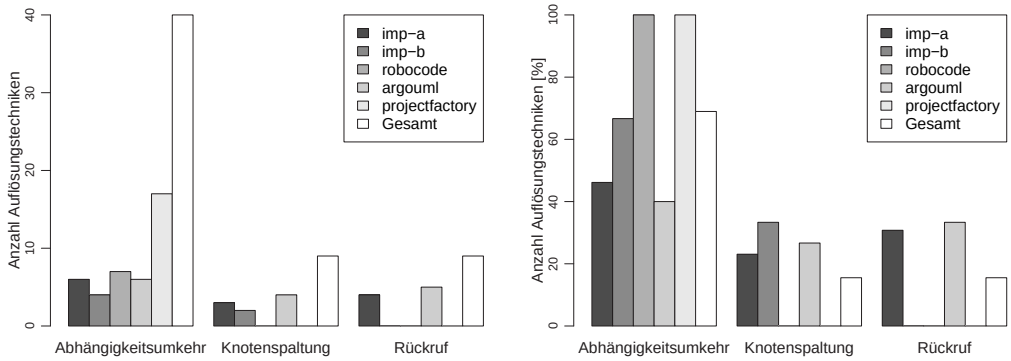
(d) imp-a



(e) imp-b

Abbildung 8.23: Arbeitsaufwand je Schritt in Phasen aufgeschlüsselt

2 Fr/imp-b



(a) Absolute Häufigkeiten (b) Relative Häufigkeiten

Abbildung 8.24: Häufigkeiten verwendeter Auflösungstechniken je Applikation und insgesamt

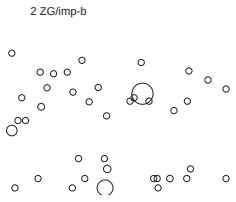
> In diesen Fällen wurde eine mehrere Schritte umfassende Strategie zur Auflösung im jeweils ersten Schritt entwickelt und über mehrere Schritte zur Ausführung gebracht.

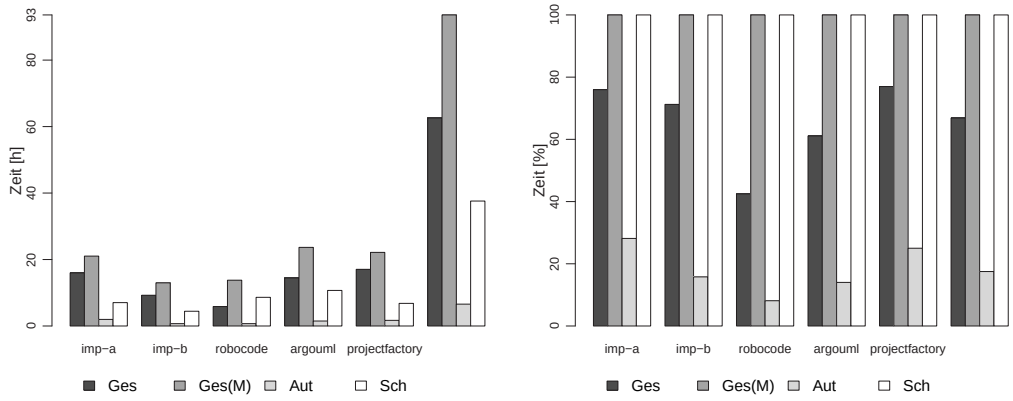
Aufteilung der Auflösungstechniken Die Abhängigkeitsumkehr wurde mit 40 Einsätzen am häufigsten eingesetzt, abgeschlagen gefolgt von Knotenspaltung mit 9 und Rückrufen mit 9 Einsätzen (siehe Abb. 8.24). Auch in jeder einzelnen Applikation kam die Abhängigkeitsumkehr am öftesten zum Zug. Für zwei Applikationen – RoboCode und Projectfactory – ist sie sogar die einzige angewandte Auflösungstechnik.

Spekulativer Zeitgewinn Wir untersuchten weiters, welche Zeitersparnis die Anwendung automatischer Auflösungstechniken im Vergleich zu manuellen Umbauoperationen am Quelltext bringen könnte. Dabei unterstellten wir eine manuelle Ausführung der Phase »Automatischer Umbau« bei ansonsten gleichbleibenden anderen Phasen.

Hier zeigte sich, dass der Gesamtaufwand bei automatischer Auflösung (Ges) ungefähr 70% des zu erwartenden Gesamtaufwands bei entsprechender manueller Auflösung (Ges(M)) betrug. Bei ausschließlichem Vergleich der Phase »Automatischer Umbau« (Aut) mit der hypothetischen Phase »Schätzung bei manuellem Umbau« (Sch) lag der Zeitgewinn sogar über dem Fünffachen (siehe Abb. 8.25).

Weiters zeigt Abbildung 8.25(b) für alle Applikationen mit Ausnahme von RoboCode einen Zeitaufwand für Ges zwischen 60% und 80% im Vergleich zu Ges(M), für Aut zu Sch zwischen 10% und 30%.





(a) Absolute Häufigkeiten

(b) Relative Häufigkeiten

Abbildung 8.25: Vergleich der Dauer automatischer Auflösung zu hypothetischer manueller Auflösung

Metrikentwicklung Tabelle 8.19 gibt einen Vergleich der Metriken im Ausgangs- und Endzustand für jedes Softwaresystem wieder. Schwarze Werte stellen eine Zunahme des Metrikwerts dar, graue Werte eine Stagnation und fettgedruckte Werte eine Abnahme. Die Metrikänderungstabellen für jeden einzelnen Schritt befinden sich in Kapitel 8.2 bei der jeweiligen Beschreibung des Softwaresystems.

Die Metriken AZG und AZG% dienten als Referenzmetriken zur Zyklenauflösung. Ihr Absinken erfolgte aus reiner Absicht. Die Entwicklungsrichtung der anderen Metriken war nicht vorgegeben. Bereits aus Tabelle 8.19 erkennen wir eine überwiegende Zunahme der Systemebenenmetriken und ZG sowie eine durchgehende Abnahme der Metriken NKKA und VK. Die Entwicklungsrichtungen der restlichen Metrikwerte sind weniger homogen, weshalb wir deren Entwicklung über Durchschnittswerte und Quartile betrachten (siehe Tab. 8.20).

Die Durchschnittswerte bestätigen die Beobachtung der Zunahme bei den Systemebenenmetriken. Diese Zunahme ist plausibel, da im Zuge der Zyklenauflösung ausschließlich Artefakte hinzugefügt, aber niemals entfernt wurden.

Bei den MOOD-Metriken ergibt sich ein anderes Bild. Vier davon sanken im Mittel, zwei stiegen, jedoch betrug die größte Änderung nur in einem Fall (PF) +0,22. Die restlichen Metrikänderungen bewegen sich im Promillebereich. Der Median unterstreicht die Insignifikanz der MOOD-Metriken mit höchstens 0,01 bei PF, ansonsten aber mit Werten unterhalb der eingestellten Wahrnehmungsschwelle.

Die konsequente Abnahme von NKKA erklärt sich aus der Tatsache, dass diese Metrik mit dem Quadrat der Zyklengruppengröße gewichtet ist. Schrumpft die Größe der Zyklengruppen, so nimmt NKKA deutlich ab. Dies bedeutet eine Verringerung des Testaufwands.

3 Fr/imp-b

	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Schritt 0	232	219	13	6	14	59	3596	2816	0,12	**0,01**	**0,25**	0,88	**0,24**	0,07	**10,5**	**0,54**	111799,3	9	**142**	**61%**
Schritt 7	239	219	20	6	14	67	3687	3162	0,34	**0,01**	**0,22**	0,88	**0,22**	0,07	**1,4**	**0,12**	114852,0	30	**97**	**41%**

(a) RoboCode

	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Schritt 0	1618	1380	238	9	176	1109	23673	11603	0,11	0,00	**0,27**	0,81	**0,70**	0,39	**65,1**	**0,53**	14705827,4	12	**905**	**56%**
Schritt 15	1631	1386	245	9	176	1116	23696	11807	0,12	0,00	**0,26**	0,81	**0,70**	0,39	**6,5**	**0,07**	15075305,4	41	**195**	**12%**

(b) ArgoUML

	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Schritt 0	279	269	10	6	39	215	7322	2059	0,07	**0,02**	**0,24**	0,43	**0,83**	**0,58**	**23,8**	**0,78**	475056,0	5	**230**	**82%**
Schritt 17	294	269	25	7	64	239	7801	2480	0,09	**0,02**	**0,20**	0,47	**0,81**	**0,56**	**2,4**	**0,15**	493096,1	19	**93**	**32%**

(c) Projectfactory

	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Schritt 0	2238	1896	342	6	46	946	26516	18575	0,27	0,00	**0,36**	0,83	**0,29**	0,21	**10,0**	**0,09**	**6383205,8**	84	**436**	**19%**
Schritt 13	2250	1901	349	6	46	954	26875	18664	0,27	0,00	**0,35**	0,83	**0,29**	0,21	**5,7**	**0,06**	**6373659,9**	105	**335**	**15%**

(d) imp-a

	AK	AKK	AAK	MVT	MVB	VB	AEM	AM	PF	CF	MHF	AHF	MIF	AIF	NKKA	VK	BL	ZG	AZG	AZG%
Schritt 0	720	643	77	4	100	389	9345	6458	0,47	0,01	**0,25**	**0,61**	**0,24**	0,09	**3,7**	**0,14**	833156,1	22	**264**	**37%**
Schritt 6	726	645	81	4	100	394	9367	6555	0,47	0,01	**0,24**	**0,61**	**0,24**	0,09	**1,7**	**0,05**	842323,8	49	**228**	**31%**

(e) imp-b

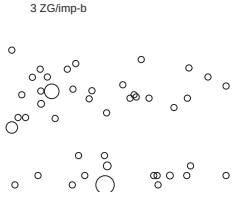
Tabelle 8.19: Metrikänderungen aller Applikationen zwischen erstem und letztem Schritt

Ebenso lässt sich die Abnahme bei VK erklären. VK misst die systemweite Kopplung, und da durch die Zyklenauflösung die Kopplung des Softwaresystems reduziert wird, ist eine Abnahme von VK zu erwarten.

Die Zunahme der Anzahl der Zyklengruppen (ZG) ist keine zwingende Folge aus der Klassenreduktion, doch sie entsteht durch die Spaltung großer Zyklengruppen in mehrere kleine. Da in der Regel Zyklengruppen nicht vollständig eliminiert, sprich, alle Klassen aus einer Zyklengruppe herausgelöst wurden, war eine Zunahme von ZG zu erwarten.

Die Beschreibungslänge (BL) nahm erwartungsgemäß im Durchschnitt zu, da durch Hinzufügung von mehr Artefakten und Beziehungen auch mehr Bits benötigt wurden, um diese zu beschreiben.

Dass BL jedoch nicht lediglich eine bessere AK-Metrik ist, beweist die Zyklenauflösung von imp-a, innerhalb derer die Beschreibungslänge um nahezu 10000 Bits abnahm. Hier sorgte die Zyklenauflösung tatsächlich für ein engeres Zusam-



Eigenschaft	μ (σ)	Min	$Q_{25\%}$	Med	$Q_{75\%}$	Max
AK	10,6 (±4)	6	7	12	13	15
AKK	2,6 (±3)	0	0	2	5	6
AAK	8,0 (±4)	4	7	7	7	15
MVT	0,2 (±0)	0	0	0	0	1
MVB	5,0 (±11)	0	0	0	0	25
VB	10,4 (±8)	5	7	8	8	24
AEM	194,8 (±211)	22	23	91	359	479
AM	231,4 (±148)	89	97	204	346	421
PF	0,0514 (±0,10)	0,00	0,00	0,01	0,02	0,22
CF	-0,0004 (±0,00)	-0,00	-0,00	0,00	0,00	0,00
MHF	-0,0156 (±0,02)	-0,04	-0,03	-0,00	-0,00	-0,00
AHF	0,0065 (±0,01)	-0,00	0,00	0,00	0,00	0,03
MIF	-0,0091 (±0,01)	-0,02	-0,02	-0,00	-0,00	-0,00
AIF	-0,0033 (±0,01)	-0,02	0,00	0,00	0,00	0,00
NKKA	-19,0787 (±23,34)	-58,60	-21,43	-9,15	-4,27	-1,95
VK	-0,3258 (±0,26)	-0,63	-0,46	-0,42	-0,09	-0,03
BL	78038,51 (±163228,3)	-9545,9	3052,6	9167,7	18040,1	369478,0
ZG	22,4 (±6)	14	21	21	27	29
AZG	-205,8 (±285)	-710	-137	-101	-45	-36
AZG%	-25,1 (±22)	-51	-44	-21	-5	-5

Tabelle 8.20: Übersicht über die Metrikveränderungen zwischen Anfangs- und Endzustand

menliegen zusammengehöriger Artefakte sowie loserer Kopplung, sodass insgesamt weniger Bits zur Darstellung der Struktur benötigt wurden.

Extrema der Metrikveränderungen Die Effektivität der Zyklenauflösung hängt stark vom unterliegenden Softwaresystem ab. Tabelle 8.21 zeigt dabei die Rekordhalter der extremstem Änderungen der Metriken NKKA, VK, AZG und AZG%.

Die größte absolute Abnahme von AZG mit 710 Klassen erfolgte bei ArgoUML, ebenso wie die Abnahmen um den größten Faktor bei VK mit 7,23 und NKKA mit 10,05. Die größte prozentuelle Differenz bei AZG% erzielte Projectfactory mit 51%.

imp-a weist Minimalrekorde bei NKKA mit 1,75, VK mit 1,44 und AZG% mit 5% auf. Die absolut kleinste Reduktion von AZG hält imp-b mit 36 Klassen.

In Summe manifestiert sich die Zyklenauflösung in zum Teil um ein Vielfaches verringerten Testaufwand sowie systemweiter Kopplung. Die Erhöhung der Anzahl Artefakte durch Anwendung von Auflösungstechniken nimmt sich mit höchstens 15 Klassen dagegen bescheiden aus. Die durch die MOOD-Metriken ausgedrückte Kopplung und Kohäsion lassen sich jedoch nicht feststellbar durch Zyklenauflösungsbestrebungen beeinflussen.

Eigenschaft	Wert	Applikation
Größter Faktor bei NKKA	10,05	ArgoUML
Kleinster Faktor bei NKKA	1,75	imp-a
Größter Faktor bei VK	7,23	ArgoUML
Kleinster Faktor bei VK	1,44	imp-a
Größte Differenz bei AZG	710	ArgoUML
Kleinste Differenz bei AZG	36	imp-b
Größte Differenz bei AZG%	51	Projectfactory
Kleinste Differenz bei AZG%	5	imp-a

Tabelle 8.21: Rekordhalter bei Metrikänderungen zwischen Anfangs- und Endzustand

Rolle	1	2	3	4
Softwareentwickler	ja	ja*	ja	nein
Softwareentwickler	ja	ja*	nein	ja*
Programmierer	ja	ja*	ja	nein
Systemarchitekt	ja	ja*	ja	–
Alles	–	ja*	ja*	nein
Softwareentwickler	ja	ja	ja	ja
Softwareentwickler	ja	nein	ja	ja
Softwareentwickler	ja	ja*	ja*	nein
–	ja	nein	ja	ja
Projektleiter	ja	nein	ja	nein
Softwareentwickler	ja	nein	ja	nein
–	ja	nein	nein	nein

Tabelle 8.22: Alle Rückläufer des Fragebogens

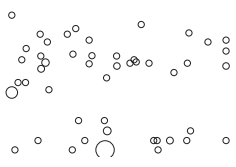
8.3.2 Akzeptanz

Die Akzeptanz der automatisiert durchgeführten Zyklenreduktionen an den Softwaresystemen wurde anhand des in Kapitel 8.1.7 aufgeführten Fragebogens festgestellt. Von den 23 an ausgewählte Mitglieder der jeweiligen Softwaresysteme elektronisch versandten Fragebögen erhielten wir 12 Rückläufer, die in Tabelle 8.22 aufgeführt sind.

Die Mehrheit der befragten Personen nahm die Rolle des Softwareentwicklers oder Programmierers ein. Diese Gewichtung überrascht nicht, da zur Begutachtung der Änderungen softwareentwicklungsspezifisches Grundwissen vorausgesetzt wird.

Die einzelnen Fragen interpretieren wir jeweils summarisch. Die angeführte Nummer entspricht der Nummer der Frage auf dem Fragebogen.

ad 1 In keinem einzigen Fall wurde durch die automatisierten Umbauoperationen die Wiedererkennbarkeit des Softwaresystems beeinträchtigt.



ad 2 Bei der Frage nach der Verbesserung durch die ausgeführten Änderungen zeigte sich bereits ein differenzierteres Bild.

Eine Mehrheit von 7 betrachtete die Änderungen als Verbesserung, 5 gewannen den Änderungen keine Verbesserungen ab.

Von den positiven Antworten knüpften jedoch 6 Personen die Zustimmung an Bedingungen (ja* in Tabelle 8.22).

Die Bedingungen lassen sich in Gruppen zusammenfassen. Die mittels Abhängigkeitsumkehr durchgeführten Zyklenauflösungen wurden überwiegend als sinnvoll betrachtet und nur in einem Fall abgelehnt.

Lediglich in einem Fall wurde bemängelt, dass die extrahierten Schnittstellen zu groß wären und in weitere, kleinere Schnittstellen unterteilt werden sollten. In einem anderen Fall wurde eine vollautomatische Quelltexterzeugung gegenüber der Extrahierung von Schnittstellen bevorzugt. Diese vollautomatische Erzeugung setzt allerdings fachspezifisches Wissen voraus, das mit einem allgemeinen Zyklenauflösungsansatz nicht erreicht werden kann.

Die Knotenspaltung wurde ebenfalls weitgehend positiv aufgenommen. Eine Person merkte sogar explizit an, dass ihr die Trennung einer Hilfsklasse in drei Hilfsklassen nach Zuständigkeiten besonders zusagte.

Die Trennung nach Zuständigkeiten entstand als unbeabsichtigte Folge der Anwendung der Knotenspaltung, für die wir uns aufgrund der Analyse von Fragmentierern und Kanten entschieden hatten. Die Zyklenreduktion bewirkte damit eine nachweisliche Verbesserung der Softwarearchitektur.

Nur zwei Personen empfanden die durch Knotenspaltung aufgelösten Abhängigkeiten als nicht nachvollziehbar.

Die Änderungen durch Rückrufe stießen mit drei Ablehnungen auf den größten Widerstand. Als vorherrschenden Grund nannten die Befragten die Abwicklung der Rückrufe über eine gemeinsame Abfertigungsklasse und betrachteten deren Aufspaltung nach Zugehörigkeit mittels manueller Nachbearbeitung als notwendig.

Eine Person bevorzugte Quelltextentfernung statt der Auflösung von Abhängigkeiten, eine weitere Person begründete ihre Ablehnung mit fehlendem Kontext, und eine andere Person begründete ihre Ablehnung überhaupt nicht.

ad 3 Die Frage, ob die Befragten grundsätzlich mit einem automatisiert geänderten System weiterarbeiten könnten, wurde mit 10 : 2 mehrheitlich positiv beantwortet.

Eine Person begründete ihre Ablehnung mit der Bevorzugung manueller Änderungen, eine andere Person mit dem Hinweis, dass einige Klassen ohnehin bereits automatisch generiert würden und daher keiner automatisierten Änderung bedurften.

Die bedingten Zustimmungen (ja*) beliefen sich zum einen nicht auf die Auflösung selbst, sondern auf die Anwendung auf eine »veraltete« Version, zum anderen auf die Bevorzugung manueller Änderungen.

Eine grundsätzliche Ablehnung automatisierten Umbaus kann daher aus den Umfrageergebnissen nicht abgeleitet werden.

ad 4 Nur 4 Personen gaben ein positives Signal, auch mit automatisiert umgebauter Software weiterarbeiten zu wollen. Eine Mehrheit von 7 Personen lehnte eine Weiterarbeit ab.

Die Gründe der Ablehnung sind vielschichtig und lassen sich in weitgehend zwei Kategorien fassen: direkte Ablehnung der Zyklenauflösung und indirekte Ablehnung durch Einbeziehung äußerer Einflüsse.

In erstere Kategorie fallen einmal die Bevorzugung der manuellen Auflösung gegenüber der automatischen, einmal die Bemängelung der Namensgebung der erzeugten Artefakte, einmal die Ablehnung von Rückrufen sowie die einmalige unspezifische Feststellung, dass die Änderungen keine Verbesserung bedeuteten.

Die zweite Kategorie betreffen einmal mangelndes Interesse sowie einmal die Ablehnung der Änderungen, da sich die Entwicklerversion bereits zu weit von der untersuchten Version entfernt hatte.

Eine Ablehnung außerhalb beider Kategorien betraf die Feststellung, die Problemstellung sei zu komplex für eine automatische Auflösung.

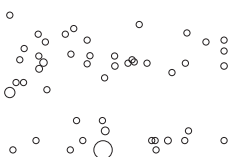
Zusammengefasst erfolgte eine Ablehnung in der Sache selbst nur vier Mal, womit sich die Ablehnungen mit den Zustimmungen die Waage halten.

Die Nützlichkeit automatisierter Auflösung zyklischer Abhängigkeiten konnte insofern bestätigt werden, als dass die über die besten Fragmentierer aufgedeckten Klassen mehrheitlich als Brennpunkte mit Verbesserungspotential identifiziert wurden.

Dabei stieß die Anwendung der Abhängigkeitsumkehr auf die höchste Akzeptanz, die Anwendung der Knotenspaltung auf die zweithöchste. Die Rückrufe wurden als solche abgelehnt.

Die Ablehnung automatischer Änderungen, soweit aus ihr überhaupt Kritik am Verfahren selbst hervorging, basierte zum großen Teil auf der Tatsache, dass Änderungen eingefordert wurden, die ein automatisches Verfahren nicht entscheiden kann. Solche Änderungen stellten eine nach Sachgebiet oder Subsystemzugehörigkeit sinnvolle Namensgebung oder eine Notwendigkeit weiterer sachgebiets- oder subsystemspezifischer Aufspaltung erzeugter Artefakte dar.

Weiters spielten externe Einflüsse bei vielen Ablehnungen eine Rolle, die weder vom angewandten Verfahren verursacht wurden noch sich auf das Verfahren selbst bezogen. Sie können daher nicht der Zyklenauflösung an sich angelastet werden.



8.3.3 Erfahrungen

Durch die Reduktion zyklischer Abhängigkeiten anhand von fünf realen Softwaresystemen erlangten wir einige Erfahrungen.

- Es reichen bereits unter 20 Schritte, um die Zyklengruppengröße jeder auflösungswürdigen Zyklengruppe auf nicht mehr als 15 Elemente zu bringen.
- Jede große Zyklengruppe hängt in der Regel durch wenige gute Fragmentierer zusammen. Damit ließen sich durch die Herauslösung der besten oder auch zweitbesten Fragmentierer jeweils in wenigen Schritten große Reduktionserfolge erzielen.
- Tiefe und breite Vererbungshierarchien, die ihrerseits in Basisklassen Objekte abgeleiteter Klassen erzeugen, erschweren die Zyklenreduktion außerordentlich, da die Vorwärtskanten unauflösbare Vererbungsbeziehungen und die Rückwärtskanten schwer auflösbare Objekterzeugungsbeziehungen enthalten.
- Für Klassen, die überwiegend über Objekte verwendet werden, empfiehlt sich die Verwendung der Abhängigkeitsumkehr.
- Für Klassen mit überwiegend statischen Methoden empfiehlt sich die Verwendung der Knotenspaltung.
- Objekterzeugungen lassen sich durch Rückrufe auflösen. Da Rückrufe jedoch auf geringe Akzeptanz stoßen, ist die manuelle Anwendung einer Lösung wie in Projectfactory empfehlenswert.
- Singletonklassen lassen sich durch eine Kombination der drei Auflösungstechniken aufspalten, wie anhand von ArgoUML praktiziert wurde.

Kapitel 9

Zusammenfassung und Ausblick

In dieser Arbeit spezifizierten wir Auflösungstechniken zur automatisierten Auflösung von zyklischen Abhängigkeiten in Softwaresystemen. Dazu implementierten wir ein Werkzeug, mit dem wir sowohl statistisch die Eigenschaften dreier Auflösungstechniken an realen Softwaresystemen untersuchten als auch exemplarisch in realen Softwaresystemen zyklische Abhängigkeiten reduzierten.

Die drei implementierten Auflösungstechniken sind die Abhängigkeitsumkehr, die gerichtete Knotenspaltung und die Rückrufe.

9.1 Statistische Ergebnisse

Im Rahmen einer großangelegten Untersuchung wurden die drei Auflösungstechniken zur Reduktion zyklischer Abhängigkeiten vollautomatisch auf alle Rückwärtskanten von 6091 Java-Applikationen angewandt und ihre Effektivität sowie die Auswirkung auf bekannte systemweite Metriken untersucht. Je nach Auflösungstechnik konnten 4,6% bis 43% der Kanten aufgelöst werden, was zu einer Zyklenreduktion zwischen 2,8% und 26% führte.

Die Rückrufe konnten die meisten Kanten auflösen, die Knotenspaltung die wenigsten. Jedoch führte eine exzessive Nutzung der Rückrufe trotz Auflösung zyklischer Abhängigkeiten zu einem (manchmal sogar exorbitanten) Anstieg des Testaufwands. Die anderen Auflösungstechniken verringerten hingegen den Testaufwand. Die systemweite Kopplung vermochten alle drei Auflösungstechniken zu verringern.

9.2 Exemplarische Ergebnisse

Noch interessantere Ergebnisse als die statistische Auswertung lieferte die exemplarische Auswertung. Hier reduzierten wir mittels der drei Auflösungstechniken die zyklischen Abhängigkeiten in fünf Softwaresystemen. Statt einer vollautomatischen Auflösung wandten wir selektiv über eine Benutzeroberfläche Auflösungstechniken auf Abhängigkeiten an und ließen sie, so weit möglich, automatisch auf-

lösen. Wir versuchten die Anzahl der Klassen in Zyklengruppen zu reduzieren, behielten jedoch immer im Auge, die ursprüngliche Architektur nicht degenerieren zu lassen. Dabei maßen wir Zeitaufwand und Veränderungen von Systemmetriken und erfragten am Schluss die Akzeptanz der Änderungen von an diesen Softwaresystemen beteiligten Personen.

Der Zeitaufwand zur Reduktion bis zu einem Endzustand betrug im Durchschnitt 12,5 Stunden und ließ sich stets mit weniger als 20 Schritten bewerkstelligen. Durch den Einsatz automatischer Auflösungstechniken ermittelten wir eine Zeitersparnis von ungefähr 30%, wenn wir stattdessen die Auflösung rein händisch durchgeführt hätten. Die Zyklenreduktion verringerte – wie bei der statistischen Auswertung – für jedes Softwaresystem den Testaufwand sowie die systemweiten Kopplung.

Die Befragung der Beteiligten an den untersuchten Softwaresystemen zeigte, dass die Änderungen die Softwaresysteme weder unkenntlich machten noch als grundlegend hinderlich zur Verbesserung der Softwaresysteme angesehen wurden. Die Reaktionen zeigten weiters, dass unser Verfahren durchaus in der Lage war, Brennpunkte aufzufinden, die die Entwickler selbst bereits als problematische Konstruktionen erkannt und teilweise bereits behoben hatten oder im Begriff sie zu beheben waren.

Die Entwickler monierten jedoch die Umsetzung einzelner Auflösungstechniken, insbesondere der Rückrufe, und zogen in einigen Fällen eine manuelle Auflösung einer automatisierten vor. Weitere Kritikpunkte betrafen Änderungen, zu deren zielgerichteterer Anwendung softwaresystemspezifisches Fachwissen erforderlich gewesen wäre. Dies betraf insbesondere die Benennung von Klassen.

9.3 Erreichte Ziele

Wir erreichten die in der Einleitung formulierten Ziele.

- Die Anzahl der Klassen in zyklischen Abhängigkeiten konnte in jedem Fall reduziert werden.

- Wir implementierten Auflösungstechniken, die eine automatisierte Auflösung von Abhängigkeiten ermöglichten.

- Wir wählten aus der Literatur nur solche Auflösungstechniken, die das Programmverhalten nicht verändern und spezifizierten sie über Umbauoperationen.

- Die befragten Entwickler der untersuchten Softwaresysteme erkannten die Softwaresysteme in jedem Fall wieder und empfanden die Änderungen mehrheitlich sinnvoll.

9.4 Ausblick

Das hier vorgestellte Verfahren setzt einen Grundstein für Entwickler und Softwarearchitekten, um zyklische Abhängigkeiten in Softwaresystemen automationsgestützt zu verringern. Die Auflösungstechniken beschränken sich jedoch nicht nur auf zyklische Abhängigkeiten, prinzipiell können sie alle Arten von Abhängigkeiten auflösen.

Das Verfahren ist damit aber nicht am Ende der Fahnenstange angelangt. Daher führen wir abschließend noch einige weiterführende Möglichkeiten auf, in diesem Feld zu forschen.

- Während die Auflösungstechniken automatisiert einzelne Abhängigkeiten auflösen, obliegt es immer noch dem Entwickler, entsprechende Auflösungsstrategien zu finden. Eine automatisierte Entdeckung von Auflösungsstrategien könnte dem Entwickler eine große Hilfe sein.

- Die Auflösungstechniken ignorieren fachspezifisches Wissen. Wege zur Integration fachspezifischen Wissens würden es ermöglichen, Auflösungstechniken zu Auflösungsstrategien zusammenzufassen oder zumindest automatisch erzeugte Artefakte akkurat zu benennen.

- Momentan nimmt die automatische Auflösung am Gesamtprozess der Softwareverbesserung nur einen geringen Anteil ein. Zukünftige Entwicklungen könnten versuchen, diesen Anteil zu erhöhen.

- Die jetzigen Hinweise auf auflösbare Abhängigkeiten sind nicht so nützlich, wie sie sein könnten. Es erscheint daher zweckmäßig, bereits durchgeführte Auflösungen zu speichern und bei ähnlichen Situationen vorzuschlagen.

- Die vorgestellten Auflösungstechniken sind statischer Natur. Eventuell lassen sich mittels Datenschürfung in einem großen Stichprobe von Softwaresystemen, an denen Verbesserungen durchgeführt wurden, weitere Auflösungstechniken finden, die der Literatur bisher entgingen.

- Gefundene Probleme werden mittels einfacher Verbindungsdiagramme visualisiert. Damit ist die Visualisierung noch nicht ausgereizt. Hier ließen sich weitere Visualisierungsarten finden, die problematische Teile eines komplexen Softwaresystems schnell auffindbar und analysierbar machen.

- Zur Zeit unterstützt die Implementierung nur Java-Softwaresysteme, obwohl das unterliegende Modell sprachunabhängig konzipiert ist. Eine Herausforderung wäre die Unterstützung weiterer objektorientierter Sprachen und die Dokumentation der damit verbundenen Probleme und Eigenheiten.

- Interessant wäre auch zu untersuchen, wie sich die automatisierte Zyklenauflösung in einen Softwareentwicklungsprozess integrieren lässt und wie sie das Ergebnis beeinflusst. Insbesondere eine vergleichende Entwicklung

(einmal mit automatisierter Auflösung, einmal ohne) würde Aussagen über die Wirtschaftlichkeit automatisierter Zyklenauflösung liefern und eine bessere Anpassung an »gelebte« Softwareentwicklung ermöglichen.

Literaturverzeichnis

[1] ABRAN, ALAIN, JEAN-MARC DESHARNAIS, SERGE OLIGNY, DENIS ST-PIERRE und CHARLES SYMONS: *COSMIC-FFP Measurement Manual 2.2*. Technischer Bericht, Common Software Measurement International Consortium, 2003.

[2] ABREU, FERNANDO BRITO E und ROGÉRIO CARAPUÇA: *Object-Oriented Software Engineering: Measuring and Controlling the Development Process*. In: *Proceedings of the 4th International Conference on Software Quality*, 1994.

[3] ABREU, FERNANDO BRITO E und WALCÉLIO MELO: *Evaluating the Impact of Object-Oriented Design on Software Quality*. In: *Proceedings of the 3rd International Software Metrics Symposium*, 1996.

[4] ADAMEK, JIRI: *Addressing Unbounded Parallelism in Verification of Software Components*. Technischer Bericht, Abteilung Software Engineering, Karlsuniversität Prag, 2006.

[5] ALBRECHT, ALLAN J. und JOHN E. GAFFNEY: *Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation*. IEEE Transactions on Software Engineering, 9(6):639–648, 1983.

[6] BENLARBI, SAÏDA und NISHITH GOEL: *Measuring Object-Oriented Software Systems*. Technischer Bericht, Cistel Technology, 1998.

[7] BEST, D. J. und D. E. ROBERTS: *Algorithm AS 89: The Upper Tail Probabilities of Spearman's Rho*. Applied Statistics, 24(3):377–379, 1975.

[8] BOEHM, BARRY W.: *Software Engineering Economics*. Prentice-Hall Inc., Englewood Cliffs, N. J. 07632, 1981.

[9] BOOTSMA, FRED: *How to Obtain Accurate Estimates in a Real-Time Environment Using Full Function Points*. In: *Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, Seite 105ff, 2000.

[10] BÄR, HOLGER: *FAMIX C++ language plug-in 1.0*. Technischer Bericht, Forschungszentrum Informatik, Karlsruhe, 1999.

[11] BRIAND, LIONEL, PREM DEVANBU und WALCELIO MELO: *An investigation into coupling measures for C++*. In: *Proceedings of the 19th international conference on Software engineering*, Seiten 412–421, 1997.

[12] *Programming Language C++*. Technischer Bericht, ISO, 1998. ISO/IEC 14882.

[13] CHIDAMBER, SHYAM R. und CHRIS F. KEMERER: *A Metrics Suite for Objekt Oriented Design*. IEEE Transactions on Software Engineering, 20(6), 1994.

[14] CLARK, JOHN und DEREK ALLAN HOLTON: *Graphentheorie*. Spektrum Akademischer Verlag, 1994.

[15] DAGPINAR, MELIS und JENS H. JAHNKE: *Predicting Maintainability with Object-Oriented Metrics – An Empirical Comparison*. In: *Proceedings of the 10th Working Conference on Reverse Engineering*, Seite 155ff, 2003.

[16] DEMETRESCU, CAMIL und IRENE FINOCCHI: *Combinatorial algorithms for feedback problems in directed graphs*. Information Processing Letters, 86(3):129–136, 2003.

[17] DEMEYER, SERGE, STÉPHANE DUCASSE und SANDER TICHELAAR: *Why FAMIX and not UML?* Technischer Bericht, Universität Bern, 1999.

[18] DEMEYER, SERGE, SANDER TICHELAAR und PATRICK STEYAERT: *FAMIX 2.0: The FAMOOS Information Exchange Model*. Technischer Bericht, Universität Bern, 1999.

[19] DUCASSE, STÉPHANE, TUDOR GÎRBA, MICHELE LANZA und SERGE DEMEYER: *Moose: a Collaborative and Extensible Reengineering Environment*, Seiten 55–71. Franco Angeli, Milano, 2005.

[20] EADES, PETER, XUEMIN LIN und W. F. SMYTH: *A Fast Effective Heuristic For The Feedback Arc Set Problem*. Information Processing Letters, 47(6):319–323, 1993.

[21] EMAM, KHALED EL, WALCELIO MELO und JAVAM C. MACHADO: *The prediction of faulty classes using object-oriented design metrics*. Journal of Systems and Software, 56(1):63–75, 2001.

[22] FAHRMEIR, LUDWIG, RITA KÜNSTLER, IRIS PIGEOT und GEHRHARD TUTZ: *Statistik*. Springer Berlin Heidelberg New York, 5 Auflage, 2007.

[23] FENTON, NORMAL: *Software Measurement: A Necessary Scientific Basis*. IEEE Transactions on Software Engineering, 20(3):199–206, 1994.

[24] FLATSCHER, RONY G.: *Metamodeling in EIA/CDIF—meta-metamodel and metamodels*. ACM Transactions on Modeling and Computer Simulation, 12(4):322–342, 2002.

[25] FOWLER, MARTIN: *Refactoring*. Addison-Wesley, 2000.

[26] FOWLER, MARTIN R.: *Reducing Coupling*. IEEE Software, 18(4):102–104, 2001.

[27] FRAKES, WILLIAM und CAROL TERRY: *Software reuse: metrics and models*. ACM Computing Surveys, 28(2):415–435, 1996.

[28] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns CD*. Addison Wesley Longman, 1996.

[29] GARRIDO, ALEJANDRA und RALPH JOHNSON: *Challenges of refactoring C programs*. In: *Proceedings of the International Workshop on Principles of Software Evolution*, Seiten 6–14, 2002.

[30] GENTILINI, RAFFAELLA, CARLA PIAZZA und ALBERTO POLICRITI: *Computing strongly connected components in a linear number of symbolic steps*. In: *Proceedings of the 14th ACM-SIAM symposium on Discrete algorithms*, Seiten 573–582, 2003.

[31] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *The Java? Language Specification*. Addison-Wesley, 3. Auflage, 2005.

[32] GRAY, ANDREW R., STEPHEN G. MACDONELL und MARTIN J. SHEPPERD: *Factors systematically associated with errors in subjective estimates of software development effort: the stability of expert judgment*. In: *Proceedings of the Sixth International Software Metrics Symposium*, Seite 216ff, 1999.

[33] GÎRBA, TUDOR: *Modeling History to Unterstand Software Evolution*. Doktorarbeit, Universität Bern, 2005.

[34] GRUNTZ, DOMINIK: *Java Design: On the Observer Pattern*. Java Report, Feb. 2002.

[35] HANNEMANN, JAN und GREGOR KICZALES: *Design pattern implementation in Java and aspectJ*. In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 161–173, 2002.

[36] HARRISON, RACHEL, STEVE J. COUNSELL und REUBEN V. NITHI: *An Evaluation of the MOOD Set of Object-Oriented Software Metrics*. IEEE Transactions on Software Engineering, 24(6):491–496, 1998.

[37] HAUTUS, EDWIN: *Improving Java Software Through Package Structure Analysis*. In: *Proceedings of the 6th IASTED International Conference Software Engineering and Applications*, 2002.

[38] HITZ, MARTIN, GERTRUDE KAPPEL, ELISABETH KAPSAMMER und WERNER RETSCHITZEGGER: *UML@Work*. dpunkt.verlag Heidelberg, 3. Auflage, 2005.

[39] KARP, RICHARD MANNING: *Reducibility among combinatorial problems*. In: *Complexity of Computer Computations*, Seiten 85–103. Plenum Press, 1972.

[40] KICZALES, GREGOR, JOHN LAMPING, ANURAG MENDHEKAR, CHRIS MAEDA, CRISTINA LOPES, JEAN-MARC LOINGTIER und JOHN IRWIN: *Aspect-Oriented Programming*. In: *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[41] KIEBUSCH, SEBASTIAN und BOGDAN FRANCZYK: *Prozess-Familien-Punkte*. Informatik – Forschung und Entwicklung, 20(4):222–229, 2006.

[42] LAKOS, JOHN: *Large-Scale C++ Software Design*. Addison Wesley Longman, 1996.

[43] LANZA, MICHELE: *CodeCrawler – A Lightweight Software visualization Tool*, 2003. Universität Bern.

[44] LANZA, MICHELE: *Object-Oriented Reverse Engineering – Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. Doktorarbeit, Universität Bern, 2003.

[45] LEHMAN, MEIR MANNY: *Laws of software evolution revisited*, Seiten 108–124. Lecture Notes in Computer Science. Springer Verlag, 1996.

[46] LI, WEI: *Another metric suite for object-oriented programming*. Journal of Systems and Software, 44(2):155–162, 1998.

[47] LUTZ, RUDI: *Evolving good hierarchical decompositions of complex systemsy*. Journal of Systems Architecture, 47(7):613–634, 2001.

[48] LUTZ, RUDI: *Recovering High-Level Structure of Software Systems Using a Minimum Description Length Principle*. In: *Artificial Intelligence and Cognitive Science : 13th Irish International Conference*, Seiten 63–80, 2002.

[49] MACCORMACK, ALAN, JOHN RUSNAK und CARLISS BALDWIN: *Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code*. Technischer Bericht, Havard Business School, 2006.

[50] MACDONELL, STEPHEN G. und MARTIN J. SHEPPERD: *Combining techniques to optimize effort predictions in software project management*. Journal of Systems and Software, 66(2):91–98, 2003.

[51] MARTIN, ROBERT C.: *Object Oriented Design Quality Metrics*. ROAD, 1995.

[52] MARTIN, ROBERT C.: *The Dependency Inversion Principle*. C++ Report, 1996.

[53] MARTIN, ROBERT C.: *Granularity*. The C++ Report, 8(11), 1996.

[54] MARUYAMA, KATSUHISA und SHINICHIRO YAMAMOTO: *Design and Implementation of an Extensible and Modifiable Refactoring Tool*. In: *Proceedings of the 13th International Workshop on Program Comprehension*, 2005.

[55] MAYER, TOBIAS und TRACY HALL: *A Critical Analysis of Current OO Design Metrics*. Software Quality Journal, 8(2):97–110, 1999.

[56] MELTON, HAYDEN und EWAN TEMPERO: *Empirical Study of Cycles among Classes in Java*. Technischer Bericht, Department of Computer Science, Universität Auckland, 2006.

[57] MELTON, HAYDEN und EWAN TEMPERO: *Identifying Refactoring Opportunities by Identifying Dependency Cycles*. In: *Proceedings of the 29th Australasian Computer Science Conference*, 2006.

[58] MELTON, HAYDEN und EWAN TEMPERO: *JooJ: Real-time Support for Avoiding Cyclic Dependencies*. Technischer Bericht, Department of Computer Science, Universität Auckland, 2006.

[59] MELTON, HAYDEN und EWAN TEMPERO: *An Empirical Study of Cycles among Classes in Java*. Empirical Software Engineering, 4(12):389–415, 2007.

[60] MENZIES, TIM, ZHIHAO CHEN, JAIRUS HIHN und KAREN LUM: *Selecting Best Practices for Effort Estimation*. IEEE Transactions on Software Engineering, 32(11):883–895, 2006.

[61] *MOF 2.0/XMI Mapping Specification, v2.1*. Technischer Bericht, Object Management Group, 2005.

[62] MOHANTY, SIBA N.: *Models and Measurements for Quality Assessment of Software*. ACM Computing Surveys, 11(3):251–275, 1979.

[63] MOLOKKEN, KJETIL und MAGNE JORGENSEN: *A Review of Surveys on Software Effort Estimation*. In: *Proceedings of the International Symposium on Empirical Software Engineering*, Seite 223ff, 2003.

[64] MOORE, IVAN: *Automatic inheritance hierarchy restructuring and method refactoring*. ACM SIGPLAN Notices, 31(10):235–250, 1996.

[65] NARAHARI, Y: *Data Structures and Algorithms*. Indisches Institut der Wissenschaften, Bangalur, ca. 2000. http://lcm.csa.iisc.ernet.in/dsa/.

[66] NIERSTRASZ, OSCAR, STÉPHANE DUCASSE und TUDOR GÎRBA: *The Story of Moose: an Agile Reengineering Environment*. In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering 2005*, 2005.

[67] NUUTILA, ESKO und ELJAS SOISALON-SOININEN: *On Finding the Strong Components in a Directed Graph*. Technischer Bericht, Laboratory of Information Processing Science, Technische Universität Helsinki, 1994.

[68] OFFUTT, JEFF und AYNUR ABDURAZIK: *Coupling-based class integration and test order*. In: *Proceedings of the international workshop on Automation of software test*, Seiten 50–56, 2006.

[69] OPDYKE, WILLIAM F.: *Refactoring Object-Oriented Frameworks*. Doktorarbeit, Universität Illinois, Urbana-Champaign, 1992.

[70] PARNAS, DAVID LORGE: *Designing software for ease of extension and contraction*. In: *Proceedings of the 3rd international conference on Software engineering*, 1978.

[71] PASSING, URSULA und MARTIN SHEPPERD: *An experiment on software project size and effort estimation*. In: *Proceedings of the International Symposium on Empirical Software Engineering*, Seite 120ff, 2003.

[72] PINZGER, MARTIN: *ArchView – Analyzing Evolutionary Aspects of Complex Software Systems*. Doktorarbeit, Technische Universität Wien, 2005.

[73] PLEWAN, HANS-JÜRGEN: *Methodische Aufwandsschätzung aus Sicht eines agilen Projektmanagements*, Seiten 83–100. dpunkt verlag, 2006.

[74] R DEVELOPMENT CORE TEAM: *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Wien, Österreich, 2007. ISBN 3-900051-07-0.

[75] SANGAL, NEERAJ, EV JORDAN, VINEET SINHA und DANIEL JACKSON: *Using Dependency Models to Manage Complex Software Architecture*. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2005.

[76] SANTILLO, LUCA, MASSIMILIANO CONTE und ROBERTO MELI: *Early & Quick Function Point: Sizing More with Less*. In: *Proceedings of the 11th IEEE International Software Metrics Symposium*, Seite 41ff, 2005.

[77] SANTILLO, LUCA und ROBERTO MELI: *Early Function Points: some practical experiences of use.* In: *Proceedings of the 9th European Software Control and Metrics Conference*, 1998.

[78] SAVERNIK, LEO: *Anatomie zyklischer Abhängigkeiten in Softwaresystemen.* Technischer Bericht 1, Institut für Systemsoftware, Universität Linz, 2007.

[79] SAVERNIK, LEO: *Empirischer Vergleich von Metriken zur Komplexitätsmessung*. Technischer Bericht, Institut für Systemsoftware, Universität Linz, 2007. (unveröffentlicht).

[80] SKIENA, STEVEN S.: *The Algorithm Design Manual.* Springer-Verlag, 1997.

[81] SOFTWARE TOMOGRAPHY GMBH: *Sotograph 2.4 User's Guide and Reference Manuals*, 2006. http://www.software-tomography.com/.

[82] SOMMERVILLE, IAN: *Software Engineering*. Pearson Studium, 6. Auflage, 2001.

[83] SPINELLIS, DIOMIDIS: *Global Analysis and Transformations in Preprocessed Languages.* IEEE Software Engineering, 29(11):1019–1030, 2003.

[84] STENTEN, HANS: *Java naar FAMIX parsen*, 2002.

[85] STOREY, MARGARET-ANNE: *Theories, Methods and Tools in Program Comprehension: Past, Present and Future.* In: *Proceedings of the 13th International Workshop on Program Comprehension*, 2005.

[86] SYMONS, CHARLES R.: *Function Point Analysis: Difficulties and Improvements.* IEEE Transactions on Software Engineering, 14(1):2–11, 1988.

[87] TEGARDEN, DAVID P., STEVEN D. SHEETZ und DAVID E. MONARCHI: *Effectiveness of traditional software metrics for object-orientedsystems.* In: *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, Seiten 359–368, 1992.

[88] TEGARDEN, DAVID P., STEVEN D. SHEETZ und DAVID E. MONARCHI: *A Software Complexity Model of Object-Oriented Systems.* Decision Support Systems: The International Journal, 13(1):241–262, 1995.

[89] TICHELAAR, SANDER: *Complete CDIF FAMIX specification*, 1999.

[90] TICHELAAR, SANDER: *FAMIX Java language plug-in 1.0.* Technischer Bericht, Universität Bern, 1999.

[91] TONELLA, PAOLO: *Concept Analysis for Module Restructuring.* IEEE Transactions on Software Engineering, 27(4):351–363, 2001.

[92] *Unified Modeling Language: Superstructure*, 2005. Object Management Group.

[93] WERMELINGER, MICHEL, KIM MENS und TORN MENS: *Supporting software evolution with intentional software views.* In: *Proceedings of the International Workshop on Principles of Software Evolution*, 2002.

[94] WONG, KENNY: *Rigi User's Manual – Version 5.4.4.* Technischer Bericht, University of Victoria, 1998. http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf.

[95] XIE, AIGUO und PETER A. BEEREL: *Implicit enumeration of strongly connected components.* In: *Proceedings of the IEEE/ACM international conference on Computer-aided design*, Seiten 37–40, 1999.