

Automatische Zyklenauflösung unter Berücksichtigung von Quell- und Binärkompatibilität

Ein Vorschlag für eine Dissertation

Leo Savernik

24. August 2006

Zusammenfassung

Zyklische Abhängigkeiten zwischen Komponenten eines Softwaresystems erschweren das Programmverständnis, die Erweiterbarkeit, Wartbarkeit und Testbarkeit. So kann der Interessierte eine in zyklischer Abhängigkeit stehende Komponente nicht für sich alleine betrachten, sondern hat die durch Rückkopplung verursachten Einflüsse bei Änderungen an der Komponente auf die Komponente selbst zu berücksichtigen. Je vielfältiger und größer die Zyklen, desto schwerer wird die Entwicklung eines Softwaresystems handhabbar. Dieser Vorschlag präsentiert einen Plan zur Entwicklung eines Verfahrens zur automatischen Auflösung zyklischer Abhängigkeiten in Softwaresystemen. Das Verfahren sucht Zyklen und löst diese – soweit möglich – selbständig auf und minimiert die Notwendigkeit manueller Eingriffe. Dabei berücksichtigt das Verfahren den Erhalt von Quell- und Binärkompatibilität öffentlicher Schnittstellen, damit externe Verwender nicht durch unbedachte Schnittstellenänderungen in Mitleidenschaft gezogen werden.

Inhaltsverzeichnis

1	Einführung	2
1.1	Was ist eine zyklische Abhängigkeit?	3
1.2	Kapitelübersicht	3
2	Motivation	3
2.1	Hintergrund	3
2.2	Zum Verfahren	4
2.3	Berücksichtigung der Wirklichkeit	4
3	Stand der Technik	5
3.1	Zyklische Abhängigkeiten in Java-Applikationen	5
3.2	Weitere Erkenntnisse zu zyklischen Abhängigkeiten	6
3.3	Erkennungstechniken	7
3.4	Auflösungstechniken	9
3.5	Umbau	13
3.6	Kompatibilität	13
3.7	Quellkompatibilität	14
3.8	Binärkompatibilität	14

3.9	Vergleich von Quell- und Binärkompatibilität	15
3.10	Zusammenfassung	16
4	Ziele und Eigenleistung	16
4.1	Ziele	16
4.2	Eigenleistung	17
4.3	Offene Fragen	17
5	Übersicht über den Ansatz	18
5.1	Spezifikation	18
5.2	Ansatz	19
5.3	Vorläufige Struktur der Arbeit	21
6	Zusammenfassung und Terminplan	22
6.1	Zusammenfassung	22
6.2	Zeitplan	23

1 Einführung

Der Schlüssel zu zeitgerechter Fertigstellung und adäquater Implementierung von Anforderungen in Softwaresystemen liegt in Qualitätsmerkmalen wie guter Erweiterbarkeit, Wartbarkeit und Testbarkeit.

Die Implementierung zusätzlicher Anforderungen oder auch nur die Anpassung existierender Funktionalität an neue Anforderungen erfordert ein Verständnis des Systems und ein Zeitkontingent zur Einfeldung in die vorherrschende Architektur. Erst auf den gewonnenen Erkenntnissen aufbauend lässt sich die Integration dieser Anforderungen durchführen.

Weiters nimmt der Testaufwand eines Softwaresystems eine in Betracht zu ziehende Größe ein, die ebenfalls ein Zeitkontingent beansprucht.

Nun besitzen viele Softwaresysteme eine Eigenschaft, die das Programmverständnis sowie die Erweiterbarkeit, Wartbarkeit und Testbarkeit empfindlich stört: zyklische Abhängigkeiten [25].

Diese zyklischen Abhängigkeiten, auch kurz Zyklen genannt, erschweren das Programmverständnis, da der Wissbegierige eine in zyklischer Abhängigkeit stehende Komponente

- nicht separat betrachten und begreifen,
- eine Komponente nicht erweitern, ohne dass sie über den Zyklus eine Rückkopplung erfährt,
- nur schwer warten, weil schwer zu begreifen und
- nur mühsam und zeitaufwendig testen kann, da sich die einzelnen Komponenten nicht getrennt betrachten lassen.

Ein erstrebenswertes Ziel zur Verbesserung von Softwaresystemen liegt daher in der Eliminierung von zyklischen Abhängigkeiten.

Ein weiteres Ziel stellt die Berücksichtigung der Quell- und Binärkompatibilität dar. Komponenten realer Softwaresysteme besitzen Schnittstellen, deren Änderung im Rahmen einer Auflösung von zyklischen Abhängigkeiten nicht in Frage kommt. Die Forschungstätigkeit soll daher auch die Frage nach Berücksichtigung der Kompatibilität behandeln.

1.1 Was ist eine zyklische Abhängigkeit?

Zunächst definieren wir den Grundbegriff der Beziehung zwischen zwei Artefakten. Ein Artefakt ist ein physisches oder abstrahiertes Element eines Softwaresystems wie zum Beispiel Symbole, Funktionen, Klassen, Pakete, Subsysteme oder Architekturschichten.

Eine Beziehung zwischen zwei Artefakten liegt vor, wenn Artefakt A Artefakt B zu seiner korrekten Funktionsweise benötigt. Eine Beziehung stellt etwa ein Funktionsaufruf von B nach A dar oder die Ableitung einer Klasse in B von A, um nur einige Beziehungstypen zu nennen.

Ferner gilt, ein Artefakt A ist von Artefakt B abhängig, wenn A Beziehungen zu B unterhält. Abhängigkeit ist transitiv.

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

1.2 Kapitelübersicht

In Kapitel 2 wird der Hintergrund zur Zyklenproblematik genauer erläutert, konzeptuelle Alternativen angeführt und beschrieben, warum eine automatische Zyklenauflösung wünschenswert ist.

Kapitel 3 führt verschiedene Erkenntnisse und Verfahren zum Stand der Technik der Zyklenerkennung, Zyklenauflösung und der Kompatibilität auf.

Kapitel 4 kontrastiert die im Rahmen dieser Forschungstätigkeit zu untersuchenden Sachverhalte von den bereits existierenden.

Kapitel 5 beschreibt die angedachte Vorgehensweise unter Nennung von Voraussetzungen, Methoden und Ergebnissen und strukturiert die Forschungsarbeit grob, ohne einen konkreten Zeitplan vorzugeben.

Zuletzt wiederholt Kapitel 6 das bereits Gesagte und präsentiert einen Zeitplan, der die in Kapitel 5 beschriebene Vorgangsweise in ein zeitliches Raster fügt.

2 Motivation

Die Motivation hinter der automatischen Zyklenauflösung liegt in der Tatsache, dass erstens – wie bereits erwähnt – Zyklen ein Softwaresystem schwer wart- und erweiterbar machen und zweitens die manuelle Auflösung von Zyklen in realistischen Softwaresystemen ab 100000 Textzeilen viel Zeit beansprucht.

2.1 Hintergrund

Im Rahmen dieser Forschung konnte sich der Autor von »Softwareentwicklung im Felde« ein gutes Bild machen. Die grundsätzlichen Erkenntnisse bezüglich Qualitätssicherung belaufen sich auf das Ergebnis, dass die Qualität nicht etwa maßgeblich durch Irrtum oder Unwissenheit beeinträchtigt wird, sondern in der Regel mit Vorsatz, damit jene direkt zum finanziellen Erfolg der Unternehmung beitragenden Zeitpläne eingehalten werden können.

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

2.2 Zum Verfahren

Ein Verfahren zur automatischen Auflösung von Zyklen hilft der langfristigen Hebung der Softwarequalität hinsichtlich Erweiterbarkeit und Wartbarkeit, 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 ebenfalls personalintensiv. An Interessensvertretern sind nicht nur der Softwarearchitekt zur Oberaufsicht, Steuerung und Kontrolle der vorgenommenen Auflösung hinsichtlich der Konformanz zur Sollarchitektur 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 Zahl von Werkzeugen unterstützt das Auffinden von Zyklen 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 sei. Dieses Ziel ist allerdings utopisch.

Jedoch genügt es bereits, wenn *große Teile* des transformierten Systems ohne manuelle Nachänderung übernommen werden können. Dadurch reduziert sich der Testaufwand auf jenen Bruchteil der tatsächlich fälligen 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 bleiben dem Personal freilich nicht erspart, da im Endeffekt nur ein Mensch entscheiden kann, ob eine Änderung dem Geiste der Entwicklung entspricht.

2.3 Berücksichtigung der Wirklichkeit

Die Forschungstätigkeit entspringt einer in der Wirklichkeit auftretenden Problemstellung und wird diese Wirklichkeit weiter in Betracht ziehen.

Daher liefert eine naive automatische Zyklenauflösung schon alleine aus dem Grund keine befriedigenden Ergebnisse, da sie die Schnittstellen von Komponenten nicht berücksichtigt. Eine umsichtige Auflösung hat daher auf öffentliche Schnittstellen bedacht zu nehmen und sie von der Transformation auszuschließen oder sie zumindest so zu transformieren, dass dem Schnittstellenverwender kein Änderungsaufwand entsteht.

Die Berücksichtigung der Wirklichkeit wird deswegen unter dem Gesichtspunkt der *Kompatibilität* Eingang in die Forschung finden. Der Erhalt der Kompatibilität in den durch automatische Zyklenauflösung umgebauten Softwaresystemen stellt somit eine notwendige Bedingung für den Erfolg der Arbeit dar.

3 Stand der Technik

Zyklen in Softwaresystemen sind ein altbekanntes Phänomen. Bereits in den 1970er Jahren stellten Forscher fest, dass Zyklen Systeme erzeugten, bei denen die gegenseitige Benutzung von Modulen die Entwicklung einzelner Module unabhängig voneinander schwierig machte bis hin zu dem Punkt, an dem kein einziger Teil des Systems übersetzt und ausgeführt werden konnte, bevor nicht sämtliche Teile ausprogrammiert waren [35].

Bedingt durch die lange Einsatzdauer von Softwaresystemen und stetig wachsenden Änderungsanforderungen an Softwaresysteme spielen Qualitätsmerkmale wie Änderbarkeit und Wartbarkeit eine große Rolle. Zyklen zwischen Modulen wirken sich direkt negativ auf diese Qualitätsmerkmale aus und erhöhen Zeitaufwand und Kosten der Softwareentwicklung.

Das vordringliche Ziel stellt also die *Zyklenauflösung* dar. Auf dem Wege dorthin stehen bereits einige Verfahren und Werkzeuge zur Verfügung, die es genauer zu betrachten gilt.

3.1 Zyklische Abhängigkeiten in Java-Applikationen

Die neuesten Untersuchungen zu zyklischen Abhängigkeiten wurden anhand von Java-Applikationen durchgeführt. Zuallererst sei die *Zyklusstudie* [25] genannt, die in einem großangelegten Aufbau eine Anzahl von repräsentativen Java-Applikationen auf ihre Verzyklung untersucht.

Dabei stellt die Studie fest, dass Zyklen selbst in nach dem heutigen Stand der Softwareentwicklungstechnik entworfenen Softwaresystemen der *Regelfall* sind, nicht die Ausnahme.

Die vorherrschenden Techniken zur Zyklusfeststellung belaufen sich auf die nachfolgenden 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 Quelltextansicht 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.

JDdepend Das Werkzeug *JDdepend*³ berechnet Metriken und Beziehungen eines Softwaresystems auf Paketebene und präsentiert diese dem Benutzer zurück. Die Metriken belaufen sich nach [20] auf afferente/effereente Kopplungen, Instabilität, Abstraktheit, 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.

¹<http://byecycle.sourceforge.net/> (17. Aug. 2006)

²<http://classycle.sourceforge.net/> (18. Aug. 2006)

³<http://www.clarkware.com/software/JDdepend.html> (17. Aug. 2006)

Jepends Das Werkzeug *Jepends* [26] dient der Auffindung von Zyklen, die es als kommagetrennte Textdatei für die Weiterverarbeitung in einer Tabellenkalkulation verwendet. Die zugrundeliegende Methode der Zyklenerkennung baut auf einer Heuristik, die im Programmabhängigkeitsdiagramm einfache Zyklen erkennt. Da das Problem der vollständigen Zyklenerkennung NP-schwierig ist, werden nur eine gewisse Mindestanzahl Zyklen erkannt, an denen ein Artefakt beteiligt ist. Dies genügt auf jeden Fall für das Auffinden von Zyklen.

JooJ Das Werkzeug *JooJ* [27] 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 solche 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 Kantenrückkopplungsmenge*, also der Minimalanzahl Kanten, die von einem Graphen zu entfernen sind, damit ein gerichteter azyklischer Graph entsteht. Die durch die minimale Rückkopplungsmenge gelieferten Kanten markiert das Werkzeug dann als zyklensbildend im Quelltext.

Lattix LDM Das Werkzeug *Lattix LDM* [38] visualisiert die Abhängigkeiten zwischen Artefakten mittels sogenannter Abhängigkeitsstrukturmatrix. Hierbei handelt es sich um eine Matrix, in der die Spalten von den Zeilen abhängen. Ein zyklensfreies System enthält keine Beziehungen oberhalb der Nebendiagonale, zyklische Abhängigkeiten sind daher einfach auszumachen.

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

PASTA Das Werkzeug *PASTA* [15] ermöglicht die Visualisierung von Beziehungen auf Java-Paketebene nach Schichten und die automatische Aufteilung stark verzykelter Pakete (die sich per definitionem in einer Schicht befinden) auf mehrere Schichten nach einer Heuristik, genannt *intelligente Schichtung*.

Das Verfahren der intelligenten Schichtung beruht auf dem Kantengewicht, also der Anzahl der elementaren Verbindungen, die eine Kante von einem Paket zu einem anderen repräsentiert, welches den Aufwand widerspiegelt, um diese Verbindung aufzulösen. Das Verfahren sucht eine Menge Kanten, deren Gesamtaufwand zu deren Auflösung es als minimal eingeschätzt, blendet diese Kanten aus und weist die Knoten des nun azyklischen Graphen zur zugehörigen Schicht zu.

Den aufgeführten Werkzeugen ist gemein, dass sie zwar Zyklen zu finden, teils zu verhindern (JooJ) und teils selbständig aufzulösen (PASTA), nicht aber das unterliegende Softwaresystem entsprechend umzubauen vermögen.

3.2 Weitere Erkenntnisse zu zyklischen Abhängigkeiten

Der Artikel »Granularität« [22] beschreibt die Nachteile hinsichtlich Wart- und Testbarkeit durch verzykelte Softwaresysteme. Darauf aufbauend definiert er das sogenannte *azyklische Abhängigkeitsprinzip*, welches besagt, dass die Abhängigkeitsstruktur zwischen Paketen ein gerichteter azyklischer Graph zu sein habe.

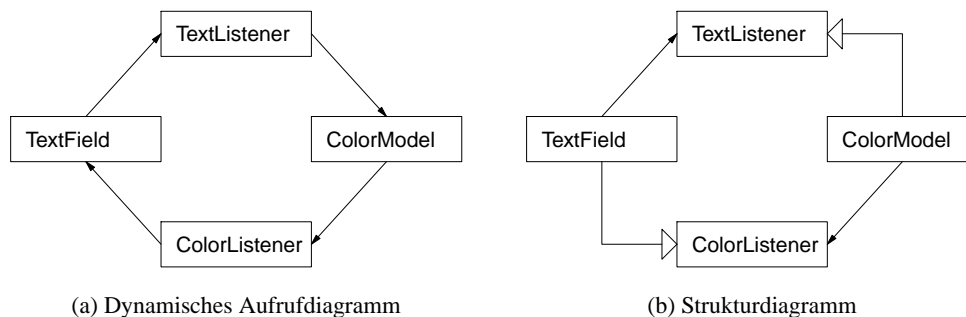


Abbildung 1: Dynamische Zyklen ohne direkte Repräsentation in statischer Struktur
Quelle von (a): [13]

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

In [35] erfolgt die Vermeidung von Zyklen durch die Einführung einer »Benutzt«-Beziehung zwischen Unterprogrammen und deren Einordnung in eine Hierarchie, in der die Tiefe eines jeden Unterprogramms der minimalen Entfernung zu einem kein anderes Unterprogramm benutzenden Unterprogramm entspricht. Auf deutsch: Benötigt B A, A aber nichts, so befindet sich A auf Ebene 0, B auf Ebene 1.

Die konsequente Einordnung der Unterprogramme in eine Hierarchie der »Benutzt«-Beziehungen gewährleistet ein zyklensystem.

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 [13] die unbeabsichtigte Einführung eines Laufzeitzyklus anhand unvorsichtiger Anwendung des Beobachtermusters [9, S. 293] in einen Java-Texteditor.

Abbildung 1(a) zeigt einen Aufrufzyklus über vier Java-Klassen, ausgelöst durch ein Änderungereignis aus `java.awt.TextField.setText`, selbst wenn der Inhalt nicht geändert wurde. Die Struktur des Systems weist hingegen keinerlei statische Zyklen auf (siehe Abb. 1(b)).

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

3.3 Erkennungstechniken

Um zyklische Abhängigkeiten entfernen zu können, müssen sie erst einmal gefunden werden. Da wir die automatische 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 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 sogenannte *Zyklengruppen*, auch als *starke Komponenten* [3, S. 256] bekannt. Eine Zyklengruppe ist ein Teil eines gerichteten Graphen, in dem zwischen jedem Knotenpaar (x, y) ein Weg von x nach y führt und ein Weg von y nach x .

Abbildung 2 veranschaulicht einen Graphen, dessen Zyklengruppen hervorgehoben sind.

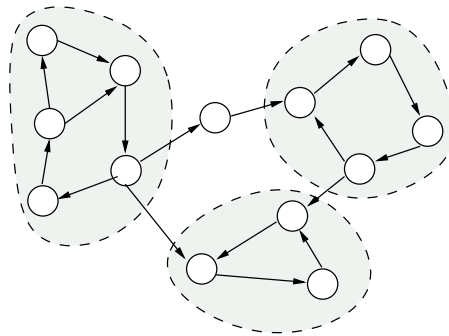


Abbildung 2: Graph mit hervorgehobenen Zyklengruppen

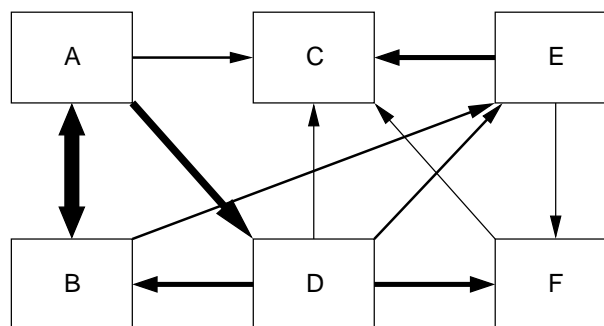


Abbildung 3: Artefakte hoher Ebene mit teilweise zyklischen Abhängigkeiten.

Zur Erkennung von Zyklengruppen existieren mehrere Algorithmen. Der *inkrementelle Zyklengruppenerkennungsalgorithmus* [43] liefert aus einem Graphen unter Verwendung binärer Entscheidungsdiagramme, kombiniert mit Erreichbarkeitsanalyse alle Zyklengruppen. Gemäß [43] 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.

Der *lineare symbolische Schrittalgorithmus* [11] berechnet die Zyklengruppen eines durch ein geordnetes binäres Entscheidungsdiagramm repräsentierten Graphen. Die Verwendung von Entscheidungsdiagrammen im Gegensatz zu herkömmlicher expliziter Graphrepräsentationen wie Adjazenzmatrizen und -listen liegt in der Platzersparnis bei riesigen Graphen. Der Algorithmus liefert zu einem Graphen in einer linearen Anzahl von symbolischen Schritten alle Zyklengruppen.

Erkennung von Zyklen auf Visualisierungsebene bieten viele Werkzeuge an. Ihnen allen ist die Darstellung des Softwaresystems über Verbindungsdiagramme gemein, sodass Zyklen insbesondere bei Betrachtung von Artefakten hoher Ebene sofort ins Auge stechen. Abbildung 3 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* [39], *Moose* [32], *ArchView* [37], *Rigi* [42] und so weiter, um nur einige zu nennen.

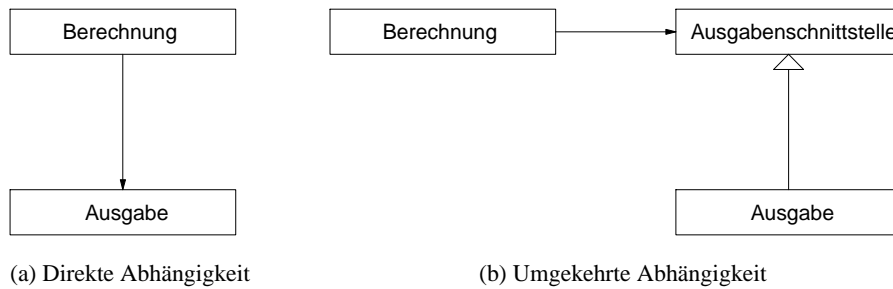


Abbildung 4: Beispiel für Abhängigkeitsumkehr

3.4 Auflösungstechniken

Da wir nun das Rüstzeug zur Auffindung von Zyklen besitzen, beschäftigen wir uns nun mit Verfahren und Techniken zur Auflösung derselben.

Wie bereits im letzten Abschnitt angedeutet, lässt sich die statische Struktur eines Softwaresystems in einem gerichteten Graphen darstellen.

Die einfachste Möglichkeit, einen Zyklus zu brechen, besteht in der Umkehrung einer Kante in einem den Zyklus konstituierenden Kreis derart, dass der Kreis dadurch aufhört, ein Kreis zu sein. Bei Softwaresystemen dürfen allerdings nicht beliebige Kanten umgedreht werden, da sich unbedachte Änderungen sofort auf das Verhalten und die Funktionsfähigkeit des Programms zur Laufzeit auswirken (man stelle sich bloß vor, einen Funktionsaufruf $f \rightarrow g$ in $g \rightarrow f$ zu ändern).

Eine einfache Lösung bringt das *Abhängigkeitsumkehrprinzip* [21]. Dieses Prinzip besagt, kein höherschichtiges Modul dürfe von einem niederschichtigen Modul, sondern beide sollen von Abstraktionen abhängen.

Weiters dürfe keine Abstraktion von der Implementierung, aber jede Implementierung solle von der Abstraktion abhängen.

Nehmen wir als Beispiel das in Abbildung 4(a) gezeigte Softwaresystem, welches aus den Klassen *Berechnung* und *Ausgabe* besteht und *Berechnung* auf Methoden von *Ausgabe* zugreift, also von *Ausgabe* abhängig ist.

Durch die Einführung einer Schnittstellenklasse *Ausgabenschnittstelle*, der Umleitung der Zugriffe seitens *Berechnung* auf diese und deren Implementierung durch die Klasse *Ausgabe* wird die Beziehung zwischen *Berechnung* und *Ausgabe* strukturell umgekehrt, wie aus Abbildung 4(b) ersichtlich. Am Laufzeitverhalten des Programms ändert sich dadurch nichts.

Das *Abhängigkeitsumkehrprinzip* ist natürlich nicht auf Klassen beschränkt. Durch Aggregation vieler solcher Abhängigkeitsumkehroperationen lassen sich zum Beispiel Verbindungen zwischen ganzen Paketen oder noch höherschichtigeren Artefakten umkehren.

Die Methode der *gerichteten Knotenspaltung* [1] 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 5(a) zeigt einen kleinen Zyklus zwischen zwei Knoten A und B. In Abbildung 5(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.



Abbildung 5: Beispiel für gerichtete Knotenspaltung

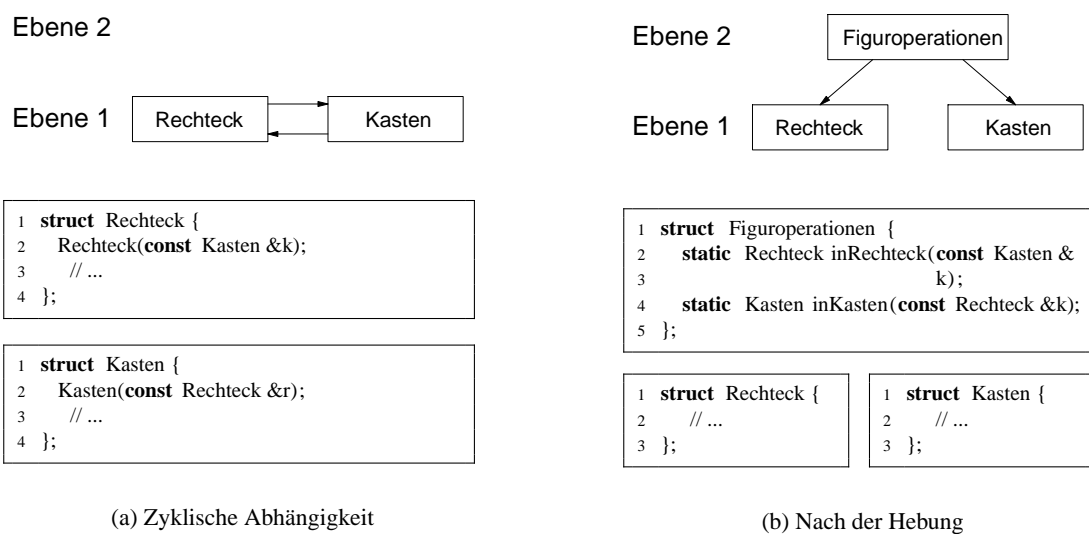


Abbildung 6: Beispiel für eine Hebung

Auf Artefakte von Softwaresystemen umgemünzt ergibt dies ein Zyklenauflösungspotential auf mehreren Ebenen mit verschiedenen Schwierigkeitsgraden. Während die Spaltung auf Paketebene unter Umständen bereits durch Aufteilung der im Paket enthaltenen Klassen auf die gesparteten Pakete ohne großen Zusatzaufwand erledigt werden kann, ist zum Beispiel bei der Spaltung auf Klassenebene auf jeden Fall beträchtlicher Umbauaufwand des Quelltextes erforderlich.

Für große C++-Softwaresysteme stellt [18] eine Reihe von Techniken zur Auflösung zyklischer Abhängigkeiten vor, die – obwohl C++-spezifisch – sich weitgehend programmiersprachenunabhängig auf beliebige Softwaresysteme anwenden lassen. Wir sehen uns einige Techniken im einzelnen an.

Die *Hebung* [18, 5.2] stellt das Herausheben jener Unterartefakte von zwei verzykelten Artefakten, die die zyklische Abhängigkeit konstituieren, auf eine höhere Ebene dar. Abbildung 6(a) zeigt zwei zyklisch voneinander abhängige Klassen nebst Quelltexten. In Abbildung 6(b) wurde eine zusätzliche Klasse *Figuoperationen* eingeführt, die die zyklenverursachenden Methoden aus den beiden anderen Klassen heraushebt und somit den Zyklus auflöst.

Die Klasse *Figuoperationen* befindet sich auf einer höheren Ebene, da sie von den beiden

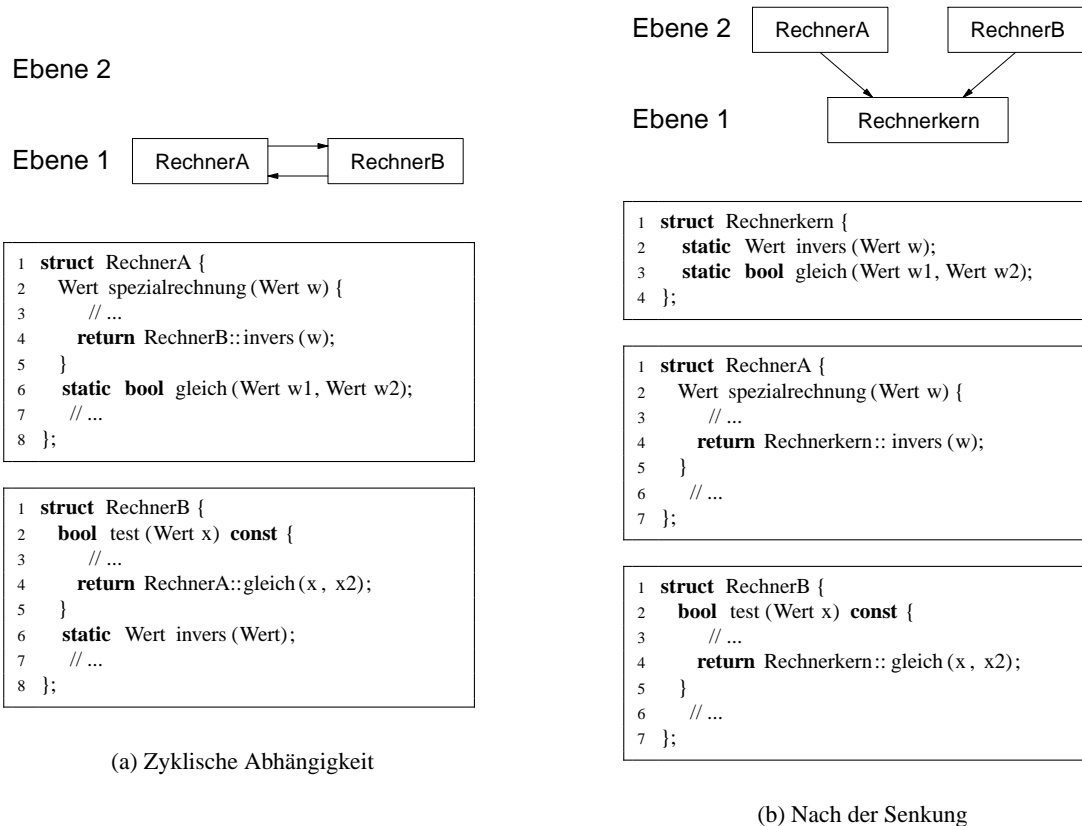


Abbildung 7: Beispiel für eine Senkung

unterliegenden Klassen abhängig ist, jedoch die unterliegenden Klassen nicht von ihr.

Des Weiteren führt [18, 5.2, S. 218] die sogenannte *Dominanz* von Artefakten auf. Ein Artefakt A dominiert ein Artefakt B, wenn A von B abhängt. Darauf aufbauend gilt folgende Regel: Eine zusätzliche Beziehung von einem Artefakt B zu einem Artefakt A erzeugt dann und nur dann keinen Zyklus, wenn A nicht B dominiert.

Die *Senkung* [18, 5.3] – das Gegenstück zur oben erwähnten Hebung – löst zyklische Abhängigkeiten durch Herausheben der zyklusverursachenden Unterartefakte zwischen höherschichtigen Artefakten und überführt sie in ein neues niederschichtiges Artefakt. Abbildung 7(a) zeigt zwei verzykelte Klassen. In Abbildung 7(b) wird eine dritte Klasse *Rechnerkern* eingeführt, welche die ehemals zyklusverursachenden Beziehungen heraushebt.

Das Buch [18] stellt noch sieben weitere Techniken zur Zyklenauflösung vor. Diese sind verdeckende Zeiger, einfache Daten, Redundanz, Rückrufe, Verwalterklassen, Herausheben und Kapselungseskalation und sollen ebenfalls als Mittel zum Zweck für die Zyklenauflösung herangezogen werden.

Einen unkonventionellen aber praxisorientierten Weg zur Zyklenauflösung verfolgt die Strategie zur Implementierung [14] von Entwurfsmustern [9] durch Aspekte [17]. Die Auflösung von Zyklen kommt vor allem durch Abhängigkeitsumkehr analog zu [21] zustande. Durch die Zusammenfassung systemweiter Belange in eigene Aspekte verschwinden die Beziehungen auf die konkreten Klassen.

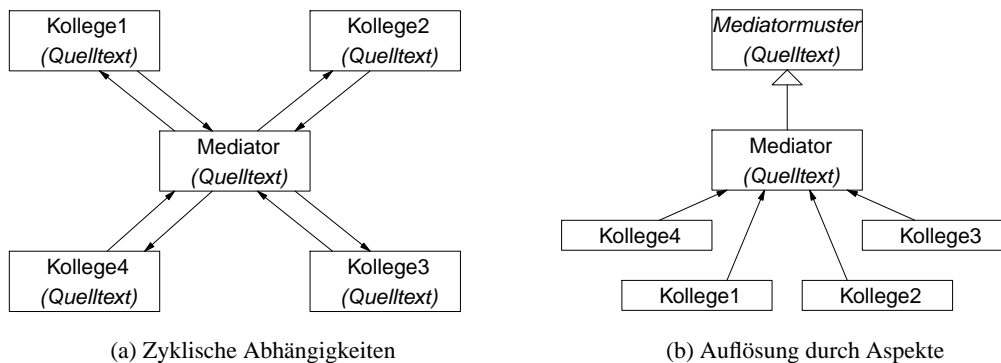


Abbildung 8: Auflösung durch Aspekte anhand des Mediatormusters
(Quelle: [14]).

Dadurch verursachte Zyklen werden somit aufgelöst.

Ein Beispiel für die Zyklenauflösung durch Aspekte ist der Mediator [9, S. 273]. Hier fasst der Aspekt die Implementierung des Mediators an einer Stelle zusammen und befreit die konkreten Klassen von ihrer Abhängigkeit zur Mediator-Klasse (siehe Abbildung 8).

Der Ausflug in die aspektorientierte Programmierung sei der Vollständigkeit halber erwähnt, da diese bis heute nicht die vorherrschende objektorientierte Programmierung verdrängen konnte. Erstens gilt die aspektorientierte Programmierung nach wie vor noch nicht als einsatzreif [41], 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.

Die Möglichkeiten zur Auflösung einzelner 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 Kantenrückkopplungsmenge. Eine *minimale Kantenrückkopplungsmenge* 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 [27, 3.2] 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 funktional eine Klasse im Zyklus 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.

Allerdings lassen sich nicht alle Klassen gleich einfach durch Scheinklassen ersetzen. Jedoch existiert ein Verfahren [33] zur Auflösung der Zyklen unter Berücksichtigung der Schwierigkeit der Scheinklassenerstellung.

Hierbei betrachtet das Verfahren die Klassenbeziehungen als Graph und versieht zunächst die Kanten mit Gewichten, die die Kosten der Ersetzung ihrer jeweiligen Ausgangsklassen durch Schein-

klassen widerspiegeln. Dann durchläuft ein Algorithmus sämtliche Zyklengruppen und löst die erhaltenen 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 Aggregation, indem es diesen ein besonders hohes Gewicht zuweist.

3.5 Umbau

Unter Umbau versteht sich die Umstrukturierung von Quelltextartefakten zur Verbesserung des Programmverständnisses oder der Wartbarkeit, ohne das nach außen beobachtbare Verhalten eines Programms zu verändern [8, Kap. 2]. Beispiele für Umbauoperationen sind das Umbenennen von Klassen/Funktionen, das Herausheben von Funktionen aus anderen Funktionen, das Aufspalten von Klassen/Funktionen und das Verschieben von Funktionen in andere Klassen/Pakete.

Zur ordnungsgemäßen Durchführung wird sich die automatische Zyklenauflösung zumindest einiger der mannigfaltigen Umbauoperationen [8] bedienen müssen. Wir stellen daher einige Erkenntnisse zum Stand der Technik des Umbaus dar.

Die grundlegenden Verfahrensweisen zum Umbau definiert [34]. Dieses Werk beschreibt 26 atomare Umbauoperationen, die sich verhaltenskonservierend auf objektorientierten Quelltext anwenden lassen und deren Verhaltenskonservierung erwiesen ist, sowie drei höherschichtige Umbauoperationen. Keine dieser Operationen nimmt Rücksicht auf Quell- oder Binärkompatibilität [34, S. 29].

[8] definiert 74 Umbauoperationen, die sich teilweise mit den in [34] genannten überschneiden. Die Beschreibungen der Umbauoperationen erfolgt unter Berücksichtigung der Verständlichkeit und verzichtet auf Formalismen. Als vorläufig interessante Umbauoperationen für die Zyklenauflösung erscheinen »bidirektionale Beziehung in unidirektionale umwandeln« [8, S. 200], »Methode verschieben« [8, S. 142], »Klasse extrahieren« [8, S. 149] sowie in Hinblick auf Erhaltung der Kompatibilität »fremde Methode einführen« und »lokale Erweiterung einführen« [8, S. 162, S. 164].

Werkzeuge zum Umbau stellen zum Beispiel *Jrbx* [23] auf Java-Basis dar, ein flexibles, erweiterbares Umbauwerkzeug, sowie *Guru* [31], 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 [40, 10] beim Umbau Bedacht.

Zum momentanen Zeitpunkt ist lediglich die Verwendung von Umbauoperationen für die Zyklenauflösung gewiss. Ausmaß und Umfang der notwendigen Umbauoperationen werden erst im Zuge der tatsächlichen Forschung zur automatischen Zyklenauflösung zu Tage treten.

3.6 Kompatibilität

Kompatibilität ist laut ANSI die Fähigkeit einer funktionalen Einheit, die Anforderungen einer spezifizierten Schnittstelle zu erfüllen⁴.

Wir sprechen hier konkret von der *Schnittstellenkompatibilität* oder kurz Kompatibilität von Softwarekomponenten in der Art und Weise, wie verschiedene Versionen von Komponenten untereinander zusammenarbeiten.

⁴http://www.fda.gov/ora/Inspect_ref/igs/gloss.html (13. Aug. 2006)

Die *Schnittstellenkompatibilität* spielt in der professionellen Komponentenentwicklung eine besondere Rolle. Sie stellt sicher, dass eine Komponente B, die ursprünglich auf Komponente A aufbaute, auch mit einer neuen Version A' ohne Änderungen an B zusammenarbeitet.

Der heutige Stand der Softwareentwicklung sieht das Verfassen eines Programms in Ausdrücken einer höheren Programmiersprache vor (Quelltext) und bedient sich eines Übersetzers zur Umwandlung in niederschichtige Repräsentation wie Bytecode oder Maschinensprache.

Daher sind zwei Arten der Schnittstellenkompatibilität zu unterscheiden: die *Quellkompatibilität* und die *Binärkompatibilität*.

3.7 Quellkompatibilität

Die *Quellkompatibilität* ist ein Spezialfall der Schnittstellenkompatibilität und gilt dann, wenn eine neue Version einer Komponente keine Änderungen am Quelltext der davon abhängigen Komponenten verursacht.

Eine Komponente B' ist also quellkompatibel zu einer Komponente B, wenn sie B ersetzen kann und kein Quelltext einer von B abhängigen Komponente an B' angepasst zu werden braucht. Eine Neuübersetzung der abhängigen Komponenten ist in der Regel notwendig.

Quellkompatibilität wird in der Regel nur für Programmierschnittstellen gefordert. Das bedeutet, dass keine Schnittstellenänderung eine Änderung an die Schnittstellen benutzenden Komponenten nach sich ziehen darf.

Die Wissenschaftsliteratur scheint sich über die explizite Behandlung von Quellkompatibilität zurückzuhalten. In [36] wird Quellkompatibilität lediglich indirekt von der Binärkompatibilität getrennt, als dass die erlaubten Änderungen an Artefakten für ein Beispiel für jede Kompatibilitätsart aufgeführt sind (zum Beispiel dürfen private Instanzvariablen einer Klasse beliebig geändert werden, ohne die Quellkompatibilität zu verletzen).

Die praxisorientierte Softwareentwicklung bringt mehr Quellen zur Quellkompatibilität hervor. KDE beschreibt Quellkompatibilität als die durch Herausgabe einer neuen Bibliothek entstehende Notwendigkeit einer Neuübersetzung eines Programms ohne Änderung seines Quelltexts⁵, während Apache APR Quellkompatibilität als die Möglichkeit der fehlerfreien Übersetzung der Anwendung ohne semantische Änderungen⁶ ansieht.

3.8 Binärkompatibilität

Die *Binärkompatibilität* definiert die erlaubten Änderungen zwischen zwei Versionen einer Komponente, sodass die Binärrepräsentationen abhängiger Komponenten ohne Notwendigkeit der Neuübersetzung auf diese Komponente zugreifen können.

Eine Komponente B' ist also *binärkompatibel* zu einer Komponente B, wenn sie alle Schnittstellen von B enthält und keinerlei Änderungen an der Binärrepräsentation sowie am Verhalten dieser Schnittstellen gegenüber B vornimmt. Alle von B abhängigen Komponenten arbeiten daher ohne Neuübersetzung auch mit B' zusammen.

Die obige Definition ist strenger als die in der Java-Sprachspezifikation [12, § 13.2] angegebene (eine Änderung eines Typs ist binärkompatibel zu existierenden Binärrepräsentationen, wenn diese zuvor ohne Fehler gebunden werden konnten und weiterhin ohne Fehler zu binden sind), da Sprachen wie zum Beispiel C++ die Binärrepräsentation von Instanzvariablen nicht als Bindeinformation ablegen, sondern transparent als Versätze in der Maschinensprache. Aus Sicht des Binders hat sich

⁵<http://developer.kde.org/documentation/other/binarycompatibility.html> (14. Aug. 2006)

⁶<http://apr.apache.org/versioning.html#source> (14. Aug. 2006)

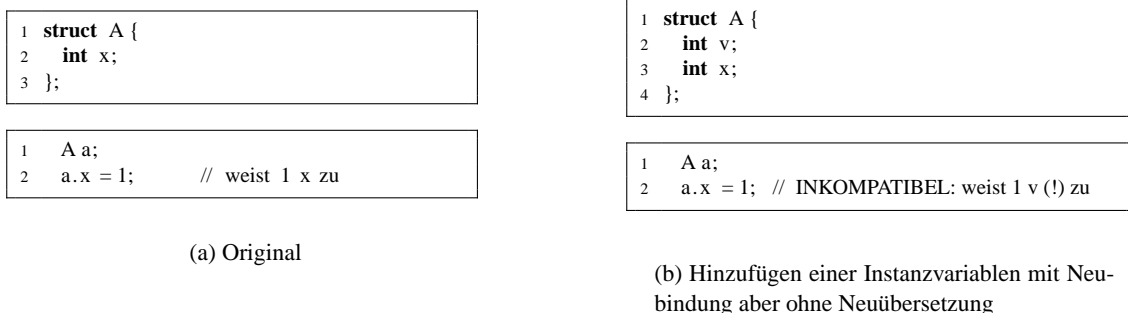


Abbildung 9: Binärinkompatible Änderung in C++, die keinen Bindefehler hervorruft

nichts geändert, die Ausführung schlägt dennoch fehl. Abbildung 9 illustriert einen solchen Fall. Unter »Bindung« versteht sich in diesem Fall eine dynamische Bindung: `struct A` ist in einer Bibliothek enthalten, welche erst zur Laufzeit des Hauptprogramm hinzugeladen wird. Die geänderte Variante wurde neu übersetzt und dynamisch gebunden, das Hauptprogramm blieb unverändert. Genau dann tritt der beschriebene Fehler auf.

Um noch einmal kurz zur Java-Binärkompatibilität zurückzukehren: Die oben erwähnte Definition der Java-Binärkompatibilität weist Schwächen auf [6] – eine konforme Auslegung der Binärkompatibilität nach den Buchstaben der Spezifikation [12, § 13] erlaubt dennoch die Erzeugung von Bytecode, der weder übersetzt noch bindet. Zum Beispiel führt die Addition einer Schnittstellenmethode (binär kompatibel [12, § 13.5.3]) zu einem Bindefehler, da die implementierende Klasse die neue Schnittstellenmethode nicht implementiert haben kann. Die Java-Sprachspezifikation spezifiziert lediglich syntaktische Kompatibilität, keine semantische [28]. Aus diesem Grund genügt die Java-Definition von Binärkompatibilität unseren Anforderungen nicht.

Varianten zum Erhalt der Binärkompatibilität gibt es viele. Eine auf Metaklassen basierende Variante ist das *Systemobjektmodell* (SOM) [7]. Dieses setzt sich aus einer Reihe von Vorschriften zusammen, die weitgehende binäre Kompatibilität bei größtmöglicher Flexibilität auf. Klassenhierarchien werden zur Ladezeit komponiert, sodass Inkonsistenzen von vornherein verhindert werden. Da die SOM-Technik allerdings keine weitere Verbreitung erfahren hat, werden wir uns nicht näher damit beschäftigen.

Die Java-Binärkompatibilität wird in der Java-Sprachspezifikation [12] explizit beschrieben und wurde bereits oben behandelt.

Die C++-Spezifikation [2] lässt Aussagen zur Binärkompatibilität in C++ missen. Stattdessen gibt es Anleitungen aus der Praxis, die einfache Regeln zur Erhaltung der Binärkompatibilität definieren wie beispielsweise diejenigen von KDE⁵.

3.9 Vergleich von Quell- und Binärkompatibilität

Die Binärkompatibilität gilt gemeinhin strenger als die Quellkompatibilität, erlaubt sie doch weniger Änderungen am Quelltext, um ihre Kompatibilitätsansprüche weiterhin einzuhalten. Daraus zu folgern, die Binärkompatibilität wäre eine Untermenge der Quellkompatibilität, lässt sich jedoch einfach durch folgendes Beispiel widerlegen.

In Abbildung 10 ist die Klasse B vor und nach ihrer Änderung zu sehen. Die Änderung ist binärkompatibel, da keine sich auf die binäre Repräsentation der Programmdatei auswirkenden Änderun-

```

1 enum Opt { B_Lesen, B_Schreiben };
2 // ...
3 class B {
4     // ...
5 };

```

(a) Ursprüngliche Version von B

```

1 class B {
2     // ...
3     enum Opt { Lesen, Schreiben };
4     // ...
5 };

```

(b) Neue Version, binärkompatibel, aber nicht quellkompatibel

Abbildung 10: Binärkompatible Quelltextänderung in C++

gen (die alten Enumerationskonstantenwerte bleiben schließlich unverändert) vorgenommen wurden. Der Quelltext wurde allerdings verändert, weswegen alle diese Klasse verwendenden Komponenten neu zu übersetzen sind.

In Java gilt sogar, dass die Anforderungen für die Quellkompatibilität in der Tat strenger sind als die der Binärkompatibilität, also Quellkompatibilität eine Untermenge der Binärkompatibilität darstellt [5]. Für C++ gilt dies nicht, da zum Beispiel durch das Hinzufügen eines Funktionsparameters mit Standardwert (nicht binärkompatibel) die Quellkompatibilität nicht beeinflusst wird.

3.10 Zusammenfassung

Dieses Kapitel behandelte Verfahren zur Zyklenerkennung und Zyklenauflösung sowie die heute standesgemäßen Definitionen von Binärkompatibilität und Quellkompatibilität. Der Fokus bei der Kompatibilität lag auf den Programmiersprachen C++ und Java, ohne jedoch einem Anspruch auf Allgemeinheit Abbruch zu tun.

4 Ziele und Eigenleistung

Die vorzunehmende Forschungsarbeit widmet sich dem Ziel der *automatischen Auflösung von zyklischen Abhängigkeiten unter Berücksichtigung von Quell- und Binärkompatibilität*. Als Resultat soll ein Verfahren entstehen, welches in imperativen objektorientierten Programmiersprachen verfasste Softwaresysteme automatisiert so transformiert, dass möglichst alle zyklischen Abhängigkeiten – soweit durch Berücksichtigung der Kompatibilität und eventueller anderer Beschränkungen erlaubt – aufgelöst werden.

4.1 Ziele

Im Einzelnen soll das zu entwickelnde Verfahren folgende Punkte erfüllen.

- Automatische Programmtransformation (im Gegensatz zu manueller, werkzeuggestützter oder überhaupt keiner)
- Vollständige Erhaltung der Semantik (das transformierte Programm legt dasselbe Verhalten an den Tag wie das ursprüngliche)
- Erhaltung der Quellkompatibilität im Hinblick auf öffentliche Schnittstellen (wie können Zyklen aufgelöst werden, ohne die Quellkompatibilität öffentlicher Schnittstellen zu kompromittieren)

- Erhaltung der Binärkompatibilität im Hinblick auf öffentliche Schnittstellen (wie können Zyklen aufgelöst werden, ohne die Binärkompatibilität öffentlicher Schnittstellen zu kompromittieren)
- Genaue Protokollierung der vorgenommenen Umbauaktionen (welche Zyklen wurden gefunden und wie wurden sie aufgelöst)

4.2 Eigenleistung

Aus Kapitel 3 geht die Existenz einer großen Anzahl von Zyklenerkennungs- und -auflösungsverfahren hervor. Das technische Fundament zum Erreichen des Ziels ist damit gegeben.

Betrachten wir nun die aufgeführten existierenden Lösungen, so stellen wir fest, dass zwar eine Reihe an Verfahren und Werkzeugen zur Entdeckung von Zyklen existieren ([39, 26, 27, 15, 38], ByeCycle¹, JDepend³, Classycle²), die entweder ausschließlich Zyklen aufzeigen oder sich auf das Vorschlagen von möglichen Umbauaktionen beschränken, den Umbau selbst jedoch nicht durchführen.

Salopp ausgedrückt: Keines der Werkzeuge enthält eine Schaltfläche mit der Aufschrift »Zyklen auflösen«.

Die Eigenleistung dieser Forschungsarbeit liegt daher in der automatischen Auflösung von zyklischen Abhängigkeiten über ein gesamtes Softwaresystem, welches in einer imperativen, objektorientierten Programmiersprache verfasst ist. Neben der Auflösung nimmt auch die *nachvollziehbare Protokollierung* der Änderungen einen nicht vernachlässigbaren Stellenwert ein, sodass dem Benutzer jederzeit Informationen über Art und Umfang der Änderungen zur Verfügung stehen.

Dabei spielt die Berücksichtigung von Quell- und Binärkompatibilität im Zuge der Zyklenauflösung eine essentielle Rolle, um die Zyklenauflösung für reale Softwaresysteme brauchbar zu gestalten. Soweit bekannt existieren keine Präzedenzfälle für die kombinierte Betrachtung der Zyklenauflösung und der Schnittstellenkompatibilität.

4.3 Offene Fragen

Während der intensiven Beschäftigung mit dem Thema traten einige Fragen zu Tage, deren Beantwortung für die Forschungsarbeit notwendig erscheint und zusätzliche Beiträge zum Stand der Technik leistet.

Die Fragen sind im nachfolgenden angeführt und kurz erläutert.

1. *Wie adäquat vermögen verschiedene Algorithmen Zyklen aufzulösen?* Als wesentlicher zu behandelnder Punkt steht die Frage nach der Qualität des Ergebnisses der Zyklenauflösung zur Debatte. Da verschiedene zyklische Abhängigkeiten unter Verwendung verschiedener Algorithmen verschieden aufgelöst werden können, ist festzustellen, welcher Algorithmus unter welchen Bedingungen das brauchbarere Resultat liefert (wobei »brauchbar« bedeutet, dass die gefundene Lösung einem händischen Umbau durch einen Entwickler weitgehend entspricht).
2. *Wie lassen sich in einem Quelltext automatisch Schnittstellen finden?* Diese Frage mag auf den ersten Blick unwesentlich erscheinen. Jedoch sei an die zweite Eigenleistung erinnert, welche die Berücksichtigung von Quell- und Binärkompatibilität in das Thema einbringt.

Die öffentlichen Schnittstellen sind nämlich von der Zyklenauflösung so zu behandeln, dass sie im transformierten Softwaresystem auf jeden Fall erhalten bleiben. Andernfalls ginge die Kompatibilität verloren.

Zur Beantwortung dieser Frage ist ein Verfahren zu entwickeln, das aus vorliegendem Quelltext automatisch öffentliche Schnittstellen zu erkennen vermag. Die getroffenen Annahmen sollten nach Möglichkeit konservativ sein, das heißt keine echte Schnittstelle soll fälschlicherweise nicht als solche identifiziert werden.

3. *Wie lassen sich »gute« zyklische Abhängigkeiten automatisch feststellen?* Nicht jede zyklische Abhängigkeit in einem Softwaresystem ist automatisch eine »schlechte« Abhängigkeit. Es gibt Fälle von Zyklen, die durchaus gewollt sind, wie zum Beispiel die gegenseitige Abhängigkeit von `java.lang.Method` und `java.lang.Class`. [26] weist auf diese Problematik hin, gibt aber keine Lösung zu deren automatischer Bestimmung. Lediglich die Vermutung, dass große Zyklen tendenziell unerwünschter sind als kleine, lässt sich aus [26] ableiten.

Gesucht ist daher ein Verfahren oder eine Methode, die für einen konkreten Zyklus zwischen Artefakten eines Programms festlegt, wie »gut« (im Sinne von »gewollt«) der Zyklus ist. Sinnvollerweise liefert ein solches Verfahren eine Wahrscheinlichkeit je Zyklus, ab dessen Überschreitung eines noch zu definierenden Schwellwertes ein Zyklus als »gut« zu betrachten und daher von der Auflösung verschont bleibt.

Können brauchbare Metriken gefunden werden? Eine Möglichkeit, der Frage nach »guten« zyklischen Abhängigkeiten nachzugehen liegt in der Berechnung entsprechender Metriken. Wie diese Metriken auszusehen haben, wie sie zu ermitteln sind und ob sich die Frage überhaupt mit Metriken zufriedenstellend beantworten lässt, soll Gegenstand weiterer Ermittlungen sein.

Da [25, Kap. 9] selbst die Erforschung »guter« Zyklen erwägt, besteht die Möglichkeit, dass sich die Frage ohne eigenen Forschungsaufwand beantworten lässt (siehe dazu auch die Anmerkungen auf S. 21).

5 Übersicht über den Ansatz

Das Erreichen der Ziele in Kapitel 4 verlangt nach einem Ansatz, der dies in endlicher Zeit (der Studienplan des technischen Doktorats sieht 4 Semester vor, alles darüber ist unendlich) gewährleistet.

In diesem Kapitel legen wir daher die Vorgehensweise sowie Art und Umfang der zu untersuchenden Sachverhalte fest, soweit dies zu diesem Zeitpunkt möglich und vorhersehbar ist.

5.1 Spezifikation

Hierin befindet sich eine notwendige Spezifikation der zu erbringenden Leistungen und der Begleitbedingungen. Die einzelnen Elemente sind nach absteigender Reihenfolge ihrer Wichtigkeit geordnet.

- § 1 Die Auflösung der zyklischen Abhängigkeiten (kurz: Verfahren) basiert auf einer abstrakten syntaktischen Gesamtrepräsentation (kurz: Repräsentation) eines Softwaresystems.
- § 2 Das Verfahren wird in einem prototypischen Werkzeug (kurz: Prototyp) implementiert.
- § 3 Das Verfahren unterstützt potentiell jede imperative objektorientierte Programmiersprache.
- § 4 Die Repräsentation liegt in einem solchen Format vor, das sämtliche Namensräume, Klassen, Funktionen und Methoden sowie die abstrakte Syntax der innerhalb von Funktionen enthaltenen Befehle widerspiegelt.

- § 5 Die Repräsentation liegt in einem solchen Format vor, das die Transformation der Repräsentation bis auf Befehlsebene herab ermöglicht.
 - § 6 Das Verfahren berücksichtigt die Binärkompatibilität.
 - § 7 Das Verfahren berücksichtigt die Quellkompatibilität.
 - § 8 Der Prototyp ist in der Programmiersprache Java zu verfassen.
 - § 9 Der Prototyp unterstützt die Sprache C++.
 - § 10 Der Prototyp ist von keinen Benutzereingaben abhängig.
 - § 11 Der Prototyp akzeptiert Kommandozeilenparameter zu seiner Konfiguration. Über diese sind auch eventuelle Betriebsmodi zu steuern.
 - § 12 Die Repräsentation kann direkt aus C++-Quelltext erzeugt werden.
 - § 13 Der Prototyp unterstützt die Sprache Java.
 - § 14 Die Repräsentation enthält Informationen über das Quelltextbild (Kommentare, Leerzeilen, Einrückungen).
 - § 15 Die Transformation erhält das Quelltextbild (Kommentare, Leerzeilen, Einrückungen) weitgehend.
- Zuletzt werden alle Teile aufgeführt, deren Untersuchung explizit nicht erfolgt, wenn die Gefahr besteht, dass deren Untersuchung sich aus den obigen Punkten implizit ergäbe.
- § 16 Dynamische Abhängigkeiten wie zum Beispiel Laufzeitzyklen stellen keinen Gegenstand der Forschungsbemühungen dar.
 - § 17 Das Verfahren impliziert keine grafische Darstellung jedweder Ergebnisse.
 - § 18 Der Prototyp impliziert keine grafische Darstellung jedweder Ergebnisse.

5.2 Ansatz

Die grundsätzliche Herangehensweise besteht zunächst in einer Evaluation geeigneter Repräsentationen. Zur Auswahl stehen FAMIX [4] (bzw. EFAMIX [37]), GXL [16], MOF/XMI [30] oder werkzeuggestützte Formate wie das des neuen ASB-Analysator von KDevelop4⁷ oder der Syntaxrepräsentation von Eclipse.

Sollten sich die existierenden Repräsentationen als ungeeignet erweisen, so ist die Entwicklung einer eigenen Repräsentation anzudenken.

Als nächster Schritt erfolgt die Suche nach einem geeigneten C++-Analysator, der in der Lage ist, typische C++-Großprojekte zu lesen (ohne über Erweiterungen wie `__attribute__((visible))` zu stolpern). `gcc-xml`⁸ böte sich an, wenn er auch die Funktionskörper auslesen könnte. Als mögliche Kandidaten kommen MCC [29] des iPlasma-Projekts [19] sowie Analysatorgeneratoren wie Elkhound [24] und dem daraus entwickelten C++-Analysator Elsa⁹ in Frage.

⁷<http://trolls.troll.no/~rraggi/robe-pg-1.0.tar.gz> (14. Aug. 2006)

⁸<http://www.gccxml.org/HTML/Index.html> (11. Apr. 2006)

⁹<http://www.cs.berkeley.edu/~smcpeak/elkhound/>

Die aus dem Analysator extrahierte Ausgabe ist dann in der anfangs gewählten Repräsentation abzulegen und erlaubt das Testen einfacher Quelltexte.

Spätestens zu diesem Zeitpunkt soll der Prototyp bereits in Existenz getreten sein. Alle nachfolgenden Implementierungen erfolgen ab dann auf seiner Basis.

Anschließend werden rudimentäre Tests und prototypische Versuche auf die Repräsentation gefahren, um ihre Eignung für die Quelltexttransformation zu ermitteln. Insbesondere die Problematik der Erhaltung der Präprozessormakros nach einer Transformation verdient besonderes Augenmerk [40, 10].

Teil dieser Tests sind ausgewählte Zyklenauflösungsverfahren, welche die grundsätzliche Funktionsfähigkeit der automatischen Zyklenauflösung aufzeigen und wertvolle Erfahrungen liefern.

Nun beginnt der erste Hauptteil der Arbeit. Sämtliche in Kapitel 3 gesichteten Zyklenauflosungstechniken unterliegen einer Begutachtung und Kategorisierung hinsichtlich ihrer Einsatzmöglichkeiten auf realen Softwaresystemen.

Hier treten das erste Mal verschiedene Techniken der Mustererkennung in Erscheinung. Als erstes gilt es, in der Repräsentation alle Zyklengruppen zu identifizieren und jede für sich zu betrachten. In Kapitel 3.3 beschriebene Techniken können dabei hilfreich sein.

In jeder Zyklengruppe gilt es nun, Muster zu finden, auf denen sich eine in Kapitel 3.4 beschriebene Zyklenauflosungstechnik anwenden lässt. Der erste kreativ-wissenschaftliche Schöpfungsakt besteht in der Entwicklung eines Verfahrens – sei es eine Heuristik, eine metrikbasierte Zuordnung, eine statistische Berechnungsmethode oder ähnliches – zur Feststellung einer für einen Zyklus bestgeeigneten Auflösungstechnik (wobei »bestgeeignet« nicht im mathematisch-absoluten Sinne zu verstehen ist).

Als Muster dienen einerseits künstliche Testfälle, andererseits reale Softwaresysteme unterschiedlicher Größe, die der Validierung der Ergebnisse dienlich sein sollen.

Nach Abschluss dieses Stadiums steht folgende Forschungserkenntnis zur Verfügung: automatische Anwendung passender Zyklenauflosungstechniken durch *(effektiv verwendetes Verfahren)*. Sie beschreibt die Verwendung jenes Verfahrens (für dessen Name noch ein Platzhalter steht) zum Zwecke der Zyklenauflösung.

Diese Erkenntnis soll in einem Artikel expliziert werden, der nach Gestalt und Inhalt einer Veröffentlichung zum Zwecke des wissenschaftlichen Diskurses zu unterwerfen ist.

Hernach beginnt der zweite Hauptteil der Arbeit. Bis dato vermochte das Verfahren zur automatischen Zyklenauflösung passende Auflösungstechniken auf entsprechende Muster zyklischer Abhängigkeiten anwenden, ohne jedoch die Schnittstellenkompatibilität zu berücksichtigen.

Der zweite Hauptteil widmet sich daher der Beeinflussung der Zyklenauflösung durch Beschränkungen aufgrund von Schnittstellen.

Als erster Schritt erfolgt die Erforschung der bereits in Kapitel 4.3 erwähnten automatischen Feststellung öffentlicher Schnittstellen. Es lässt sich im gegenwärtigen Zustand nicht einfach abschätzen, wie effektiv oder wie schwierig eine solche Feststellung ist. Die sich bietenden Möglichkeiten von primitiver Heuristik bis hin zum letzten Ausweg, der manuellen Spezifikation der Schnittstellen, sind in Betracht zu ziehen. (Letzterer Punkt erwies sich jedoch als Rückschlag, da ein Großteil des Verfahrensautomatismus verloren ginge.)

Konnten vielversprechende Methoden zur Schnittstellenerkennung gefunden werden, so ist auch hiervon die interessierte Öffentlichkeit in gefälliger Form eines entsprechenden Artikels zu unterrichten.

Wie auch immer die Schnittstellen in letzter Instanz der automatischen Zyklenauflösung gewahrt gemacht werden, nun ist der Zeitpunkt gekommen, das Verfahren so abzuändern, dass Struktur und Semantik der öffentlichen Schnittstellen hinsichtlich Quell- und Binärkompatibilität über die Transformation hinweg erhalten bleiben.

Dabei impliziert »Erhaltung der Schnittstelle« nicht notwendigerweise »Erhaltung der implementierenden Klasse«. Die Realisierung größtmöglicher Zyklenauflösungspotentials stellt dabei den Kernbereich der in diesem Hauptteil durchgeführten Forschung dar. Zu untersuchen sind also Mittel und Wege, die diesem größtmöglichen Potential entgegenstreben.

Die so gewonnenen Erkenntnisse fließen in die automatische Zyklenauflösung ein und runden damit die Forschungsarbeit ab. Tests mit realen Softwaresystemen zeigen, ob und wie gut sich das Verfahren im Felde schlägt.

Selbstverständlich sind die Forschungsergebnisse des zweiten Hauptteils ebenfalls einer potentiell interessierten Wissenschaftsgemeinde durch Veröffentlichung zugänglich zu machen.

Da die bislang existierende automatische Zyklenauflösung sämtliche nicht durch physikalische Einschränkungen (wie zum Beispiel einer Vererbungsbeziehung) betroffenen Zyklen auflöst, richten wir damit »Kollateralschäden« an. Das heißt, auch »gute« Zyklen wurden aufgelöst und dadurch potentiell das Umbauergebnis degradiert.

Hier fließt nun die Forschung zur Entdeckung »guter« zyklischer Abhängigkeiten ein, die von der Auflösung ausgespart zu bleiben haben. Die Möglichkeiten rangieren wie auch bei der Erkennung der Muster zu Auflösungstechniken von Heuristiken bis zu Statistiken. Jedoch ist zuvorderst eine Untersuchung einer repräsentativen Menge an Softwaresystemen durchzuführen, um typische »gute« Zyklen festzustellen.

Konnte ein weitgehend befriedigendes Forschungsergebnis erzielt werden, ist eine Testreihe auf denselben wie bereits am ursprünglichen Verfahren der Zyklenauflösung angewandten Softwaresystemen durchzuführen. Damit offenbaren sich eventuelle Verschlechterungen.

Auch die Forschungserkenntnis über die Entdeckung »guter« Zyklen sollte der Wissenschaftsgemeinde nicht vorenthalten werden. Die Erstellung und Präsentation einer entsprechenden Publikation bietet sich hierbei an.

Die Erforschung »guter« Zyklen findet lediglich unter Vorbehalt statt und ist aus diesem Grund als letzter Teil der Forschungstätigkeit angesetzt, da [25, Kap. 9] ebenfalls die Untersuchung »guter« Zyklen erwägt. Somit besteht die Hoffnung, dass bereits verwertbare Ergebnisse zum Ende der Forschungstätigkeit vorliegen und uns eine eigene Untersuchung erspart bleibt.

Hiermit sind die geplanten Schritte der Forschungsarbeit weitgehend beschrieben. Wie dem aufmerksamen Leser nicht entgangen sein mag, nimmt der Grad der Detailtreue mit der zeitlichen Entfernung immer weiter ab. Dies ist nur natürlich, da über die kurzfristig anstehenden Aufgaben größere Klarheit herrscht als über die mittel- und langfristigen.

5.3 Vorläufige Struktur der Arbeit

Die Dissertationsschrift soll in etwa die nachfolgend im Format eines Inhaltsverzeichnisses angegebene Kapitelstruktur aufweisen.

1 Einführung

- 1.1 Grundbegriffe
- 1.2 Beitrag zum Stand der Technik

1.3	Kapitelübersicht	
2	Stand der Technik	
2.1	Zyklenerkennung	
2.2	Zyklenauflösung	
2.3	Umbau	
2.4	Schnittstellenkompatibilität	
2.5	Quellkompatibilität	
2.6	Binärkompatibilität	
3	Techniken	
3.1	Technik1	
3.2	Technik2	
3.3	Technik3	
4	Zyklenauflösung	
5	Gewollte Zyklen	
6	Schnittstellenfeststellung	
7	Kompatibilität	
7.1	Binärkompatibilität	
7.2	Quellkompatibilität	
7.3	Berücksichtigung bei der Auflösung	
8	Prototyp	
9	Schluss	
9.1	Rückblick	
9.2	Ausblick	

Sollte die Untersuchung »guter« Zyklen wegfallen (siehe S. 21), so verschiebt sich Kapitel 5 in Kapitel 2.

6 Zusammenfassung und Terminplan

Das Ziel der Forschungstätigkeit soll in ein Verfahren zur Auflösung von zyklischen Abhängigkeiten unter Berücksichtigung von Quellkompatibilität für in imperativen objektorientierten Programmiersprachen münden und damit einen Beitrag zur Hebung des Programmverständnisses, der Wartbarkeit und der Testbarkeit.

6.1 Zusammenfassung

In Kapitel 2 wurde der Hintergrund zur Zyklenproblematik genauer erläutert, konzeptuelle Alternativen angeführt und beschrieben, warum eine automatische Zyklenauflösung wünschenswert ist.

Kapitel 3 führte verschiedenste Verfahren zum Stand der Technik der Zyklenerkennung, Zyklenauflösung und der Kompatibilität auf.

Kapitel 4 kontrastierte die im Rahmen dieser Forschungstätigkeit zu untersuchenden Sachverhalte von den bereits existierenden.

Kapitel 5 beschrieb die angedachte Vorgehensweise unter Nennung von Voraussetzungen, Methoden und Ergebnissen und strukturierte die Forschungsarbeit grob, ohne einen konkreten Zeitplan vorzugeben.

6.2 Zeitplan

Da in Kapitel 5 kein Zeitplan angegeben wurde, holen wir dies hier nach. Der folgende Zeitplan besitzt Wochengranularität und beschreibt je Kalenderwoche das zu bewältigende Pensum. Der Plan ist vorläufig – Änderungen sind jederzeit vorbehalten.

Einträge mit der Inschrift *nicht existent* bedeuten, dass in dieser Woche aufgrund voraussichtlicher physikalischer Abwesenheit des Forschungspersonals keine Forschungstätigkeit stattfindet.

KW	Datum	Tätigkeit
40	Okt. 2006	C++-Umbauverfahren untersuchen
41	Okt. 2006	geeigneten C++-Analysator wählen
42	Okt. 2006	geeigneten C++-Analysator wählen
43	Okt. 2006	Prototyp erstellen
44	Nov. 2006	Prototyp auf C++ testen
45	Nov. 2006	Repräsentation wählen/erstellen
46	Nov. 2006	Repräsentation erstellen/testen
47	Nov. 2006	<i>Reserve</i>
48	Nov. 2006	Vollst. Liste der Zyklenauflösungsverf. aufstellen
49	Dez. 2006	Mustererkennungstechniken untersuchen
50	Dez. 2006	Mustererkennungstechniken untersuchen
51	Dez. 2006	Auflösungstechniken finden
52	Dez. 2006	<i>Nicht existent</i>
1	Jan. 2007	Auflösungstechniken finden
2	Jan. 2007	Auflösungstechniken finden
3	Jan. 2007	Auflösungstechniken implementieren
4	Jan. 2007	Auflösungstechniken testen (kleine Testfälle)
5	Jan. 2007	Große Testfälle vorbereiten
6	Feb. 2007	Auflösungstechniken testen (große Testfälle)
7	Feb. 2007	Auflösungstechniken implementieren reparieren
8	Feb. 2007	<i>Reserve</i>
9	Feb. 2007	Artikel erstellen
10	Mär. 2007	Artikel erstellen
11	Mär. 2007	Artikel erstellen
12	Mär. 2007	Artikel erstellen
13	Mär. 2007	<i>Reserve</i>
14	Apr. 2007	Automatisch Schnittstellen feststellen Feldversuch
15	Apr. 2007	Automatisch Schnittstellen feststellen Feldversuch
16	Apr. 2007	Automatisch Schnittstellen feststellen Feldversuch
17	Apr. 2007	Automatisch Schnittstellen feststellen implementieren
18	Mai 2007	Automatisch Schnittstellen feststellen implementieren
Fortsetzung folgt...		

KW	Datum	Tätigkeit
19	Mai 2007	Automatisch Schnittstellen feststellen testen
20	Mai 2007	Automatisch Schnittstellen feststellen implementieren reparieren
21	Mai 2007	<i>Reserve</i>
22	Mai 2007	Artikel erstellen
23	Jun. 2007	Artikel erstellen
24	Jun. 2007	Artikel erstellen
25	Jun. 2007	Artikel erstellen
26	Jun. 2007	Binärkompatibilität definieren Literatur
27	Jul. 2007	Quellkompatibilität definieren Literatur
28	Jul. 2007	Verfahren um Erhalt von Kompatibilität erweitern
29	Jul. 2007	Erhalt von Kompatibilität implementieren
30	Jul. 2007	Erhalt von Kompatibilität implementieren
31	Aug. 2007	Erhalt von Kompatibilität implementieren
32	Aug. 2007	<i>Reserve</i>
33	Aug. 2007	Erhalt von Kompatibilität testen
34	Aug. 2007	<i>Nicht existent</i>
35	Aug. 2007	<i>Nicht existent</i>
36	Sep. 2007	<i>Nicht existent</i>
37	Sep. 2007	Erhalt von Kompatibilität testen
38	Sep. 2007	Erhalt von Kompatibilität testen
39	Sep. 2007	Erhalt von Kompatibilität implementieren reparieren
40	Okt. 2007	<i>Reserve</i>
41	Okt. 2007	Artikel erstellen
42	Okt. 2007	Artikel erstellen
43	Okt. 2007	Artikel erstellen
44	Okt. 2007	Artikel erstellen
45	Nov. 2007	<i>Reserve</i>
46	Nov. 2007	Gute Zyklen finden Literatur
47	Nov. 2007	Gute Zyklen finden Feldversuch
48	Nov. 2007	Gute Zyklen berücksichtigen implementieren
49	Dez. 2007	Gute Zyklen berücksichtigen implementieren
50	Dez. 2007	Gute Zyklen berücksichtigen testen
51	Dez. 2007	Gute Zyklen berücksichtigen implementieren reparieren
52	Dez. 2007	<i>Nicht existent</i>
1	Jan. 2008	<i>Reserve</i>
2	Jan. 2008	Artikel erstellen
3	Jan. 2008	Artikel erstellen
4	Jan. 2008	Artikel erstellen
5	Jan. 2008	Artikel erstellen
6	Feb. 2008	Dissertationsschrift verfassen
7	Feb. 2008	Dissertationsschrift verfassen
8	Feb. 2008	Dissertationsschrift verfassen
9	Feb. 2008	Dissertationsschrift verfassen
10	Mär. 2008	Dissertationsschrift verfassen

Fortsetzung folgt...

KW	Datum	Tätigkeit
11	Mär. 2008	Dissertationsschrift verfassen
12	Mär. 2008	Dissertationsschrift verfassen
13	Mär. 2008	Dissertationsschrift verfassen
14	Apr. 2008	Dissertationsschrift vorab prüfen lassen
15	Apr. 2008	Dissertationsschrift überarbeiten
16	Apr. 2008	Dissertationsschrift überarbeiten
17	Apr. 2008	Dissertationsschrift überarbeiten
18	Apr. 2008	Verbesserungsvorschläge einarbeiten
19	Mai 2008	Verbesserungsvorschläge einarbeiten
20	Mai 2008	Verbesserungsvorschläge einarbeiten
21	Mai 2008	Dissertationsschrift zur Begutachtung einreichen
22	Mai 2008	Dissertationsschrift zur Begutachtung einreichen
23	Jun. 2008	Dissertationsschrift zur Begutachtung einreichen
24	Jun. 2008	Dissertationsschrift zur Begutachtung einreichen
25	Jun. 2008	Dissertationsschrift zur Begutachtung einreichen
26	Jun. 2008	Rigorosum

Tabelle 1: Zeitliche Planung der Forschungstätigkeit

Literatur

- [1] ADAMEK, JIRI: *Addressing Unbounded Parallelism in Verification of Software Components*. Technischer Bericht, Abteilung Software Engineering, Karlsuniversität Prag, 2006.
- [2] *Programming Language C++*. Technischer Bericht, ISO, 1998. ISO/IEC 14882.
- [3] CLARK, JOHN und DEREK ALLAN HOLTON: *Graphentheorie*. Spektrum Akademischer Verlag, 1994.
- [4] DEMEYER, SERGE, SANDER TICHELAAR und PATRICK STEYAERT: *FAMIX 2.0: The FAMOOS Information Exchange Model*. Technischer Bericht, Universität Bern, 1999.
- [5] DMITRIEV, MIKHAIL: *Language-specific make technology for the Java programming language*. In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 373–385, 2002.
- [6] DROSSOPOULOU, SOPHIA, SOPHIA DROSSOPOULOU und SUSAN EISENBACH: *What is Java binary compatibility?* In: *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 341–361, 1998.
- [7] FORMAN, IRA R., MICHAEL H. CONNER, SCOTT H. DANFORTH und LARRY K. RAPER: *Release-to-release binary compatibility in SOM*. In: *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, Seiten 426–438, 1995.
- [8] FOWLER, MARTIN: *Refactoring*. Addison-Wesley, 2000.
- [9] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns CD*. Addison Wesley Longman, 1996.
- [10] 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.
- [11] 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.
- [12] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *The Java? Language Specification*. Addison-Wesley, 3. Auflage, 2005.
- [13] GRUNTZ, DOMINIK: *Java Design: On the Observer Pattern*. Java Report, Feb. 2002.
- [14] 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.
- [15] HAUTUS, EDWIN: *Improving Java Software Through Package Structure Analysis*. In: *Proceedings of the 6th IASTED International Conference Software Engineering and Applications*, 2002.
- [16] HOLT, RICHARD C., ANDREAS WINTER und ANDY SCHÜRR: *GXL: Toward a Standard Exchange Format*. Technischer Bericht, GUPRO, Universität Koblenz, 2000.

- [17] 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.
- [18] LAKOS, JOHN: *Large-Scale C++ Software Design*. Addison Wesley Longman, 1996.
- [19] MARINESCU, CRISTINA, RADU MARINESCU, PETRU FLORIN MIHANCEA, DANIEL RATIU und RICHARD WETTEL: *iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design*. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005.
- [20] MARTIN, ROBERT: *Object Oriented Design Quality Metrics*. ROAD, 1995.
- [21] MARTIN, ROBERT C.: *The Dependency Inversion Principle*. C++ Report, 1996.
- [22] MARTIN, ROBERT C.: *Granularity*. The C++ Report, 8(11), 1996.
- [23] 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.
- [24] MCPEAK, SCOTT und GEORGE C. NECULA: *Elkhound: A fast, practical GLR parser generator*. In: *Compiler Construction Proceedings*, Seiten 73–88. Universität Kalifornien, 2004.
- [25] MELTON, HAYDEN und EWAN TEMPERO: *Empirical Study of Cycles among Classes in Java*. Technischer Bericht, Department of Computer Science, Universität Auckland, 2006.
- [26] MELTON, HAYDEN und EWAN TEMPERO: *Identifying Refactoring Opportunities by Identifying Dependency Cycles*. In: *Proceedings of the 29th Australasian Computer Science Conference*, 2006.
- [27] MELTON, HAYDEN und EWAN TEMPERO: *JooJ: Real-time Support for Avoiding Cyclic Dependencies*. Technischer Bericht, Department of Computer Science, Universität Auckland, 2006.
- [28] MEZINI, MIRA und JENS UWE PIPKE: *A Study of Java's Binary Compatibility*. Technischer Bericht, Technische Universität Darmstadt, 1999.
- [29] MIHANCEA, PETRU-FLORIN: *The extraction of detailed design information from C++ software systems*, 2004.
- [30] *MOF 2.0/XMI Mapping Specification, v2.1*. Technischer Bericht, Object Management Group, 2005.
- [31] MOORE, IVAN: *Automatic inheritance hierarchy restructuring and method refactoring*. ACM SIGPLAN Notices, 31(10):235–250, 1996.
- [32] 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.
- [33] 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.

- [34] OPDYKE, WILLIAM F.: *Refactoring Object-Oriented Frameworks*. Doktorarbeit, Universität Illinois, Urbana-Champaign, 1992.
- [35] PARNAS, DAVID LORGE: *Designing software for ease of extension and contraction*. In: *Proceedings of the 3rd international conference on Software engineering*, 1978.
- [36] PARRISH, ALLEN, BRANDON DIXON und DAVID CORDES: *Binary software components in the undergraduate computer science curriculum*. In: *Proceedings of the 32nd SIGCSE technical symposium on Computer Science Education*, Seiten 332–336, 2001.
- [37] PINZGER, MARTIN: *ArchView – Analyzing Evolutionary Aspects of Complex Software Systems*. Doktorarbeit, Technische Universität Wien, 2005.
- [38] 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.
- [39] SOFTWARE TOMOGRAPHY GMBH: *Sotograph 2.4 User’s Guide and Reference Manuals*, 2006. <http://www.software-tomography.com/>.
- [40] SPINELLIS, DIOMIDIS: *Global Analysis and Transformations in Preprocessed Languages*. IEEE Software Engineering, 29(11):1019–1030, 2003.
- [41] 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.
- [42] 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>.
- [43] 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.