Entwicklung von Verfahren zur statischen Analyse dynamischer Programmeigenschaften für das Refactoring aspektorientierter Programme

zur Erlangung des akademischen Grades
Diplom-Informatiker

vorgelegt dem
Fachbereich Informatik Fakultät IV der Technischen Universität Berlin

Joachim Hänsel
Matrikelnr. 166487, haebor@cs.tu-berlin.de

Gutachter: Prof. Dr.-Ing. Stefan Jähnichen
2.Gutachter: Dr. Ing. Stephan Hermann
Betreuung: Jan Wloka

Entstanden im BMBF geförderten Forschungsprojekt TOPPrax am Fraunhofer Institut für Rechnerarchitektur und Softwaretechnik – FIRST

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, 3. Mai 2006

_____

Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Aspektorientierte Programmierung

Softwareprodukte werden zunehmend komplexer und stellen die Entwickler vor die Herausforderung, diese Komplexität in den Griff zu bekommen. Die Zerlegung von Software in einzelne Module ist schon seit geraumer Zeit ein Mittel, Komplexität beherrschbar zu machen. Ein Modul erlaubt es einem Entwickler, sich auf die Bearbeitung eines Teils der Software zu beschränken. Was ein Modul ist, hängt dabei von dem verwendeten Programmiermodell ab. In der objektorientierten Programmierung sind Module z. B. Pakete und Klassen.

Modularisierung kann aber nur dann Komplexität von Software verringern, wenn sie nicht wahllos durchgeführt wurde. Für eine sinnvolle Zerlegung sollten Kriterien aufgestellt werden. **Separation of Concerns** ist in der Softwareentwicklung das Konzept, mit dem Kriterien in Form von Concerns entstehen. Der Begriff **Concern** wird von Dijkstra in Dijkstra (1997) wie folgt beschrieben:

> „Ein Concern ist ein Teil des Problems, welches als eine einzelne konzeptionelle Einheit behandelt werden soll.“ [1]

Die Aufgabe (das Problem), die eine Software erfüllen soll, wird also in Concerns zerlegt, die jeweils in Modulen implementiert werden. Durch die Zusammenführung der Module in einer Software kann dann die gestellte Aufgabe im Zusammenspiel der Module erfüllt werden. Beispiele für Concerns sind Anforderungen, Anwendungsfälle, Features, Datenstrukturen, Fragestellungen des Quality-of-Service, Varianten, Kollaborationen, Patterns und Verträge.[2]

Mit der Aspektorientierten Programmierung ist ein neues Programmiermodell entstanden. Es soll insbesondere die modularisierte Implementierung sog. Crosscutting Concerns verbessern. **Crosscutting Concerns** sind solche Concerns, die sich *nicht* mit dem verwendeten Programmiermodell modularisiert implementieren lassen[3]. Sobald in einer Software ein Crosscutting Concern implementiert werden muss, schädigt das die Gesamtstruktur in zweierlei Hinsicht: Einerseits werden andere Module, die eigentlich ausschließlich einen Concern implementieren sollten, mit Code versehen, der dem Crosscutting Concern zuzuordnen ist, andererseits kann die Implementierung dieses Concerns auf Grund seiner Verteilung nicht überblickt werden. In einem

[1] A concern is some part of the problem that we want to treat as a single conceptual unit. Zitiert nach: von Flach G. Chavez u. a. (2005)

[2] In einem seiner Essays (vgl. Dijkstra 1974) beschreibt Dijkstra das Prinzip des Separation of Concerns sogar als "die einzig verfügbare Technik", Gedanken zu sortieren.

[3] Ein Crosscutting Concern kann also nur im Hinblick auf ein verwendetes Programmiermodell von einem gewöhnlichen Concern unterschieden werden.

objektorientierten Programm verteilt sich z. B. die Implementierung des Crosscutting Concerns quer zur Programmstruktur über eine Vielzahl von Klassen und Methoden.

Aspektorientierte Programmierung führt den **Aspekt** als neue Moduleinheit ein. Im Sinne der Modularisierung sollen Aspekte Implementierungen von Crosscutting Concerns kapseln. Mit der Kapselung in einen Aspekt wird erreicht, dass ein sonst Crosscutting Concern die Struktur des restlichen Programms nicht mit seiner Implementierung „stört“ . Ein Aspekt bedient sich dabei zweier konzeptioneller Richtungen. Ein Aspekt kann sowohl eine strukturelle Erweiterung vornehmen als auch das Programmverhalten anpassen.

1.1.1 Strukturelle Erweiterungen

Konzepte zur Umsetzung von Aspekten verfolgen jeweils unterschiedliche Ziele, indem sie sich der Kapselung von speziellen Teilen der Implementierung eines Crosscutting Concerns annehmen. Eine Implementierung eines Crosscutting Concerns kann einerseits Strukturen und andererseits Programmverhalten umfassen. Strukturen werden gebraucht, um Kontextinformationen an der richtigen Stelle zur Verfügung zu haben, auf die das im Aspekt implementierte Programmverhalten zugreifen kann.

Erreicht wird das, indem Aspekte, die eine strukturelle Erweiterung vornehmen, Strukturen zu wohldefinierten Punkten in anderen Modulen des Programms hinzufügen oder diese abändern. Die wohldefinierten Punkte in anderen Modulen eines Programms können z. B. Klassen sein, hinzufügbare Strukturen können z. B. Felddeklarationen sein. Für Konzepte, die strukturelle Erweiterungen verwirklichen, sind **Rollen** im **ObjectTeams** Programmiermodell ein Beispiel, **Introductions** aus der **AspectJ** Programmiersprache ein weiteres[4].

1.1.2 Verhaltensanpassung

Eine Verhaltensanpassung wird mit dem Ziel vorgenommen, fehlendes Verhalten hinzuzufügen oder unzureichendes Verhalten zu ergänzen oder sogar ganz zu ersetzen. Aspekte zur Verhaltensanpassung passen Programmverhalten anderer Module an. Die Anpassung findet an wohldefinierten Interaktionspunkten statt. Eine Möglichkeit der Verhaltensanpassung ist durch die Konzepte von Advice, Pointcut und Joinpoints umgesetzt worden.

Abbildung 1.1 zeigt schematisch ein aspektorientiertes Programm. Neben einer strukturellen Erweiterung ist hier auch eine Verhaltensanpassung mittels Pointcuts und Advices dargestellt. In **Advices** befinden sich die Teile des gekapselten Verhaltens eines Aspekts. **Pointcuts** spezifizieren die Interaktionspunkte in anderen Modulen, an denen bestimmte Advices gebunden werden. Die Interaktionspunkte, an die Advices gebunden werden können, werden als **Joinpoint** bezeichnet. Konzeptionell bedeutet das, dass das Verhalten eines Advice durch den Pointcut an Verhalten aus anderen Modulen gebunden wird. Das gebundene Verhalten der Advices eines Aspektes führt zu dem gewünschten angepassten Verhalten.

[4] (vgl. Herrmann 2002; Kiczales u. a. 1997)

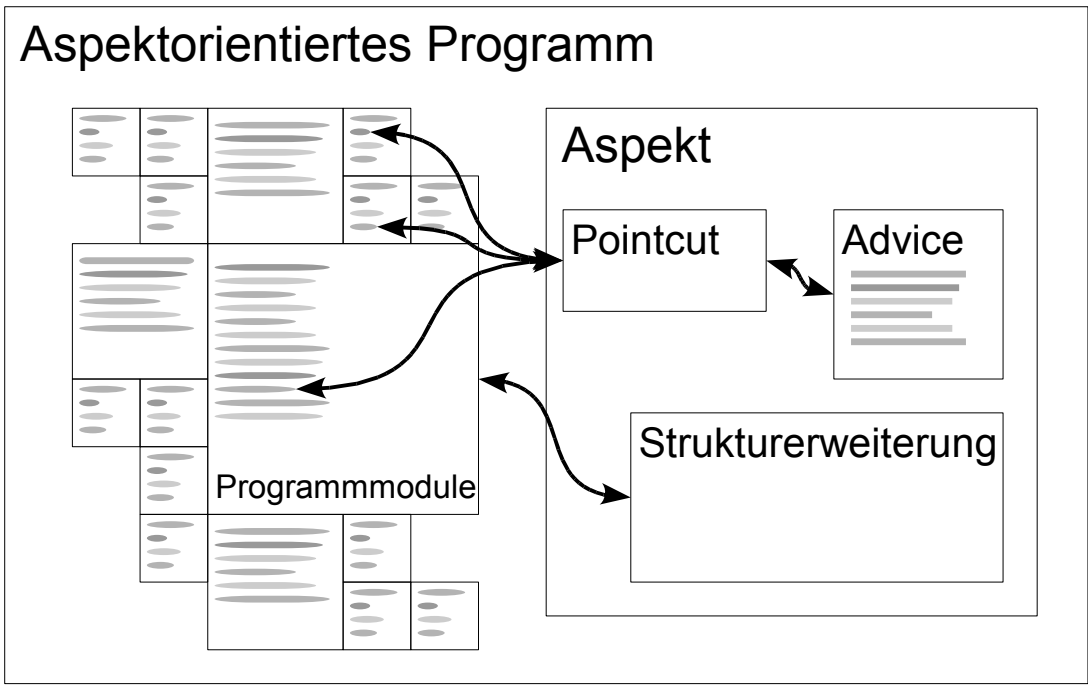


Abbildung 1.1: Aspektorientierte Programmierung

1.2 Refactoring

Software hat im Gegensatz zu vielen anderen Produktarten die Eigenschaft, dass die Entwicklung eher einem Evolutionsprozess als einem Konstruktionsprozess entspricht. Ursachen dafür sind sich immer wieder ändernde Anforderungen oder neue hinzukommende Anforderungen. Im fortlaufenden Entwicklungsprozess wird dann sowohl versucht, diesen sich ändernden Anforderungen gerecht zu werden, als auch neue Anforderungen zu realisieren und Fehler im bestehenden Produkt zu beseitigen.

Evolution von Software kann einen entscheidenden Nachteil mit sich bringen: Je weiter der Evolutionsprozess fortgeschritten ist, desto schlechter gelingt es, neue Evolutionsschritte zu tätigen. Dass kontinuierliche Änderungen problematisch sind, ist schon seit längerem bekannt (vgl. Frederick P. Brooks 1975), und schon früh ist darauf hingewiesen worden, dass Softwareentwicklung sich damit auseinandersetzen muss:

> „As an evolving program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.“ [5]

Statt wie von Lehman gefordert, die Struktur aufrechtzuerhalten oder zu reduzieren, nachdem sie sich schon durch Änderungen verschlechtert hat, scheint es sinnvoller zu sein zuerst die Struktur auf die Änderung vorzubereiten. Wenn die Software dann geändert wird, geschieht das einvernehmlich mit der angepassten Struktur.

Im Sinne dieser Idee hat Fowler in (Fowler u. a. 1999, Seite 7) das **Refactoring** als Weg für Strukturanpassungen beschrieben. Das Refactoring erlaubt Änderungen an der Struktur eines Programms vorzunehmen, ohne dass sie das beobachtbare Verhalten ändern. Eine Änderung

[5] (vgl. Lehman u. Belady 1985)

kann so, im Sinne der vorangegangenen Überlegung in zwei Schritten vollzogen werden: Erst wird die Struktur mittels Refactoring angepasst, dann wird die eigentliche Änderung eingebracht.

Der relativ allgemeine Begriff des Refactorings meint in der Literatur häufig verschiedene Dinge. Zur besseren Unterscheidung sollen hier Refactoring, refaktorisieren und Refactoringpattern als getrennte Begriffe eingeführt und definiert werden.

Fowler definiert Refactoring und refaktorisieren in Fowler u. a. (1999)[6] folgendermaßen:

> „**Refaktorisierung** (Substantiv): Eine Änderung an der internen Struktur einer Software, um sie leichter verständlich zu machen und einfacher zu verändern, ohne ihr beobachtbares Verhalten zu ändern.“ [7]

> „**Refaktorisieren** (Verb): Eine Software umstrukturieren, ohne ihr beobachtbares Verhalten zu ändern, indem man eine Reihe von Refaktorisierungen anwendet.“

In demselben Buch werden darüber hinaus eine Reihe von Refactorings als wieder verwendbare Muster beschrieben. Solche Muster werden in anderen Arbeiten auch mit **Refactoringpattern** bezeichnet. Ein Refactoringpattern definiert folgende Bestandteile:

- Ein Refactoringpattern hat einen Namen.
- Es beschreibt das Problem im Quellcode, das gelöst werden kann.
- Es werden Umstände beschrieben, unter denen das Refactoringpattern angewendet werden kann oder unter denen es nicht angewendet werden darf.
- Einzelne Schritte werden beschrieben, die getätigt werden müssen, um die Änderung durchzuführen.

Refactoringpatterns sind Refactorings, die sich unter bestimmten Umständen im Einsatz bewährt haben. Sie sind wohl durchdacht und in ihren Transformationsschritten mehrfach erprobt worden. Verschiedene Autoren haben solche Refactoringpatterns, nachdem sie eine gewisse Reife erlangt haben, als sichere Wege des Refaktorisierens in Katalogen oder Büchern zusammengefasst. Auf Basis von Refactoringpatterns können Entwickler ihre eigenen Refactorings vornehmen oder ein größeres Refactoring aus mehreren Refactoringpatterns aufbauen.

1.2.1 Charakterisierung eines Refactoringprozesses

Zur besseren Veranschaulichung, welche Aktivitäten das Refaktorisieren eines Programms beinhaltet, erfolgt eine Charakterisierung, die der üblichen Auffassung eines Refactoringprozesses entspricht.[8]

[6] Übersetzung aus der deutschen Fassung Fowler u. a. (2005)

[7] Wie auch schon vorher im Text wird statt des deutschen Wortes **Refaktorisierung** weiterhin das englische **Refactoring** benutzt.

[8] In einem Papier von Mens und Tourwé ist ein Refactoringprozess beschrieben worden, an dem sich der hier beschriebene orientiert. Der dort beschriebene Prozess ist allerdings etwas weiter gefasst (vgl. Mens u. Tourwé 2004).

Identifizierung eines Designmangels

In dem zu bearbeitenden Programm werden Designmängel identifiziert, die einer Weiterentwicklung entgegenstehen. Designmängel lassen sich in Form von strukturellen Schwächen im Programm erkennen. Für objektorientierte Sprachen gibt es bereits Kataloge, die sog. *Bad Code Smells* beschreiben, die auf strukturelle Schwächen im Quellcode hinweisen.

Bestimmung eines geeigneten Refactoringpatterns

Der Entwickler wählt Refactoringpatterns aus, die zu dem identifizierten Designmangel passen. Von den passenden Refactoringpatterns wird das gewählt, welches das beste Verhältnis von Strukturverbesserung und negativen Seiteneffekten auf den restlichen Quellcode aufweist.

Gewährleistung der Verhaltenserhaltung

Die im Refactoringpattern beschriebenen Transformationsschritte sind so gewählt, dass ein sicherer, verhaltenserhaltender Übergang entsteht. Sie genügen Regeln, die sicherstellen, dass einerseits die Syntax korrekt bleibt und zum anderen semantische Eigenschaften beibehalten werden. In Opdyke (1992) werden solche syntaktischen und semantischen Eigenschaften für das objektorientierte Programmiermodell beschrieben.

Anwendung des Refactoringpatterns

Konnte im vorherigen Schritt eine Verhaltenserhaltung gewährleistet werden, so werden die einzelnen, im Refactoringpattern beschriebenen, Änderungsschritte durchgeführt. Wenn sich ein Schritt des Refactoringpatterns nicht erfolgreich durchführen lässt, werden die einzelnen Schritte rückgängig gemacht.

Überprüfung, ob Designmangel beseitigt

Zuletzt wird überprüft, ob die Anwendung des Refactoringpatterns den identifizierten Designmangel wirklich beseitigt hat. Dazu wird der Quellcode daraufhin untersucht, ob er evtl. immer noch die anfangs identifizierten *Bad Code Smells* enthält.

1.2.2 Werkzeuggestütztes Refactoring

Eine besondere Eigenschaft von Refactoringpatterns ist, dass viele automatisiert[9] angewendet werden können. Eine automatisierte Anwendung eines Refactoringpatterns ist der manuellen Anwendung sogar überlegen. Das begründet sich dadurch, dass einem Entwickler Fehler bei der Prüfung von Bedingungen und während der Ausführung der einzelnen Transformationsschritte unterlaufen können. Entsprechende Werkzeugunterstützung existiert für verschiedene Entwicklungsumgebungen und Programmiersprachen. Eine automatisierte Anwendung kann für die verschiedenen Schritte des Prozesses erfolgen:

[9] Automatisiert ist von automatisch zu unterscheiden, ein Mitwirken des Entwicklers ist unbedingt erforderlich

Identifizierung eines Designmangels
Der Entwickler kann die Identifizierung von Designmängeln anstoßen. Ein Werkzeug kann anhand verschiedener Kriterien mögliche Designmängel identifizieren und diese dem Entwickler aufzeigen. Die vom Werkzeug automatisch ermittelten Designmängel können manche Designmängel beinhalten, die keine sind. Andererseits besteht die Möglichkeit, dass manche Designmängel nicht identifiziert wurden. Der Entwickler wird dann entscheiden, welches wirkliche Designmängel sind und welche einem Refactoring unterzogen werden sollen.

Bestimmung eines geeigneten Refactoringpatterns
Ein Werkzeug kann zu den gewählten Designmängeln passende Refactoringpatterns anbieten. Der Entwickler sucht sich das seines Erachtens passende Refactoringpattern heraus.

Gewährleistung der Verhaltenserhaltung
Werkzeuge können automatisch die ausgewählten Refactoringpatterns daraufhin untersuchen, ob sie sich anwenden lassen oder ob sich bestimmte Schritte nicht ausführen lassen werden.

Anwendung des Refactoringpatterns
Zusammen mit der Überprüfung ob die Verhaltenserhaltung gewährleistet werden kann, ist das automatische Anwenden eines Refactoringpatterns besonders geeignet von einem Werkzeug ausgeführt zu werden. Die sich immer weiter verbreiternde Basis an Werkzeugen veranschaulicht dies (vgl. Fowler 2006).

1.3 Refactoring aspektorientierter Programme

Aspektorientierte Software unterliegt, genauso wie Software, die mit anderen Progammiermodellen erstellt wird, der Evolution. Daraus lässt sich schließen, dass auch aspektorientierte Software so angepasst werden muss, dass sie evolutionär weiterentwickelt werden kann. Die weitere Schlussfolgerung, dass auch hier Refactorings eingesetzt werden sollten, liegt gleichfalls nahe.

1.3.1 Aspekt berücksichtigendes Refactoring

Aspektorientierte Sprachen sind meist eine Erweiterung einer existierenden Sprache, welche einem anderen Programmiermodell folgt. Die erweiterte Sprache wird auch als Basissprache bezeichnet. Diese Basissprache wird um Konzepte des aspektorientierten Programmiermodells, wie strukturelle Erweiterungen und Verhaltensanpassung, erweitert. Dazu wird die Basissprache mit neuen Sprachelementen zur Realisierung der Konzepte ergänzt. Abbildung 1.2 zeigt schematisch die Verhältnisse von Modellen und Sprachen.

Während das Refactoring von Sprachen vieler anderer Programmiermodelle (insbesondere des objektorientierten Modells) schon eingehend bearbeitet worden ist, steht das Refactoring von Sprachen des aspektorientierten Modells noch am Anfang und ist Gegenstand aktueller Forschung. Das Refactoring für aspektorientierte Sprachen von Grund auf neu zu entwickeln ist dabei nicht das Ziel, sondern die Anpassung bestehender Refactoringprozesse und Refactoringpatterns der Basissprache. Für eine Anpassung ist es notwendig, auch die neuen Sprachelemente

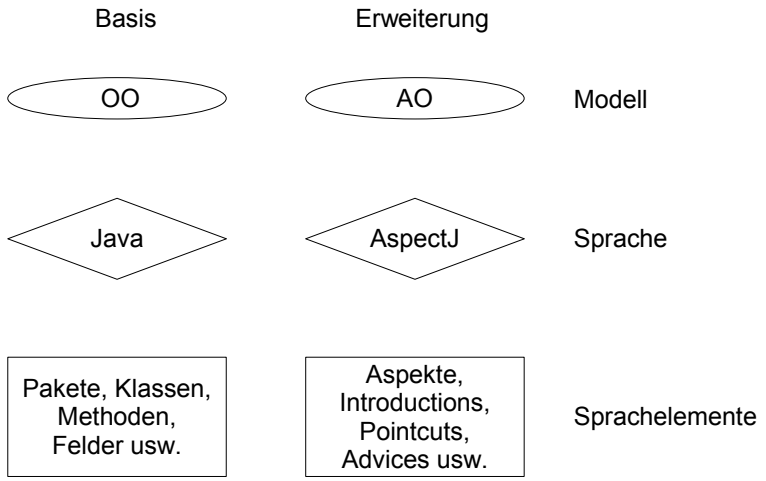


Abbildung 1.2: Erweiterung einer Basissprache zur aspektorientierten Sprache am Beispiel der Programmiersprachen Java und AspectJ

beim Refaktorisieren zu berücksichtigen. Genauso wie beim Refactoring von Programmen in der Basissprache dürfen ausschließlich verhaltenserhaltende Änderungen vorgenommen werden.

Einige Arbeiten haben sich damit auseinandergesetzt, mit welchen Zielsetzungen aspektorientierte Programme refaktorisiert werden können (vgl. Wloka 2005; Hannemann u. a. 2005). Im Folgenden sind die vorgefundenen Zielsetzungen in einer Klassifizierung aspektorientierter Refactorings zusammengetragen, von denen eine Klasse ausgewählt wurde, die als Basis für die weitere Arbeit dienen soll.

- **Basisrefactorings unter Berücksichtigung von Aspekten**
  Diese Klasse deckt Refactorings ab, wie sie bereits in der Basissprache vorhanden sind. Bei der Refaktorisierung von Elementen der Basissprache müssen die Refactoringpatterns zusätzlich angepasst werden, um die neuen aspektorientierten Sprachelemente zu berücksichtigen.
- **Aspektrefactorings**
  In diese Klasse fallen Refactorings von Konstrukten, die auf Sprachelementen der aspektorientierten Erweiterung beruhen. Dafür müssen für die Sprachelemente der aspektorientierten Erweiterung neue Refactorings erarbeitet werden.

Die beiden folgenden Klassen beinhalten Refactorings, die sowohl Sprachelemente der Basissprache als auch die der Erweiterung betreffen.

- **Das Refaktorisieren von Elementen des Basiscodes in den Aspektcode**
  Mit Refactorings dieser Klasse wird das Ziel verfolgt, die Implementierung eines Crosscutting Concerns in einen Aspekt zu kapseln. Das bedeutet, dass aus verschiedenen Modulen Codeteile identifiziert werden müssen, die zum Crosscutting Concern gehören. Die identifizierten Teile werden dann entfernt und mit einer Implementierung in einem Aspekt ersetzt.
- **Das Refaktorisieren von Elementen des Aspektcodes in den Basiscode**
  Umgekehrt kann es vorkommen, dass ein Aspekt nicht mehr gebraucht wird und entsprechende Implementierungen in anderen Modulen untergebracht werden müssen.

Aspekte können sowohl Konstrukte der Basissprache als auch die der erweiternden Aspektsprache betreffen. Deswegen müssen bei den beiden ersten Klassen von Refactorings die Auswirkungen von Aspekten auf die zu refaktorisierenden Elemente berücksichtigt werden. Bei Refactorings der beiden letzten Klassen werden Refactorings aus den beiden ersten Klassen auf mehrere Elemente des Programms angewendet, um das jeweilige Ziel zu erreichen. Diese Refactorings sind also auf das Vorhandensein von entsprechenden Refactorings der ersten beiden Klassen angewiesen.

Das bedeutet insgesamt, dass die Berücksichtigung von Aspekten beim Refaktorisieren allererste Voraussetzung für die Entwicklung von aspektorientierten Refactorings ist. Refactorings für die Basissprache sollten schon existieren. Es liegt also nahe, sich zuerst mit aspektberücksichtigenden Basisrefactorings zu beschäftigen, bevor Refactorings auf neuen Sprachelementen entwickelt werden. Die Berücksichtigung von Aspekten für Basisrefactorings steht im Zusammenhang mit den in Abschnitt 1.1 beschriebenen Konzepten strukturelle Erweiterung und Verhaltensanpassung. Diese Arbeit soll sich ausschließlich mit den Auswirkungen von Verhaltensanpassungen auf das Refaktorisieren beschäftigen.

Wie bereits dargestellt, erfordert Refaktorisieren, dass nur solche Quellcodeänderungen vorgenommen werden, die die Erhaltung des Verhaltens eines Programms gewährleisten können. Opdyke hat Bedingungen aufgestellt, die, wenn sie bei Quellcodeänderungen eingehalten werden, Verhaltenserhaltung gewährleisten (vgl. Opdyke 1992, Abschnitt 4.1). In Gegenwart von Aspektbindungen müssen diese Bedingungen erweitert werden. Verhaltenserhaltung kann offensichtlich nicht gewährleistet werden, wenn Aspektbindungen verloren gehen, die Erhaltung von Aspektbindungen ist insofern eine weitere Bedingung.

Ein Ziel dieser Arbeit soll es sein, für eine bestimmte Form von Aspektbindungen herauszufinden, ob bei einer konkreten Quellcodeänderung eine Aspektbindung betroffen ist oder nicht. Dadurch können Quellcodeänderungen, die keine Aspektbindungen betreffen, als verhaltenserhaltend hinsichtlich der Aspektbindung durchgeführt werden. Darüber hinaus sollen dem Entwickler für von Quellcodeänderungen betroffene Aspektbindungen Informationen zur Verfügung gestellt werden, die Hinweise dafür liefern können, welche Anpassungen vorzunehmen sind, um das Verhalten beizubehalten.

1.4 Aufbau der Diplomarbeit

Der Aufbau der Arbeit ergibt sich folgendermaßen: Im folgenden *Kapitel 2* wird das aspektorientierte Konzept Verhaltensanpassung duch Advices Pointcuts und Joinpoints noch einmal näher erläuterte. Es wird gezeigt welche Arten von Joinpointeigenschaften es gibt. Anhand verschiedener Arbeiten werden Vorteile dynamischer Joinpointeigenschaften erläutert, gefolgt von einer näheren Betrachtung unterschiedlicher Möglichkeiten dynamische Joinpointeigenschaften zu spezifizieren. Am Ende des Kapitels wird auf spezielle Problematiken bei der Berücksichtigung von Spezifikationen dynamischer Joinpointeigenschaften bei Refactorings eingegangen, und es wird aufgezeigt, wie eine in dieser Arbeit zu entwickelnde Lösung aussehen könnte.

Kapitel 3 vertieft zunächst den Begriff der Verhaltenserhaltung und setzt diesen zur aspektorientierten Programmierung in Beziehung. Dabei wird ein Schwerpunkt auf die Behandlung dynamischer Pointcuts gelegt. Es wird dargelegt, inwiefern statische Repräsentationen dynamischer Programmeigenschaften für die Berücksichtigung dynamischer Joinpoints bei einem Refactoring sinnvolle Voraussetzung sind. Repräsentationen werden am Beispiel von zwei dynamischen Joinpointeigenschaften entwickelt. Anschließend werden Überlegungen angestellt, wie

sich Quellcodeänderungen als Teil eines Refactorings auf die Repräsentationen auswirken und ob diese Auswirkungen einen Effekt auf die Durchführbarkeit eines Refactorings haben.

In Kapitel 4 werden Verfahren erarbeitet, die den Aufbau einer spezifizierten dynamischen Joinpointeigenschaft Cflow ermöglichen. Darauf aufbauend wird ein Vergleichsverfahren von den Repräsentationen erläutert um Auswirkungen auf die spezifizierte Eigenschaft zu identifizieren.

Das Kapitel 5 gibt einen Überblick darüber, wie die Verfahren zum Aufbau und Vergleich der Repräsentation der dynamischen Joinpointeigenschaft Cflow prototypisch realisiert wurden. Die testweise Durchführung von Refactorings wird beschrieben und inwieweit das Werkzeug die richtigen oder falschen Ergebnisse ermittelt hat.

Den Abschluss der Arbeit bildet das Kapitel 6, hier werden die Ergebnisse nocheinmal zusammengefasst und Schlussfolgerungen aus den Ergebnissen gezogen. Außerdem enthält das Kapitel einen Ausblick darüber, wo Potential zur Verbesserung der Verfahren liegt und wie weiterführende Arbeiten in dieser Richtung aussehen könnten.

Anhang A beinhaltet eine Aufstellung untersuchter Algorithmen zur Erstellung von Aufrufgraphen.

Kapitel 2

Besonderheiten dynamischer Pointcuts

Das Programmiermodell der aspektorientierten Programmierung ist auf vielfältige Weise in konkreten Ansätzen realisiert worden[1]. Bei den meisten wurde eine Umsetzung in die Definition einer Sprache vorgenommen. Darüber hinaus sind diese Sprachen durch die Bereitstellung entsprechender Werkzeuge anwendbar gemacht geworden (Übersetzer und Laufzeitumgebung). Häufig ist das dazugehörige Basissprachenmodell die Objektorientierung, und oft wird auf der Basissprache Java aufgebaut.

Durch das testweise Arbeiten mit einigen der genannten Ansätze und das Untersuchen der Sprachdefinitionen konnten Erkenntnisse über Möglichkeiten in der Spezifikation von Pointcuts gewonnen werden. Die Unterteilung der verschiedenen Möglichkeiten soll später die Grundlage für eine systematische Untersuchung im Hinblick auf die Berücksichtigung von Pointcuts für Basisrefactorings bilden.

2.1 Pointcuts

In Abschnitt 1.1.2 sind Pointcuts als Sprachelemente beschrieben worden, die die Aspektbindung an Joinpoints spezifizieren. Eine Aspektsprache hält dafür eine sog. **Pointcutsprache** bereit, welche die Syntax beschreibt, mit der die Spezifikation vorgenommen werden kann.

Die in Abschnitt 1.1.2 beschriebene Verhaltensanpassung wird im Folgenden genauer anhand der Abbildung 2.1 erläutert. Dort wird symbolhaft die Auswirkung eines Aspektes durch die Nutzung eines Pointcuts und Advices dargestellt.

Im oberen Teil der Abbildung ist der Aspekt mit einem Pointcut und einem Advice als Teil des Programms zu sehen, im unteren Teil ein Modell der Ausführung des Programms. Die Instruktionen des Programms, die nacheinander ausgeführt werden, spiegeln sich in auftretenden Ereignissen wieder. Die Ereignisse sind dabei nicht zwingend atomar, das heißt, dass, während ein Ereignis noch nicht abgeschlossen ist, bereits andere Ereignisse eintreten können. Das **Joinpointmodell** einer aspektorientierten Sprache ist ausschlaggebend dafür, welche dieser

[1] vgl. Kiczales u. a. (1997) (AspectJ), Herrmann (2002) (Object Teams), Gybels u. Brichau (2003) (LMP), Eichberg u. a. (2004) (Magellan), Allan u. a. (2005) (Trace Matches), De Fraine u. a. (2005) (Stateful Aspects), Ostermann u. a. (2005) (ALPHA)

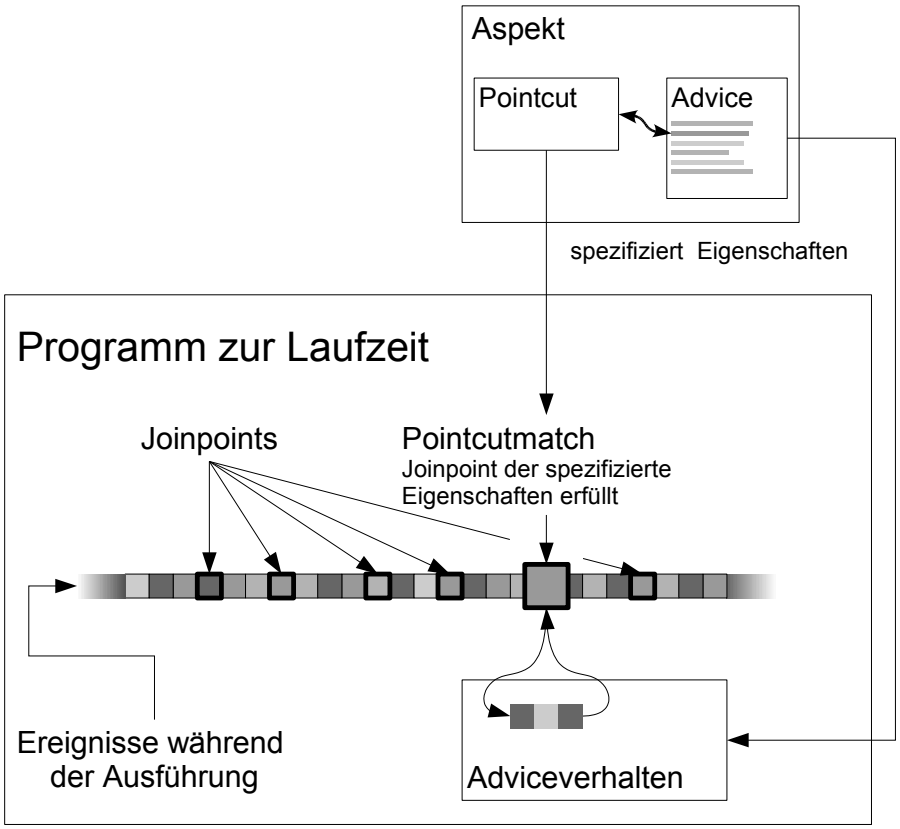


Abbildung 2.1: Aspekt, Pointcut, Joinpoint und Pointcutmatch

Ereignisse als Joinpoints behandelt werden. Gängige Beispiele für Ereignisse, die Teil eines Joinpointmodells sind, sind Methodenausführungen, Konstruktoraufrufe oder Feldzugriffe[2].

Die auftretenden Joinpoints haben bestimmte Eigenschaften, die von Pointcuts spezifiziert werden können, es handelt sich hierbei um **Joinpointeigenschaften**. Erfüllt ein als Ereignis eintretender Joinpoint alle Eigenschaften eines Pointcuts, so handelt es sich um einen **Pointcutmatch**. Wie in der Abbildung dargestellt, führt ein Pointcutmatch zu einer Verhaltensanpassung, indem das durch den Advice beschriebene Verhalten ausgeführt wird. Wenn an einem Pointcutmatch das Adviceverhalten ausgeführt wird, dann hat eine Aspektbindung stattgefunden. Die Bindung eines Aspektes bedeutet genauer, dass der Code eines Advices des Aspektes vor statt oder nach einem Pointcutmatch ausgeführt wird. Ein Advice enthält zum einen die Implementierung des auszuführenden Verhaltens (vergleichbar einem Methodenrumpf) und zum anderen direkt den Pointcut bzw. einen Verweis auf einen Pointcut. Dadurch wird eine direkte Verbindung von Pointcut und auszuführendem Verhalten erreicht.[3]

[2]Selten werden Joinpointmodelle direkt beschrieben, oft ergibt sich aus der Pointcutsprachdefinition welche Ereignisse Joinpoints sein können und somit Teil des Modells sind.

[3]Pointcuts haben in verschiedenen Aspektsprachen, neben der Aufgabe der Spezifikation von Joinpointeigenschaften, noch andere. Pointcuts können z. B. Informationen aus dem Kontext des Pointcutmatch an den Advice weitergeben. Solche Informationen sind z. B. bestimmte Objekte, die durch den Pointcut an Variablen gebunden werden können. Unter den entsprechenden Variablenbezeichnern sind sie dann im Advicerumpf einsetzbar. Diese anderen Aufgaben werden im Verlauf dieser Arbeit nicht weiter betrachtet, da sie nicht im Zusammenhang mit der Problem- und Aufgabenstellung stehen. Sie sind hier lediglich der Vollständigkeit halber exemplarisch dargestellt worden.

Bei der Untersuchung welche Eigenschaften beschrieben werden können, ist deutlich geworden, dass diese Eigenschaften einer von zwei Gruppen zuordenbar sind, entweder den Statischen oder den Dynamischen.

Statische Eigenschaften: Es ist möglich, Eigenschaften von statischen Strukturen eines Programms zu spezifizieren. Strukturen eines Programms sind statisch, wenn sie vorhanden sind, ohne dass das Programm ausgeführt wird. Statische Strukturen zeichnen sich insbesondere dadurch aus, dass sie beschreiben, *wie* ein Programm *implementiert* ist.

Beispiel: In AspectJ würde der Pointcut in Abbildung 2.2 Folgendes spezifizieren:
Ein Joinpoint ist ein Pointcutmatch, wenn es sich um die Ausführung einer Methode handelt und diese Methode den Namen `get` hat. Die Struktur, auf der die Eigenschaft beschrieben wurde, ist der Namensraum eines Programms (auch: lexikalische Struktur), die Eigenschaft ist *Methodenname*.

```
pointcut staticProperty(): execution(* *.get())

after(): staticProperty(){ //Advicebody A }
```

Abbildung 2.2: Beispiel einer Spezifikation einer statischen Eigenschaft

Dynamische Eigenschaften: Ebenso können Eigenschaften dynamischer Strukturen eines Programms spezifiziert werden. Dynamische Strukturen sind gekennzeichnet durch ihr ausschließliches Vorhandensein während der Ausführung des Programms. Sie charakterisieren während der Ausführung den Zustand des Programms und wie es sich bis zum aktuellen Zeitpunkt „verhalten“ hat.

Beispiel: Analog zu einem Pointcut, der über Eigenschaften der Implementierungsstruktur spezifiziert, wird in Abbildung 2.3 ein über eine dynamische Struktur spezifizierender Pointcut gezeigt. Auch dieser ist in AspectJ Syntax gehalten und spezifiziert Folgendes:
Ein Joinpoint ist ein Pointcutmatch, wenn der aktuelle Typ während der Ausführung einer Methode `String` ist. Alle Objekte und deren Zusammenhang bilden die dynamische Struktur (Objektgraph). Die Eigenschaft ist *Typ der `this`-Referenz*, also des Typs des Objektes, an dem die Methode aufgerufen wird[4].

```
pointcut dynamicProperty(): execution(* *.*()) && this(String)

after(): dynamicProperty(){ //Advicebody A }
```

Abbildung 2.3: Beispiel einer Spezifikation einer dynamischen Eigenschaft

Die in den Beispielen eingesetzten syntaktischen Elemente **execution**(..) und this (..) werden **Pointcutdesignatoren** genannt. Wie in dem Beispiel 2.3 zu sehen, können verschiedene Pointcutdesignatoren in einem Pointcut zu einer Spezifikation zusammengesetzt werden. Ein Joinpoint wäre z. B. dann ein Pointcutmatch, wenn er die Eigenschaft *„Methodenname ist* `get`“ und die Eigenschaft *„aktueller Typ ist* `String`“ besitzt. Üblicherweise sind Veroderung und Negation der Eigenschaften, die durch Designatoren spezifiziert werden, neben der Verundung weitere Verbindungsmöglichkeiten. Es gibt darüber hinaus Sprachansätze, die durch den Einsatz von Elementen aus der logischen Metaprogrammierung nahezu beliebige Verbindungen zwischen

[4]In Java wird der aktuelle Typ über das Schlüsselwort `this` referenziert

Spezifikationselementen zulassen (vgl. z. B. Volder u. D'Hondt (1999) oder Gybels (2001)[5]). Ein Pointcutdesignator (als einzelnes syntaktisches Element) kann aber auch mehrere Eigenschaften gleichzeitig spezifizieren.

Anhand der Art der Eigenschaften, die sie spezifizieren, werden Pointcuts folgendermaßen unterschieden:

Statische Pointcuts: Pointcuts, die *ausschließlich* Eigenschaften der Implementierungsstruktur spezifizieren, werden **statische Pointcuts** genannt.

Dynamische Pointcuts: Pointcuts, die *mindestens* eine Eigenschaft dynamischer Programmstrukturen spezifizieren, werden als **dynamische Pointcuts** bezeichnet.

2.2 Dynamische Pointcuts

Pointcutsprachen sind noch immer kein abgeschlossenes Forschungsthema, wie sich an den fortlaufenden Publikationen neuer Ansätze erkennen lässt. Im Allgemeinen lässt sich ein Trend hin zu Möglichkeiten der Spezifikation von Eigenschaften dessen, *was* ein Programm realisiert, erkennen, weg von der Spezifikation von Eigenschaften, *wie* ein Programm seine Funktionalität realisiert.

2.2.1 Motivationen für dynamische Pointcuts

Es lässt sich gut nachvollziehen, worin die Motivation für diesen Trend unter anderem liegt. Offensichtlich bringt die Nutzung von Eigenschaften der Art der Implementierung zur Spezifikation von Joinpoints einen Nachteil mit sich. Wird ein aspektorientiertes Programm weiterentwickelt und dazu z. B., wie in Abschnitt 1.2.1 erläutert, einem Refactoring zur Restrukturierung unterzogen, so hat dies für Pointcuts, die auf Implementierungsstrukturen Eigenschaften beschreiben, einen meist unerwünschten Seiteneffekt. Ein Entwickler sollte gewöhnlich davon ausgehen können, dass sich das Programm in dem, was es tut, nicht ändert, solange sich nur die Art und Weise ändert, wie diese Funktionalität implementiert ist. Dies ist für diese Art von Pointcuts aber nicht der Fall. Eine aus objektorientierter Sicht rein strukturelle Änderung, kann zu fehlerhaften zusätzlichen oder verloren gegangenen Aspektbindungen führen, welche eine Verhaltensänderung zur Folge haben kann. Für Pointcuts, die gegenüber strukturellen Änderungen in ihrem Bindungsverhalten anfällig sind, wurden in einer Veröffentlichung von Störzer u. a. (vgl. Koppen u. Störzer (2004)), die dieses Problem beschreibt, der Begriff **fragiler Pointcut** eingeführt.

Die einleuchtende Konsequenz ist, keine Eigenschaften der Implementierung für einen Pointcut zu nutzen, sondern zu spezifizieren, bei welchem Verhalten das Verhalten angepasst werden soll. Eine Spezifikation hat eine gewisse Einschränkung dieser Anforderung gegenüber, da sie Verhalten mit formalen Mitteln beschreiben muss. Formal kann Verhalten nicht unbedingt beschrieben

[5] Dort werden in den Abschnitten 3.1.1 und 3.1.2 auch die Begriffe Metaprogrammierung und logische Metaprogrammierung erläutert

werden, es lässt sich aber nachvollziehen, dass Eigenschaften dessen, was während der Ausführung eines Programms passiert, meist eine bessere Annäherung an eine Verhaltensbeschreibung sind als die Beschreibung von Eigenschaften von Implementierungsdetails[6].

Ostermann u. a. haben in diesem Zusammenhang in einer Veröffentlichung (vgl. Ostermann u. a. (2005)) ein Experiment aufgestellt, indem sie fünf verschiedene Pointcuts, die dieselben Joinpoints für eine Verhaltensanpassung spezifizieren sollten, daraufhin untersucht haben, wie sich jeweils das Bindungsverhalten ändert, wenn sich der Quellcode ändert.

Jeder der untersuchten Pointcuts sollte dieselbe Eigenschaft – die Aktualisierung der Anzeige in einer Zeichenapplikation – spezifizieren. Dabei werden von den verschiedenen Pointcuts unterschiedlich implementierungsnahe Eigenschaften für die Spezifikation genutzt: 1. eine einfache Aufzählung der syntaktischen Elemente mittels lexikalischer Struktur, 2. ein Namensmuster mit Wildcards, welches die Elemente der Aufzählung in einem Ausdruck bündelt, 3. Eigenschaften eines approximierten Kontrollflusses, 4. Eigenschaften des tatsächlichen Kontrollflusses und 5. Eigenschaften sowohl des tatsächlichen Kontrollflusses als auch von Objekten und ihrer Relation zueinander. Die beiden ersten Pointcuts beschreiben Joinpoints anhand der Implementierungsstruktur. Die restlichen drei können dagegen in ihrer Spezifikation näher an dem liegen, was mit dem Pointcut spezifiziert werden sollte: ein bestimmtes Programmverhalten. Entsprechend kommen sie in der Auswertung ihres Experiments, wie zu erwarten, zu dem Ergebnis, dass weniger implementierungsnahe Pointcuts weniger fragil bzw. robuster gegenüber Änderungen sind.

Werden neben strukturellen Änderungen auch Änderungen des Verhaltens durch Quellcodemodifikationen betrachtet, so eröffnet sich ein weiterer Vorteil von Pointcuts, die näher an einer Verhaltensbeschreibung liegen. Gybels u. a. haben in einer ihrer Arbeiten (vgl. Gybels u. Brichau (2003)) beobachtet, dass die Herstellung einer Aspektbindung bei Nutzung von Pointcuts, die Eigenschaften der Implementierung beschreiben, dazu führen kann, dass die Struktur des Programms angepasst wird, damit ein Pointcut zu richtigen Bindungen führt. Ein Programmcode, der Verhalten implementiert, welches ebenfalls mittels der Pointcuts angepasst werden soll, muss dann so geschrieben werden, dass er der Spezifikation dieser Pointcuts genügt. Pointcuts hingegen, die in ihrer Spezifikation näher an der Beschreibung des Verhaltens liegen, erfordern keinen speziell entwickelten Code.

Die Motivation, Pointcuts zu ermöglichen, die Eigenschaften dynamischer Programmstrukturen beschreiben, liegt daher in der höheren Robustheit dieser Art von Pointcuts gegenüber Änderungen.

2.2.2 Eigenschaften dynamischer Programmstrukturen

In diesem Abschnitt werden verschiedene Joinpointeigenschaften dynamischer Programmstrukturen vorgestellt. Die Untersuchung der Joinpointeigenschaften hat für die behandelten Pointcutsprachen gezeigt, auf welchen dynamischen Programmstrukturen Joinpointeigenschaften gelten können. Diese dynamischen Programmstrukturen werden näher erläutert, um anschließend auf ihnen spezifizierbare Eigenschaften zu erklären. Später werden diese dynamischen Programmstrukturen für die Aufstellung von Modellen benötigt, um Effekte von Quellcodeänderungen auf die Eigenschaften untersuchen zu können.

[6] Es gibt Ausnahmefälle, in denen sich an Implementierungsmerkmalen sehr präzise festmachen lässt, welches Verhalten vorliegt. Dazu gehören die Nutzung von Namensmustern für die Spezifikation von *JUnit* Testmethoden, die mit der Zeichenkette `test` anfangen müssen, Namen spezieller Methoden, wie `main` in einer Javaapplikation oder spezielle Methodensignaturen von *JavaBeans*

2.2.2.1 Eigenschaften des Objektgraphen

Ein Objektgraph ist eine dynamische Programmstruktur, welche die Objekte und deren Verbindungen während der Ausführung des Programms repräsentiert. Die Objekte werden dabei durch Knoten repräsentiert und die Verbindungen zwischen den Objekten (also die Referenzen) durch Kanten. Der Objektgraph ist ein gerichteter Graph, da eine Referenz zwischen zwei Objekten bedeutet, dass das eine Objekt das andere referenziert und daraus nicht die Umkehrung folgt.

Der Objektgraph kann sich während der Ausführung in zweierlei Hinsicht ändern. Zum einen in der Anzahl der Knoten und zum anderen in der Struktur, also wie die Knoten über Kanten verbunden sind. Die Anzahl der Knoten im Graphen kann sich ändern, indem neue Objekte instanziiert werden und so Knoten hinzukommen oder Objekte von keinem anderen Objekt mehr referenziert werden. Die Struktur des Graphen ändert sich hingegen, wenn zum Beispiel dem Feld eines Objektes ein anderes Objekt zugewiesen wird. Das Feld referenziert jetzt das andere Objekt, wodurch der Graph eine neue Kante erhalten hat (oder es wurde eine Kante geändert, wenn das Feld vorher ein anderes Objekt referenziert hat).

Würde die Ausführung des Programms angehalten werden, so wäre ein bestimmter Teil des Objektgraphen sichtbar. Der sichtbare Teil wird durch einen Ausführungskontext festgelegt, zum Beispiel dem Methodenrumpf, in dem sich die Ausführung gerade befindet. Dort kann auf bestimmte Variablen zugegriffen werden, wie zum Beispiel die Parameter der Methode, Felder des Objektes, an dem die Methode aufgerufen wurde, oder lokal deklarierte Variablen. Der Ausführungskontext bietet Einstiegspunkte in den Objektgraphen, indem bestimmte Knoten referenzierbar werden.

Ein Beispiel soll die Struktur eines Objektgraphen noch einmal verdeutlichen sowie den Aspekt der Veränderlichkeit darstellen und zeigen, wie entsprechende Ausführungskontexte aussehen. In den Abbildungen 2.5 bis 2.9 ist ein Objektgraph und dessen Veränderung während der Ausführung des Programms dargestellt, in Abbildung 2.4 steht der zugehörige Programmcode. Auf der linken Seite der Abbildungen ist jeweils der Ausführungskontext gezeigt, wobei der schwarze Pfeil den gerade aktuellen anzeigt.

Ausgehend von der Methode `main(String[] args)` in der Klasse `ObjectGraphExample` werden nacheinander Objekte erzeugt und durch entsprechende Methodenaufrufe miteinander in Beziehungen gesetzt. In Zeile 6 wird zunächst eine Liste instanziiert. In den nächsten drei Zeilen werden dann die Objekte a, b und c instanziiert (bis Zeile 10). Abbildung 2.5 zeigt den Objektgraphen nach diesen Instanziierungen. Es lässt sich erkennen, dass bei der Instanziierung des Objektes vom Typ `MyLinkedList` ein weiteres Objekt mitinstanziiert wurde und eine Kante von der Liste zu einem Objekt vom Typ `Entry` erstellt wurde.

In Abbildung 2.6 ist der Zeitpunkt vor dem Rücksprung von Methode `add(Object o)` zurück in Methode `main(String[] args)` zu sehen, der Ausführungskontext ist also noch der der Methode `add(Object o)`. Durch die Zuweisungen, die in `add(Object o)` stattgefunden haben, wurde der Graph um einen Knoten, das Objekt *elem2*, erweitert und es wurden sowohl Kanten geändert als auch neue hinzugefügt. Die Änderungen an den Kanten, zusammen mit dem Hinzufügen eines neuen Knotens, haben dafür gesorgt, dass das Objekt *a* Teil der Liste geworden ist. Die nächste Abbildung 2.7 zeigt den Zustand des Graphen, nachdem drei weitere Aufrufe von `add(Object o)` stattgefunden haben und die Ausführung wieder in Methode `main(String[] args)` angelangt ist (Zeile 14 im Programmcode). Die Elemente in der Liste sind jetzt nicht mehr direkt im Ausführungskontext sichtbar, sondern nur noch die Liste selber und die in der Liste gespeicherten Objekte;

```
public class MyLinkedList{
  private Entry header =
    new Entry(null, null, null);

  public MyLinkedList(){
    header.next = header;
    header.previous = header;
  }

  public void add(Object o){
    Entry newEntry =
      new Entry(o,
      header,
      header.previous);
    newEntry.previous.next =
      newEntry;
    newEntry.next.previous =
      newEntry;
  }

  public boolean remove(Object o){
    for (Entry e = header.next;
      e != header;
      e = e.next)
    {
      if (o.equals(e.element)){
        remove(e);
        return true;
      }
    }
    return false;
  }
  private void remove(Entry e) {
    e.previous.next = e.next;
    e.next.previous = e.previous;
  }
}
```

(a)

```
public class Entry{
  Object element;
  Entry next;
  Entry previous;

  Entry(Object element,
      Entry next,
      Entry previous)
  {
      this.element = element;
      this.next = next;
      this.previous = previous;
  }
}
```

(b)

```
1  public class ObjectGraphExample
2  {
3    public static void
4      main(String[] args)
5    {
6      MyLinkedList list =
7        new MyLinkedList();
8      Object a = new Object();
9      Object b = new Object();
10     Object c = new Object();
11     list.add(a);
12     list.add(b);
13     list.add(c);
14
15     list.remove(c);
16   }
17 }
```

(c)

Abbildung 2.4: Programmcode für das Objektgraph Beispiel

Abbildungen 2.8 und 2.9 zeigen, wie ein Knoten aus dem Objektgraph gänzlich verschwinden kann. Abbildung 2.8 zeigt zunächst, wie die Methode `remove(Object o)` Kanten aus dem Graphen entfernt, sodass das Objekt *elem4* nur noch aus dem Aufrufkontext von `remove(Object o)` zu erreichen ist. Durch den Rücksprung in die Methode `main(String[] args)` existiert der Aufrufkontext von `remove(Object o)` als einziger Halter einer Referenz auf *elem4* nicht mehr, und damit ist *elem4* nicht mehr Teil des Aufrufgraphen.

Ein Joinpoint als ein Punkt in der Ausführung des Programms kann Joinpointeigenschaften hinsichtlich des Objektgraphen anhand seines Ausführungskontextes aufweisen. In den untersuchten Pointcutsprachen sind dabei nur bestimmte Variablen aus dem Ausführungskontext verfügbar. Zu den spezifizierbaren Variablen gehören das aktuelle Objekt, die Parameter einer Methode, das Objekt, an dem eine Methode aufgerufen wird bzw. das Objekt, dessen Feld gelesen oder geschrieben wird, das Rückgabeobjekt einer Methode und global zugreifbare Felder[7]. Joinpointeigenschaften des Objektgraphen werden entsprechend über diese Variablen beschrieben.

[7]In Java sind das Felder, die die Modifier `public static` tragen

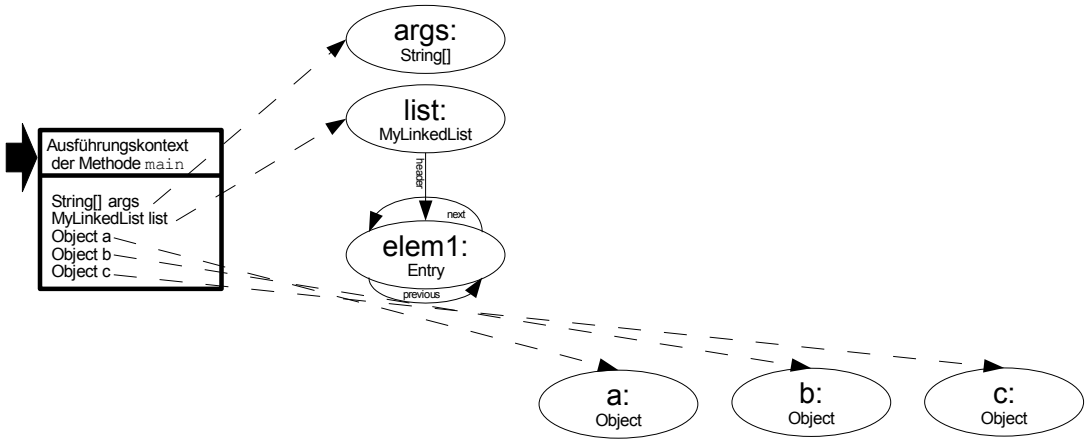


Abbildung 2.5: Objektgraph nach der Ausführung aller Instanziierungen in der Methode `main`

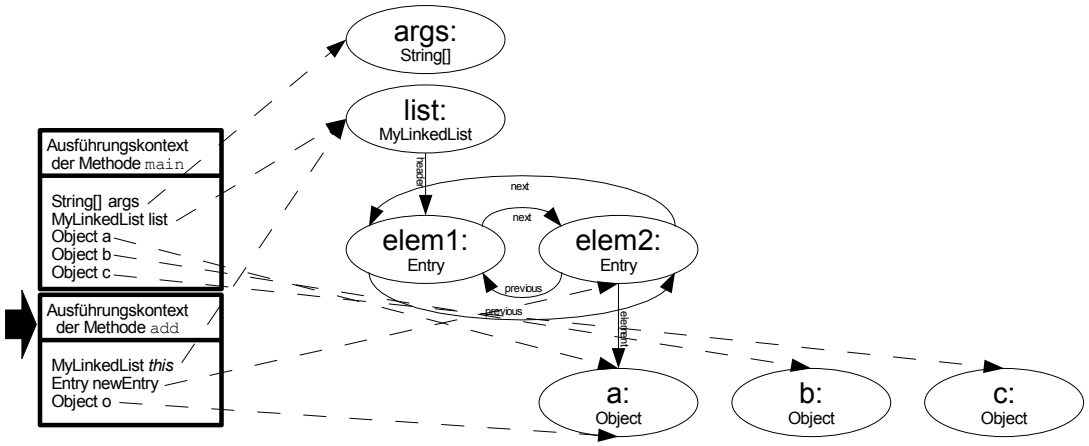


Abbildung 2.6: Objektgraph vor Beendigung des ersten Aufrufs der Methode `add(Object o)`

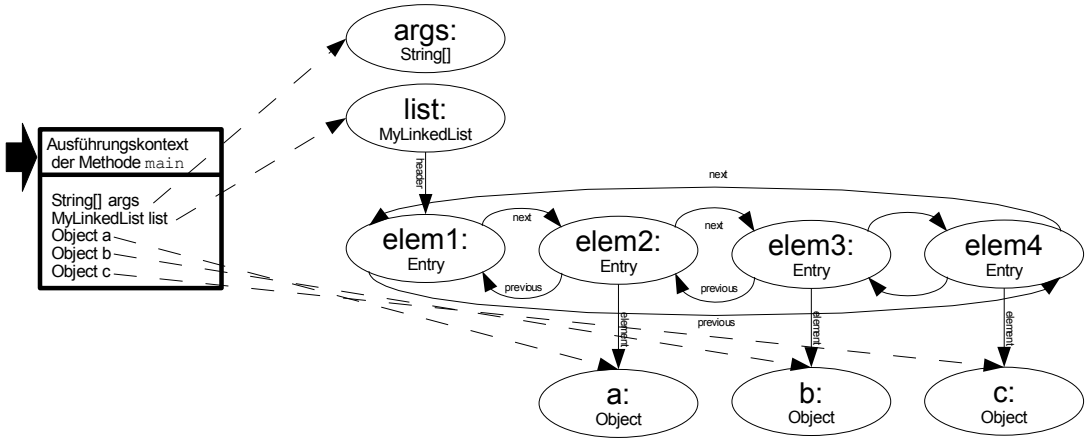


Abbildung 2.7: Objektgraph nach Beendigung des letzten Aufrufs der Methodes `add(Object o)`

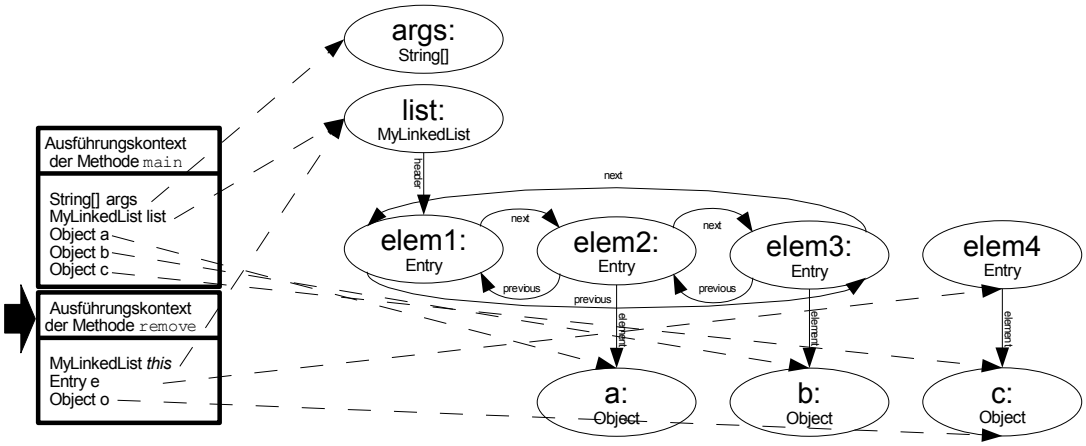


Abbildung 2.8: Objektgraph vor Beendigung des Aufrufs der Methode `remove(Object o)`

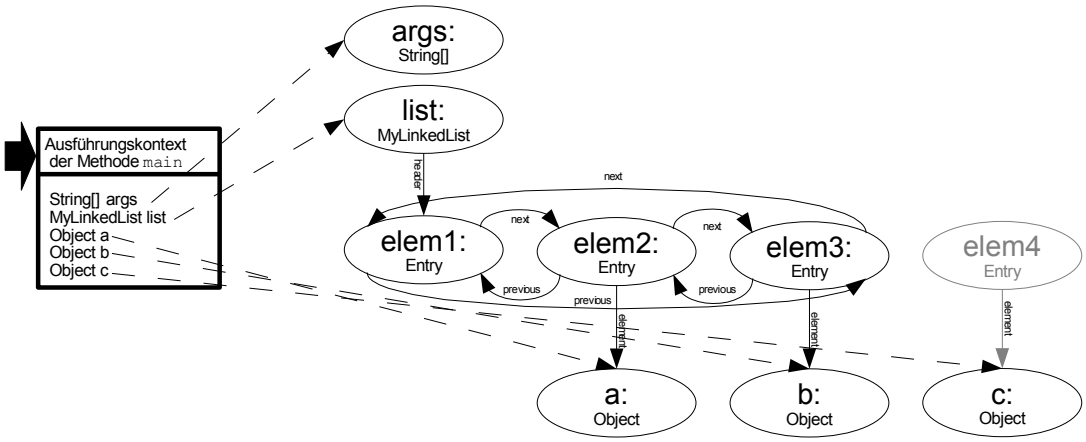


Abbildung 2.9: Objektgraph nach Beendigung des Aufrufs der Methode `remove(Object o)`, es gibt keinen Ausführungskontext, der noch eine Referenz auf das Object elem4 hat.

Typübereinstimmung eines Knotens

In den meisten Pointcutsprachen kann für einen Joinpoint gefordert werden, dass ein spezifizierter Knoten in seinem Ausführungskontext einen bestimmten Typ oder auch diesen Typ oder Subtypen davon haben muss. Die meisten Sprachen erlauben es, dies für direkt verfügbare Knoten des Ausführungskontextes zu fordern (z. B. aktuelles Objekt, bestimmter Parameter usw.) andere, wie zum Beispiel *ALPHA*, lassen es auch für andere spezifizierte Knoten des Objektgraphen zu. Der Pointcut in Abbildung 2.3 aus Abschnitt 2.1 ist ein Beispiel für eine Typübereinstimmung als geforderte Joinpointeigenschaft.

Identität von Knoten

Eine weitere spezifizierbare Joinpointeigenschaft auf dem Objektgraphen ist die Eigenschaft der Übereinstimmung zweier (durch andere Eigenschaften) spezifizierter Knoten. In Abbildung 2.10 ist ein solches Beispiel gezeigt, welches der Veröffentlichung Gybels u. Brichau (2003) entnommen ist. In diesem Beispiel sind die Knoten als zwei aufeinander folgende Argumente einer ansonsten beliebigen Ausführung einer Methode an einem beliebigen Objekt spezifiziert worden. Die Identität wird über den Mechanismus der Unifikation gefordert (Beide Argumente werden mit demselben Bezeichner `?arg` versehen).

```
?jp matching
  reception(?jp, ?selector, <?arg, ?arg>)
```

Abbildung 2.10: Identität zweier Knoten im Objektgraphen

Erreichbarkeit von Knoten im Objektgraph

Eine neuere Spezifikationsmöglichkeit auf dem Objektgraphen ist unter anderem mit *ALPHA* eingeführt worden (vgl. Ostermann u. a. (2005)). Als Eigenschaft, die ein Joinpoint erfüllen muss, kann spezifiziert werden, dass ein spezifizierter Knoten von einem anderen spezifizierten Knoten aus im Objektgraphen erreichbar ist. In Abbildung 2.11 ist ein solches Beispiel dargestellt. Der eine Knoten wird als Empfängertyp, an dem eine Variablenzuweisung stattfindet, spezifiziert, der andere muss die weiter oben beschriebene Eigenschaft der Typübereinstimmung erfüllen. Die Eigenschaft der Erreichbarkeit zwischen zwei Knoten im Objektgraphen wird nun für diese Knoten im Graphen gefordert. Sobald also irgendeine Feldzuweisung stattfindet und ein Objekt vom Typ `String` vom Knoten, an dem gerade das Feld zugewiesen wird, erreichbar ist, handelt es sich bei dem Joinpoint um einen Pointcutmatch. Auch hier wird Unifikation über die Variablen `Q` und `P` eingesetzt, um die Eigenschaft zu spezifizieren.

```
class Exmpl extends Object{
  after set(P, F, _), reachable(Q, P), instanceof(Q, 'String')
  {
    /* advicebody: do something */
  }
}
```

Abbildung 2.11: Erreichbarkeit zweier Knoten im Objektgraphen

Wertebasierte Bedingungen

Der Pointcut in Abbildung 2.12 zeigt, wie eine wertebasierte Bedingungen als Eigenschaft auf einem Knoten des Objektgraphen, spezifiziert werden kann. Das Beispiel ist ein Pointcut in der Syntax von AspectJ. Für die Spezifikation des Knotens wird eine Verundung eines `execution`-Designators und eines `args`-Designators genutzt. Der Knoten muss also ein Argument einer Methode sein und den Typ `Withdrawal` haben. Die geforderte Eigenschaft, die hier dargestellt werden soll, ist die durch den `if`-Designator beschriebene Anforderung, dass das Feld `value` des Objektgraphknotens `amount` größer als 1000 sein soll. Sobald also eine Methode `withdraw` mit einer Auszahlung, die einen höheren Wert als 1000 hat, ausgeführt wird, findet eine Aspektbindung statt und der Advice wird ausgeführt.

```
aspect AnAspect {
  before(Withdrawal amount) : if(amount.value > 1000.0)
  && execution(void *.withdraw(Withdrawal)) && args(amount)
  {
    /* advicebody: since this is much money make a more intense check*/
  }
}
```

Abbildung 2.12: Bedingung an ein Feld eines Knoten des Objektgraphen

2.2.2.2 Eigenschaften der Struktur Programmablauf

In diesem Abschnitt werden Eigenschaften der dynamischen Struktur Programmablauf beschrieben. Der Programmablauf (engl. *execution history* oder *execution trace*) kann als Sequenz von Ereignissen, die bis zum betrachteten Zeitpunkt eingetreten sind, gesehen werden. Die Ereignisse im Programmablauf sind auf verschiedenen Ebenen abstrahierbar. Die unterste Ebene wäre die Ausführung einzelner Bytecodeinstruktionen auf Maschinenebene bzw. im Falle von Java auf virtueller Maschinenebene. Eine geeignete Abstraktionsebene für Joinpointeigenschaften des Programmablaufs, sind offensichtlich die Punkte in der Ausführung, bei denen es sich um einen Joinpointeintritt oder einen Joinpointaustritt handelt. So ist es möglich die Ereignisse sequentiell einzuordnen. Abbildung 2.13 zeigt eine solche Sequenz an einem kleinen Beispiel, für eine der möglichen Ausführungen des Programms.

Die Eigenschaft Cflow

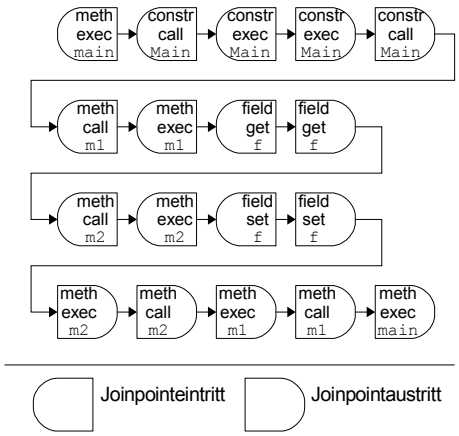
Eine der ersten Möglichkeiten, eine Eigenschaft des Programmablaufs zu beschreiben, war die Spezifizierbarkeit der Eigenschaft, dass ein Joinpoint im Kontrollfluss einer Methode liegt. Diese Spezifikationsmöglichkeit wurde mit dem `cflow` Pointcutdesignator in AspectJ eingeführt.

Anhand des folgenden Beispiels soll diese Eigenschaft genauer beschrieben werden. Als Grundlage dient das Programm 2.14a und der Pointcut 2.15, der der AspectJ Syntax entspricht: In 2.14b sind die Kontrollflussverhältnisse der einzelnen Methoden abgebildet. An den Kontrollflussverhältnissen lässt sich erkennen, welche Methoden im Kontrollfluss welcher anderen Methoden liegen.

Der Pointcut spezifiziert, dass ein Joinpoint dann ein Pointcutmatch ist, wenn es sich um die Ausführung einer Methode mit der Signatur **void** Example.m9() handelt *und* sich diese Methodenausführung im Kontrollfluss einer Methode mit der Signatur **void** Example.m3() befindet.

```
public class Main{
  private int f;

  public static void main(String[] args){
    boolean runtimeValue = args == null
      ? true
      : (args.length % 2) == 0;

    Main o = new Main();
    if(runtimeValue){
      o.m1();
    } else {
      o.m2();
    }
  }
  private void m1(){
    int i = f;
    m2();
  }
  private void m2(){
    f = 5;
  }
}
```

(a)



(b)

Abbildung 2.13: Ein Beispielprogramm (a) und ein Programmablauf (b). Die Art des Joinpoints ist in (b) jeweils oben in das Symbol eingetragen

In Abbildung 2.14c ist dargestellt, wie verschiedene Ausführungen derselben Methode `m9()` zu Joinpoints mit verschiedenen Eigenschaften führen. Zuerst liegt die Ausführung der Methode im Kontrollfluss der Methode `m3()`, und der Joinpoint ist ein Pointcutmatch, im folgenden Programmverlauf liegt eine weitere Ausführung der Methode `m9()` nicht mehr im Kontrollfluss der Methode `m3()`, und der Joinpoint ist bezüglich des Pointcuts kein Pointcutmatch mehr. Genauso wie gefordert werden kann, dass ein Joinpoint im Kontrollfluss liegen soll, besteht ebenso die Möglichkeit, die Negation zu spezifizieren: Ein Joinpoint darf *nicht* im Kontrollfluss einer Methode liegen.

Einige Pointcutsprachen, wie zum Beispiel die von AspectJ, erfordern einen aufmerksamen Umgang in der Spezifikation dieser Eigenschaft. Dort kann der Joinpoint, in dessen Kontrollfluss ein anderer liegen soll, auf beliebige Joinpoints bezogen sein. Für die Joinpointart Feldzuweisung macht das keinen Sinn, da dort kein Kontrollfluss entsteht.

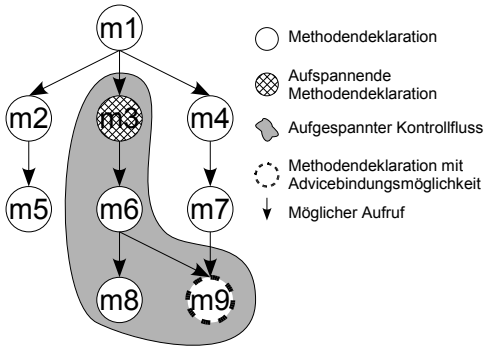
Die Eigenschaft Ereignissequenz

Pointcuts zur Spezifikation von Ereignissequenzen wurden in diversen Sprachansätzen eingeführt. Beispielhaft seien eine Erweiterung der Sprache AspectJ von Allan et al. mit dem Namen *tracematches* und eine Erweiterung der Sprache JAsCo von De Fraine et al. unter dem Namen *Stateful Aspects* erwähnt (vgl. Allan u. a. (2005), De Fraine u. a. (2005)). Beide beziehen sich auf ein formales Modell von Douence et al. (vgl. Douence u. a. (2001)).

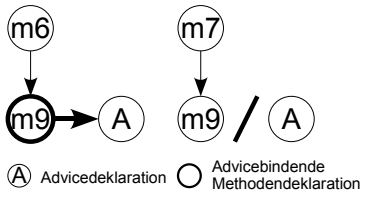
Ein Beispiel in der Sprache *tracematches* soll diese Form der Spezifikation verdeutlichen. Grundlage ist wieder das Programm aus Abbildung 2.14a, welches bei Ausführung den in Abbildung 2.16 dargestellten Programmablauf aufweist. In diesem Programmablauf sind die einzelnen Ereignisse Eintritte in und Austritte aus Methoden. Daneben können auch Eintritt und Austritt aus allen anderen Joinpoints Ereignisse sein.

```
public class Example {
  public static void
    main(String[] args) {m1();}

  public static void m1() {
    m2();
    m3();
    m4();
  }

  public static void m2() {m5();}

  public static void m3() {m6();}

  public static void m4() {m7();}

  public static void m5() {}

  public static void m6() {
      m8();
      m9();
  }

  public static void m7() {m9();}

  public static void m8() {}
  public static void m9() {}
}
```

(a)



(b)



(c)

Abbildung 2.14: Ein Beispielprogramm (a) mit seinen Kontrollflussverhältnissen(b) und unterschiedlichen Advicebindungen (c)

```
pointcut examplePC():
  cflow(execution(void Example.m3())) && execution(void Example.m9())
after(): examplePC()
{
  //Advicebody A
}
```

Abbildung 2.15: Beispiel eines cflow Pointcuts

Mit der Spezifikation einer Ereignissequenz kann für einen Pointcutmatch gefordert werden, dass im Programmablauf vor dem Auftreten des Pointcutmatches ein bestimmtes Muster von Ereignissen eingetreten ist. Im Beispiel sollen zuerst die Ereignisse *Eintritt in Methode* `m5()` und *Eintritt in die Methode* `m3()` aufgetreten sein. Wenn dann das Ereignis *Eintritt in die Methode* `m9()` auftritt soll der Pointcutmatch vorliegen. In der Syntax von *tracematches* sähe das dann wie in Abbildung 2.17 aus.

Abbildung 2.16 zeigt, dass das Ereignis *Eintritt in Methode* `m9()` zwar an verschiedenen Stellen vorkommt, aber ein Pointcutmatch nur an der Stelle im Programmablauf vorliegt (hervorgehoben), vor der das spezifizierte Ereignismuster eingetreten ist. Ein weiteres Auftreten des Ereignisses *Eintritt in Methode* `m9()` führt nicht zu einem Pointcutmatch, das Muster dafür müsste so aussehen: `entM5 entM3 entM9 entM9`.

Die Definition einer Sequenz erfolgt in der Sprache *tracematches* in zwei Teilen. Einerseits werden sog. Symbole definiert, andererseits wird aus diesen Symbolen ein regulärer Ausdruck gebildet, der mit den im Programmablauf auftretenden Ereignissen verglichen wird. Wenn, wie oben beschrieben, das letzte Symbol des regulären Ausdruckes im Programmablauf auftritt, liegt ein

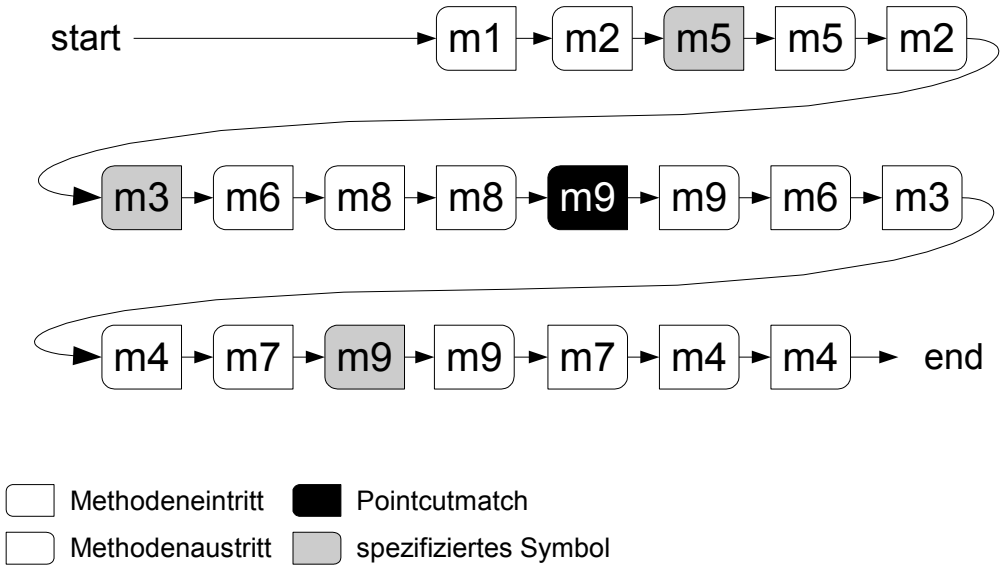


Abbildung 2.16: Eine Programmausführung zum Beispielprogramm aus Abbildung 2.14a

```
tracematch() {
  sym entM5 before : call(void Example.m5());
  sym entM3 before : call(void Example.m3());
  sym entM9 before : call(void Example.m9());

  entM5 entM3 entM9

  { /* Advicebody A */}
}
```

Abbildung 2.17: Ein Beispiel für die Spezifikation einer Ereignissequenz in tracematches

Pointcutmatch vor. Symbole werden mit Pointcuts und der `before`, `after` Anweisung beschrieben, nebenbei entfällt dadurch diese Angabe im Adviceteil des Aspektes. Die Angabe von `before`, `after` für die Symbole ist notwendig, damit es sich um atomare Ereignisse handelt, die zeitlich angeordnet werden können[8].

Stateful Aspects in JAsCo unterscheiden sich zwar in der Syntax, formal gesehen bestehen aber nahezu dieselben Spezifikationsmöglichkeiten[9]. In einem JAsCo *Stateful Aspects* Pointcut werden Zustände und Zustandsübergänge beschrieben, wodurch ein Zustandsautomat definiert wird. Die formale Gleichheit der Ansätze besteht darin, dass jeder reguläre Ausdruck in einen entsprechenden Zustandsautomaten transformiert werden kann und umgekehrt jeder Zustandsautomat äquivalent zu einem regulären Ausdruck ist.

Beide Ansätze kennen zwei Interpretationen einer Spezifikation: So wie sie im obigen Beispiel geschildert wurde, wo das zweite Auftreten von *Eintritt in die Methode* `m9()` nicht zu einem

[8]Das letzte Symbol des regulären Ausdrucks darf auch mit around versehen werden, die spezielleren Formen von after sind auch zulässig (after returning, after throwing)

[9]Unterschiede sind: 1. Die unterliegenden Pointcutsprache, die nicht AspectJ sondern JaSCo entstammt 2. In tracematches kommen bei einem *Oder* (|) beide Seiten als nächster Zustand gleichberechtigt in Betracht, bei JAsCo hingegen wird der linke bevorzugt (wirkt sich nur bei Variablenbindungen aus, siehe 2.2.2.3) 3. In Stateful Aspects kann jeder Zustand auch als Pointcutmatch spezifiziert werden

```
tracematch (Subject s, Observer o) {
  sym create_observer after returning(o):
      call(Observer.new(..))
   && args(s);
  sym update_subject after:
      call(* Subject.update(..))
   && target(s);

  create_observer update_subject*
  {
    o.update_view();
  }
}
```

Abbildung 2.18: Implementierung des Updatemechanismus des Observerpatterns durch eine Variablenbindung in *tracematches*

Pointcutmatch führt. Alternativ ist auch eine Interpretation möglich, in der spezifizierte Symbole ignoriert werden können. Für das Beispiel würde das bedeuten, dass für beide Ereignisse *Eintritt in die Methode* `m9()` ein Pointcutmatch vorliegen würde.

2.2.2.3 Eigenschaften verschiedener Strukturen

Gerade im Bereich der Pointcutsprachen, die auf logischer Programmierung basieren, können Eigenschaften spezifiziert werden, die von Objektgraph und Programmablauf gleichermaßen abhängen. Aber auch die Sprachansätze *Stateful Aspects* und *tracematches* erlauben es, Eigenschaften zu spezifizieren, die auf den kombinierten dynamischen Strukturen gelten.

Unifikation
Mit den Pointcutsprachen von *ALPHA* und einer von Gybels et. al beschriebenen Sprache ist die Nutzung einer logischen Programmiersprache als Pointcutsprache eingeführt worden. Die Sprachen leiten sich im Grunde genommen aus dem Bereich der logischen Metaprogrammierung ab, reduzieren sich aber auf die Pointcutspezifikation im aspektorientierten Umfeld.

Die Sprachen stellen jeweils statische und dynamische Strukturen als Datenbasis zur Verfügung, von denen die dynamischen im Programmverlauf ständig aktualisiert werden. Im Pointcut stehen prologähnliche Terme über die Datenbasis, die, wenn sie am aktuellen Punkt in der Programmausführung zu wahr ausgewertet werden können, zu einer Aspektbindung führen. In den dazugehörigen Publikationen wird dies ausführlicher diskutiert und erklärt (ebenso Ostermann u. a. (2005) und Gybels u. Brichau (2003) sowie Gybels (2002) und Gybels (2001)).

Variablenbindungen
Eine Form der Unifikation gibt es auch in *Stateful Aspects* und *tracematches*. Hier können zusätzlich zu Ereignissequenzen noch die Identität zwischen Elementen von verschiedenen Objektgraphen gefordert werden. Verschiedene Objektgraphen ergeben sich zu verschiedenen Punkten im Programmablauf. In der Publikation für *tracematches* haben Allan et al. auch diese Spezifikationsmöglichkeit eingeführt und unter anderem das in Abbildung 2.18 dargestellte Beispiel zur Implementierung des Updatemechanismus des Observerpatterns[10] gegeben. Hier wird die Identi-

[10] Programmbeispiel aus Allan u. a. (2005), Begriff *Observerpattern* nach (Gamma u. a. 1995, Seiten 239 ff.)

tät des *Subject* Objektes im Observerpattern, zum Zeitpunkt der Instanziierung eines Observers, mit einem Objekt während eines Updatevorganges gefordert.

2.3 Refactoring und Pointcuts

In Abschnitt 1.3.1 ist die Berücksichtigung von Aspektbindungen für Basisrefactorings als übergeordnetes Ziel gesetzt worden. In demselben Abschnitt wurde erläutert, dass die Erhaltung der Aspektbindung bei Quellcodeänderungen Voraussetzung für die Verhaltenserhaltung ist. Um Aspektbindungen zu erhalten sind zwei Hilfestellungen identifiziert worden, die einen Entwickler dabei unterstützen: zum einen die Identifizierung von den von der Änderung betroffenen Pointcuts und zum anderen das Zur-Verfügung-Stellen von Informationen darüber, wie sich die Aspektbindung geändert hat.

Am Anfang dieses Kapitels wurden Aspektbindungen zwischen Joinpoints und Advicerümpfen mittels Pointcuts genauer erklärt. Welche Strukturen zur Spezifikation von Eigenschaften in gängigen Pointcutsprachen zum Einsatz kommen und insbesondere welche dynamischen Eigenschaften spezifiziert werden können, ist in den vorherigen Abschnitten beschrieben worden. Im Rahmen erster Überlegungen, wie sich durch Refactorings hervorgerufene Quellcodeänderungen auf Aspektbindungen auswirken, sind Unterschiede zwischen der Berücksichtigung statischer und dynamischer Pointcuts aufgefallen.

2.3.1 Berücksichtigung statischer Pointcuts

Eine Aspektbindung durch einen statischen Pointcut bei einer Quellcodeänderung zu berücksichtigen ist nicht in jedem Fall eine triviale Aufgabe. Dennoch wird, bei einer überschaubaren Menge von statischen Pointcuts, welche nicht besonders komplex aufgebaut sind, und bei einer Software von gleichfalls überschaubarer Komplexität ein Entwickler sogar ohne Werkzeugunterstützung in der Lage sein, bei einfachen Quellcodeänderungen herauszufinden, ob Aspektbindungen von einer Änderung betroffen sind.

Dies liegt darin begründet, dass die in einem statischen Pointcut spezifizierten Eigenschaften, wie beschrieben, ausschließlich statische Strukturen betreffen, deren Erstellung und Umgang für einen Entwickler einen zentralen Bestandteil seiner Arbeit bilden. Für ein von einer Änderung betroffenes Programmelement, werden dadurch betroffene Strukturen bekannt sein, die wiederum mit den Pointcuts verglichen werden, um herauszufinden, ob betroffene Strukturen spezifiziert wurden. Ist das der Fall, ist auch der Pointcut betroffen und damit die Aspektbindung. In diesem Szenario wird ein Entwickler auch in der Lage sein zu erkennen, wie er die betroffenen Pointcuts anpassen muss, um das Verhalten zu erhalten.

Bei aspektorientierten Softwareprojekten größeren Umfangs und höherer Komplexität, die viele Aspekte mit vielen statischen Pointcuts enthalten, steigt der Anspruch an den Entwickler, die Zusammenhänge zu verstehen. Wie es schon für nicht aspektorientiertes Refactoring üblich ist, wird ab einem Komplexitätsgrad, den der Entwickler nicht mehr zu bewältigen in der Lage ist, eine Unterstützung durch ein Werkzeug erforderlich sein.

Ein Werkzeug für die Identifizierung der von Änderungen betroffenen statischen Pointcuts zu entwickeln sollte eine zu bewältigende Aufgabe sein. Vorhandene Refactoringwerkzeuge nutzen bereits verschiedene statische Strukturen eines Programms (z. B. den Abstrakten Syntax Baum

oder Vererbungshierarchien), um den Effekt von Quellcodeänderungen auf diese Strukturen zu ermitteln und diesen bezüglich der Bedingungen eines Refactoringpatterns zu prüfen. Die Verbindung eines statischen Pointcuts zu statische Strukturen und Elemente darin herzustellen, wird ebenso unproblematisch sein. Damit könnte ein Werkzeug diese Verbindung herstellen zwischen Refactoring, Quellcodeänderung, betroffener Programmstruktur, betroffenem Element in der Struktur und betroffenen Pointcuts, die dieses Element spezifiziert haben.

In Abbildung 2.19 sind beispielhaft zwei solche Verbindungen dargestellt. Zentral sind symbolhaft statische Strukturen abgebildet. Die Pfeile, die von der Quellcodeänderung ausgehen, zeigen welche Elemente der jeweiligen Struktur betroffen sind. Umgekehrt sind auch die Verbindungen der Pointcuts zu den Elementen der Struktur durch Pfeile aufgezeigt. Die dunkel markierten Elemente in den Strukturen sind diejenigen, die einerseits von Änderungen betroffen sind und andererseits von Pointcuts referenziert werden. Die gestrichelten Pfeile vom Refactoring zu einem Pointcut zeigen, wie über die markierten Elemente eine Verbindung von einem Refactoring zu den betroffenen Pointcuts aufgebaut werden kann.

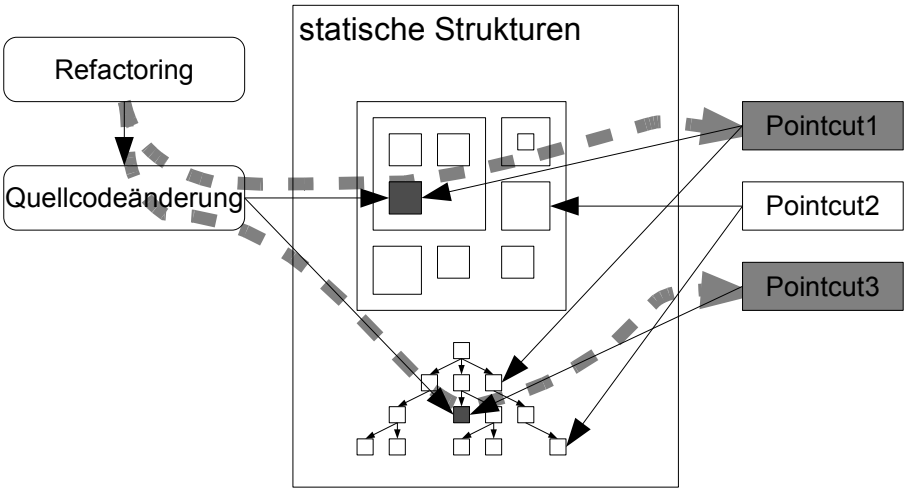


Abbildung 2.19: Identifizierungskette von einem Refactoring zu betroffenen Pointcuts

2.3.2 Berücksichtigung dynamischer Pointcuts

Die Verbindung von einer Quellcodeänderung zu einem dynamischen Pointcut aufzubauen stellt sowohl einen Entwickler als auch ein Werkzeug vor ein wesentliches Problem. Während statische Pointcuts sich auf statische Strukturen beziehen, die entweder vom Entwickler überschaut oder von einem Werkzeug genau berechnet werden können, sind dynamische Strukturen in den wenigsten Fällen einem Entwickler präsent und für ein Werkzeug prinzipbedingt nicht beliebig präzise berechenbar. Das ist dem Umstand zu verdanken, dass dynamische Strukturen, wie sich später zeigen wird, beim Versuch, sie zu berechnen, sehr schnell sehr komplex werden. Das führt dann dazu, dass von Änderungen betroffene Programmelemente nicht mit Pointcuts in Zusammenhang gebracht werden können.

Um diesem Problem zu begegnen, soll ein Mittelweg untersucht werden, der zwar keine definitiven Aussagen darüber machen kann, ob ein Pointcut betroffen ist oder nicht, der aber mit

unscharfen Aussagen genügend Anhaltspunkte zum vernünftigen Umgang mit Quellcodeänderungen liefern sollte. Für diesen Mittelweg sollen dynamische Strukturen konservativ angenähert werden. Das bedeutet, dass die eigentlichen Strukturen nicht präzise berechnet werden, sondern dass Strukturen angenähert werden, die *mindestens* alle möglichen Elemente der Struktur enthalten. Für Pointcuts lassen sich dann bei Quellcodeänderungen Aussagen in folgender Form machen: ist definitiv betroffen, ist vielleicht betroffen und ist auf keinen Fall betroffen. Außerdem können dem Entwickler aus den angenäherten Strukturen Informationen bereitgestellt werden, anhand derer er entscheiden kann, ob er eine Änderung durchführen möchte oder nicht, oder es wird ihm mittels der angenäherten dynamischen Strukturen eine Vorstellung vermittelt, wie ein Pointcut angepasst werden muss.

Kapitel 3

Auswirkungen von Codeänderungen auf dynamische Joinpointeigenschaften

Mit seiner Arbeit *Refactoring Object-Oriented Frameworks* (vgl. Opdyke 1992) war Opdyke der erste, der Refactoring zum Thema einer wissenschaftlichen Arbeit gemacht hat. Dort hat er zentrale Vorgehensweisen und Bedingungen für das Refaktorisieren objektorientierter Programme erarbeitet und insbesondere die Grundlagen für automatisiertes, werkzeuggestütztes Refactoring geschaffen. Von besonderer Bedeutung ist die von ihm formulierte Definition der Erhaltung von Verhalten bei Umstrukturierungen von Programmen. Seine Überlegungen bilden die Grundlage für die folgenden Betrachtungen zum Thema Verhaltenserhaltung, das für Basisrefactorings aspektorientierter Programme (siehe Abschnitt 1.3.1) ebenso eine zentrale Rolle spielt. In Bezug auf Aspektbindungen von Advices, Pointcuts und Joinpoints wird anhand des Begriffes der semantischen Äquivalenz von Referenzen und Operationen eine Erweiterung vorgenommen. Diese Erweiterung wird im Speziellen für dynamische Pointcuts betrachtet und beispielhaft erarbeitet.

3.1 Verhaltenserhaltung

Opdyke identifizierte eine Reihe spezieller Programmeigenschaften, die von Quellcodeänderungen im Rahmen eines Refactorings leicht verletzt werden können. Das Besondere an diesen Programmeigenschaften ist, dass ihre Verletzung ein Hinweis darauf ist, dass das Programmverhalten sich geändert hat.

Die Programmeigenschaften sind zum einen eine Menge von syntaktischen Eigenschaften, die ausschlaggebend dafür sind, dass das Programm der Sprachdefinition gerecht wird, und zum anderen sog. semantische Programmeigenschaften. Opdyke hat die Eigenschaften für die Programmiersprache C++ identifiziert, welche sich auf die für diese Arbeit betrachtete Sprache Java leicht adaptieren lassen. In einer Arbeit von Rura, welche sich mit dem Refactoring von AspectJ Programmen auseinandersetzt, ist bereits eine Anpassung dieser Programmeigenschaften auf Java vorgenommen worden (vgl. Rura 2003, Abschnitt 2.2.1 bis 2.2.3)[1].

[1] Die Eigenschaften werden zwar für AspectJ insgesamt und nicht für Java im Speziellen beschrieben, die für Java relevanten Teile lassen sich aber leicht herauslösen.

3.1.1 Anforderungen der Sprachdefinition

Offensichtlich ist, dass ein Programm nur dann refaktorisiert werden kann, wenn es davor mittels eines Übersetzers in einen ausführbaren Zustand versetzbar ist. Ebenso einleuchtend ist, dass die Quellcodeänderungen eines Refactorings nur dann zulässig waren, wenn danach ein ebenfalls übersetzbares Programm vorliegt. Bei den ersten sechs der für C++ identifizierten sieben Programmeigenschaften handelt es sich um die syntaktischen Eigenschaften, also genau diejenigen, deren Nichtverletzung zu einem übersetzbaren Programm führen.

3.1.1.1 Anforderungen der Sprachdefinition von Java

Die syntaktischen Eigenschaften ergeben sich aus der jeweiligen Sprachdefinition, in diesem Fall also zunächst für Java. Die hier vorgestellten Definitionen sind der oben erwähnten Arbeit von Rura entnommen (vgl. Rura 2003, Abschnitt 2.2.1).

1. Jede Klasse muss eine eindeutige Superklasse haben.
2. Jeder Typ (Klasse oder Interface) muss einen eindeutigen Namen haben.
3. Jede Variable muss einen eineindeutigen Namen in ihrem Gültigkeitsbereich haben.
4. Jede Methode eines Typs muss eine eineindeutige Signatur haben.
5. Das Programm muss typsicher sein.
6. Geerbte Felder dürfen nicht überschrieben werden.
7. Regeln für `extends` und `implements`
   a) Zwischen zwei Interfaces darf eine Vererbungsbeziehung nur mittels `extends` hergestellt werden (nicht durch `implements`).
   b) Zwischen zwei Klassen darf eine Vererbungsbeziehung ebenfalls nur mittels `extends` hergestellt werden (nicht durch `implements`).
   c) Für eine Vererbungsbeziehung zwischen Klasse und Interface gilt, dass nur Klassen mittels `implements` diese Beziehung herstellen dürfen (einzig mögliche Vererbungsbeziehung zwischen Klasse und Interface).
8. Wenn zwei Methoden eine bestimmte Methodensignatur überladen, muss eine (eineindeutig) konkreter sein als die andere.

Hinsichtlich der Vererbung gelten zusätzlich folgende Eigenschaften:

1. Wenn eine geerbte Methode geändert werden soll, müssen die entsprechenden Methoden in Subtypen angepasst werden.
2. Eine Klasse, die ein Interface oder eine abstrakte Klasse beerbt, muss nach dem Refactoring immer noch den Anforderungen des Interfaces bzw. der Klasse genügen.

3.1.1.2 Berücksichtigung aspektorientierter Spracherweiterungen

Aufbauend auf den für Java geltenden Anforderungen der Java-Sprachdefinition müssen zusätzliche Regeln hinsichtlich der aspektorientierten Erweiterung gelten. Da hier keine spezielle aspektorientierte Sprache behandelt wird, sondern allgemein aspektorientierte Sprachen, die die Spezifikation dynamischer Programmeigenschaften zulassen, können an dieser Stelle auch keine entsprechenden Regeln aufgestellt werden. Als Beispiel sei aber auf die in der bereits erwähnten Arbeit von Rura erstellten Regeln für AspectJ hingewiesen (ebenso Rura 2003, Abschnitt 2.2.1 und 2.2.2). Dort gibt es zum Beispiel ein neues syntaktisches Element `Aspect`, welches in seiner Behandlung der von Typen entspricht. Folglich wird es bei Regel 2 mit einbezogen, und es wird definiert, dass auch ein `Aspect` einen eindeutigen Namen haben muss.

Ein anderes Beispiel liefert Object Teams (vgl. Herrmann 2002, Abschnitt 2.1) mit einem erweiterten Vererbungskonzept, der *Implicit Inheritance* welches das mehrfache Erben erlaubt. So wird offenbar die erste Regel bezüglich der Java-Sprachdefinition nicht mehr gelten, und die Regeln für `extends` und `implements` bedürfen einer Veränderung. Zusätzlich werden weitere Anpassungen erforderlich sein, um die verschiedenen Regeln zu adaptieren, und es werden sicherlich einige neue hinzukommen. Eine umfassendere Auseinandersetzung mit dem Thema Refactoring von ObjectTeams/Java Programmen findet sich in (Brcan 2005).

Neben den aus der Sprachdefinition hervorgehenden syntaktischen Programmeigenschaften, deren Verletzung mit Hilfe eines Übersetzers festgestellt werden kann, gibt es andere Programmeigenschaften, die ebenfalls nicht durch ein Refactoring verletzt werden dürfen.

3.1.2 Semantische Äquivalenz von Programmen

Es ist leicht nachzuvollziehen, dass Verhaltenserhaltung mehr fordert als die oben beschriebene Berücksichtigung syntaktischer Programmeigenschaften. Die Tatsache, dass ein geändertes Programm noch der Sprachdefinition genügt, sagt nicht ausreichend viel darüber aus, ob das Verhalten erhalten geblieben ist.

Opdyke hat dazu die semantische Äquivalenz von Referenzen und Operationen eingeführt, um für einige spezielle Quellcodeänderungen argumentieren zu können, dass sie verhaltenserhaltend sind. Den Begriff semantische Äquivalenz zwischen einem Programm vor und nach einer Quellcodeänderung definiert er zunächst wie folgt:

> „Let the external interface to the program be via the function *main*. If the function *main* is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same.“ [2]

Diese Definition semantischer Äquivalenz von Programmen in ihrer Gesamtheit wird in den meisten Fällen aber keine überprüfbare Bedingung für eine gegebene Änderung hinsichtlich der Erhaltung des Verhaltens ergeben. Die Frage, ob zwei Programme semantisch äquivalent sind, ist nicht entscheidbar.

Stattdessen wird von Opdyke für bestimmte Quellcodeänderungen, die definierten Regeln genügen, argumentiert, dass sie zwar Änderungen von Referenzen und Operationen im Programm darstellen, diese aber semantisch äquivalent zu den Referenzen und Operationen sind, wie sie vor

[2] (vgl. Opdyke 1992, Abschnitt 4.1.1), *function* ist in diesem Kontext als C++ Entsprechung von Methoden in Java zu sehen

den Quellcodeänderungen im Programm bestanden. Die Idee dahinter ist, dass sich der Effekt eines Refactorings auf ein Programm in zwei Bereiche teilt, zwischen denen eine Schnittstelle steht. Für den Bereich außerhalb der Schnittstelle verhält sich der Bereich innerhalb der Schnittstelle gleich, im Sinne der oben genannten Definition semantischer Äquivalenz.

3.1.2.1 Semantisch äquivalente Referenzen und Operationen

Wird auf die semantische Äquivalenz von Referenzen und Operationen bei Quellcodeänderungen geachtet, ergeben sich eine Reihe von unter bestimmten Bedingungen zulässigen Änderungen, die die semantische Äquivalenz des gesamten Programms nach der Änderung zum Programm vor der Änderung nicht beeinträchtigen. Die hier aufgelisteten Änderungen sind zusammengeführt aus den Arbeiten von Opdyke und Rura (vgl. Opdyke 1992; Rura 2003).

1. Die Vereinfachung von Ausdrücken (engl. Expressions) durch äquivalente Ausdrücke ist zulässig.
2. Nicht erreichbarer Quellcode kann entfernt werden.
3. Bedingungen können vereinfacht werden, z. B. durch das Herauslösen von Invarianten.
4. Variablen, Methoden und Klassen können hinzugefügt werden, wenn sie nicht referenziert werden.
5. Variablen, Methoden und Klassen können gelöscht werden, wenn sie nicht referenziert werden.
6. Der Typ einer Variable darf geändert werden, wenn jede referenzierte Operation auf dem neuen Typ äquivalent definiert ist und alle Zuweisungen, die diese Variable betreffen, typsicher bleiben.
7. Referenzen auf ein Feld oder eine Methode, die in einer Klasse definiert sind, können mit Referenzen von Feldern oder Methoden ersetzt werden, die äquivalent definiert sind.

Für objektorientierte Programme sind die oben beschriebenen Quellcodeänderungen unter den angegebenen Bedingungen zulässige Änderungen für ein Refactoring. Zulässig sind sie, da sie die semantische Äquivalenz von Referenzen und Operationen gewährleisten, für die argumentiert wurde, dass auch das gesamte Programm unter dieser Bedingung vor und nach der Quellcodeänderung semantisch äquivalent ist. In aspektorientierten Programmen kann hingegen aus verschiedenen Gründen nicht mehr argumentiert werden, dass diese Quellcodeänderungen nur mit diesen Bedingungen verhaltenserhaltend sind.

3.1.2.2 Semantische Äquivalenz in aspektorientierten Programmen

In den Abschnitten 1.1.1 und 1.1.2 sind die durch das aspektorientierte Programmiermodell hinzugekommenen Konzepte zur strukturellen Erweiterung und Verhaltensanpassung eingeführt worden. In Abschnitt 1.3.1 ist schon erwähnt worden, dass die Sprachelemente, die diese Konzepte anwendbar machen, beim Refaktorisieren zusätzlich berücksichtigt werden müssen.

Für die Konzepte zur strukturellen Erweiterung sind bereits Arbeiten erschienen, die die entsprechenden Sprachelemente berücksichtigen, so dass die semantische Äquivalenz von Referenzen und Operationen erreicht werden kann, indem angepasste, erweiterte oder neue mögliche Änderungen

und zugehörige Bedingungen erarbeitet wurden. Zu diesen Arbeiten gehören zum Beispiel (Brcan 2005) und (Rura 2003), die dies für die Sprache *ObjectTeams/Java* bzw. *AspectJ* in Teilen realisiert haben.

Auch für das Konzept der Verhaltensanpassung, in der Umsetzung durch Advices und Pointcuts, gibt es erste Realisierungen, die das Refaktorisieren in Gegenwart von Pointcuts und Advices unter bestimmten Umständen erlauben. Rura hat sich in seiner Arbeit nicht nur mit strukturellen Erweiterungen auseinandergesetzt, sondern auch Pointcuts und Advices für *AspectJ* untersucht und bestimmte Refactorings beschrieben (ebenso Rura 2003).

Im Folgenden soll ebenfalls für das Konzept der Verhaltensanpassung durch Pointcuts und Advices versucht werden, Wege zu finden,s um Quellcodeänderungen so durchzuführen, dass semantisch äquivalente Programme entstehen. An dieser Stelle sollen, ausgehend von einem Beispiel, Advices und Pointcuts als neue Arten von „Rümpfen“ und Referenzen im Zusammenhang mit Opdykes Definition von semantisch äquivalenten Programmen untersucht werden.

3.1.2.3 Semantische Äquivalenz von Pointcuts

Warum eine zusätzliche Betrachtung von Aspektbindungen unbedingt erforderlich ist, kann an einem einfachen Beispiel veranschaulicht werden. Abbildung 3.1b zeigt ein Programm vor einem *Rename Method*[3] Refactoring und Abbildung 3.1c danach. Die Umbenennung der Methode `m2()` inklusive der aufrufenden Operation aus Methode `m1()` ist zulässig im Sinne der semantischen Äquivalenz von Referenzen und Operationen für objektorientierte Programme. Für ein objektorientiertes Programm würde sich das Verhalten nicht ändern, was aufgrund des Umfangs des Beispiels leicht zu erkennen ist. Wenn allerdings der Aspekt aus Abbildung 3.1a ebenfalls Teil des Programms ist, ist das nicht mehr so. Dadurch, dass der erste Advice aus 3.1a in 3.1b zweimal bindet, aber in 3.1c nur einmal, hat sich das Verhalten des Programms geändert. Wegen der Änderung des beobachtbaren Verhaltens handelt es sich bei der Quellcodeänderung nicht mehr um ein Refactoring.

Obwohl alle Referenzen und Operatoren in den Programmen aus objektorientierter Sicht semantisch äquivalent sind, hat sich das Verhalten des Programms geändert. Offensichtlich muss auch der Aspekt mit in die Betrachtung des Verhaltens einbezogen werden, um Verhaltenserhaltung zu gewährleisten. In diesem Beispiel hätte der bestehende Zusammenhang zwischen der umbenannten Methode und dem Pointcut berücksichtigt werden müssen, das heißt die Aspektbindung, die nach der Änderung nicht mehr existiert.

Es ist deutlich geworden, dass zusätzliche Überlegungen im Umgang mit Pointcuts gemacht werden müssen. Bezogen auf dieses Beispiel liegt die Vermutung nahe, dass Pointcuts vielleicht ähnlich wie normale Programmreferenzen bei Refactorings behandelt werden könnten, es lässt sich aber leicht zeigen, wo hier die Grenzen liegen.

In den Pointcuts aus Abbildung 3.1a sind Beispiele für vollständige Methodensignaturen in Form der drei spezifizierten Namensmuster gegeben. Wenn ein Pointcut Joinpoints ausschließlich über vollständige Methoden- oder Feldsignaturen spezifiziert, entsteht zwischen Programmelementen und dem Pointcut ein Zusammenhang, vergleichbar mit gewöhnlichen Methodenaufrufen und Feldreferenzen. In diesem Fall könnten tatsächlich die spezifizierten Namensmuster wie ein Methodenaufruf oder eine Feldreferenz behandelt werden. Auch in Bezug auf durch solche Pointcuts

[3] vgl. Fowler u. a. (2005), Seite 273

```
public aspect AnAspect {
  private int _counter = 0;

  after() :
    execution(void Program.m1())
    || execution(void Program.m2())
  {
    _counter++;
  }

  after() : execution(void Program.lastAct())
  {
    System.out.println("Aspekt " + _counter + " mal aufgerufen");
  }
}
```

(a)

```
public class Program {
  public static void main(String[]
      args) {
    m1();
    lastAct();
  }
  private static void m1() {
    m2();
  }
  private static void m2() {}
  private static void lastAct() {
  }
}
```

(b)

```
public class Program {
  public static void main(String[]
      args) {
    m1();
    lastAct();
  }
  private static void m1() {
    p2();
  }
  private static void p2() {}
  private static void lastAct() {
  }
}
```

(c)

Abbildung 3.1: Beispielprogramm vor (b) und nach der Änderung (c) mit einem für beide Versionen unveränderten Aspekt (a)

hergestellte Aspektbindungen ließe sich die semantische Äquivalenz von Programmen, gemäß Opdykes Definition, herstellen.

Eine Behandlung von Pointcuts dieser Art würde für Aspektbindungen eine vergleichbar einfache Analyse wie für Referenzen und Operationen bedeuten. Für Referenzen und Operationen gilt nämlich, dass immer auf Programmstrukturen verwiesen wird, die sich eindeutig auflösen lassen. Als Beispiel sei hier die Berücksichtigung von Referenzen beim Hinzufügen einer neuen Methode genannt. Regel 4 besagt, dass eine Methode hinzugefügt werden kann, wenn sie im Programm nicht referenziert wird. Ob eine Methode referenziert wird, lässt sich durch eine verhältnismäßig einfache Überprüfung aller Methodenreferenzen (Methodenaufrufe) in einem Abstract Syntax Tree (AST) feststellen.

Wenn hingegen etwas anderes als eine vollständige Signatur für die Spezifikation im Pointcut benutzt wird, erfordert dies eine andere Art der Analyse. Hier liegt die Grenze zwischen der Behandlung von Referenzen und Operationen in einem objektorientierten Programm und von Pointcuts für aspektorientierte Programme. Werden `execution` Joinpoints zum Beispiel anhand einer Klassenhierarchie spezifiziert (siehe Abbildung 3.2), bildet nicht mehr ein AST die Grundlage für die Analyse, welche Programmelemente in welcher Form von einer Quellcodeänderung betroffen sind, sondern zusätzlich die Klassenhierarchie.

Geht es außerdem noch darum, wie ein Pointcut angepasst werden muss, der nicht ein konkretes Programmelement *vollständig* beschreibt, sondern mehrere Programmelemente über eine Eigenschaft wie die Klassenhierarchie, dann tauchen noch weitere Probleme auf. Wird nämlich

ein Programmelement geändert, auf das die Spezifikation vor der Änderung passte, stellt sich die Frage, wie die Spezifikation geändert werden kann, damit sie danach einerseits die anderen spezifizierten Elemente und andererseits das geänderte Element noch spezifiziert.

```
after():
    execution(void Programm+.m1())
    || execution(void Programm+.m2())
{
  _counter ++;
}
```

Abbildung 3.2: Spezifikation von Namenseigenschaften und einer Eigenschaft der Klassenhierarchie

In den weiteren Betrachtungen wird wiederholt auf ein Programm vor der Quellcodeänderung und eine entsprechende Folgeversion des Programms eingegangen werden. Für die bessere Lesbarkeit wird für das Programm vor der Änderung das Zeichen P definiert und für das Programm nach der Änderung das Zeichen P'.

Wenn vorausgesetzt wird, dass das geänderte Programm ohne Aspekt semantisch äquivalentes Verhalten erzeugt, dann würde es in einem ersten Schritt der Überlegung erforderlich sein, dass bei gleicher Eingabe für P und P' nur Joinpoints entstehen, die exakt dieselben Joinpointeigenschaften aufweisen. Dann würden auch die Aspekte an denselben Joinpoints gebunden werden, und das Verhalten sollte, im Sinne von Opdykes Definition der semantischen Äquivalenz von Programmen erhalten geblieben sein.

Es liegt aber nahe, dass jede Quellcodeänderung bewirkt, dass bei bestimmten oder sogar jeder möglichen Eingabe für P und P' bei der Ausführung von P' Joinpoints mit anderen Joinpointeigenschaften entstehen als bei der Ausführung von P. Das kann zur Folge haben, dass aufgrund fehlender oder verlagerter Joinpointeigenschaften ein Pointcut nicht mehr die richtigen Joinpoints spezifiziert. Die Folgen können dabei unterschiedlich ausfallen. Advices können gar nicht mehr gebunden werden, sie können an falsche Joinpoints gebunden werden oder an zusätzliche Joinpoints.

Theoretisch vollständig wäre eine Betrachtung, in der für jede Eingabe für P und P' jeweils alle auftretenden Joinpoints aufgestellt werden und dann miteinander verglichen werden. Danach könnte versucht werden, äquivalente Joinpoints zu identifizieren, deren Joinpointeigenschaften sich aber so unterscheiden, dass die Aspekte, die in P gebunden waren, in P' nicht mehr binden. Die Auswertung der Differenz in den Joinpointeigenschaften äquivalenter, nur in einer der Versionen gebundener Joinpoints wäre die Grundlage, um Pointcuts so anzupassen, dass sie durch Änderungen in den in Pointcuts spezifizierten Joinpointeigenschaften die äquivalenten Joinpoints aus P' spezifizieren würden.

Dieser vollständige Ansatz würde aber erfordern, dass jeweils für P und P' eine Repräsentation erzeugt wird, die die Joinpoints für alle möglichen Ausführungen widerspiegelt. So eine Repräsentation ist jedoch bereits für relativ kleine Programme nicht mehr realisierbar. Genauso problematisch ist die Erkennung äquivalenter, hinzugekommener oder nicht mehr vorhandener Joinpoints. In einem objektorientierten Refactoring können Elemente von P und P' direkt als verändert hinzugekommen oder entfernt identifiziert werden, da die Quellcodeänderungen direkt auf diesen Elementen getätigt werden. Joinpoints hingegen sind nur indirekt von der Quellcodeänderung betroffen, was einen Vergleich von aus den verschiedenen Versionen hervorgehenden Joinpoints nicht mehr ermöglicht.

Sinnvoller ist dagegen die Analyse der Auswirkungen von Quellcodeänderungen auf Joinpointeigenschaften in entsprechenden Strukturen selbst, die in den konkreten Pointcuts spezifiziert worden sind. Das Ziel der Analyse besteht dann darin, Änderungen in den entsprechenden Strukturen festzustellen, anhand derer erkannt werden kann, ob sich spezifizierte Eigenschaften von Joinpoints geändert haben, sodass vorhandene Pointcuts sie nicht mehr spezifizieren können bzw. andere oder zusätzliche Joinpoints spezifizieren. Statt also direkt zu versuchen, durch eine Art Simulation der Programmausführung verloren gegangene, hinzugekommene oder geänderte Aspektbindungen von Advices zu detektieren, um dann Anpassungen der Pointcuts vorzunehmen, wird hier umgekehrt versucht, die Auswirkungen von Quellcodeänderungen auf die Referenzen von Pointcuts zu den Joinpointeigenschaften in den entsprechenden Strukturen zu untersuchen.

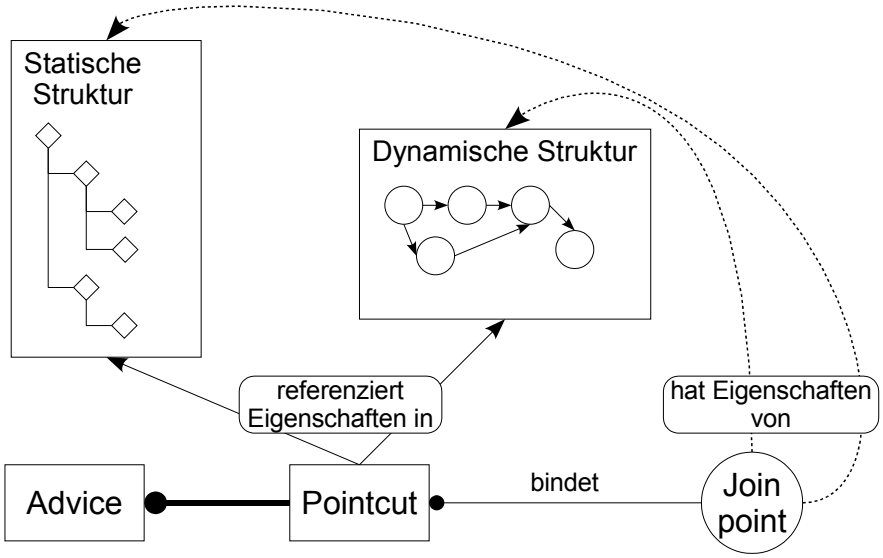


Abbildung 3.3: Bindung von Advicecode am Joinpoint (Ausführung), zustande gekommen durch die Übereinstimmung der im Pointcut spezifizierten Eigenschaften und der Eigenschaften, die dieser Joinpoint hat.

Der Zusammenhang zwischen Pointcuts und Joinpoints, in Bezug auf dynamische und statische Strukturen, kann erklären, warum die Analyse der Auswirkungen von Quellcodeänderungen auf Repräsentationen der Joinpointeigenschaften dabei hilft, Verhaltensänderungen zu erkennen. In Abbildung 3.3 sind symbolhaft eine dynamische und eine statische Struktur dargestellt. Der Pointcut referenziert Eigenschaften in diesen Strukturen. Sobald ein Joinpoint existiert, der mindestens dieselben Eigenschaften in denselben Strukturen aufweist, wird dieser gebunden; das heißt, der mit dem entsprechenden Pointcut gekoppelte Advice wird ausgeführt.

Für dynamische Strukturen gilt dabei, dass ein Pointcut die Eigenschaften auf einer Struktur spezifiziert, die alle möglichen Ausführungen repräsentiert, da er unverändert in jeder Ausführung des Programms vorhanden ist. Ein Joinpoint hat dagegen eine Eigenschaft bezogen auf den Teil der gesamten dynamischen Struktur, der der konkreten Programmausführung entspricht, in welcher der Joinpoint existiert. Da statische Strukturen unabhängig von der Ausführung sind, ist die statische Struktur, auf der eine Eigenschaft für einen Joinpoint gilt, die gleiche wie die Struktur, auf der ein Pointcut die Eigenschaft spezifiziert hat.

Wenn eine Quellcodeänderung eine Struktur ändert, auf der eine Eigenschaft spezifiziert wur-

de, und sich die spezifizierte Eigenschaft danach nicht mehr finden lässt, dann wird auch kein Joinpoint mehr diese Eigenschaft aufweisen können. An einer oder mehreren Stellen, an denen vorher Verhalten aufgrund des gebundenen Advices vorhanden war, wird dieses Verhalten verloren gegangen sein, und die Verhaltenserhaltung konnte nicht gewährleistet werden.

Auch das häufigere Vorhandensein einer Eigenschaft in einer Struktur nach der Quellcodeänderung bedeutet, dass das Verhalten nicht erhalten wurde. In Fällen, in denen die Eigenschaft noch genauso häufig in der Struktur zur Version P' vorhanden ist wie in der Struktur zur Version P des Programms, kann erst dann von Verhaltenserhaltung bezüglich dieser Eigenschaft gesprochen werden, wenn es sich um semantisch äquivalente Stellen in der Struktur vor und nach der Änderung handelt.

3.1.2.4 Verhaltenserhaltung unter Verwendung der semantischen Äquivalenz von Pointcuts

Die Erhaltung des Verhaltens kann jedoch erreicht werden, indem versucht wird die Pointcuts so anzupassen, dass sich für Referenzen des Pointcuts auf Joinpointeigenschaften in den entsprechenden Strukturen für das Programm P vor der Änderung Referenzen auf *semantisch äquivalente Eigenschaften* in den Strukturen für das Programm P' ergeben. Mit der semantischen Äquivalenz von Pointcutreferenzen sollte es möglich sein, analog zum Umgang mit Opdykes Begriff der semantischen Äquivalenz von Referenzen und Operationen, für bestimmte Quellcodeänderungen unter bestimmten Bedingungen argumentieren zu können, dass eine Quellcodeänderung zu einem semantisch äquivalenten Programm geführt hat.

Wie die Analyse der Auswirkungen von Quellcodeänderungen auf Repräsentationen dazu führt, dass erkannt werden kann, ob sich das Verhalten geändert hat, lässt sich an dem Beispiel aus Abbildung 3.1 erläutern. Der erste Pointcut ist in zwei Teilen aufgebaut, die durch eine Veroderung verbunden sind. Neben anderen statischen Strukturen wird der Namensraum benutzt, um darin Methodensignaturen zu spezifizieren. Bei der dargestellten Änderung des Programms verliert der Namensraum den Methodennamen `m2()` als Folge der Änderung und bekommt stattdessen zusätzlich einen Methodennamen `p2()`. Die Verhaltensänderung kann daran festgemacht werden, dass der Methodenname `m2()` im geänderten Programm nicht mehr als Eigenschaft für einen Joinpoint zur Verfügung steht. Eine Aspektbindung, die vorher wegen dieser Eigenschaft stattgefunden hat, wird nicht mehr eintreten, und das Verhalten des Advices wird an dieser Stelle nicht mehr ausgeführt.

Ist aber die Äquivalenz des Methodennamens `m2()` vor der Änderung und `p2()` nach der Änderung feststellbar, so kann der Pointcut so angepasst werden, dass er statt des Namens `m2()` den Namen `p2()` spezifiziert. Folgt man der Argumentation, dann ist mit der semantischen Äquivalenz der Pointcuts (des ursprünglichen Pointcuts und des angepassten Pointcuts) auch die semantische Äquivalenz der Programme hergestellt.

Die Regeln und Bedingungen einiger objektorientierte Refactorings sind bereits für den Einsatz in aspektorientierten Programmen angepasst worden. Die Anpassung der Regeln und Bedingungen war Inhalt verschiedener Arbeiten, wie zum Beispiel (Rura 2003; Hanenberg u. a. 2003), in denen auch spezifizierte statische Pointcuts berücksichtigt wurden. Um die zusätzliche Berücksichtigung von dynamischen Pointcuts zu ermöglichen, sollen Repräsentationen dynamischer Strukturen erarbeitet werden, in denen dynamische Joinpointeigenschaften auffindbar sind, entsprechend den Überlegungen dieses und des vorherigen Abschnitts.

3.2 Repräsentationen dynamischer Joinpointeigenschaften

Für statische Pointcuts und die entsprechenden statischen Joinpointeigenschaften wird es nicht notwendig sein, spezielle Überlegungen bezüglich der Ausführung anzustellen. Änderungen dieser Joinpointeigenschaften lassen sich bereits an statischen Strukturen des Programms identifizieren, wie es in Beispiel 3.1a gezeigt worden ist. Ein Modell der Ausführung in die Analyse von Auswirkungen auf statische Joinpointeigenschaften mit einzubeziehen macht vor allem deswegen keinen Sinn, weil sich für verschiedene Ausführungen die Auswirkungen auf statische Eigenschaften nicht ändern. Für dynamische Pointcuts ist es hingegen unbedingt erforderlich, dass ein Modell der Programmausführung genutzt wird, andernfalls kann eine von der Ausführung abhängige Eigenschaft nicht erkannt werden.

Ein Modell der Programmausführung und der Strukturen in diesem Modell, auf denen dynamische Joinpointeigenschaften zu finden sind, ermöglicht es Betrachtungen auch hinsichtlich dieser Joinpointeigenschaften zu machen. In diesem Rahmen soll die Entwicklung von Repräsentationen von Strukturen für dynamische Joinpointeigenschaften durchgeführt werden. Die Repräsentationen sollen es ermöglichen, auf diesen zu erkennen, ob eine bestimmte spezifizierte Eigenschaft in einem Pointcut durch eine Quellcodeänderung betroffen ist, und wenn das so ist, wie sich die Auswirkung der Quellcodeänderung auf diese Eigenschaft gestaltet.

In Abschnitt 2.2.2 wurden verschiedene dynamische Joinpointeigenschaften erläutert. Mit den vorgestellten Joinpointeigenschaften ist ein Überblick über die Spezifikationsmöglichkeiten aktueller Pointcutsprachen gegeben worden, mit der Einschränkung, dass es sich um einen Ausschnitt des Möglichen handelt wegen des sich kontinuierlich erweiternden Feldes an neuen Pointcutsprachen und Spezifikationsmöglichkeiten.

Die verschiedenen dynamischen Joinpointeigenschaften galten für verschiedene dynamische Strukturen und fielen so in eine von drei Gruppen:

- Joinpointeigenschaften des Objektgraphen.
- Joinpointeigenschaften des Programmablaufs.
- Joinpointeigenschaften beider Strukturen.

Joinpointeigenschaften der beiden letzten Gruppen unterscheiden sich darüber hinaus von denen der ersten Gruppe. Das Besondere dieser Eigenschaften ist, dass sie von durch weitere Eigenschaften spezifizierten „Pointcutmatches“ abhängen, die nicht direkt gebunden werden. Da es sich bei den benutzten Pointcutmatches also um keine Pointcutmatches im eigentlichen Sinne handelt, werden sie im Folgenden auch nicht als solche bezeichnet. Es wird stattdessen von **Triggern** gesprochen. Die Spezifikation einer Cflow- oder Ereignissequenz- Eigenschaft fordert zum Beispiel die in Abschnitt 2.2.2 erläuterten Abhängigkeiten unter diesen Triggern und führt dann dazu, dass an einem der Trigger wirklich ein Aspekt gebunden wird. Dieser Trigger wird mit **Endtrigger** bezeichnet. Da die Eigenschaften immer von mehreren Triggern abhängen, heißen diese **Multitriggereigenschaften**. Die Joinpointeigenschaften der ersten Gruppe sind dagegen, wie auch die statischen Joinpointeigenschaften, Singletrigger-Joinpointeigenschaften.

Die den Repräsentationen der Multitrigger-Joinpointeigenschaften zugrunde liegenden Strukturen sind (wie sich später zeigen wird) der Aufrufgraph und der Kontrollflussgraph. Objektgraphstrukturen, die für die dynamischen Singletrigger-Joinpointeigenschaften offensichtlich notwendig sein werden, bauen bei ihrer Erstellung auf dem Aufrufgraph und dem Kontrollflussgraphen

auf. Für eine durchgängige Erarbeitung einer Analyse der Auswirkungen von Quellcodeänderungen auf eine Joinpointeigenschaft – einschließlich einer Realisierung in einem Werkzeug – versprechen die Multitrigger-Joinpointeigenschaften ein guter Einstiegspunkt zu sein.

Am Beispiel der Eigenschaft *Cflow* soll eine Analyse erarbeitet und realisiert werden, um zu prüfen, inwieweit der in 2.3.2 aufgezeigte Mittelweg zum Umgang mit dynamischen Pointcuts einen Beitrag zum Ziel der Durchführbarkeit von Refactorings in Gegenwart dynamischer Pointcuts liefert. Für die Eigenschaft *Ereignissequenz* werden ausblickhaft Möglichkeiten aufgezeigt und die Erreichbarkeit der Ziele diskutiert.

Die Repräsentationen werden Joinpoints in Verbindung mit Programmelementen bringen, die existentielle Voraussetzung eines Joinpoints sind. In diesem Zusammenhang sollen an dieser Stelle ein paar Begriffe eingeführt werden. Beispielsweise ist ein Statement `_i = 5;` (`_i` ist ein Feld) die Grundlage dafür, dass bei jeder Ausführung an dieser Stelle ein `get` Joinpoint entsteht. Für jeden Joinpoint gibt es Programmcode, der in dieser oder ähnlicher Weise für die Existenz bestimmter Joinpoints zur Ausführungszeit verantwortlich ist[4]. Dieser Programmcode wird als **Joinpointshadow** bezeichnet, ein Begriff, der für die Realisierung von aspektorientierten Compilern eine zentrale Rolle spielt (vgl. z. B. Masuhara u. a. 2002; Hilsdale u. Hugunin 2004). Zur besseren Unterscheidung werden zusätzlich die Begriffe **Triggershadow** und **Endtriggershadow** benutzt, um diejenigen Joinpointshadows zu bezeichnen, die jeweils für Trigger und Endtrigger Voraussetzung sind.

3.2.1 Repräsentation der Joinpointeigenschaft Cflow

In Abschnitt 2.2.2.2 wurde die Eigenschaft *Cflow* detailliert beschrieben. Dort wurde erläutert, dass die Eigenschaft *Cflow* für die Aufrufabhängigkeit zweier Joinpoints steht. Ein Endtrigger bindet dann einen Advice, wenn er in einem Kontrollfluss steht, der durch einen anderen Trigger (dem Starttrigger) aufgespannt wird. Die Starttrigger und Endtrigger werden dabei durch weitere Joinpointeigenschaften spezifiziert[5]. In einer konkreten Ausführung handelt es sich bei einem aufgespannten Kontrollfluss um den, der ab dem Eintritt in eine Methode bis zum Beenden der Methode bzw. ab dem Aufruf einer Methode bis zum Rücksprung aus der Methode vorliegt.

Für eine, für ein Programm allgemeingültige, Aussage hinsichtlich der Joinpointeigenschaft *Cflow*, reicht die Betrachtung einer konkreten Ausführung nicht aus. Es muss ein vollständiges Modell dieser Eigenschaft für alle möglichen Ausführungen eines Programms entstehen. Als Einstiegspunkt für die allgemeine Repräsentation der Eigenschaft *Cflow* werden alle möglichen Kontrollflüsse ausgehend von einem Starttriggershadow aufgebaut. Das heißt, ausgehend von einer Methodendeklaration oder einem Methodenaufruf werden mögliche Kontrollflüsse anhand von Aufrufabhängigkeiten zusammengeführt.

Ist der Starttriggershadow eine Methodendeklaration, so gehören die möglichen Kontrollflüsse des Methodenrumpfes zur ersten Ebene der aufgespannten Kontrollflüsse. Für jeden Methodenaufruf in der ersten Ebene der Kontrollflüsse ergeben sich jeweils wieder neue Kontrollflüsse, die sich durch die möglichen Kontrollflüsse in den Methodenrümpfen der aufgerufenen Methoden definieren. Die weiteren Ebenen sind jeweils wieder Bestandteil der aufgespannten Kontrollflüsse. Es werden sich solange neue Ebenen ergeben, bis keine weiteren Methodenaufrufe in den jeweiligen Kontrollflüssen vorhanden sind.

[4] Komplexere Zusammenhänge entstehen lediglich bei Joinpointarten wie z. B. `preinitialization` in AspectJ

[5] Die weiteren Eigenschaften dürfen auch die Eigenschaft *Cflow* beinhalten

Handelt es sich beim aufspannenden Starttriggershadow nicht um eine Methodendeklaration sondern um einen Methodenaufruf, so gehört der Aufruf ebenfalls zu den aufgespannten Kontrollflüssen. Ansonsten werden die Kontrollflüsse der weiteren Ebenen in der gleichen Weise aufgebaut.

Die so aufgebaute Kontrollflussmenge eines Starttriggershadows repräsentiert alle möglichen aufgespannten Kontrollflüsse. Für die Endtriggershadows lässt sich nun feststellen, ob sie in einem Kontrollfluss dieser Kontrollflussmenge liegen. Ist das der Fall, so führt der Endtriggershadow, bei einer Ausführung, die diesem Kontrollfluss entspricht, zu Joinpoints, die die spezifizierte *Cflow* Eigenschaft haben.

Diese Repräsentation enthält allerdings viele Informationen, von denen bezüglich der Eigenschaft *Cflow* abstrahiert werden kann. Zum Beispiel spielt die Reihenfolge der einzelnen Elemente in einem Kontrollfluss zwischen Starttrigger und Endtrigger keine Rolle. Entscheidend ist dagegen, ob ein Endtriggershadow in Aufrufabhängigkeit eines Starttriggershadows liegt. Aufrufabhängigkeiten werden üblicherweise mit Aufrufgraphen repräsentiert.

3.2.1.1 Aufrufgraph

Die Aufrufabhängigkeiten eines Programms lassen sich in einem Aufrufgraphen darstellen. Bei einem Aufrufgraphen handelt es sich um einen gerichteten, unter Umständen zyklischen, Graphen. Zyklen kann dieser Graph wegen der Möglichkeit direkter bzw. indirekter Rekursion enthalten. Die Knoten des Aufrufgraphen stehen für Methodendeklarationen. Kanten stehen für die Möglichkeit eines Aufrufes einer Methode aus einer anderen Methode heraus. Kann also eine Methode `m1()` eine Methode `m2()` aufrufen, dann gibt es im Aufrufgraph eine gerichtete Kante von dem Knoten, der `m1()` repräsentiert, zu dem Knoten, der `m2()` repräsentiert.

Ein reiner Aufrufgraph repräsentiert zwar die Aufrufabhängigkeiten, repräsentiert damit aber noch nicht genau genug die Joinpointeigenschaft *Cflow*. Die erste Grenze, die sich auftut, ist die Eigenschaft des Aufrufgraphen, als einzige Programmelemente Methodendeklarationen zu beinhalten. Eine Spezifikation der Eigenschaft *Cflow* kann aber einerseits als Starttrigger einen `call` Joinpoint spezifizieren, dessen Starttriggershadow einen Methodenaufruf ist. Zum anderen kann als Endtrigger jede mögliche andere Joinpointart spezifiziert werden, die als Endtriggershadows auch verschiedene Arten von Anweisungen sein können.

Außerdem spielen Kontrollflussanweisungen eine Rolle. Kontrollflussanweisungen können eine Aufrufabhängigkeit zu einer bedingten Aufrufabhängigkeit machen, oder es besteht durch Schleifenanweisungen die Möglichkeit, dass eine Aufrufabhängigkeit häufiger vorkommt. Ein Aufrufgraph enthält auch keine parallelen Kanten, das heißt, dass bei mehrmaligen Aufrufen derselben Methode aus einer Methode nur eine Kante entsteht. Für die Repräsentation der Eigenschaft *Cflow* spielt die Häufigkeit der, an einem Endtriggershadow möglichen, Joinpoints aber eine Rolle, die Repräsentation muss dies soweit möglich identifizierbar machen.

Ein gerichteter Graph kann Zyklen enthalten, dies ist auch dann eine Eigenschaft wenn der gerichtete Graph ein Aufrufgraph ist. Verursacht werden solche Zyklen durch direkt oder indirekt rekursive Aufrufe. Problematisch ist ein gerichteter zyklischer Graph in diesem Zusammenhang deswegen, weil sich durch Zyklen im Prinzip unendlich viele verschiedene Pfade ergeben können. Benötigt wird aber eine feste Anzahl von Pfaden, die die Eigenschaft eindeutig repräsentieren.

Diese Zyklen können aber in Analogie zu vorhandenen Schleifenanweisungen behandelt werden. Liegt in einem Pfad zwischen Starttriggershadow und Endtriggershadow ein Zyklus vor, so handelt es sich genauso um einen mehrfachen Pfad wie bei dem Vorhandensein einer Schleife. Ist der Endtriggershadow selbst Teil des Zyklus, handelt es sich um dieselbe Situation. Für Starttriggershadows als Teil eines Zyklus verhält es sich dagegen anders. Hier bedeutet der Zyklus, dass der Starttriggershadow immer wieder durchlaufen werden kann, was zwar die Häufigkeit der auftretenden Joinpoints am Endtriggershadow beeinflusst, aber dies ist unabhängig von der Joinpointeigenschaft *Cflow*, die beschreibt, ob der Endtrigger im Kontrollfluss des Starttriggers liegt. Mit anderen Worten bedeutet das, dass es unwichtig ist, wie oft der Starttriggershadow durchlaufen wird, die Häufigkeit bleibt gleich, mit der Joinpoints mit der *Cflow* Eigenschaft in dem Kontrollfluss des jeweiligen Starttriggers liegen.

3.2.1.2 Erweiterter Aufrufgraph

Der reine Aufrufgraph muss also hinsichtlich mehrerer Punkte erweitert werden. Damit auch Aufrufabhängigkeiten auf Anweisungsebene dargestellt werden können, muss der Graph dahingehen erweitert werden, dass auch Methodendeklarationen, die selbst keinen Endtriggershadow darstellen aber Endtriggershadows enthalten, entsprechend markiert als Knoten im Graphen stehen.

Gibt es mehrere Aufrufe derselben Methode aus einer anderen, so wird es entsprechend parallele Kanten im erweiterten Aufrufgraphen geben.

Für eine bessere Differenzierung, auf welche Art und wie oft es an einem Endtriggershadow zu Joinpoints mit der spezifizierten *Cflow* Eigenschaft kommen kann, werden die Kanten qualifiziert. Das bedeutet, wenn eine Kante für einen bedingten Aufruf steht (dynamisch gebunden, `if-else` usw.), so wird sie so gekennzeichnet, dass sich erkennen lässt, von welcher Bedingung sie welche Alternative darstellt. Das heißt, es wird die Anzahl der möglichen Verzweigungen an einer Bedingung aufgestellt, und diese werden als Alternativen unterschieden. Bei Methodenaufrufen, die Teil eines Schleifenrumpfes sind, werden die Kanten als mehrfache Kante gekennzeichnet. Gibt es Zyklen im Graphen, so werden diese als einzelner Knoten dargestellt und die abgehenden Kanten werden ebenso als mehrfache Kanten gekennzeichnet.

3.2.1.3 Der erweiterte Aufrufgraph als Repräsentation der dynamischen Joinpointeigenschaft Cflow

Mit dem Aufbau von erweiterten Aufrufgraphen zwischen den Starttriggershadows und den Endtriggershadows ergibt sich für jede Kombination aus Starttriggershadow und Endtriggershadow eine Menge von möglichen qualifizierten Aufrufpfaden zwischen den jeweiligen Starttriggershadows und Endtriggershadows. Je nach Art der in einem Pfad enthaltenen Kanten ergeben sich dadurch sichere, bedingte und mehrfache Pfade.

Ein sicherer Pfad bedeutet, dass zwischen dem Starttriggershadow und Endtriggershadow entlang dieses Aufrufpfades die Joinpointeigenschaft *Cflow* am Endtriggershadow sicher eintritt. Ein bedingter Pfad bedeutet, dass zwischen Start- und Endtriggershadow die Möglichkeit besteht, dass die Eigenschaft *Cflow* für Joinpoints am Endtriggershadow zutrifft. Ein mehrfacher Pfad heißt, dass entlang dieses Pfades die spezifizierte Eigenschaft *Cflow* für Joinpoints am Endtriggershadow n - mal vorkommt. Der Wert n ist dabei von einer konkreten Ausführung abhängig, in dieser Repräsentation wird davon abstrahiert. Abbildung 3.4 zeigt ein Codebeispiel

```
public void st()
{
  m1();
}
public void m1()
{
  if(condition){
    mt();
  }
}
public void mt(){}
```

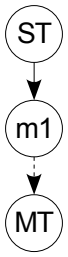


Abbildung 3.4: Quellcode und Visualisierung für einen bedingten Matchpfad

```
public void st()
{
  m1();
}
public void m1()
{
  while(condition){
    mt();
  }
}
public void mt(){}
```

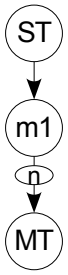


Abbildung 3.5: Quellcode und Visualisierung für einen mehrfachen Matchpfad

und ein Visualisierung eines bedingten Pfades, Abbildung 3.5 eines mehrfachen Pfades. Für beide Beispiele gilt die Spezifikation einer Eigenschaft *Cflow* deren Spezifikation für den Starttrigger zum Starttriggershadow `st()` und zum Endtriggershadow `mt()` führt.

Die Abhängigkeit der Qualifikation des Pfades von den Qualifikationen der enthaltenen Kanten ergibt sich dabei folgendermaßen:

- Ein Pfad, in dem es ausschließlich sichere Kanten gibt, ist auch ein sicherer Pfad.
- Ein Pfad, in dem es mindestens eine bedingte Kante, aber keine Schleifenkante und keinen Zyklus gibt, ist ein bedingter Pfad.
- Ein Pfad, der mindestens eine Schleifenkante oder mindestens einen Zyklus hat, ist ein mehrfacher Pfad.

Am Ende steht für ein Programm eine Anzahl von sicheren, bedingten und mehrfachen Pfaden für eine spezifizierte *Cflow* Eigenschaft fest, jeweils in Verbindung mit den dazugehörigen Starttriggershadows und Endtriggershadows.

Für die bedingten Pfade ist allerdings noch eine gesonderte Behandlung notwendig, damit die Anzahl der sicheren und die der bedingten Pfade nicht verfälscht ist. Ein bedingter Aufruf kann im gegenseitigen Ausschluss zu einem oder mehreren anderen bedingten Aufrufen stehen. Dies ist der Fall, wenn zum Beispiel in einem `if-then-else` Konstrukt im `if`-Zweig und im `else`-Zweig dieselbe Methode aufgerufen wird. Am Endtriggershadow kommt es auf jeden Fall zu einem Endtrigger, der im Kontrollfluss des Starttriggers liegt, wenn beide Pfade im gegenseitigen Ausschluss existieren. Statt zwei bedingter Pfade gibt es also einen sicheren Pfad.

Diese Repräsentation steht für eine spezifizierte *Cflow* Eigenschaft in allen möglichen Ausführungen. Abschnitte in den Ausführungen, in denen die *Cflow* Eigenschaft zutrifft, sind durch die

Pfade beschrieben, die allerdings von Wertebelegungen der Variablen abstrahieren. Variablen, die in Bedingungen stehen, von denen die Ausführung eines Pfades abhängt, müssten im Prinzip mit in die Betrachtungen mit einbezogen werden, wodurch Pfade noch genauer unterscheidbar wären.

Eine Repräsentation, die so genau ist, hat den Nachteil, dass sie nur als Modell beschreibbar ist, jedoch nicht physisch abgebildet werden kann. Schon relativ kleine Programme können einen sehr großen Raum von möglichen Zuständen und Zustandsübergängen erzeugen. Auch unendlich große Zustandsräume können leicht für ein Programm entstehen. Die Ermittlung der Zustände ist jedoch für die Bestimmung von Werten in Variablen notwendig. Da die physische Umsetzung dieser Repräsentation Voraussetzung dafür ist, um mittels eines Werkzeuges den Einfluss von Quellcodeänderungen auf die durch Pointcuts spezifizierte *Cflow* Eigenschaft zu analysieren, muss auch eine physisch abbildbare Repräsentation genutzt werden. Eine Repräsentation, die von Variablenbelegungen abstrahiert, scheint ein tragbarer Kompromiss. Die Auswertung der Untersuchung der Auswirkungen von Quellcodeänderungen auf die Eigenschaft *Cflow* wird zeigen, ob diese Repräsentation für die Zwecke des Werkzeugs ausreichen werden.

3.2.2 Repräsentation der Eigenschaft Ereignissequenz

Auch die Eigenschaft *Ereignissequenz*, die ebenfalls schon in Abschnitt 2.2.2.2 beschrieben wurde, ist eine Multitriggereigenschaft. Hier sind die Trigger nicht Joinpoints wie bei der *Cflow* Eigenschaft, sondern der Zeitpunkt vor oder nach einem Joinpoint. In Abschnitt 2.2.2.2 sind als Beispiele für solche Trigger Methodeneintritte und Methodenaustritte gezeigt worden.

Für eine konkrete Ausführung wird die Folge der eintretenden Trigger „beobachtet“ und mit einem spezifizierten regulären Ausdruck verglichen. Passt der reguläre Ausdruck vollständig auf die Folge, dann handelt es sich beim aktuellen Trigger um den Endtrigger und der Advice wird ausgeführt. Zusätzlich kann gefordert werden, dass sich bestimmte Objekte aus dem Endtriggerkontext und einem anderen spezifizierten Trigger gleichen (siehe Abschnitt 2.2.2.3).

Für diese Joinpointeigenschaft wird, genau so wie für die Eigenschaft *Cflow*, eine für alle Ausführungen eines Programms gültige Repräsentation gefunden werden müssen, um zwei Programmversionen bezüglich dieser Eigenschaft vergleichen zu können. Diese Repräsentation wird in zweierlei Hinsicht komplexer sein als die der Eigenschaft *Cflow*.

Während die Eigenschaft *Cflow* sich in verhältnissmäßig kleinen Abschnitten des Kontrollflusses repräsentiert und dort auch nur bestimmte Informationen für diese Eigenschaft Relevanz haben, sind für die Eigenschaft *Ereignissequenz* genauere Betrachtungen des Kontrollflusses notwendig, und die Menge an zu berücksichtigender Information wird sich über weitere Teile des Kontrollflusses verteilen.

Ist die Beschreibung der Verhältnisse der Trigger und Endtrigger zueinander für die Eigenschaft *Cflow* noch relativ einfach, werden für die Beschreibung der Verhältnisse der Trigger der Eigenschaft *Ereignissequenz* reguläre Ausdrücke genutzt, die komplexere Zusammenhänge beschreiben können.

Als sinnvolle Grundlage für die Repräsentation der Eigenschaft *Ereignissequenz* erscheint ein Kontrollflussgraph. In ihm sind alle möglichen Ausführungen des Programms repräsentiert und ein Kontrollflussgraph enthält die notwendigen Informationen im dem Sinne, dass sich die Trigger identifizieren lassen und sich für die Trigger Aussagen darüber treffen lassen in welcher zeitlichen

Abhängigkeit sie zueinander stehen, sodass sich darauf die spezifizierte Ereignissequenz erkennen lässt.

3.2.2.1 Kontrollflussgraph

Der Kontrollflussgraph ist in der Programmanalyse eine geläufige Repräsentation des Programms. Diese Repräsentation wird häufig für Analysen von Programmübersetzern (engl. Compiler) eingesetzt, die der Optimierung des übersetzten Quellcodes dienen (vgl. Aho u. a. 1987, Abschnitt 9.4 und Kapitel 10). Ein Kontrollflussgraph beschreibt für alle möglichen Ausführungen eines Programms, welche „Wege“ durch das Programm für eine konkrete Ausführung genommen werden können, also welche Reihenfolgen von Anweisungen möglich sind.

Grundlage bilden sog. **Basic Blocks**, die einen Abschnitt von Anweisungen darstellen, der immer am Stück durchlaufen wird. Das heißt, dass sobald die erste Anweisung eines Basic Blocks ausgeführt wird, auch alle anderen Anweisungen des Basic Blocks ausgeführt werden. Eine weiter Eigenschaft von Basic Blocks besteht darin, dass ein Sprung zu einem Basic Block nur an den Anfang des Block geschehen kann, es gibt nicht mehrere Einsprungziele pro Block. Die Basic Blocks bilden die Knoten des Kontrollflussgraphen. Zwischen den Basic Blocks befinden sich Kanten, die anzeigen, in welcher Reihenfolge einzelne Basic Blocks ausgeführt werden können.

Die Kanten stellen im Gegensatz zum Aufrufgraphen keine Aufrufabhängigkeiten dar, sondern zeigen an von welchem Basic Block zu welchem anderen Basic Block (bei Schleifen kann das auch derselbe sein) die Ausführung des Programms laufen kann. Dies setzt sich solange fort, bis ein Punkt erreicht wird an dem es keine ausgehenden Kanten mehr gibt, hier ist dann das Ende der Programmausführung erreicht.

Für Sprachen, die Funktionsaufrufe, Prozeduraufrufe oder Methodenaufrufe als Sprachkonstrukt beinhalten, müssen Kontrollflussgraphen zunächst aus den Rümpfen der entsprechenden Deklarationen aufgebaut werden. Die einzelnen Kontrollflussgraphen der Deklarationen werden dann mit dem Aufrufgraph des Programms zu einem Kellerautomaten zusammengesetzt. Der Kellerautomat kann so für Methodenaufrufe und insbesondere für Methodenrücksprünge die richtigen Basic Blocks bestimmen.

3.2.2.2 Einschränkungen des Kontrollflussgraphen

Für den Einsatz als Repräsentationsgrundlage der Eigenschaft *Ereignissequenz* weist ein Kontrollflussgraph im Wesentlichen zwei Nachteile auf. Zum einen kann nur ein Teil der Eigenschaft *Ereignissequenz* abgebildet werden und zum anderen ist die Identifizierung der Eigenschaft unter Umständen aufwendig und kann sich über einen sehr großen Bereich des Graphen erstrecken.

Eine Repräsentation, die es erlaubt, die Gleichheit von Objekten zu verschiedenen Zeitpunkten zu ermitteln, muss Informationen zu Variablen, die Objekte referenzieren und zu Instanziierungspunkten der Objekte enthalten. Diese Anforderungen ergeben sich einerseits daraus, dass in der Spezifikation einer *Ereignissequenz* die Objekte, von denen Identität gefordert wird, über Variablennamen angegeben werden und andererseits für zwei Trigger im Kontrollflussgraphen der Inhalt der Variablen vergleichbar sein muss. In einem reinen Kontrollflussgraphen sind diese Informationen nicht enthalten. In dem Gebiet der Programmanalyse zur Optimierung während der Codeübersetzung gibt es, neben der Analyse von Kontrollflussgraphen, auch Datenflussanalysen mit ähnlichen Anforderungen, wie sie hier beschrieben wurden (ebenso Aho u. a. 1987, Abschnitt

10.3). Eine weitere Auseinandersetzung mit diesen Analysen könnte Lösungswege aufzeigen, sie ist im Rahmen dieser Arbeit im Hinblick auf den Umfang aber ausgeblieben.

Die Einschränkung hinsichtlich der Identifizierbarkeit und der Ausdehnung in der Repräsentation kann an einem Beispiel dargestellt werden. In Abbildung 3.6 ist eine Art Kontrollflussgraph dargestellt, der an den Kanten die entsprechenden Trigger aufweist. Um die Komplexität der Darstellung nicht unnötig zu erhöhen, sind lediglich Trigger für den Joinpointtyp `execution` eingefügt worden. Trigger, die in dem oben links abgebildeten Pointcut einer Ereignissequenz spezifiziert wurden, sind mit Symbolen markiert worden. Der Pointcut spezifiziert einen einfachen Ausdruck, die Schwierigkeiten, die mit einem komplexeren regulären Ausdruck eventuell entstehen könnten, sollen hier nicht erläutert werden.

Die spezifizierte Eigenschaft in diesem Graph geht von allen Triggershadows aus, die das erste Triggersymbol spezifiziert hat (`sym nonCritAqu`), über alle möglichen Triggershadows, die durch das zweite Triggersymbol spezifiziert (`sym execCritOp`) sind, und endet bei den jeweiligen erreichbaren Triggershadows, die durch das letzte Triggersymbol spezifiziert sind(`sym nonCritRel`). In den jeweiligen Pfaden im Kontrollflussgraph dürfen dabei keine Triggershadows enthalten sein, die durch die Triggersymbole spezifiziert wurden, die nicht Teil der geforderten Sequenz `nonCritAqu execCritOp nonCritRel` sind.

Das manuelle Nachvollziehen der Ermittlung der Pfade, die durch den Pointcut spezifiziert sind, macht deutlich, dass eine Analyse von Kontrollflusspfaden sich leicht über den gesamten assoziierten Aufrufgraph zieht. In der Darstellung sind die Methoden, in denen sich die aktuelle Anweisung befindet, als umschließender Kasten dargestellt. Ausgehend von der aktuellen umschließenden Methode muss sowohl zu aufgerufenen als auch zu aufrufenden Methodendeklarationen gesprungen werden. Für die Ermittlung der Pfade einer Eigenschaft *Cflow* sind dagegen nur die aufgerufenen Methodendeklarationen von Bedeutung, der Umfang der zu analysierenden Daten reduziert sich dadurch erheblich.

Auch der Umfang des Ergebnisses wird in aller Regel höher sein. Hier wird ein vergleichendes Beispiel Anhaltspunkte für diese Aussage liefern. Für die Repräsentation der Eigenschaft *Cflow* werden alle Folgeknoten der Starttriggershadows untersucht. Damit der Vergleich ausgewogen bleibt, wird davon ausgegangen, dass für den Starttriggershadow der Eigenschaft *Cflow* die gleiche Anzahl an Elementen in Betracht kommt wie für den ersten Trigger der Sequenz der Eigenschaft *Ereignissequenz*. Der Einfachheit halber soll auch *ein* weiteres Triggersymbol der Eigenschaft *Ereignissequenz* betrachtet werden, bei dem ebenfalls von einer vergleichbaren Menge an dadurch spezifizierten Elementen ausgegangen wird, wie bei dem Endtrigger der Eigenschaft *Cflow*. Für die Eigenschaft *Cflow* seien die Endtriggershadows `Sub1Top1.critAqu()` und `Sub2Top1.critAqu()` spezifiziert.

Die Pfade, die sich für die Eigenschaft *Cflow* in einem Aufrufgraphen ergeben, befinden sich alle in einem abgegrenzten Bereich des Programms. Der Kontrollflussgraph, der für dasselbe Programm erstellt wird, baut, wie oben beschrieben, auf dem Aufrufgraphen auf. Die Elemente eines Kontrollflussgrahpen, die die Eigenschaft *Ereignissequenz* repräsentieren, lassen sich in dem Aufrufgraphen entsprechenden Knoten zuordnen. Werden in Abbildung 3.7 die Bereiche des Aufrufgraphen, die von der Eigenschaft *Ereignissequenz* betroffen sind, (3.7b) mit den Bereichen die durch die Eigenschaft *Cflow* betroffen sind (3.7a), verglichen, wird deutlich, dass eine Repräsentation der Eigenschaft *Ereignissequenz* weit weniger lokal ist.

Obwohl eine solche Analyse zur Lokalisierung der Repräsentation um einiges aufwändiger erscheint, ist es wohl nicht unmöglich. Selbst wenn komplexere reguläre Ausdrücke benutzt wer-

den, gibt es Wege, um Abschnitte eines Kontrollflussgraphen zu berechnen, die die spezifizierte Eigenschaft repräsentieren.

In einer Arbeit aus dem Gebiet des Model Checkings ist ein Werkzeug vorgestellt worden, das dies für die Programmiersprache *C* verwirklicht hat (vgl. Chen u. Wagner 2002). Der Fokus dieses Werkzeugs liegt auf dem Erkennen von sicherheitskritischen Zuständen in einer Software, die durch einen regulären Ausdruck beschrieben werden. Wie die Abschnitte des Kontrollflussgraphen aussehen, die dieses Werkzeug als sicherheitskritisch aufgrund des Ausdrucks eingestuft hat, ist nicht detailliert genug aus der Arbeit hervorgegangen. An dieser Stelle lässt sich noch nicht abschließend behaupten, ob eine Repräsentation für die Eigenschaft *Ereignissequenz* möglich ist, und wenn sie möglich ist, ob sich sinnvoll damit arbeiten lässt. Wie eingangs angekündigt, wird die Repräsentation für diese Eigenschaft in dieser Arbeit ausblickhaft bleiben.

3.2.2.3 Repräsentation der Eigenschaft Ereignissequenz

Für die Repräsentation der Joinpointeigenschaft *Ereignissequenz*, aufbauend auf einem Kontrollflussgraphen, gelten zunächst dieselben Betrachtungen wie für die Eigenschaft *Cflow*. Es werden sich ebenfalls Pfade ergeben, mit dem Unterschied, dass hier ein anderer Graph zugrunde liegt. Sinnvollerweise werden auch hier die Kanten nach demselben Schema qualifiziert werden müssen, und es werden sich dadurch Mengen von Pfaden ergeben, die eine Qualifizierung aufweisen (sicher, bedingt, mehrfach).

Für die Eigenschaft *Cflow* galt, dass sie auf der Grundlage von zwei oder mehr, in der Repräsentation hintereinander stehenden, Triggern spezifiziert wurde. Für die hier behandelte Joinpointeigenschaft werden reguläre Ausdrücke spezifiziert, die eine unendliche Menge von Reihenfolgen der Triggershadows definieren können. Obwohl sich im Kontrollfluss eine repräsentative endliche Menge darstellen lässt, kann auch diese sehr groß sein. An einem Beispielpointcut, der eine Joinpointeigenschaft *Ereignissequenz* spezifiziert, wird dies deutlich werden (siehe Abbildung 3.8)[6].

Dieser Pointcut ist im Kontext eines Editors zu sehen, der durch diesen Aspekt automatisch, nach jeder fünften Aktion ein automatisches Speichern durchführt. Die Vorstellung der Menge möglicher Kontrollflüsse in einer Editorapplikation offenbart, wie hoch die Anzahl an Pfaden sein muss, für die die spezifizierte Ereignissequenz zutrifft. Die Sinnhaftigkeit einer solchen Repräsentation der Eigenschaft *Ereignissequenz*, als Grundlage für den Einsatz beim Refaktorisieren aspektorientierter Programme, bleibt hier ungeklärt. Von weiteren Betrachtungen zu der Joinpointeigenschaft *Ereignissequenz* wird von hier an abgesehen, da sie in einem der Arbeit angemessenen Umfang voraussichtlich nicht zu einem besser verwertbaren Ergebnis führen werden.

Die weiteren Betrachtungen werden am Beispiel der Joinpointeigenschaft *Cflow* fortgeführt. Die gefundene Repräsentation dieser Joinpointeigenschaft wird daraufhin untersucht, wie sich, beim Vergleich von Programmversionen vor und nach einer Quellcodeänderung, die Repräsentation verhalten hat und welche Aussagen sich ableiten lassen bezüglich einer Verhaltenserhaltung oder der Änderung des Verhaltens.

[6] Beispiel entnommen aus: (Allan u. a. 2005)

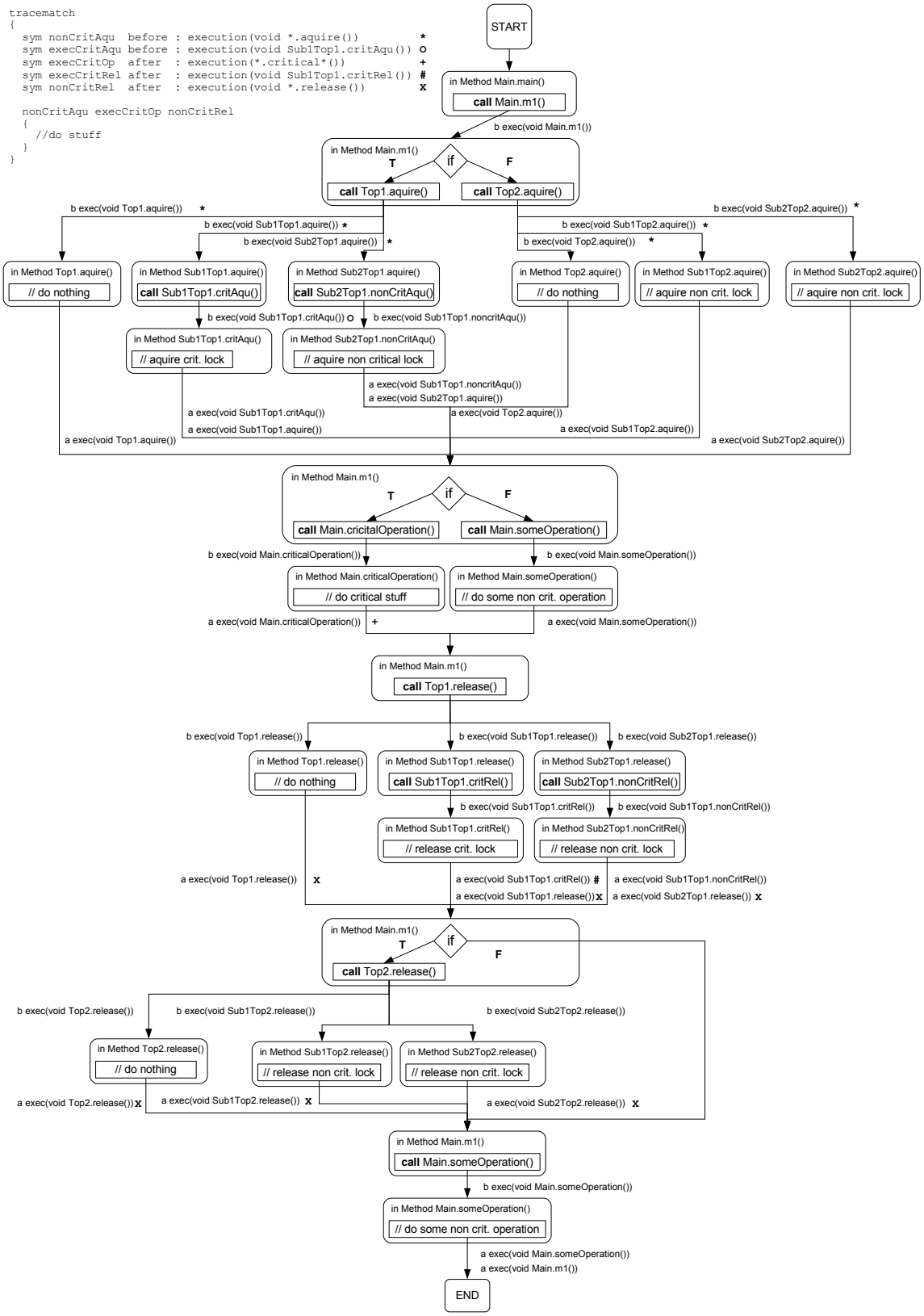


Abbildung 3.6: Repräsentation angelehnt an einen Kontrollflusgraph, mit den für Ereignissequenzen wichtigen Triggern

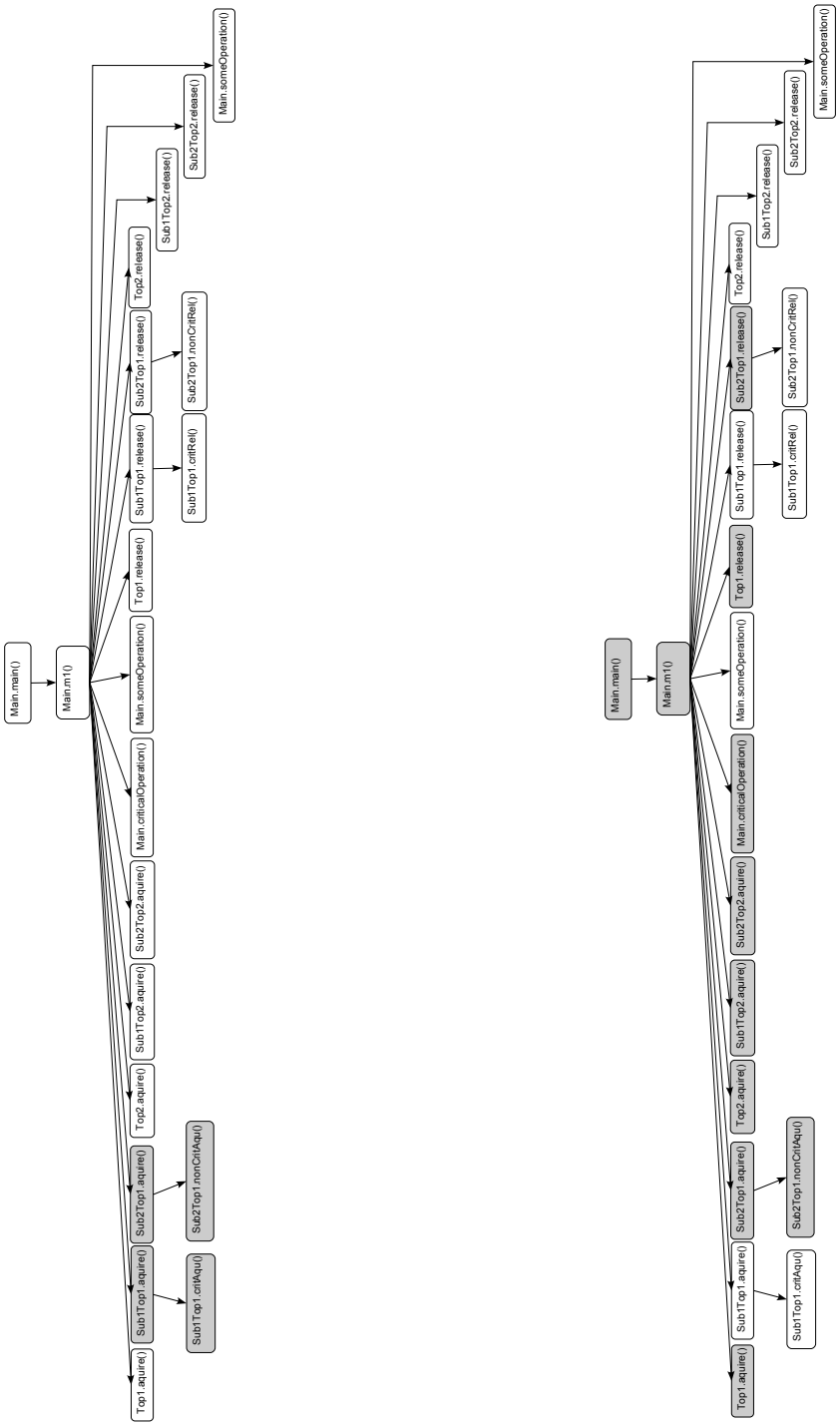


Abbildung 3.7: Bereiche in einem Aufrufgraph, die jeweils durch eine Eigenschaft *Cflow* und eine Eigenschaft *Ereignissequenz* betroffen sind

```
tracematch
{
  sym save after:
       call (* Application.save())
       || call(* Application.autosave())
  sym action after:
       call (* Command.execute())

  action[5]
  {
    Application.autosave();
  }
}
```

Abbildung 3.8: Ein Aspekt der das automatische Speichern unter Verwendung der Joinpointeigenschaft *Ereignissequenz* realisiert.

3.3 Quellcodeänderungen und ihre Auswirkungen in den Repräsentationen

Die Untersuchung der Auswirkung von verschiedenen Quellcodeänderungen soll erkenntlich machen, welche der Änderungen tatsächlich dazu führen, dass eine Joinpointeigenschaft nicht mehr zutreffen kann, und es soll herausgearbeitet werden, welche Änderungen zwar einen Einfluss auf die Repräsentation der Struktur haben aber zu einem äquivalenten Bindungsverhalten führen werden. Auch Quellcodänderungen, die keinen Einfluss auf die Repräsentation haben können, sollen identifiziert werden, sodass für diese Quellcodeänderungen Analysen der Repräsentation vermieden werden können.

Für die Joinpointeigenschaft *Cflow* ist eine verhältnismäßig vollständige Repräsentation entwickelt worden, auf der sich Änderungen relativ genau ablesen lassen sollten. Für die Eigenschaft *Ereignissequenz* ist eine entsprechende Repräsentation noch undeutlich geblieben, sie wird deswegen nicht weiter untersucht.

3.3.1 Auswirkungen auf Multitrigger-Joinpointeigenschaften

Für alle Multitrigger-Joinpointeigenschaften wird sich die Analyse der Auswirkung von Quellcodeänderungen in zwei (gegebenenfalls mehr) Phasen unterteilen. Bevor eine Analyse auf der Struktur stattfindet, die die Multitrigger–Joinpointeigenschaft repräsentiert, müssen die Quellcodeänderungen hinsichtlich ihrer Auswirkungen auf die Joinpointeigenschaften, die zur Spezifikation der Trigger benutzt wurden, untersucht werden. Für den Fall, dass hier ebenfalls Multitrigger–Joipointeigenschaften eingesetzt wurden, müssen die triggerspezifizierenden Joinpointeigenschaften solange aufgelöst werden, bis es sich um Singletrigger-Joinpointeigenschaften handelt (in diesem Fall wird es mehr als zwei Phasen geben). Erst wenn die Eigenschaften, auf denen die Multitrigger–Joinpointeigenschaft aufbaut, keine Beeinflussung durch die Quellcodeänderung zu verzeichnen haben (bzw. entsprechende Anpassungen dazu geführt haben), macht es Sinn, nach Auswirkungen in der Repräsentation der darüber liegenden Multitrigger–Joinpointeigenschaft zu suchen.

Bei den in einer Multitrigger-Joinpointeigenschaft eingesetzten Singletrigger-Joinpointeigenschaften können sowohl statische als auch dynamische Joinpointeigenschaften spezifiziert worden

sein. Wie in 3.1.2.2 bereits erwähnt, sind schon Wege zum Umgang mit statischen Joinpointeigenschaften während des Refaktorisierens erarbeitet worden. Für dynamische Singletrigger-Joinpointeigenschaft trifft dies nicht zu. Im weiteren Verlauf wird dennoch davon ausgegangen, dass für jede Art von Joinpointeigenschaft, die für die Spezifizierung der Trigger eingesetzt werden kann, Wege bekannt sind, Auswirkungen auf diese Joinpointeigenschaft festzustellen und gegebenenfalls Anpassungen vorzunehmen.

Sollte sich herausstellen, dass für bestimmte dynamische Singletrigger-Joinpointeigenschaften kein Weg zur Berücksichtigung in einem Refactoring gefunden werden kann, so sollte von der Nutzung solcher Eigenschaften Abstand genommen werden. In Abschnitt 1.3 ist bereits dargelegt worden, dass jede Software, also auch aspektorientierte, Teil eines stetigen Änderungsprozesses sein wird und aus diesem Grunde Refactorings erforderlich sein werden. Scheitert also ein Refactoring an einem Element der aspektorientierten Sprache, wird es sinnvoller sein auf dieses Element von vornherein zu verzichten.

Sind die zugrunde liegenden Joinpointeigenschaften berücksichtigt worden, werden die Quellcodeänderungen auf ihre Auswirkungen auf die aufbauenden Multitrigger-Joinpointeigenschaften hin untersucht werden müssen.

3.3.2 Auswirkungen auf die Repräsentation der Eigenschaft Cflow

Ziel der Analyse sollte es sein herauszufinden, ob eine Quellcodeänderung Auswirkungen auf die Joinpointeigenschaften in den zugrunde liegenden Strukturen hat, sodass die Referenzen eines spezifizierten Pointcuts zu diesen Joinpointeigenschaften in den Strukturen verloren geht, zusätzliche Joinpointeigenschaften referenziert oder falsche. Dafür werden Repräsentationen der jeweiligen Joinpointeigenschaften aufgestellt, wie sie jeweils vor und nach der Quellcodeänderung zu finden wären. Ein Vergleich der Strukturen soll zu dem geforderten Analyseergebnis führen.

Für die Eigenschaft *Cflow* ist die Repräsentation der Joinpointeigenschaft in der Struktur beschrieben worden. Im Folgenden sollen verschiedene Quellcodeänderungen betrachtet werden, um festzustellen, welche Änderungen tatsächlich Auswirkungen haben und welche Änderungen vielleicht Änderungen in der Repräsentation zur Folge haben, an der spezifizierten Eigenschaft aber nichts ändern.

Grundlage für die weiteren Überlegungen sind die in 3.2.1.3 beschriebenen sicheren, unsicheren und mehrfachen Pfade. Sie werden im folgenden **Matchpfade** genannt (abgeleitet aus: „Die Pfade, die zu einem Pointcutmatch führen“). In dieser Phase, in der die Analyse der unterliegenden Singletrigger-Joipointeigenschaften abgeschlossen ist, werden die Matchpfade für die beiden Programmversionen aufgestellt, untersucht und verglichen. Es gibt dann eine Repräsentation R des Programm P vor der Änderung und eine Repräsentation R' für das Programm P' nach der Änderung.

Ein erster Bereich, der sich vorneweg behandeln lässt, sind all diejenigen Änderungen, die sich auf weggefallene Triggershadows beziehen (in R' nicht mehr existent). Weggefallene Triggershadows werden auf jeden Fall zu weniger Matchpfaden führen. Eine gesonderte Behandlung für die Eigenschaft *Cflow* wird aber nicht notwendig sein, da bereits die Analyse der Auswirkungen der Änderungen auf die zugrunde liegenden Singletrigger spezifizierenden Eigenschaften den Umstand festgestellt haben wird.

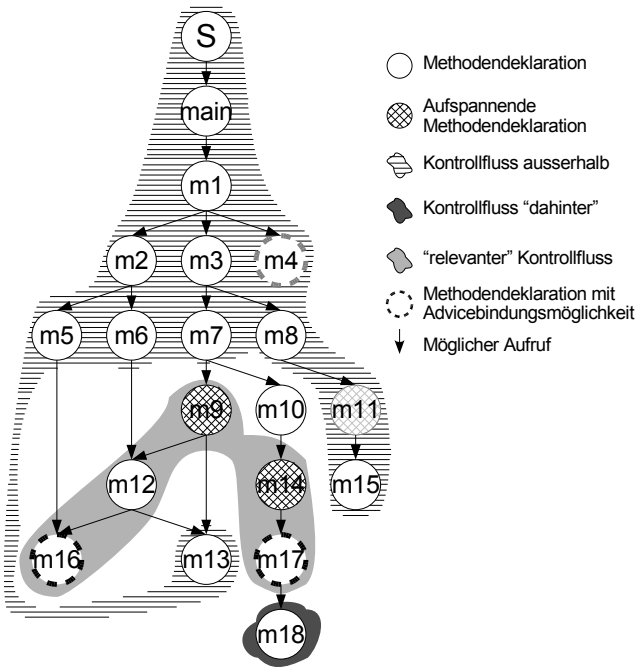


Abbildung 3.9: Abschnitt auf dem gesamten Aufrufgraphen, der relevante von nicht relevanten Programmelementen trennt

In Abbildung 3.9 ist eine Repräsentation für eine Programmversion dargestellt. Darin wird sichtbar, wie das Programm in Bereiche teilbar ist, abhängig davon, ob Änderungen an einem Programmelement Auswirkungen auf die Eigenschaft *Cflow* haben können.

Änderungen an Programmelementen, die sowohl in P als auch in P' im jeweiligen Aufrufgraph im gestrichelten Bereich liegen, werden keinen Einfluss auf die Eigenschaft gehabt haben. Dies sind die Elemente, die in keiner der Programmversionen im aufgespannten Kontrollfluss eines Starttriggershadows liegen. Liegt dagegen eine Änderung von Programmelementen im „relevanten“ Bereich, müssen Vergleiche der Matchpfade betrieben werden. Änderungen an Programmelementen, die im Bereich hinter den Endtriggershadows liegen, können nur in einem Fall eine Auswirkung gehabt haben. Dieser Fall liegt vor, wenn ein rekursiver Aufruf aus dem aufgespannten Bereich des Endtriggershadows in den relevanten Bereich bzw. zum Endtriggershadow selbst vorliegt. Entsprechende Änderungen und deren Auswirkungen werden noch erläutert.

Erwähnt werden müssen noch die Bereiche von der Programmeintrittsmethode, vertreten durch `main()`, bis zu den Starttriggershadows. Es könnte zunächst argumentiert werden, dass es bei Änderungen in diesen Bereichen zu zusätzlichen oder weniger Joinpoints mit der spezifizierten Eigenschaft am Endtriggershadow kommen könnte. Das ist nicht auszuschließen, es sind dann aber nicht zusätzliche oder weniger Joinpoints, die aufgrund von Änderungen existieren, welche die Eigenschaft *Cflow* betreffen. Vielmehr handelt es sich um eine Änderung von möglichen Joinpoints am Starttriggershadow, die durch Änderungen bezüglich der für die Starttrigger spezifizierten Joinpointeigenschaften zustande kommen.

Für die verbliebenen Programmelemente wird es leicht nachzuvollziehen sein, dass die einzigen

```
public void st()
{
  m1();
}
public void m1()
{
  mt();
}
public void mt()
{
}
```

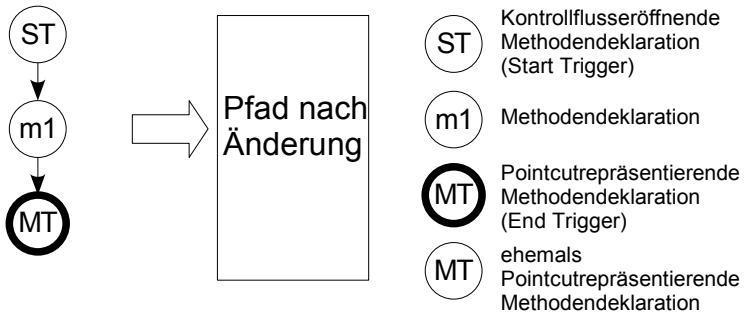


Abbildung 3.10: Quellcode vor der Änderung und eine Visualisierung des Matchpfades

Quellcodeänderungen, die für einen *Cflow* eine Auswirkung haben können, diejenigen Änderungen sind, die etwas an den Aufrufverhältnissen ändern. Andere Änderungen, wie zum Beispiel das Umbenennen von Programmelementen die im objektorientierten Sinne zu semantisch äquivalenten Referenzen und Operationen führen, werden keinen Einfluss haben.

3.3.2.1 Auswirkungen einzelner Quellcodeänderungen

An dieser Stelle sollen verschiedene Quellcodeänderungen betrachtet werden, die Auswirkungen auf Aufrufabhängigkeiten haben. Für jedes der Beispiele soll eine Aussage getroffen werden, ob sich dadurch eine Änderung in einer spezifizierten Eigenschaft *Cflow* ergeben könnte. Die Änderungen führen immer von einem einzelnen sicheren Matchpfad – für die Repräsentation vor der Quellcodeänderung – zu einem andersartigen Matchpfad (oder mehreren), in der Version nach der Quellcodeänderung. Kann für eine Quellcodeänderung argumentiert werden, dass sie nicht zu einem geänderten Bindungsverhalten führt, stellen die beiden Versionen semantisch äquivalente Ausprägungen dar.

In Abbildung 3.10 ist der Quellcode, durch den der Ausgangs-Matchpfad entsteht, dargestellt. Die Methode `st()` ist ein Starttriggershadow, bei der Methode `mt()` handelt es sich um einen Endtriggershadow. Daneben sind die Aufrufverhältnisse visualisiert. Quellcode und Abbildung stellen den Ausgangspunkt der im Folgenden beschriebenen Änderungen dar.

Die Änderungen sind bezüglich der Art involvierter Kanten in drei Gruppen geordnet. Die erste Gruppe behandelt nur sichere Methodenaufrufe, in der zweiten Gruppe finden sich Änderungen, die bedingte Pfade erzeugen können, und in der dritten Gruppe werden Änderungen unter Einbeziehung von Iterationen und Rekursionen betrachtet.

Uneingeschränkte Aufrufe

Die erste Gruppe behandelt Änderungen an direkten, uneingeschränkten Methodenaufrufen. Das bedeutet, es werden keine Aufrufe behandelt, die durch Kontrollflussanweisungen oder den Mechanismus des dynamischen Bindens beeinflusst werden.

```
public void st()
{
  mt();
}
public void mt()
{
}
```

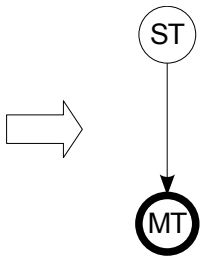


Abbildung 3.11: Geänderter Quellcode. Statt über die Methode `m1()` wird der Endtriggershadow direkt aufgerufen

```
public void st(){
  int result = m2();
}
public void m2(){
  int i = 1;
  mt();
  return i * 2;
}
public void mt(){}
```

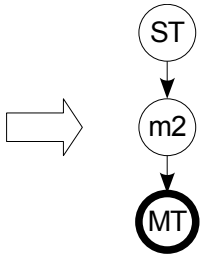


Abbildung 3.12: Methode `m1()` wird durch eine andere Methode `m2` ersetzt

Ersetzen einer Methodendeklaration durch direkten Aufruf
Vor der Änderung verlief der Matchpfad von einem Starttriggershadow über einen Zwischenknoten zu einem Endtriggershadow. Im Rahmen der Änderung wird der Zwischenknoten entfernt und durch einen direkten Aufruf vom Starttriggershadow zum Endtriggershadow ersetzt (siehe Abbildung 3.11). Diese Änderung entspricht den Änderungen, die während eines *Inline Method* Refactorings vorgenommen werden (vgl. Fowler u. a. 2005, Seite 117). In der Repräsentation würde die Änderung wie eine Änderung von Abbildung 3.10 zu Abbildung 3.11 aussehen. Es wird deutlich, dass sich der Matchpfad zwar geändert hat, die Eigenschaft *Cflow* aber noch im gleichen Umfang zutrifft.

Ersetzen einer Methodendeklaration im Matchpfad
Das Ersetzen einer Methodendeklaration durch ein andere Methodendeklaration ist ebenfalls unbedeutend für die Eigenschaft *Cflow*, solange keine weiteren Aufrufe zum Endtriggershadow in der ersetzten Methode erscheinen oder vom Starttriggershadow keine weiteren Aufrufe zu dieser Methode existieren (Abbildung 3.12). Dabei spielt es für die Eigenschaft *Cflow* keine Rolle, ob es sich bei der ausgetauschten Methode um eine semantisch äquivalente Methode handelt oder nicht.

Hinzufügen einer Methodendeklaration in einen Matchpfad
In diesem Szenario wird im Matchpfad eine weitere Methode aufgerufen (vor oder nach der bestehenden Methode im Pfad). Auch diese Änderung beeinflusst die Eigenschaft *Cflow* nicht, da es keine weiteren oder weniger Umstände gibt unter denen ein Joinpoint die spezifizierte Eigenschaft aufweist (siehe auch Abbildung 3.13).

```
public void st()
{
  m1();
}
public void m1()
{
  m2();
}
public void m2()
{
  mt();
}
public void mt()
{
}
```

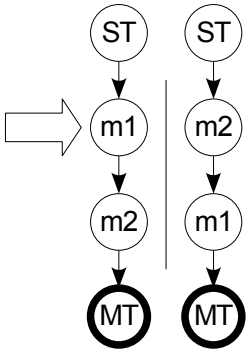


Abbildung 3.13: Einfügen einer weiteren Methode `m2()` in den Aufrufpfad

```
public void st()
{
  m1();
}
public void m1()
{
}
public void mt()
{
}
```

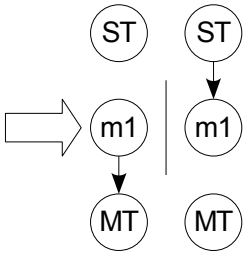


Abbildung 3.14: Entfernen eines Methodenaufrufs

Entfernen eines Aufrufs im Matchpfad
Die Eigenschaft *Cflow* kann nicht mehr zustande kommen, wenn ein Aufruf, der für die Existenz einer Aufrufabhängigkeit zwischen Starttriggershadow und Endtriggershadow notwendig ist, nicht mehr vorhanden ist. Das Ergebnis der Änderung, wie es en Abbildung 3.14 dargestellt wird, kann also nicht mehr zu einer semantisch äquivalenten Pointcutreferenz für diese Eigenschaft führen.

Entfernen einer Methodendeklaration aus einem Matchpfad
Auch wenn eine Methodendeklaration, inklusive ihrer Aufrufe, vollständig entfernt wird, sind offensichtlich Joinpoints mit der spezifizierten Eigenschaft *Cflow* verloren gegangen (siehe Abbildung 3.15).

Hinzufügen eines parallelen Methodenaufrufs in einem Matchpfad
In diesem Beispiel wird der Endtriggershadow einmal mehr aufgerufen. Entsprechend gibt es einen sicheren Matchpfad mehr, denn sobald der Starttrigger am Starttriggershadow eintritt wird der Endtriggershadow nicht mehr einmal, sondern zweimal aufgerufen, und es wird zwei Endtrigger geben, die die geforderten Eigenschaft aufweisen (siehe Abbildung 3.16).

Hinzufügen eines Aufrufs und einer Methodendeklaration
Diese Änderung am Programm wird keine Auswirkung auf Aspektbindungen haben, die von einer spezifizierten Eigenschaft *Cflow* abhängen. In der Graphik aus Abbildung 3.17 wird deutlich,

```
public void st()
{
}
public void mt()
{
}
```

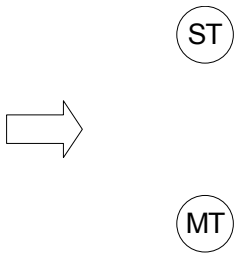


Abbildung 3.15: Entfernen einer Methode und ihrer Aufrufe

```
public void st()
{
  m1();
}
public void m1()
{
  mt();
  mt();
}
public void mt()
{
}
```

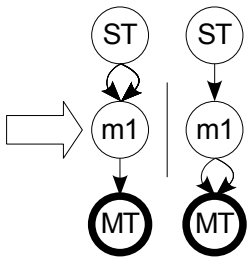


Abbildung 3.16: Hinzufügen eines Aufrufs

dass kein weitere Aufrufabhängigkeit zwischen den beiden Triggershadows entsteht, es wird folglich auch keinen zusätzlichen Matchpfad geben. Die Änderung hat also keinen Einfluss, es bleibt bei einem Endtrigger pro Starttrigger und damit bei einem gleichen Bindungsverhalten.

Einbinden einer Methodendeklaration in einen Pfad

Dieses Beispiel unterscheidet sich in der Auswirkung auf die spezifizierte Eigenschaft *Cflow* nicht von dem Beispiel, in dem ein zusätzlicher Aufruf zu einer vorhandenen Methode im Pfad eingefügt wurde. Es ist ebenso ein weiterer Matchpfad vorhanden, die Tatsache, dass dieser über eine andere Methodendeklaration geht, ist bezüglich der Eigenschaft nicht relevant. Abbildung 3.18 stellt das Beispiel dar.

```
public void st(){
  m1();
}
public void m1(){
  mt();
}
public void m2(){
  mt();
}
public void mt(){}
```

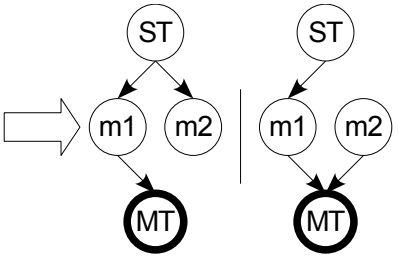


Abbildung 3.17: Hinzufügen eines Aufrufs und einer Methodendeklaration

```
public void st(){
  m1();
  m2();
}
public void m1(){
  mt();
}
public void m2(){
  mt();
}
public void mt(){}
```

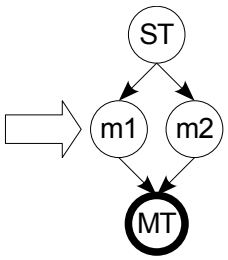


Abbildung 3.18: Einbinden einer Methodendeklaration in einen Pfad

```
public void st(){
  m1();
}
public void m1(){
  if(condition){
    mt();
  }
}
public void mt(){}
```

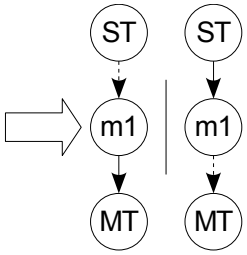


Abbildung 3.19: Ersetzen eines definitiven Aufrufs durch einen bedingten

Bedingte Aufrufe

Die nachfolgenden Beschreibungen von Quellcodeänderungen und ihren Auswirkungen werden aus sicheren Kanten bedingte Kanten machen. In den Beispielcodeabschnitten wird dafür immer eine `if`-Anweisung verwendet werden. Es sind natürlich auch alle anderen den Kontrollfluss beeinflussenden Anweisungen möglich. In objektorientierten Programmen besteht zusätzlich die Möglichkeit, dass ein Methodenaufruf dynamisch zu einer Methode aufgelöst wird. Für die Aufrufabhängigkeit stellt das auch einen bedingten Methodenaufruf dar.

Ersetzen eines definitiven Aufrufs durch einen bedingten

Durch den Einschluss des Aufrufs des Endtriggershadows in ein `if`-Konstrukt wird sich das Bindungsverhalten ändern. Während vorher bei jedem Eintreten in den Starttriggershadow am Endtriggershadow ein Endtrigger existierte, der die spezifizierte Eigenschaft *Cflow* besaß, ist dies jetzt abhängig von der Bedingung. Dabei wird offensichtlich keine Rolle spielen, an welcher Stelle im Pfad der Aufrufabhängigkeiten die Bedingung hinzukommt. Diese Änderung, deren Resultat in Abbildung 3.19 dargestellt ist, hat das Bindungsverhalten geändert.

Hinzufügen eines bedingten Aufrufs

Wird parallel zu einem definitiven Aufruf ein bedingter Aufruf hinzugefügt, so ist bei jedem Eintritt des Starttriggers nach wie vor ein Endtrigger sicher. Hinzu kommt die Möglichkeit eines weiteren Endtriggers. Dieses Beispiel (siehe Abbildung 3.20) hat Ähnlichkeiten mit dem in Abbildung 3.16 dargestellten. Die Änderung unterscheidet sich nur durch die andere Art des hinzugekommenen Matchpfades.

```
public void st()
{
  m1();
}
public void m1()
{
  mt();
  if(condition)
  {
    mt();
  }
}
public void mt(){}
```

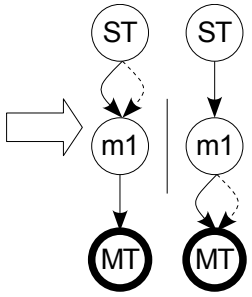


Abbildung 3.20: Hinzufügen eines bedingten Aufrufs

```
public void st(){
  m1();
  m2();
}
public void m1(){
  mt();
}
public void m2(){
  if(condition){
    mt();
  }
}
public void mt(){}
```

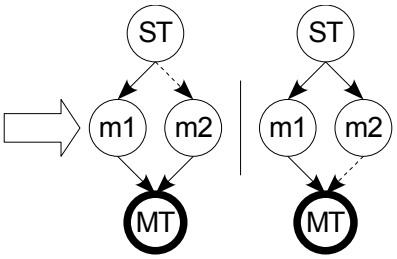


Abbildung 3.21: Hinzufügen eines bedingten Aufrufs zu/von einer zusätzlichen Methode

Hinzufügen eines bedingten Aufrufs zu/von einer zusätzlichen Methode

Wie schon bei sicheren Pfaden unterscheidet sich die Auswirkung der Änderung nicht, wenn der zusätzliche Aufruf über eine andere Methode geht. Das zu diesem in Abbildung 3.21 dargestellte Beispiel entsprechende ist in Abbildung 3.18 zu sehen.

Hinzufügen eines Aufrufs im gegenseitigen Ausschluss

Mit bedingten Aufrufen kommt die Möglichkeit hinzu, dass zwar andere Wege in der Ausführung in Betracht kommen, für bestimmte Konstruktionen aber effektiv nur ein Pfad existieren kann. In Abbildung 3.22 sind im Quellcode nun zwei Methodenaufrufe statt einem vorhanden. Dadurch, dass sich die Aufrufe aber in entsprechenden Codeteilen befinden, schließen sie sich gegenseitig aus. Im Beispiel sind das der „then“- und der „else“-Block der `if`-Anweisung. Das Ergebnis ist immer noch ein Endtrigger, wenn der Starttrigger vorliegt. Es handelt sich also um eine Änderung, die zu äquivalentem Bindungsverhalten führt.

Hinzufügen eines Aufrufs im gegenseitigen Ausschluss über eine zusätzliche Methode

Analog zu den Fällen für sichere Aufrufe und normal bedingte Aufrufe spielt es auch in diesem Beispiel (siehe Abbildung 3.23) keine Rolle, dass der zusätzliche Methodenaufruf über eine andere Methode geht. Das Ergebnis ist dasselbe wie bei einem parallelen Aufruf. Diese Änderung führt also auch zu äquivalentem Bindungsverhalten.

```
public void st()
{
  m1();
}
public void m1()
{
  if(condition){
    mt();
  }
  else{
    mt();
  }
}
public void mt(){}
```

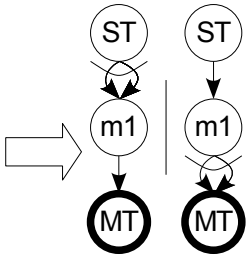


Abbildung 3.22: Hinzufügen eines Methodenaufrufs im gegenseitigen Ausschluss zu einem anderen

```
public void st(){
  if(condition){
    m1();
  }
  else{
    m2();
  }
}
public void m1(){
  mt();
}
public void m2(){
  mt();
}
public void mt(){}
```

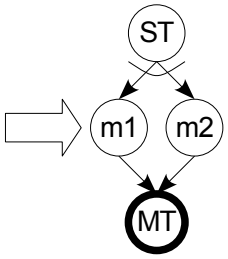


Abbildung 3.23: Hinzufügen eines Methodenaufrufs im gegenseitigen Ausschluss zu einem anderen

```
public void st(){
  m1();
}
public void m1(){
  mt();
}
public void mt(){
  if(condition){
    mt();
  }
  return;
}
```

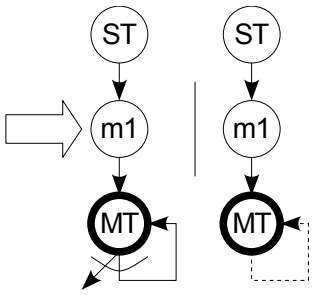


Abbildung 3.24: Hinzufügen einer direkten Rekursion

```
public void st()
{
  m1();
}
public void m1()
{
  mt();
}
public void mt()
{
  if(condition){
    m1();
  }
}
```

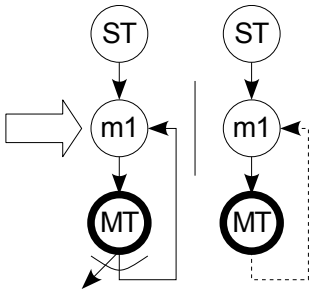


Abbildung 3.25: Hinzufügen einer indirekten Rekursion vom Endtriggershadow zu einem Zwischenelement

Iterative/Rekursive Aufrufe

Die letzte Gruppe bilden die Änderungen, die Iterationsanweisungen oder rekursive Aufrufe einfügen. Im Falle der rekursiven Aufrufe werden hier auch Änderungen betrachtet, die Programmelemente betreffen, die in Bereichen liegen, die für die anderen Änderungen nicht von Bedeutung sind.

Hinzufügen einer direkten Rekursion

Diese Änderung führt dazu, dass es ab einem am Starttriggershadow ausgelösten Starttrigger zu beliebig vielen Endtriggern kommen kann, abhängig von der Bedingung `condition`. Die Änderung hätte auch in einem zusätzlichen `else`-Block an eine andere Stelle in das Programm verzweigen können. Abbildung 3.24 stellt diese Änderungen dar, die ein geändertes, nicht äquivalentes Bindungsverhalten ergeben.

Hinzufügen einer indirekten Rekursion vom Endtriggershadow zu einem Zwischenelement

Dieses Beispiel illustriert, dass es nicht relevant ist, ob es sich bei der Änderung um eine direkte oder eine indirekte Rekursion handelt. Auch hier wird es statt eines Endtriggers am Endtriggershadow zu beliebig vielen Endtriggern pro am Starttriggershadow ausgelöstem Starttrigger kommen (siehe Abbildung 3.25).

```
public void st()
{
  m1();
}
public void m1()
{
  mt();
  mt();
}
public void mt()
{
}
```

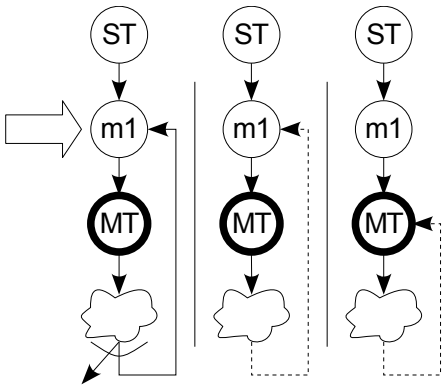


Abbildung 3.26: Hinzufügen einer indirekten Rekursion aus einem Knoten im Kontrollfluss des Endtriggershadows

```
public void st()
{
  m1();
}
public void m1()
{
  if(condition){
    m1();
  }
  mt();
}
public void mt()
{
}
```

```
public void st()
{
  m1();
}
public void m1()
{
  while(condition)
  {
    mt();
  }
}
public void mt()
{
}
```

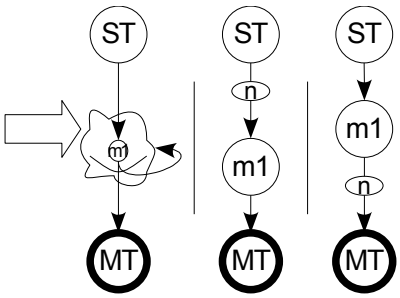


Abbildung 3.27: Hinzufügen einer direkten/indirekten Rekursion an einem Zwischenknoten bzw. Hinzufügen einer iterativen Kante

Hinzufügen einer indirekten Rekursion aus einem Knoten im Kontrollfluss des Endtriggershadows

Die in Abbildung 3.26 dargestellte Änderung stellt schon eine Ausnahme dar. Die Quellcodeänderung findet, im Gegensatz zu den anderen, nicht im relevanten Bereich statt, sondern im vom Endtriggershadow aufgespannten Kontrollflussbereich. Dies führt, entgegen der ersten Behauptung, zu mehr Endtriggern am Endtriggershadow pro eingetretenem Starttrigger.

Hinzufügen einer direkten/indirekten Rekursion an einem Zwischenknoten bzw. Hinzufügen einer iterativen Kante

Das Ergebnis der in Abbildung 3.27 dargestellten Änderungen führt ebenfalls zu einer von der Bedingung `condition` abhängigen Zahl von Endtriggern am Endtriggershadow für jeden Starttrigger am Starttriggershadow. Entsprechend wirken sich diese Änderung wie die drei vorherigen aus.

Hinzufügen einer indirekten Rekursion zum Starttriggershadow

Auch in diesem in Abbildung 3.28 abgebildeten Beispiel handelt es sich, wie in dem in Abbildung 3.26 dargestellten um einen Sonderfall. Allerdings nicht, weil Programmelemente des nicht

```
public void st()
{
  m1();
}
public void m1()
{
  mt();
  mt();
}
public void mt()
{
}
```

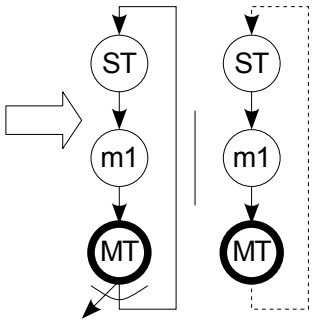


Abbildung 3.28: Hinzufügen einer indirekten Rekursion zum Starttriggershadow

relevanten Bereichs betroffen sind. Obwohl hier eine Rekursion eingeführt wurde und es Programmausführungen geben wird, in denen es mehr Joinpoints gibt, die diese Eigenschaft haben, ist die Eigenschaft selber nicht betroffen. Das liegt daran, dass nach wie vor pro aufgetretenem Starttrigger am Starttriggershadow immer noch nur ein Endtrigger am Endtriggershadow eintreten wird.

Änderung ausgehend von bedingten und mehrfachen Pfaden

Alle vorher beschriebenen Änderungen gehen von einem sicheren Pfad aus. Die folgenden Änderungen gehen von den noch nicht behandelten bedingten und mehrfachen Matchpfaden aus und zeigen wo hier äquivalente Strukturen entstehen oder sich die Strukturen ändern, das heißt, dass der Typ des Matchpfades sich ändert.

Bedingter Matchpfad zu mehrfachem Pfad

Wird in einen Matchpfad zusätzlich, wie im Beispiel in Abbildung 3.29 gezeigt, eine Iteration eingefügt, dann wird der gesamte Matchpfad zu einem mehrfachen Pfad. Dies resultiert direkt aus der in Abschnitt 3.2.1.3 definierten Aussage, dass eine mehrfache Kante eine bedingte Kante überdeckt. Statt der Iteration hätte auch eine Rekursion stehen können. Diese Änderung bewirkt also auch eine Änderung im Bindungsverhalten.

Bedingter Matchpfad zu sicherem Pfad

In diesem Beispiel (siehe Abbildung 3.30) wurde zu dem `if`-Konstrukt neben dem `then`-Block ein `else`-Block eingeführt, in dem dieselbe Methode aufgerufen wird. Dadurch, dass die beiden Aufrufe nur im gegenseitigen Ausschluss stattfinden können, wird aus dem bedingten Pfad ein sicherer.

Änderungen in einem mehrfachen Pfad ohne Auswirkung

In Abbildung 3.31 ist dargestellt, dass das Ändern von sicheren Kanten im mehrfachen Matchpfad in bedingte Kanten oder mehrfache Kanten keine Auswirkung auf die Art des Matchpfads und damit auf das Bindungsverhalten haben wird. Statisch feststellbar ist lediglich, dass es zu beliebig vielen Endtriggern am Endtriggershadow kommen kann, angefangen bei keinem bis hin zu unendlich vielen.

```
public void st()
{
  if(condition){
    m1();
  }
}
public void m1()
{
  mt();
}
public void mt()
{
}
```

(a) Quellcode vor Änderung

```
public void st(){
  if(condition){
    m1();
  }
}
public void m1(){
  while(condition2){
    mt();
  }
}
public void mt()
{
}
```

(b) Quellcode nach Änderung

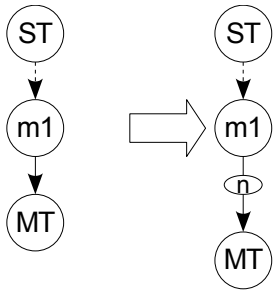


Abbildung 3.29: Änderung von einem bedingten Matchpfad in einen mehrfachen Matchpfad durch Hinzufügen einer Iteration

```
public void st()
{
  if(condition){
    m1();
  }
}
public void m1()
{
  mt();
}
public void mt()
{
}
```

(a) Quellcode vor Änderung

```
public void st(){
  if(condition){
    m1();
  }
  else {
    m1();
  }
}
public void m1()
{
  mt();
}
public void mt(){}
```

(b) Quellcode nach Änderung

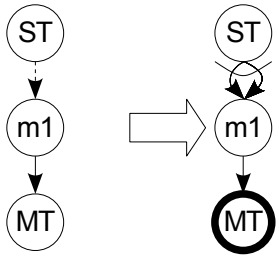
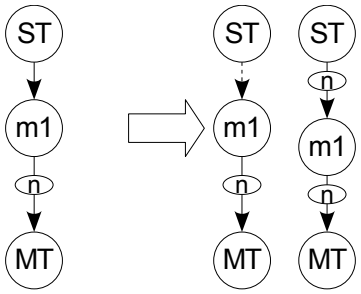


Abbildung 3.30: Änderung von einem bedingten Matchpfad in einen sicheren Matchpfad durch Hinzufügen eines weiteren Aufrufs im gegenseitigen Ausschluss

```
public void st()
{
  m1();
}
public void m1()
{
  while(cond)
  {
    mt();
  }
}
public void mt()
{
}
```



(a) Quellcode vor Änderung

```
public void st()
{
  if(cond1){
    m1();
  }
}
public void m1()
{
  while(cond){
    mt();
  }
}
public void mt(){}
```

(c) Quellcode nach Änderung

```
public void st(){
  while(cond1){
    m1();
  }
}
public void m1(){
  while(cond){
    mt();
  }
}
public void mt(){}
```

(d) Quellcode nach Änderung

Abbildung 3.31: Änderungen in einem mehrfachen Matchpfad ohne Auswirkung

3.3.2.2 Bedeutung der ermittelten Auswirkungen für das Analyseergebnis

In den vorangegangenen Betrachtungen sind verschiedene Änderungen bezüglich eines Ausgangsquellcodes gezeigt worden, in dem es einen Matchpfad gab. Es haben sich jeweils unterschiedliche Auswirkungen auf eine spezifizierte Eigenschaft *Cflow* ergeben. Die Auswirkungen sind im Rahmen der zuvor entwickelten Repräsentation beschrieben worden. Die beschriebenen Änderungen sind nicht umfassend, in dem Sinne, dass nicht jede Ausgangssituation und jede Endsituation betrachtet und zueinander in Beziehung gesetzt wurde. Die angeführten Änderungen stehen aber repräsentativ für weitere und komplexere Änderungen, deren Auswirkungen sich analog verhalten werden, oder als Verbindungen der dargestellten Änderungen und deren Auswirkungen aufgefasst werden können.

Die erste große Gruppe von Änderungen gingen von einem sicheren Matchpfad aus und haben in ihrer Auswirkung zu verschiedenen Endszenarien geführt. Es gab Fälle, in denen es zu einem äquivalenten Matchpfad gekommen ist, zu einem zusätzlichen oder zu keinem Matchpfad. Auch eine andere Qualität des Matchpfades hat sich bei manchen Änderungen ergeben, sodass danach statt des sicheren ein mehrfacher oder bedingter Matchpfad vorhanden war. Danach sind noch solche Änderungen betrachtet worden, die von einem bedingten oder mehrfachen Matchpfad ausgegangen sind. Die folgende Auflistung gibt eine Zuordnung der einzelnen Beispiele zu der Art der Auswirkung wieder.

- Es hat keine Änderung gegeben, im Ergebnis wird semantisch Äquivalentes referenziert (Abbildungen: 3.11, 3.12, 3.13, 3.17, 3.22, 3.23, 3.28, 3.31).
- Es gibt zusätzliche sichere, bedingte, mehrfache Pfade (Abbildungen 3.16 und 3.18 für zusätzlich sichere, Abbildungen 3.20 und 3.21 für zusätzliche bedingte).
- Es gibt weniger sichere, bedingte, mehrfache Pfade (Abbildungen 3.14 und 3.15 für weniger sichere).
- Pfade die vorher sicher, bedingt, mehrfach waren haben nach der Änderung eine andere Qualifizierung (mehrfach, bedingt, sicher). In Abbildung 3.19 ist ein sicherer Pfad zu einem bedingten geworden, in den Abbildungen 3.24, 3.25, 3.26, 3.27 zu einem mehrfachen. Abbildung 3.30 zeigt die Änderung von einem bedingten zu einem sicheren und Abbildung 3.31 zu einem mehrfachen.

Da es in einer Repräsentation R nicht immer nur einen Matchpfad geben wird, sondern mehrere, ist es notwendig festzustellen, welche Matchpfade in R zu welchen Matchpfaden in R' in Beziehung stehen. Die Bezeichnung R meint hier die Repräsentation einer spezifizierten Eigenschaft *Cflow* des Programms P vor der Änderung, R' entsprechend die Repräsentation, die sich durch dieselbe Spezifikation im Programm P' nach der Änderung ergibt. Der Versuch, jeden Matchpfad aus R auf Gleichheit mit jedem anderen Matchpfad aus R' zu untersuchen, ist hier nicht sinnvoll, da es nicht nur darum geht, gleiche Pfade zu finden, sondern Pfade, die sich entsprechen. Vorteilhafter wird es sein, die Information über die Änderung mit einzubeziehen. Wenn zum Beispiel um einen Methodenaufruf herum eine Bedingung aufgebaut wird, enthält die Änderung die Information, welche beiden Methodendeklarationen in P und P' diese Modifikation eines enthaltenen Aufrufs beinhalten. Die Knoten der Repräsentationen R und R' referenzieren Methodendeklarationen der zugehörigen Programmversion. Jeder Matchpfad in einer der Repräsentation kann Knoten enthalten, die diese geänderte Methodendeklaration referenziert. Ist das der Fall, so stehen die Matchpfade zueinander in Beziehung, die ein äquivalentes Programmelement referenzieren.

Ein Verfahren, welches die Auswirkungen von Quellcodeänderungen auf eine spezifizierte dynamische Joinpointeigenschaft *Cflow* analysiert, wird entsprechend der vorangegangenen Überlegungen zunächst Auswirkungen auf die genutzten Singletrigger-Joinpointeigenschaften ermitteln. Danach werden die Repräsentationen R und R' aufgebaut. Diejenigen Teile der Repräsentationen, in denen es eine Änderung gegeben hat, werden in beiden Repräsentationen miteinander verglichen. Die Teile, in denen es in keiner der beiden Versionen eine Änderung gegeben hat, bedürfen hingegen keiner weiteren Betrachtung, denn sie können keinen Einfluss auf die spezifizierte Eigenschaft gehabt haben.

Bei den Vergleichen der betroffenen Teile in den Repräsentationen wird ermittelt, ob die unterschiedlichen Matchpfade der beiden Repräsentationen äquivalent im Sinne der spezifizierten Eigenschaft *Cflow* sind. Hat sich für keinen der ermittelten Matchpfade eine Änderung in der Qualifizierung ergeben, so hat es sich um eine Änderung ohne Auswirkung auf die spezifizierte Eigenschaft *Cflow* gehandelt, selbst wenn in R und R' korrespondierende Matchpfade sich in anderer Hinsicht verändert haben. Solche Pfade sind äquivalente Pfade im Sinne der Eigenschaft, eine Anpassung des Pointcuts, um andere semantisch äquivalente Teile in der Repräsentation zu spezifizieren, wird nicht notwendig sein.

In Fällen, in denen es weniger, mehr oder anders qualifizierte Matchpfade in R' gibt, wird die Quellcodeänderung so nicht durchführbar sein. Die Werkzeugunterstützung soll hier nur so weit gehen, dass weggefallene, hinzugekommene oder sich in ihrer Qualifizierung geänderte Matchpfade dem Entwickler präsentiert werden, der dann entscheiden kann, wie er weiter verfahren möchte.

3.3.3 Auswirkungen auf die Repräsentation der Eigenschaft Ereignissequenz

Für die Eigenschaft Ereignissequenz sind im vorigen Abschnitt 3.2.2.3 bereits mögliche Probleme mit einer Repräsentation dieser Eigenschaft dargestellt worden. Da für diese Eigenschaft keine entsprechende Repräsentation erarbeitet werden konnte, wird auch die Untersuchung von Auswirkungen von Quellcodeänderungen ausbleiben müssen. Im Sinne eines Ausblicks kann aber vermutet werden, dass ein Vergleich von Repräsentationen einer Eigenschaft *Ereignissequenz* ungleich schwerer fallen wird als für Repräsentationen einer Eigenschaft *Cflow*.

3.4 Kapitelzusammenfassung

Im ersten Teil dieses Kapitels wurde, ausgehend von Opdykes Begriff der semantischen Äquivalenz von Referenzen und Operationen, eine ähnliche Betrachtung der Verhaltenserhaltung für Pointcuts erarbeitet. Ein Pointcut spezifiziert Joinpointeigenschaften, die auf entsprechenden Strukturen gelten und durch Teile der Struktur repräsentiert sind. Die Anforderung, um Verhaltenserhaltung auch in Gegenwart von Pointcuts zu detektieren und unter Umständen sogar wieder herzustellen, lief darauf hinaus, dass ein Pointcut für die geänderten Programmversion entweder dieselben Eigenschaftsrepräsentation referenzieren muss oder semantisch äquivalente.

Anschließend wurde für die dynamische Joinpointeigenschaft *Cflow* eine Repräsentation modelliert, die von bestimmten dynamischen Umständen abstrahiert (Variablenbelegungen), aber

dennoch jede mögliche Ausführung eines Programms darstellen kann. Für eine Repräsentation der dynamischen Joinpointeigenschaft *Ereignissequenz* ist dies nicht gelungen.

Die Repräsentation der Eigenschaft *Cflow* wurde für verschieden Quellcodeänderungen untersucht. Dabei hat sich für bestimmte Änderungen herausgestellt, dass das Referenzieren semantisch äquivalenter Eigenschaftsrepräsentationen automatisch geschieht. Bei anderen Änderungen wurden keine, mehr oder Repräsentationsteile anderer Art referenziert. In diesen Fällen muss der Pointcut, soweit es möglich ist, entsprechend angepasst werden, um äquivalente Repräsentationsteile zu referenzieren.

Bis zu dieser Stelle sind Änderungen und ihre Auswirkungen auf das Bindungsverhalten durch Pointcuts nur auf theoretischer Ebene beschrieben worden. Es fehlt noch der Zusammenhang zum Quellcode. In den Beschreibungen wurde immer davon ausgegangen, dass die Repräsentationen der Eigenschaften für den Quellcode des Programms vor und nach den Änderungen existieren und verglichen werden können. Wie es vom Quellcode zu einer solchen Repräsentation kommen kann, ist nur ansatzweise und nicht explizit erläutert worden. Außerdem sollen die Repräsentationen nicht vom Entwickler aufgebaut und in ihrer Gesamtheit verglichen werden, sondern von einem Werkzeug berechnet werden und nur in Fällen, in denen es nicht zu denselben oder semantisch äquivalenten Referenzen gekommen ist, der Entwickler mit entsprechenden Teilen der Repräsentation konfrontiert werden.

Kapitel 4

Appoximation und Analyse von Laufzeitverhalten

Im vorangegangenen Kapitel ist beschrieben worden, wie es einem Entwickler mit Hilfe einer Repräsentation der dynamischen Joinpointeigenschaft *Cflow* gelingen sollte, die Auswirkungen von Quellcodeänderungen auf eine spezifizierte Eigenschaft *Cflow* zu erkennen und an dieser Repräsentation die Durchführbarkeit eines Refactorings festzustellen. Diese Repräsentation ist für die meisten Programme, allein wegen des Umfangs, offensichtlich nicht manuell erstellbar. Ebenso wenig wird es möglich sein, ohne zusätzliche Unterstützung, Auswirkungen von Quellcodeänderungen in einer Repräsentation festzustellen. In Abschnitt 3.4 wurde bereits erwähnt, dass diese Unterstützung durch ein Werkzeug realisiert werden soll.

Bevor die Analyse, der Auswirkungen von Quellcodeänderungen auf die dynamische Joinpointeigenschaft *Cflow*, in einem Werkzeug realisiert werden kann, ist es erforderlich Überlegungen anzustellen, wie die in Abschnitt 3.2.1 entwickelte Repräsentation aus dem Quellcode aufgebaut werden kann. Da es sich um eine statische Repräsentation dynamischer Gegebenheiten handelt, kann für diese Repräsentation nur eine Approximation erreicht werden. Schon die Abstraktion von verschiedenen Wertebelegungen für verschiedene Matchpfade stellt eine konservative Approximation dar. Hinzu kommt, dass die Repräsentation auf einem Aufrufgraph aufbaut, der mit einer von verschiedenen Methoden erstellt werden muss, die alle nur bis zu einem gewissen Grad den *idealen* Aufrufgraphen berechnen können.

Anhand einer Übersicht verschiedener Algorithmen und der Erarbeitung relevanter Faktoren bezüglich der Eignung für die Repräsentation, wird sich ein Algorithmus identifizieren lassen, der für die Zwecke dieser Arbeit die besten Voraussetzung hat. Darauf aufbauend wird ein Verfahren beschrieben werden können, welches die Repräsentation aus dem Aufrufgraphen anhand einer spezifizierten Joipnointeigenschaft *Cflow* ermittelt. Für das zu entwickelnde Werkzeug wird dann ein Verfahren benötigt, dass anhand der Repräsentationen Aussagen darüber bereitstellen kann, ob eine Quellcodeänderung Auswirkungen auf eine spezifizierte Eigenschaft *Cflow* hat und um welche Art von Auswirkung es sich handelt.

4.1 Approximation von Aufrufgraphen

In den vergangenen Jahren sind eine große Zahl von Arbeiten und Papieren veröffentlicht worden, die sich mit Algorithmen zur Approximation von Aufrufgraphen beschäftigten. Mit der Einführung der objektorientierten Programmierung hat sich, durch das dynamische Binden von Methoden, die Komplexität der Berechnung eines genauen Graphen weiter erhöht. In diesem

Kontext sind ebenfalls eine Vielzahl von weiteren Veröffentlichungen erschienen. Der Aufrufgraph, der sich bei der Berechnung mit einem konkreten Algorithmus ergibt, unterscheidet sich dabei in verschiedenen Hinsichten von einem Ergebnis eines anderen Algorithmus. Eine Aufstellung der untersuchten Algorithmen findet sich in Anhang A.

4.1.1 Klassifikation der Algorithmen

Die Menge der beschriebenen Algorithmen kann in verschiedene Gruppen eingeteilt werden, die jeweils eine grundsätzliche Herangehensweise darstellen. Ein Algorithmus kann auch verschiedene Herangehensweisen kombinieren, wodurch er mehreren Gruppen gleichzeit angehört. Im Folgenden sollen diese Gruppen kurz erläutert werden, um einen Überblick zu gewinnen.

4.1.1.1 Kontextsensitive, Kontextinsensitive Algorithmen

Kontextsensitive Algorithmen unterscheiden sich von den kontextinsensitiven durch das Miteinbeziehen des Aufrufkontextes bei der Generierung des Aufrufgraphen. Der Aufrufkontext kann Verschiedenes umfassen und die Anzahl von Ebenen, deren Kontext miteinbezogen wird, unterscheidet sich für verschieden Algorithmen oder ist sogar variabel. Als Kontextinformation kommen zum Beispiel die letzten Methodenaufrufe (repräsentiert als Zeichenkette) zum Einsatz oder die Objekte an denen die letzten Methodenaufrufe stattgefunden haben (vgl. (Sharir u. Pnueli 1981, zitiert nach Ryder (2003)) und (Milanova u. a. 2002)).

4.1.1.2 Propagation Based Algorithmen

Propagation based Algorithmen übergeben (propagieren) Typinformationen von Methodenaufruf zu Methodenaufruf, jeweils in Aufrufrichtung und in Rückkehrrichtung entsprechend den übergebenen Parametern und dem zurückerhaltenen Rückgabewert. Durch die Propagierung der Typinformationen werden, abhängig vom Algorithmus, verschiedene Typmengen beeinflusst (durch Hinzufügen bzw. Entfernen von Typen). Algorithmen dieser Gruppe werden in Tip u. Palsberg (2000) (Algorithmen: CTA, MTA, FTA und XTA) und in Sundaresan u. a. (2000) (Algorithmen: VTA, DTA) beschrieben.

4.1.1.3 Algorithmen die auf einer Points-to Analyse basieren

Points-To Analysen erlauben es, für eine gegebene Variable im Programm eine Menge von möglichen Typen angeben zu können, da sie die Beziehung zwischen Pointern (in Java: Objektreferenzen) und Daten (in Java: instanziierten Objekten) eines Programmes herstellen. Insofern dienen Points-To Analysen indirekt dem Aufbau von Aufrufgraphen, indem mittels der aufgebauten Objektmengen für einen Methodenaufruf entschieden werden kann, welche Empfängertypen und Parametertypen in Frage kommen. Points-To Analysen bieten den zusätzlichen Vorteil, die statisch ermittelten Objektinstanzinformationen nicht nur für den Aufrufgraphalgorithmus zur Verfügung zu haben, sondern auch für andere Analysen.

Gleichheitsbasierte und submengenbasierte Algorithmen

Eine Points-To Analysen lässt sich einer von zwei Gruppen von Algorithmen zuordnen, gleichheitsbasiert (engl. equality-based) und submengenbasiert (engl. subset-based). Equality-based Algorithmen setzen die Mengen für zwei Variablen gleich, sobald eine Zuweisung stattfindet: z. B. `p = q;` $\rightarrow PointsTo(p) = PointsTo(q)$. Subset-based Algorithmen erzeugen Aussagen über Teilmengenverhältnisse für Zuweisungen: z. B. `p = q;` $\rightarrow PointsTo(p) \supseteq PointsTo(q)$. Entsprechend werden von Equality Based Algorithmen wesentlich kleinere Mengen erzeugt als von Subset Based Algorithmen.

Für statisch getypte Sprachen wie Java sind equality-based Algorithmen vorsichtig zu behandeln. Die Zuweisung `p = q;` hat wie beschrieben $PointsTo(p) = PointsTo(q)$ zur Folge, was wiederum das Ergebnis von `q = p;` sein kann. Diese Vertauschung kann im Widerspruch zur Typisierung liegen (ungültiger Downcast).

Flowsensitive und flowinsensitive Algorithmen

Points-To Analysen die den Kontrollfluss miteinbeziehen gehören der Gruppe der Flowsensitiven Algorithmen an, entsprechend ist ein Algorithmus flowinsensitiv wenn die Analyse den Kontrollfluss außen vor lässt.

4.1.2 Genauigkeit von Aufrufgraphen

Bei der Auseinandersetzung mit den verschiedenen Algorithmen ist die Frage der Präzision des resultierenden Aufrufgraphen von wesentlicher Bedeutung. Die Spanne, in der ein Aufrufgraph bezogen auf seine Präzision liegen kann, wird dabei von drei, nur unter theoretischen Gesichtspunkten relevanten, Aufrufgraphen bestimmt.

Am unteren Ende der Spanne steht der Graph $G_{\perp}$ (G bottom). Hier gibt es von jeder Methode zu jeder Methode einen Aufruf. Demgegenüber steht am oberen Ende der Graph $G_{\top}$ (G top), der zwar alle Methoden als Knoten enthält, aber keine Kante (also keinen Aufruf). In der Mitte der Spanne ist der Graph $G_{\mathbf{ideal}}$, der den theoretisch präzisen Graphen darstellt, der sich in der Regel aber nicht berechnen lässt. Theoretisch präzise bedeutet hier, dass der Graph nur die Kanten zwischen Knoten enthält, die in jeder erdenkliche Ausführung auftreten können. Anders formuliert heißt das, dass er nur Kanten enthält, für die es auch eine Ausführung geben kann in der ein Aufruf vorkommt, der durch diese Kante repräsentiert wird.

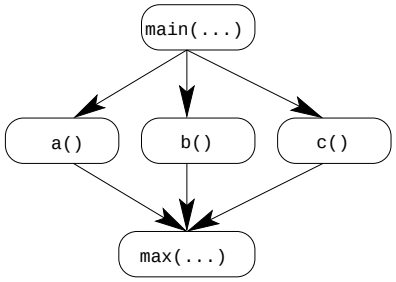
In dieser Spanne gibt es zwei Felder denen sich ein berechneter Aufrufgraph zuordnen lässt. Die Aufrugraphen, die zwischen $G_{\perp}$ und $G_{\mathbf{ideal}}$ angesiedelt sind, werden „vernünftig“ genannt (engl. sound), alle Aufrufgraphen zwischen $G_{\mathbf{ideal}}$ und $G_{\top}$ sind entsprechend „unvernünftig“(engl. unsound). Ob ein Aufrufgraph sound oder unsound ist, ist für fast alle statischen Programmanalysen, die mit Aufrufgraphen arbeiten, eine entscheidende Eigenschaft. Für einen Übersetzer (engl. Compiler), der mit eine Aufrufgraphanalyse den übersetzten Quellcode optimieren will, ist es zum Beispiel entscheidend, dass ein verwendeter Aufrufgraph sound ist. Wäre das nicht so, würde das übersetzte Programm eventuell Optimierungen enthalten, die zu einem Programmverhalten führen, das nicht der im Quellcode programmierten Implementierung entspricht.

In der gesamten Spanne kann ein Aufrufgraph konservativer oder optimistischer sein als ein Vergleichsgraph. Konservativer bedeutet hier, dass der Aufrufgraph näher an $G_{\perp}$ liegt als der andere Graph. Umgekehrt bedeutet optimistischer, dass ein Aufrufgraph näher an $G_{\top}$ liegt.

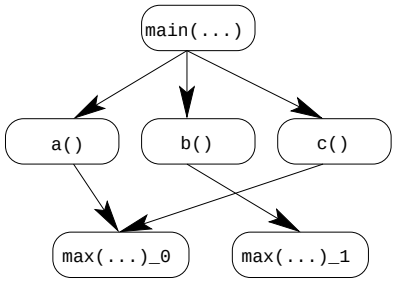
4.1.3 Eingesetzte Datenstrukturen

```
public class AClass
{
  public static void
    main(String[] args, int c)
  {
    AClass X = new AClass();
    X.a();
    X.b();
    X.c();
  }

  public void a()
  {
    max(new Integer(4), new Integer(2));
  }

  public void b()
  {
    max(new Float(4), new Float(2));
  }

  public void c()
  {
    max(new Integer(4), new Integer(2));
  }

  public void max(Number i, Number j)
  {}
}
```

(a)



(b)



(c)

Abbildung 4.1: Ein Beispielprogramm (a), ein kontext*in*sensitiver (b) und ein kontextsensitiver (c) Aufrufgraph

Im Grunde genommen gibt es zwei unterschiedliche Datenstrukturen, die bei der Konstruktion von Aufrufgraphen zum Einsatz kommen. Die einfachere Variante besteht darin, dass es für jede Methodendeklaration einen Knoten und für jeden Aufruf eine Kante gibt. Dies ist in Abbildung 4.1b für das Programm in 4.1a dargestellt. Diese naheliegende Art der Datenstruktur kommt bei sog. kontext*in*sensitiven Algorithmen zustande (siehe 4.1.1.1).

Eine andere Datenstruktur wird für kontextsensitive Algorithmen eingesetzt. Hier gibt es unter Umständen mehrere Knoten für eine Methode. Knoten in kontextsensitiven Aufrufgraphen werden auch Konturen genannt. Die verschiedenen Konturen entstehen hierbei durch die unterschiedlichen Kontexte (z. B. Typen der verwendeten Parameter), in denen eine Methode aufgerufen werden kann. Als Folge enthalten kontextsensitive Aufrufgraphen mehr Knoten und Kanten als kontextinsensitive.

In Abbildung 4.1c ist das an einem Beispiel ausgeführt. Wenn hier eine Methode `max(Number, Number)` von einer Methode `a()` mit zwei Subtypen `Integer` aufgerufen wird, dann wird eine andere Kontur erzeugt, als wenn dieselbe Methode `max(Number, Number)` von einer Methode `b()` mit zwei Subtypen `Float` aufgerufen wird.

4.1.4 Eignung der Algorithmen

Für die Berrechnung der Repräsentation der Joinpointeigenschaft *Cflow* durch ein Werkzeug wird ein konkreter Algorithmus benötigt. Unter den untersuchten Algorithmen soll der gefunden werden, der die besten Voraussetzungen hinsichtlich verschiedener Faktoren für die Repräsentation hat.

Soundness

Einer der wohl entscheidensten Faktoren, im Zusammenhang mit der Erstellung der Repräsentation für die Joinpointeigenschaft *Cflow*, ist die Lage des Ergebnisgraphen hinsichtlich des idealen Aufrufgraphen $G_{\mathbf{ideal}}$. Alle Algorithmen, die nicht zusichern können, dass das Ergebnis konservativer oder gleich $G_{\mathbf{ideal}}$ ist, sind nicht geeignet einen Aufrufgraphen bereitzustellen, der für die Erstellung der Repräsentation ausreicht. Würde ein Aufrufgraph nicht mindestens alle Kanten, die während irgendeiner möglichen Ausführung eintreten könnten, enthalten, so wäre die Repräsentation unter Umständen auch nicht für jede Ausführung des Programms gültig. Gerade diese Forderung muss aber an die Repräsentation gestellt werden, wie bereits in Abschnitt 3.1.2.3 ausgeführt wurde. Der Algorithmus muss also einen Aufrufgraphen erstellen, der sound ist.

Präzision

Ein weiterer Faktor ist die Präzision des Graphen. Wie optimistisch darf ein Aufrufgraph sein, damit er für die Repräsentation noch eine hinreichend sichere Approximation darstellt, und wie konservativ kann er sein, um nicht zu viele unmögliche Pfade zu repräsentieren, die in der Repräsentation zu falschen Matchpfaden ausgewertet werden? Wie optimistisch das Ergebnis sein darf, ist schon im vorherigen Absatz beschrieben worden, $G_{\mathbf{ideal}}$ stellt die obere Grenze dar. Die andere Richtung ist dabei weniger klar abgesteckt. Hier kann nur verallgemeinernd gesagt werden, dass zu konservative Graphen zu viele Matchpfade ergeben werden. Bei genauerer Betrachtung fallen hier weitere Faktoren an, denen sich die Anschauung der Präzision unterordnen muss.

Aufwand

Von Bedeutung ist die Frage, wie hoch der Aufwand ist, aus dem Quellcode einen Aufrufgraphen, passend für die Repräsentation, zu erstellen. Aufwand meint hier zum einen die Zeit, die ein Algorithmus zum Berechnen eines Aufrufgraphen braucht, und zum anderen wieviel Speicher benötigt wird. Der zeitliche und speicherbezogene Aufwand variiert bei den Algorithmen enorm, führt aber auch zu verschieden genauen Graphen. Eine der einfachsten Algorithmen, der einen konservativen Graphen berechnet, stellt Kanten anhand der Methodennamen auf und kann mit sehr geringem Aufwand die Berechnung durchführen. Das Ergebnis ist aber nahezu unbrauchbar, da in umfangreicheren Programmen offensichtlich eine große Menge von Matchpfaden erstellt werden wird, die als Analysegrundlage kaum sinnvoll einsetzbar sein werden. Andere Algorithmen kommen zwar zu präziseren Ergebnissen, werden sich aber nicht in einem Refactoringwerkzeug einsetzten lassen. Der Vorteil, durch ein Refactoring die Struktur des Programms verbessern zu können, wird nicht mehr vermittelbar sein, wenn Berechnungszeiten von mehreren Minuten zur Durchführung eines Refactorings benötigt werden.

Teilanalysen

Die zu verarbeitende Menge an Quellcode, um einen Aufrufgraphen zu erstellen, wird sich ebenso auf den Aufwand der Berechnung der Analyse auswirken, wie der oben beschriebene Aufwand für die Algorithmen selbst. Das Aufstellen der Repräsentation wird nicht notwendigerweise den gesamten Aufrufgraphen brauchen. Je nach Lage der Starttriggershadows und Endtriggershadows im theoretisch vollständigen Aufrufgraphen, werden die möglichen Matchpfade nur relativ kleine Abschnitte des gesamten Graphen abdecken. Algorithmen, die nicht auf die Analyse des ganzen Programms angewiesen sind, ist der Vorzug zu geben, da sich naheliegenderweise der Aufwand hinsichtlich der Berechnung der Matchpfade erheblich verringert, wenn dafür nur kleine Teile des gesamten Programms traversiert werden müssen. Der Einfluss dieses Faktors ist aber für diese Arbeit nicht belegbar, da Untersuchungen, wie weit Starttriggershadows im Aufrufgraphen von der Startmethode (z. B. `public static void main(String[] args)`) im Mittel entfernt liegen, nicht stattgefunden haben.

Zusätzliche anfallende Informationen

Im Verlauf des Aufbaus eines Aufrufgraphen werden von manchen Algorithmen Informationen genutzt und entsprechend vorher berechnet, die auch bei der Identifizierung von Joinpointeigenschaften Verwendung finden könnten. Werden zum Beispiel Algorithmen eingesetzt, die auf *Points-To*-Analysen basieren, stehen Typinformationen für Variablen zur Verfügung. Diese Typinformation könnte bei der Ermittlung von dynamischen Singletriggereigenschaften eingesetzt werden, auf denen eine Spezifikation einer Eigenschaft *Cflow* basiert.

Nachvollziehbarkeit des Analyseergebnisses

Im Ergebnis soll der Vergleich von Repräsentationen der Eigenschaft *Cflow* für eine Version vor einer Quellcodeänderung und einer Version nach der Quellcodeänderung stattfinden und bei vorhandenen Änderungen soll der Entwickler in bestimmten Fällen mit den Matchpfaden konfrontiert werden, an denen sich etwas geändert hat. Da der Algorithmus, der den Aufrufgraphen erstellt, anhand algorithmusspeziefischer Kriterien für das Vorhandensein oder das Nicht-Vorhandensein von Pfaden verantwortlich ist, werden diese algorithmusspeziefischen Kriterien auch für unterschiedliche Anzahlen von Matchpfaden in den beiden Repräsentationen verantwortlich sein.

Für einen eingesetzten Points-To basierten Algorithmus könnte sich zum Beispiel folgendes Szenario ergeben. Durch eine Quellcodeänderung wird eine Variableninstanziierung aus dem Programm entfernt. Die Entfernung der Instantziierung ist, wegen des eingesetzten Algorithmus, dafür verantwortlich, dass es einen Matchpfad weniger in der Repräsentation gibt. Diese Instanziierung muss vor dem Starttriggershadow liegen, damit sie einen Einfluss auf den Matchpfad haben kann, sie fällt also nicht mehr in den Bereich des Matchpfades selbst. Die Verbindung kann deswegen nicht mehr durch einen Vergleich der beiden Repräsentationen erfolgen, da das verantwortliche geänderte Quellcodeelement in keiner der Repräsentationen liegen wird. Damit ein Entwickler mit diesem Ergebnis etwas anfangen kann, müsste ihm der durch den Algorithmus erzeugten Zusammenhang präsentiert werden. Dies wird mit der erarbeiteten Repräsentation nicht ohne weiteres möglich sein.

4.1.5 Auswahl eines Algorithmus

Voraussetzung für die Realisierung der Repräsentation ist ein geeigneter Aufrufgraph. Die erarbeiteten Faktoren zur Auswahl eines Algorithmus sollten, bezüglich der Erstellbarkeit eines

geeigneten Aufrufgraphen, eine ausreichende Entscheidungsgrundlage darstellen. Jeder Algorithmus, der potentiell eingesetzt werden soll, muss einen Aufrufgraphen berechnen, der sound ist, dies ist schon bei der Aufstellung dieses Faktors deutlich geworden. Da Methoden zum Erstellen von unsound Graphen schon von Anfang an als nicht einsetzbar identifiziert wurden, ist es nicht notwendig bestimmte untersuchte Algorithmen auszuschließen.

Die Frage der erforderlichen Präzision eines Aufrufgraphen gestaltet sich hier interessanter. Das untere Ende eines, für objektorientierte Sprachen, sinnvoll einsetztbaren Algorithmus bildet der *Class Hierarchy Analysis* (CHA) Algorithmus ((vgl. Dean u. a. 1995)). Ein Algorithmus wie $G_{\mathbf{selector}}$ ist zum Beispiel wegen bestimmter Namenskonventionen in objektorientierten Programmen unbrauchbar. Mit diesem Algorithmus würden die üblicherweise verwendeten `get()` und `set()` Methoden große Mengen sinnloser Aufrufabhängigkeiten erzeugen. Auch von einem anderen Standpunkt aus gesehen machen Algorithmen keinen Sinn, die konservativer als CHA sind. Das besondere an CHA ist, dass es keine Kanten geben wird, die Methodenaufrufe repräsentieren, die aus Typgründen nicht existieren dürften.

Alle Algorithmen, die optimistischer als CHA sind, können aber nur als Analyse auf dem ganzen Programm funktionieren. Wie wichtig es ist, ohne die Analyse des ganzen Programms die geforderten Ausschnitte eines Aufrufgraphen zu produzieren, konnte nicht abschließend geklärt werden, deswegen wird diese Fragestellung nicht das entscheidende Kriterium sein können.

Da das Werkzeug letzten Endes auch für Refactorings von Software größeren Umfangs einsetzbar sein soll, sollte ein Algorithmus einen Aufrufgraphen in angemessener Zeit aufbauen können und dabei die Menge an Arbeistspeicher berücksichtigen, die für das Werkzeug vorhanden ist. Die Gründe dafür sind bereits bei der Einführung dieses Faktors beleuchtet worden. Aufgefallen ist, dass bei möglichen Eingschränkungen, die sich durch den Aufwand ergeben, immer zuerst die beanspruchte Zeit den limitierenden Faktor darstellte und nicht der beanspruchte Speicher. Unter den untersuchten Algorithmen befinden sich einige, die in dieser Hinsicht nicht einsetzbar sind.

Kontextsensitive Algorithmen schließen sich durch diese Anforderung aus, da keiner der Algorithmen ausreichend skaliert. Das trifft ebenso auf den *Cartesian Product* und den *Object Sense* Algorithmus zu (für eine ausführlichere Diskussion (vgl. Grove u. a. 1997, Abschnitt 4.1) bzw. für Benchmarks des *Object Sense* Algorithmus Milanova u. a. (2002)).

Algorithmen, die auf *Points-To* Analysen beruhen, scheinen bei Betrachtung entsprechender Benchmarks ebenso wenig geeignet zu sein. Selbst für Programme mittleren Umfangs werden dort Zeiten benötigt, die weit über den Anforderungen an ein Refactoringwerkzeug liegen (Streckenbach u. Snelting 2000). Das gilt bei Java Programmen sowohl für *Equality-Based* als auch für *Subset-Based* Ansätze. Lange Zeit galten insbesondere die *Subset-Based* Ansätze mit schlimmstenfalls kubischem Berechnungsaufwand als nicht einsetzbar. Eine Ausnahme bildet eine verhältnismäßig neue Entwicklung, die mit einer besonderen Datenstruktur arbeitet. Mit diesem Ansatz scheint ein *Subset-Based* Algorithmus in vertretbarer Zeit Aufrufgraphen berechnen zu können (vgl. Berndl u. a. 2003). *Propagation Based* Alogrithmen nach Tip und Palsberg scheinen ebenfalls auch für größere Programme in angemessener Zeit zu einem Ergebnis zu kommen (vgl. Tip u. Palsberg 2000, Tabelle 7).

Von den betrachteten Algorithmen kommen, unter Berücksichtigung des Faktors Aufwand, der *CHA* Algorithmus (vgl. Dean u. a. 1995), der *Rapid Type Analysis* Algorithmus (sowohl in der optimistischen als auch pessimistischen Variante) (vgl. Bacon u. Sweeney 1996), die *Propagation Based* Algorithmen (Tip u. Palsberg 2000) und der auf einer *Points-to* Analyse aufbauende,

mit *Binary Decision Diagrams* realisierte *Subset-Based* Algorithmus in Frage (vgl. Berndl u. a. 2003).

Die Singletriggereigenschaft, die spezifiziert, welchen Typ eine bestimmte Variable aus dem Joinpointkontext haben muss, könnte Grundlage für die Spezifikation einer Eigenschaft *Cflow* sein. In diesem Fall könnten Informationen genutzt werden, die durch den *Rapid Type Analysis* Algorithmus, die *Propagation Based* Algorithmen oder *Points-To* basierten Algorithmen entstehen. Bei der Erarbeitung der Repräsentation der Eigenschaft *Cflow* sind solche Informationen nicht berücksichtigt worden. Ebenso wurden für die beschriebene dynamische Singletriggereigenschaft keine Überlegungen hinsichtlich einer eigenen Repräsentation angestellt. Der Einsatz dieser Algorithmen würde in diesem Stadium, der Entwicklung von Repräsentationen dynamischer Joinpointeigenschaftn, keinen zusätzlichen Vorteil und Nutzen bringen.

Bezüglicher der Nachvollziehbarkeit des Analyseergebnisses schließen sich bei Beibehaltung der Repräsentation, wie sie in dieser Arbeit entwickelt wurde, alle behandelten Algorithmen aus, die optimistischer sind als der *CHA* Algorithmus. Ein weiterer Grund für die Entscheidung für den *CHA* Algorithmus ist die absehbar unkomplizierte Implementierung im Verhältnis zu den anderen Algorithmen.

Ein Argument, welches gegen den *CHA* Algorithmus sprechen könnte, ist die höhere Anzahl an Matchpfaden, die dieser Algorithmus im Verhältnis zu den optimistischeren erzeugt. Die Frage ist, in welcher Weise sich diese möglicherweise zu große Menge an errechneten Matchpfaden negativ, in Bezug auf die Brauchbarkeit der Repräsentation, auswirken wird.

Bei einer Repräsentation, die auf dem so erzeugten Aufrufgraphen beruht, können mehrere Effekte auftreten, die das Ergebnis verfälschen. Das Ersetzen einer Objektinstanziierung durch eine eines anderen Typs, irgendwo im Programm, kann dafür verantwortlich sein, dass es einen Matchpfad weniger gibt oder einen mehr gibt. Das ist der Fall, wenn die Klasse des ersetzenden Objekts Methoden anders implementiert als die des ersetzten Objekts. Da der *CHA* Algorithmus aber immer davon ausgeht, dass entsprechende Methoden aller Subtypen aufgerufen werden können, würde eine solche Änderung nicht zu unterschiedlichen Aufrufgraphen führen. In einem objektorientierten Refactoring zugelassene Änderungen erlauben aber nur semantisch äquivalente Referenzen und Operationen in der geänderten Version des Programms. Dies schließt das Ersetzen eines Objektes durch ein anderes aus, dessen Klasse die Methoden nicht äquivalent implementiert.

Nicht nur Objekttypen können einen Einfluss auf die Menge und Art der Matchpfade haben, sondern auch Werte in Variablen. Der Wert einer Variablen, die in einer Kontrollflussanweisung ausgewertet wird, welche im Kontrollfluss zwischen Starttriggershadow und Endtriggershadow liegt, darf von einer Quellcodeänderung nicht so verändert werden, dass in der Kontrollflussanweisung anders verzweigt wird. Solche Quellcodeänderungen sollten aber in einem Refactoring ausgeschlossen sein, da sie auch in einem objektorientierten Programm Seiteneffekte haben würden.

Der verbleibende Nachteil gegenüber Algorithmen, die optimistischere Aufrufgraphen erzeugen, liegt in einer möglicherweise zu großen Menge an Matchpfaden, die zu falschen Identifizierungen der Betroffenheit einer spezifizierten Eigenschaft *Cflow* führen. Führt die Analyse zu dem Ergebnis, dass eine Quellcodänderung vorliegt, die die spezifizierte Eigenschaft betrifft, ist die Konsequenz, dass ein Refactoring entweder nicht durchgeführt werden darf oder die Spezifikation angepasst werden muss. In diesem Fall muss der Entwickler mit den betroffenen Matchpfaden konfrontiert werden.

Relevanz bekommt die höhere Menge an Matchpfaden, wenn sie im Umfang über dem liegen, was ein Entwickler an Informationsmenge verarbeiten kann. Bei dieser Überlegung spielen drei Faktoren eine Rolle. Es muss untersucht werden, welche Informationsmenge ein Entwickler sinnvoll verarbeiten kann, das heißt es muss festgestellt werden, bei welcher Menge an Information noch sinnvolle Konsequenzen gezogen werden können (z. B. „Wie muss die Spezifikation angepasst werden“ oder „Ist der Pointcut wirklich betroffen“). Weiterhin muss verglichen werden, wie viel weniger Information (in Form von Matchpfaden) andere Algorithmen liefern und ob dies eine relevante Reduktion darstellt (wenige Prozent oder signifikant weniger Matchpfade). Als letztes bleibt die Frage, wieviel Information überhaupt entsteht. Ist diese Menge sowieso gering (z. B. unter zehn Matchpfade), dann entfallen die beiden anderen Fragestellungen.

Wie hoch die von einem Entwickler verarbeitbare Informationsmenge ist, wird sich ohne eine Untersuchung nicht feststellen lassen. Eine Untersuchung dieser Fragestellung muss, im zeitlich begrenzten Rahmen dieser Arbeit, ausbleiben. Die Frage nach der Menge an Matchpfaden, die im Rahmen eines Refactorings unter dem Einsatz eines Werkzeugs entsteht, ist, mangels eines vorhandenen Prototyps, noch nicht beantwortbar und würde auch eine umfangreicher Untersuchung mit verschiedenen Programmen und verschiedenen Refactorings erfordern.

Ob andere Algorithmen zu signifikant weniger Matchpfade führen würden als der CHA Algorithmus, kann in den meisten Fällen anhand der Arbeiten nachvollzogen werden, in denen die entsprechenden Algorithmen vorgestellt wurden. Die Algorithmen werden dort häufig dahingehend mit dem CHA Algorithmus verglichen, wie viel weniger Aufrufziele für dynamisch bindbare Methodenaufrufe in dem Ergebnisgraphen in Frage kommen. In (Tip u. Palsberg 2000, Abschnitt 4.5) variieren die Ergebnisse zwar für die unterschiedlichen untersuchten Programme, es zeigt sich aber, dass alleine die Verwendung des RTA Algorithmus dazu führen kann, dass zwischen $80,6\%$ und $96,6\%$ der dynamisch bindbaren Methodenaufrufe zu einem spezifzischen Methodenaufruf aufgelöst werden können.

Die Diskussion der Faktoren ergibt, dass hinsichtlich der Nachvollziehbarkeit des Analyseergebnisses, nur der CHA Algorithmus sinnvoll einsetzbar ist. Dagegen steht die unter Umständen zu große Anzahl an Matchpfaden, die daraus folgen, dass der CHA Algorithmus zu konservativ arbeitet (mangelnde Präzision). Konkreten Erfahrungen im Umgang mit den Analyseergebnissen liegen noch nicht vor, deswegen kann dieser Faktor zunächst außen vor gelassen werden. Ebenso sollte ein Ausbau der Realisierung des Algorithmus zu einer *Rapid Type* Analyse, bei entsprechend nachteiligen Ergebnissen, unproblematisch sein. Da der Algorithmus zu den verbleibenden Faktoren nicht im Widerspruch steht und eventuell einen Vorteil aufweisen kann gegenüber Algorithmen, die den kompletten Aufrufgraphen aufbauen müssen, wird der Prototyp mit dem CHA Algorithmus realisiert werden.

4.2 Ein Verfahren zur Ermittlung der Repräsentation der Joinpointeigenschaft Cflow aus dem Aufrufgraphen

Der Aufrufgraph bildet die Grundlage, um die Matchpfade zu bestimmen, die es ermöglichen sollen, die Auswirkungen von Quellcodeänderungen auf eine spezifizierte Eigenschaft *Cflow* zu ermitteln. Dazu soll ein Verfahren erläutert werden, dass aus dem Aufrufgraphen alle möglichen Matchpfade bezüglich einer Spezifikation einer Eigenschaft *Cflow* ermittelt.

In einem ersten Schritt werden die Starttriggershadows und Endtriggershadows gefiltert, die sich durch die vorangegangene Identifizierung anhand der Singletriggereigenschaften ergeben haben. Dabei werden diejenigen Starttriggershadows und Endtriggershadows entfallen, die nicht in einer Aufrufabhängigkeit stehen können. Gleichzeitig entsteht der für diese Filterung notwendige Aufrufgraph zwischen Starttriggershadows und Endtriggershadows. Eine nachgelagerte Bestimmung der Matchpfade wird die dynamischen Umstände feststellen unter denen es an einem Endtriggershadow zu einem Endtrigger kommen kann, der zu einer Bindung des Aspektes und damit zur Ausführung des Advicecodes führt. Am Ende der Analyse sollten die Matchpfade für eine Programmversion und eine Spezifikation einer Eigenschaft *Cflow* stehen.

4.2.1 Filterung der Starttriggershadows und Endtriggershadows

Für die Filterung und das gleichzeitige Aufbauen der Teilgraphen zwischen Starttriggershadows und Endtriggershadows wird der CHA Algorithmus genutzt, entsprechend der Festlegung in Abschnitt 4.1.5. Teilgraphen werden ausgehend von jedem einzelnen Starttriggershadow gebaut, so dass am Ende ein Graphwald entsteht, dessen Wurzeln die Starttriggershadows darstellen. Die einzelnen Ergebnisse werden noch weiter für den nächsten Schritt angepasst, indem Zyklen in den Graphen detektiert und entsprechend markiert zu einem Knoten zusammengefasst werden. Danach werden die Teile des Graphen herausgefiltert, die nicht zwischen Starttriggershadow und Endtriggershadow liegen können. Mit diesem Verfahren sind bereits unqualifizierte Matchpfade entstanden. Sie stehen zwar noch in einem Aufrufgraphen, da dieser aber nicht mehr zyklisch ist, kann der Graph in einzelne Pfade zwischen Starttriggershadows und Endtriggershadows zerlegt werden, die die unqualifizierten Matchpfade darstellen.

Für die Beschreibung, wie das Filtern, die Zyklenerkennung und das Bereinigen der Teilgraphen von unwichtigen Pfaden realisiert weden kann, wird von dem Vorhandensein bestimmter Informationen ausgegangen. Als Eingabe für die Analyse müssen alle Methodendeklarationen des Programms zugreifbar sein, für jede Methodendeklaration, anhand derer der Graph aufgebaut werden soll, müssen die enthaltenen Methodenaufrufe zur Verfügung stehen und diejenigen Methodendeklarationen, die einen Starttriggershadow oder Endtriggershadow darstellen, müssen als solche identifizierbar sein. In den Fällen, in denen ein Starttriggershadow ein Methodenaufruf ist, müssen die umschließenden Methodendeklarationen ermittelbar sein. Analog wird für Endtriggershadows, die keine Methodendeklarationen sind, wie zum Beispiel Feldzuweisungen, ebenfalls die umschließende Methodendeklaration gebraucht. Der CHA Algorithmus benötigt eine Klassenhierarchie anhand derer mögliche Aufrufziele erschlossen werden können. Da der CHA Algorithmus kontextinsensitiv ist, stellen die Methodendeklarationen die Knoten dar, für jeden Methodenaufruf gibt es eine Kante.

Der Aufbau der Teilgraphen wird entsprechend dem CHA Algorithmus vorgenommen. Bei dem Aufbau der Graphen wird der Tiefensuche Algorithmus (engl. **Depth-First Search**, abgekürzt **DFS**) angewandt (vgl. z. B. Cormen u. a. 2001, Abschnitt 22.3). Dieser Algorithmus hat den Vorteil, dass sich gleichzeitig die Detektion von Zyklen vorbereiten lässt. Die Aufrufgraphen werden dabei komplett bis zum Erreichen der Blätter aufgestellt. Wenn in diesem Schritt zusätzlich, als solche deklarierte, Rückwärtskanten in den Graphen eingeführt werden, wird damit ebenfalls eine vorbereitende Basis für die Erkennung von Zyklen geschaffen, indem ein inverser Graph erstellt wird.

In den mit DFS aufgestellten Aufrufgraphen werden nun die Zyklen detektiert und zu einzelnen Knoten zusammengefasst. Nur so wird aus dem gerichteten zyklischen Aufrufgraph ein

gerichteter azyklischer Aufrufgraph, aus dem garantiert eine endlichen Menge von Pfaden herausgelöst werden kann. Zyklen in einem Graphen können durch die Ermittlung der starken Zusammenhangskomponenten (engl. **Strongly Connected Components**, abgekürzt **SCC**) herausgefunden werden. Starke Zusammenhangskomponenten sind die Teilgraphen, in denen jeder Knoten von jedem anderen Knoten erreicht werden kann. Ein ursprünglich von Tarjan entwickelter Algorithmus kann die SCC in linearer Zeit berechnen und baut auf der Nummerierung der Knotenentdeckung auf, die durch den DFS Algorithmus erzeugt werden und benötigt den inversen Graphen (vgl. Tarjan (1972), zitiert nach Cormen u. a. (2001), Abschnitt 22.5). Die entdeckten SCC können dann jeweils zu einem Knoten zusammengefasst werden (für jede SCC ein Knoten).

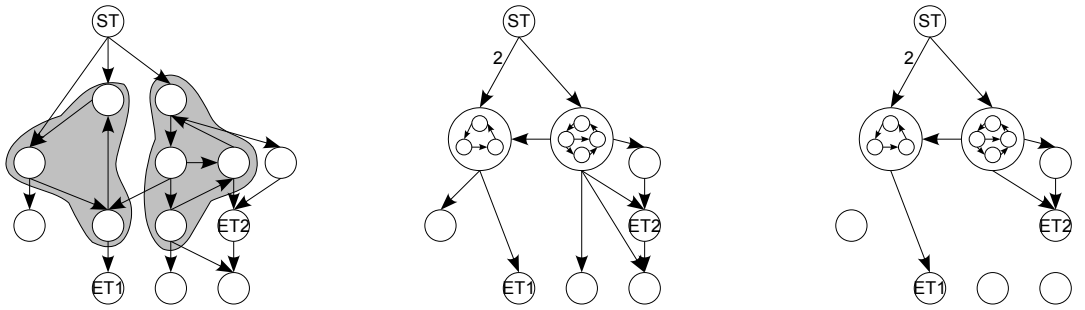
Die Teilgraphen enthalten jetzt noch viele Abschnitte, die nicht Teil eines Pfades zwischen Starttriggershadow und Endtriggershadow sein können. Dazu wird der in Abbildung 4.2 dargestellte Algorithmus angewandt. Der Graph wird ausgehend vom Starttriggershadow (Wurzel) per Tiefensuche traversiert (Zeilen 3 und 4). Dabei wird jede Kante aus dem Graphen entfernt, die zu einem Knoten führt, der kein Endtriggershadow ist oder keine Nachfolgeknoten hat (Zeilen 5, 6 und 7). So werden zunächst ausgehend von einem Knoten alle Nachfolgeknoten, die keine Endtriggershadow sind oder keine eigenen Nachfolgeknoten haben, als Nachfolger entfernt. Sind alle Nachfolger so gefiltert worden, dann bleiben entweder keine Nachfolger mehr übrig, und falls der Knoten selber kein Endtriggershadow ist, wird er über seinen Vorgängerknoten entfernt werden oder es gibt noch Nachfolgeknoten, dann muss der aktuelle Knoten Bestandteil eines Pfades zwischen Starttriggershadow und einem Endtriggershadow sein.

```
PRUNE(v)
1  v mark visited
2  for each successor s from v
3        do if s is not visited
4              then PRUNE(s)
5           if s is not an endtriggershadow or has no successors
6              then remove v as predecessor from s
7                   remove s as successor from v
```

Abbildung 4.2: Der Algorithmus zur Beseitigung überflüssiger Pfade in den Teilgraphen

Die Teilphasen der hier beschriebenen ersten Phase sind in Abbildung 4.3 noch einmal dargestellt. Abbildung 4.3a zeigt einen Graphen wie er nach dem Aufbau aussehen könnte. Ausgehend von einem Starttrigger (ST) wurden alle Pfade aufgebaut. In der Abbildung sind die Endtriggershadows (oder die umschließende Methodendeklaration) dargestellt (ETn) und die SCC als graue unterlegte Bereiche, die rekursive Teile im Graphen bilden. Abbildung 4.3b zeigt den Graphen nachdem die SCC identifiziert und zu Knoten zusammengefasst wurden. Dadurch ist eine parallele Kante entstanden. Die Ziffer Zwei macht das Vorhandensein von zwei Kanten kenntlich. Die dritte Abbildung 4.3c veranschaulicht das Ergebnis des Bereinigungsalgorithmus, es sind nur noch Kanten vorhanden, die zwischen dem Starttriggershadow und einem der Endtriggershadows liegen.

In Abschnitt 3.2.1.1 wurde erläutert inwiefern ein reiner Aufrufgraph nicht für eine Repräsentation der Eigenschaft *Cflow* ausreicht. Die in dieser Phase erreichte Repräsentation beinhaltet



(a) (b) (c)

Abbildung 4.3: Die einzelnen Schritte der ersten Phase zur Bestimmung der Matchpfade

zwar schon einige der geforderten zusätzlichen Eigenschaften (z. B. die aufgelösten Rekursionen), die Qualifizierung der Pfade kann aber noch nicht vorgenommen werden und Aufrufe im gegenseitigen Ausschluss sind ebenso wenig vorhanden.

4.2.2 Bestimmung der qualifizierten Matchpfade

Um die in der ersten Phase ermittelte Repräsentation für die Analyse der Quellcodeänderungsauswirkungen auf die dynamische Joinpointeigenschaft *Cflow* nutzbar zu machen, müssen sowohl Kontrollflussanweisungen, die Methodenaufrufe umschließen (Verzweigungen und Schleifen), als auch dynamisch bindbare Methodenaufrufe gesondert berücksichtigt werden.

Für die Bewältigung dieser Aufgabe, wird die Datenstruktur dahingehend erweitert, dass Kanten Eigenschaften bekommen können. Die Knoten des Graphen, der in der ersten Phase berechnet wurde, werden untersucht, um Kontrollflussanweisungen zu identifizieren, die Methodenaufrufe umschließen. Es werden hier nur diejenigen Methodenaufrufe beachtet, die in Form von Kanten Teil des Graphen sind. Außerdem werden die dynamisch zu bindenden Methodenaufrufe, die in mehr als einen Methodenaufruf aufgelöst werden konnten, identifiziert. Aus der identifizierten Information kann für jede Kante eine entsprechende Eigenschaft bestimmt werden aus der sich, zusammen mit den Eigenschaften der anderen Kanten eines Matchpfades, die Art des gesamten Matchpfades ermitteln lässt.

4.2.2.1 Verzweigungen

Jede der gefundenen umschließenden Verzweigungsanweisungen bekommt eine Identifizierung zugeordnet. Je nach aufgetretener Art und Form der Kontrollflussanweisung muss festgestellt werden, für wie viele verschiedene Verzweigungen die Kontrollflussanweisung steht. Die Anzahl der Blöcke die wegen unterschiedlicher Auswertungen der Bedingung in der Verzweigungsanweisung erreicht werden können, wird ebenso ermittelt. Zusätzliche wird jeder Block mit einer Nummer versehen, so dass sich Blöcke untereinander unterscheiden lassen, in die von einer Verzweigungsanweisung aus verzweigt werden kann.

Methodenaufrufe, die Methoden dynamisch binden werden, werden auch mit einer eindeutigen Identifizierung versehen. Dazu kommt die Anzahl der Methoden, zu der sich der Methodenaufruf auflösen lässt. Analog zu den Blöcken, erhält auch hier jede Methode eine Nummer.

Für jede Kante wird aus diesen Informationen eine Eigenschaft zusammengestellt. Sie besteht aus der Identifizierung, der Anzahl möglicher Verzweigungen und der Nummer des Blocks bzw. der Methode in der sich der Methodenaufruf befindet bzw. der Variante des Methodenaufrufs.

In der Programmiersprache Java, die die Grundlage für diese Arbeit bildet, gibt es, gemäß der *Java Language Specification*, die folgenden beiden Verzweigungsanweisungen (vgl. Gosling u. a. 2005, Kapitel 14)[1]:

- Mit `if` bzw. `if-else` wird ein Block bedingt ausgeführt oder zwei Blöcke werden im gegenseitigen Ausschluss ausgeführt. Das bedeutet für eine reine `if`-Verzweigungsanweisung gibt es einen Block aber zwei Möglichkeiten. Für eine `if-else`-Verzweigungsanweisung gibt es zwei Blöcke die jeweils eine Möglichkeit darstellen. Auch das Konstrukt `? :` stellt eine Form der `if-else` Verzweigungsanweisung dar bzw. sie muss als solche berücksichtigt werden. Auch müssen `return` Anweisungen untersucht werden, die in einem Block stehen, der zu einer `if`-Anweisung gehört. Hier wird der darauf folgende Quellcode als `else`-Block behandelt werden müssen.

- Für die `switch`-Verzweigungsanweisung werden die einzelnen Blöcke, die sich am Schlüsselwort `case` und `default` festmachen lassen, anhand eines Wertes ausgewählt. Es wird also eine Anzahl an Möglichkeiten ergeben, die sich aus der Anzahl der `case`-Blöcke und der „Anzahl“ der `default`-Blöcke (kein oder ein `default`-Block) herleitet[2].

Abbildung 4.4 zeigt ein Beispiel für Kanten, die mit Eigenschaften versehen werden, die im Zusammenhang mit Verzweigungsanweisungen entstehen. In dem Beispiel gibt es zwei `if-else`-Konstrukte, welche jeweils mit einer Identifizierung versehen werden ($b1$ und $b2$[3]). Beide Verzweigungsanweisungen haben genau zwei verschiedene Möglichkeiten zu verzweigen.

Bei Methodenaufruf `m3()` kann zum Beispiel gesehen werden, dass die entsprechende Kante die Identifizierung $b1$ enthält und die Information, dass der Aufruf in dem zweiten von zwei Blöcken steht. Da der Knoten `m4()` nicht Teil des gefilterten Teilgraphen ist, wird er schattiert dargestellt.

Werden später die Matchpfade aufgestellt, so können durch die Auswertung der Kanteneigenschaften genau die Anzahl der sicheren und bedingten Pfade unterschieden werden. In diesem Beispiel wird es drei Matchpfade geben.

$P1:\ \{startTriggerShadow(), m1(), m2(), endTriggerShadow()\}$, mit Kante $b1 : 1/2$

$P2:\ \{startTriggerShadow(), m1(), m3(), endTriggerShadow()\}$, mit Kante $b1 : 2/2$

$P3:\ \{startTriggerShadow(), m1(), m2(), endTriggerShadow()\}$, mit Kante $b2 : 1/2$

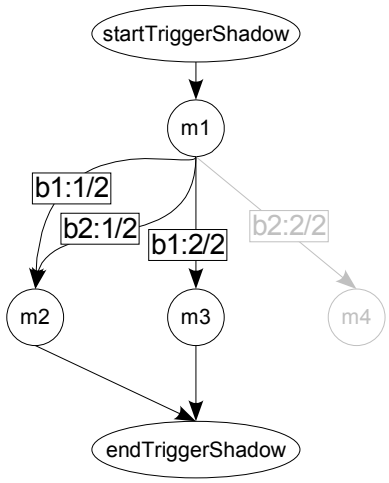
[1] Einen Fall der in dieser Arbeit nicht behandelt werden wird, stellt das `try-catch` bzw. `try-catch-finally` Konstrukt dar. Die damit zusammenhängende `throws` Anweisung wird schon beim Erstellen des Aufrufgraphen nicht beachtet. Ebensowenig werden Sprünge zu Marken (engl. label) behandelt. Eine genauere Auseinandersetzung mit diesem Thema wird im zeitlichen Rahmen der Arbeit nicht möglich sein.

[2] Eine vertiefende Betrachtung, insbesondere hinsichtlich der Anweisungen `break`, `return` und `continue`, soll hier nicht geführt werden. Es lässt sich festhalten, dass in jedem Fall ein Ergebnis ermittelt werden kann, bezüglich der Anzahl der Möglichkeiten und welche Möglichkeit für einen speziellen Aufruf zutrifft

[3] Der Quellcode wird dabei nicht, wie in diesem Beispiel mit Kommentaren versehen, die Identifizierung wird von dem zu realisierenden Werkzeug erzeugt und gehalten.

```
public class ForkExample
{
  public void startTriggerShadow(
    boolean b1, b2) {
    m1(b1, b2);
  }

  public void m1(boolean b1, b2) {
    if(b1) {  // ID: b1, zwei Moeglichkeiten
      m2(); // erste von zwei
    } else {
      m3(); // zweite von zwei
    }
    if(b2) {  // ID: b2, zwei Moeglichkeiten
      m2(); // erste von zwei
    } else {
      m4(); // zweite von zwei
    }
  }

  public void m2() { endTriggerShadow(); }
  public void m3() { endTriggerShadow(); }
  public void m4() {}

  public void endTriggerShadow() {}
}
```

(a) Programmcode



(b) Qualifizierte Kanten

Abbildung 4.4: Die Ausstattung mit Informationen anhand von Verzweigungen

Die Pfade $P1$ und $P2$ bilden zwei sich ausschließende Matchpfade und somit einen sicheren Matchpfad. Erkennen lässt sich das daran, dass beide von zwei Möglichkeiten für eine Verzweigungsanweisung vorhanden sind. Der Pfad $P3$ ist hingegen ein bedingter Pfad, da nur eine von zwei Möglichkeiten für $b2$ vertreten ist.

4.2.2.2 Schleifen und Rekursionen

Die Repräsentation wird, wie in Abschnitt 3.2.1.2 beschrieben, von der genauen Anzahl von Aufrufen abstrahieren, die durch Schleifen oder Rekursionen bedingt sind. Stattdessen werden die Kanten markiert falls sie in einer Schleifenanweisung stehen oder aus einem Rekursionsblock hervorgehen.

Die *Java Language Specification* gibt die drei verschiedenen Anweisungen `while` `do-while` und `for` als Schleifenanweisungen vor. Offensichtlich muss hier nicht zwischen den verschiedenen Schleifenanweisungen unterschieden werden, Aufrufe in einem Block dieser Anweisungen lassen sich offensichtlich gleich behandeln. Die entsprechenden Kanten werden als mehrfache Kante und mit einem, die Schleifenanweisung identifizierendem, Symbol markiert.

In der vorherigen Phase sind bereits die SCC bestimmt worden. Daraus sind speziell markierte Knoten hervorgegangen, die rekursive Teile des Graphen darstellen. Alle Kanten die aus diesen Knoten hervorgehen (abgehende Kanten) können ebenso als merhfache Kanten markiert werden und sollten ebenso eine Identifizierung enthalten. Diese ist notwendig, um in Gegenwart sich gegenseitig ausschließender Matchpfade unterschiedliche mehrfache Pfade zu unterscheiden.

4.2.2.3 Aufteilen der Teilgraphen in einzelne Matchpfade

Sind alle Kanten markiert, so muss der Teilgraph, der von einem konkreten Starttriggershadow ausgeht, in einzelne Pfade zerlegt werden, die den Matchpfaden entsprechen. Da ein Teilgraph mehrere Endtriggershadows beinhaltet aber nur einen Starttriggershadow, ist es am einfachsten von den Endtriggershadows aus auf dem inversen Graphen Pfade zu dem Starttriggershadow aufzubauen.

Der in Abbildung 4.5 in Pseudocode dargestellte Algorithmus wird dafür angewendet. Den Einstiegspunkt bildet SPLIT(E), der als Eingabe E bekommt, die die Menge aller Endtriggershadows in dem aufzuteilenden Teilgraphen darstellt. Von dort wird mit GETPATHSET(v) für jeden Endtriggershadow e aus E der Teilgraph in einzelne Matchpfade aufgeteilt.

```
SPLIT(E)
1  F ← ∅                                  ▷ Pfadwald mit Endtriggershadow Wurzeln.
2  for each endtriggershadow e from E
3        do F ← F ∪ {GETPATHSET(e)}
4  return F
```

```
GETPATHSET(v)
 1  R ← ∅                                  ▷ Ergebnismenge aller Pfade.
 2  if v has no predecessors
 3     then P ← {v}                        ▷ Erzeuge neuen Pfad mit v.
 4          R ← R ∪ {P}
 5     else                                ▷ Füge v zu Pfaden der Vorgänger hinzu.

 6          for each predecessor p from v
 7              do PS ← GETPATHSET(p)
 8                 for each P from PS
 9                     do P ← P ∪ {v}
10                        R ← R ∪ {P}
11  return R
```

Abbildung 4.5: Ein Algorithmus, um den von einem Starttriggershadow ausgehenden Teilgraphen in Pfade zu zerlegen

Gibt es Vorgängerknoten, so wird jedem Pfad, der durch die Vorgänger entstanden ist, der aktuelle Knoten hinzugefügt und die Pfade der Vorgänger werden zu einer Menge an Pfaden des aktuellen Knotens zusammengefasst und zurückgegeben (Zeilen 6 bis 10).

Ein neuer Pfad entsteht immer dann durch die Vorgänger, wenn der Algorithmus in Tiefensuche den Starttriggershadow erreicht hat. Der Starttriggershadow zeichnet sich in den Teilgraphen dadurch aus, dass er als einziger Knoten keinen Vorgänger hat (Zeilen 3 und 4). Dadurch, dass dabei alle möglichen Pfade durchlaufen werden und bei jedem Mal der Starttriggershadow erreicht wird, entsteht genau die Menge der Matchpfade.

Die so ermittelte Menge an Matchpfaden entspricht der Repräsentation *Cflow*, wie sie in Abschnitt 3.2.1.3 beschrieben wurde. Für jeden Pfad lässt sich ermitteln, um welche Art von Pfad

es sich handelt. Um bei Matchpfaden, die bedingte Kanten enthalten, herauszufinden, ob es sich um einen bedingten Matchpfad oder einen Teil eines sicheren Matchpfades handelt, müssen gegebenenfalls mehrere Pfade hinsichtlich der in den Kanten vergebenen Informationen verglichen werden.

4.3 Verfahren zur Identifizierung der Auswirkungen von Quellcodeänderungen auf die Joinpointeigenschaft Cflow

Die Zielsetzung für die Entwicklung von Verfahren zur statischen Analyse dynamischer Programmeigenschaften, war Auswirkungen von Quellcodeänderungen im Rahmen eines Refactorings auf eine spezifizierte dynamischer Joinpointeigenschaft (in diesem Fall *Cflow*) festzustellen. Die Auswirkungen sollten sich in eine von drei Kategorien einordnen lassen. Entweder ist die spezifizierte Eigenschaft nicht von der Quellcodeänderung betroffen oder sie ist von der Quellcodeänderung betroffen oder es lässt sich keine Aussage machen, in diesem Fall muss der Entwickler mit möglichst aussagekräftiger Information versorgt werden.

Mit den vorangegangenen Verfahren lässt sich für eine Programmversion eine Repräsentation der Eigenschaft *Cflow* aufstellen. Bei einer Quellcodeänderung wird es für das Erreichen der Zielsetzung notwendig sein, eine vollständige Repräsentation vor und eine nach der Quellcodeänderung aufzubauen. Die Quellcodeänderung wird als Teil eines Refactorings dem Werkzeug als Information zur Verfügung stehen müssen (das heißt sie muss nicht passiv aus dem Quellcode herausinterpretiert werden). Mit diesen Informationen soll es möglich sein, eine der drei oben beschriebenen möglichen Auswirkungen zu identifizieren.

Eindeutig keine Auswirkung auf die spezifizierte Eigenschaft wird vorliegen, wenn eine Quellcodeänderung in keinem der Matchpfade anzusiedeln ist (weder in der Repräsentation vor noch nach der Änderung). Auch wenn alle Matchpfade in äquivalente Matchpfade überführt wurden, wird es keine Auswirkung gegeben haben. Liegt keine Auswirkung auf die Eigenschaft vor, so kann die Quellcodeänderung ohne Anpassung der Spezifikation der Eigenschaft *Cflow* vorgenommen werden.

Wenn eine Quellcodeänderung Singletrigger-Joinpointeigenschaften betrifft, müssen diese ausgewertet werden, um festzustellen, ob eine Anpassung der Singletrigger-Joinpointeigenschaften möglich ist. Die spezifizierte Eigenschaft wird betroffen sein, wenn es, nach der Berücksichtigung von Änderungen an den zugrundeliegenden Singletrigger-Joinpointeigenschaften, immer noch mehr Matchpfade oder weniger Matchpfade als vor der Änderung gibt. Ändern sich Matchpfade, zum Beispiel von sicher nach bedingt, dann gibt es mehr Matchpfade der einen Art und weniger der anderen. In diesen Fällen kann die gewünschte Änderung nicht durchgeführt werden, oder es muss versucht werden die Spezifikation so anzupassen, dass die Quellcodeänderung keine Auswirkung mehr auf die Eigenschaft hat. Hier können die betroffenen Matchpfade die Grundlage für die Entscheidung des Entwicklers bilden.

Bestimmte Quellcodeänderungen werden Matchpfade der Repräsentation betreffen, ohne dass eine Aussage darüber gemacht werden kann, ob eine Änderung hinsichtlich der spezifizierten Eigenschaft vorgelegen hat. Dies ist in Situationen der Fall, in denen zwar äquivalente Matchpfade im Sinne der Repräsentation vorliegen, die aber nicht in aller Klarheit als solche gewertet werden können. Ein Beispiel dafür ist eine Änderung, die in einen Matchpfad eine weitere Schleife

einführt. Es liegt dann zwar immer noch ein mehrfacher Pfad vor, aber es wird mit den Mitteln der Repräsentation und der Auswertung nicht eindeutig zu klären sein, ob dieser Matchpfad äquivalent zu dem entsprechenden Matchpfad vor der Änderung ist. Ist das Analyseergebnis in dieser Weise unklar, kann der Entwickler mit den betroffenen Matchpfaden und den entsprechend Matchpfaden in der Repräsentation vor oder nach der Änderung konfrontiert werden.

Die Aufgabe des Werkzeuges wird es sein, drei verschiedene Ergebnisse zu ermitteln. Es müssen zusätzliche Matchpfade erkannt werden, es müssen weggefallene Matchpfade identifiziert werden und es müssen Situationen erkannt werden, in denen äquivalente Matchpfade vorliegen. Die Ergebnisse müssen zusätzlich bewertet werden, so dass in Fällen mit unklarem Ergebnis der Entwickler mit den entsprechenden Matchpfaden konfrontiert werden kann.

4.3.1 Identifizierung zusätzlicher und weggefallener Matchpfade

Zusätzliche Matchpfade lassen sich identifizieren, indem ermittelt wird, ob ein von einer Änderung betroffener Matchpfad in der Repräsentation vor der Quellcodeänderung zu zusätzlichen oder anderen Matchpfaden in der Repräsentation danach geführt hat. Umgekehrt kann auch ein Matchpfad in der Repräsentation nach der Änderung von ihr betroffen sein. Es wird sich um einen zusätzlichen Matchpfad handeln, wenn es keinen korrespondierenden Matchpfad vor der Änderung gegeben hat. Ebenfalls in diese Kategorie fallen Änderungen in der Art von Matchpfaden als Folge einer Quellcodeänderung.

Bei der Änderung der Art der Matchpfade muss darauf geachtet werden, ob es sich wirklich um zusätzliche Pfade handelt, oder ob diese Pfade ein Äquivalent zu einem Pfad vor der Änderung bilden. Dies ist bei der Änderung von einem sicheren Pfad in zwei bedingte Pfade der Fall, wenn diese im gegenseitigen Ausschluss liegen. Diese sind infolgedessen als ein sicherer Pfad zu werten, wodurch es sich nicht um zusätzliche bedingte Pfade handelt.

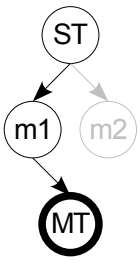
Eine andere Art von Matchpfad liegt aber vor, wenn vorher ein bedingter Matchpfad ermittelt wurde und danach zwei bedingte Matchpfade berechnet wurden, für die das Werkzeug anhand der Information über Verzweigungen in den Kanteneigenschaft feststellen kann, dass die Matchpfade nur im gegenseitigen Ausschluss ausgeführt werden können.

In Abbildung 4.6 ist ein Beispiel dargestellt für eine Quellcodänderung, die in einem Pfad nach der Änderung identifiziert werden kann und die zu einem zusätzlichen sicheren Pfad geführt hat. Für die Repräsentation vor der Änderung wurde nur einen Matchpfad ermittelt. Der Grau gezeichnete Pfad ist nicht enthalten, da er nicht zum Endtriggershadow führt (er ist herausgefiltert worden). Für die Repräsentation nach der Änderung sind hingegen zwei Matchpfade ermittelt worden. Dass einer von diesen von einer Quellcodeänderung betroffen ist, kann durch die Information über die Quellcodänderung (gekennzeichnet durch das Ä) ermittelt werden. Bei diesem Matchpfad handelt es sich um einen zusätzlichen Matchpfad, da kein Matchpfad in der vorherigen Repräsentation von der Quellcodeänderung betroffen ist.

Die Identifizierung einer Überführung von einem Matchpfad einer Art in einen Matchpfad einer anderen, wird anhand der zusammengestellten Kanteneigenschaften möglich sein. Ein betroffener Pfad, der vor der Änderung keine Verzweigungen und keine mehrfachen Kanten hatte, und dessen entsprechender Pfad in der geänderten Version des Programms zum Beispiel eine Kante enthält, die mit einer Verzweigung gekennzeichnet ist, bedeutet, dass der Matchpfad ein bedingter Matchpfad geworden ist. Die Identifizierung ist hier durch die Ermittlung der Kanteneigenschaften geschehen.

```
public void st(){
  m1();
  m2();
}
public void m1(){
  mt();
}
public void m2(){
}
public void mt(){}
```

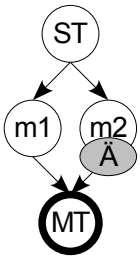
(a) Quellcode vor der Änderung



(b)

```
public void st(){
  m1();
  m2();
}
public void m1(){
  mt();
}
public void m2(){
  mt(); // zusaetzlicher Aufruf
}
public void mt(){}
```

(c) Quellcode nach der Änderung



(d)

Abbildung 4.6: Identifizierung eines zusätzlichen Pfades anhand einer Änderung Ä

Die Identifizierung von weggefallenen Matchpfaden wird offensichtlich analog zur Identifizierung hinzugekommener Matchpfade verlaufen können.

4.3.2 Identifizierung äquivalenter Matchpfade

Wenn einer der Pfade in einer der Repräsentationen betroffen ist, so muss untersucht werden, ob in der Repräsentation nach der Änderung äquivalente Matchpfade vorhanden sind. In Abschnitt 3.3.2.2 sind solche äquivalenten Pfade beschrieben worden.

Verhältnismäßig trivial sind äquivalente Matchpfade zu ermitteln, wenn die gleiche Anzahl der jeweiligen Arten von Matchpfaden für beide Repräsentationen berechnet wurde und jeder Matchpfad, an dem eine Änderung vorliegt, zu genau einem Matchpfad in der anderen Repräsentation in Beziehung steht. Dabei muss ein genaues eins zu eins Verhältnis dieser Beziehungen vorliegen, andernfalls wird sich etwas geändert haben. Liegt bei gleicher Anzahl von Matchpfaden kein eins zu eins Verhältnis vor, so wird ein Eingreifen des Entwicklers erforderlich sein.

In den anderen Fällen spielen die Ermittlung von mehrfach Kanten und die Analyse sich gegenseitig ausschließender Matchpfade die grundlegende Rolle.

Die Aufstellung der Matchpfade, so wie sie beschrieben wurde, führt dazu, dass an einem Matchpfad genau abgelesen werden kann, welche Verzweigungen durchlaufen wurden und welche alternativen Wege die Matchpfade bilden. Anhand der Identifizierung der Kanten können dann, wie in 4.2.2.1 beschrieben, Matchpfade identifiziert werden, die zusammen einen sicheren Pfad bilden. Äquivalente Matchpfade, die durch einen sicheren Matchpfad in einer Repräsentation und durch sich gegenseitig ausschließende Pfade in der jeweiligen anderen Repräsentation gekennzeichnet sind, werden auf diese Weise identifizierbar.

Es gibt Sonderfälle für äquivalente Matchpfade der eben beschriebenen Art. Es wird möglich sein, dass bei einer Verzweigung unter jeder Bedingung ein Pfad von einem Starttriggershadow zu einem Endtriggershadow existiert und es sich trotzdem nicht um einen sicheren Matchpfad handelt. Dies ist der Fall, wenn die Anzahl der Matchpfade, die jeweils einer Variante in der Verzweigung zuzuordnen sind, nicht ausgeglichen ist. Es ist zum Beispiel denkbar, dass für eine Variante einer Verzweigung ein Matchpfad und für die andere zwei Matchpfade existieren. In dieser Situation wird es entweder zu einem oder zu zwei Endtriggern nach Eintritt in den Starttriggershadow kommmen. Sind solche Matchpfade Gegenstand einer Änderung, wird der Entwickler mit den entsprechenden Matchpfaden konfrontiert werden müssen.

Eine erste Art der Äquivalenz von Matchpfaden, in der mehrfach Kanten involviert sind, ist auf eine besondere Rekursion zurückzuführen. Wenn entweder vom Endtriggershadow selbst oder von eine Folgeknoten aus eine rekursiver Aufruf zum Starttriggershadow oder zu einer Methode im Aufrufgraphen davor hinzugefügt wird, so handelt es sich um einen Matchpfad der Äquivalent zu dem ist, was er umschließt (siehe auch Abbildung 3.28 in Abschnitt 3.3).

Die Berechnung der Matchpfade wird eine SCC erkennen und unter Umständen eine ganze Menge von Matchpfaden zu einem Matchpfad mit einem Knoten zusammenschließen. Das bedeutet, dass beim Aufbau des Aufrufgraphen darauf geachtet werden muss, dass der Starttriggershadow nicht Teil einer SCC ist. Ist das der Fall, muss die Kante, welche zu der Rekursion führt, ignoriert werden. Wenn nach dieser Anpassung der Repräsentation die gleichen Matchpfade vorliegen wie davor, dann hat die Quellcodeänderung keine Auswirkung auf die spezifizierte Eigenschaft.

Es liegen auch äquivalente Pfade vor, wenn unterschiedliche Formen von mehrfach Matchpfaden als korrespondierende Matchpfade in den Repräsentationen enthalten sind. Im Sinne der Repräsentation handelt es sich zwar um äquivalente Matchpfade, statisch lässt sich keine Aussage darüber treffen, ob es nach der Änderung zu einer unterschiedlichen Anzahl von Endtriggern nach Eintritt in den Starttriggershadow kommt, aber es kann zumindest vermutet werden, dass zum Beispiel eine zusätzliche mehrfach Kante in einem Pfad zu mehr Endtriggern führen wird. Klarheit über die Auswirkungen kann in solchen Fällen lediglich der Entwickler haben, dem die entsprechenden Matchpfade präsentiert werden müssen.

In diesem Abschnitt 4.3 sind die Auswertungen beschrieben worden, mit denen zusätzliche, weggefallene und äquivalente Matchpfade in den ermittelten Repräsentationen identifiziert werden können. Zusammen mit der Möglichkeit eine Auswirkung ausschließen zu können, wenn keiner der Matchpfade betroffen ist, kann für Quellcodeänderungen, die die Matchpfade betreffen, in den meisten Fällen ermittelt werden, ob die spezifzierte Eigenschaft definitiv nicht betroffen ist oder in jedem Fall betroffen ist. In bestimmten Fällen wurde gezeigt, dass keine geeignete Zuordnung vorgenommen werden kann. Hier kann dem Entwickler das Ergebnis der Analyse in Form der betroffenen Matchpfade zur Verfügung gestellt werden.

4.4 Kapitelzusammenfassung

In diesem Kapitel sind zuerst verschiedene Arten von Algorithmen vorgestellt worden, die für den Aufbau eines Aufrufgraphen eingestetzt werden können. Unter den Algorithmen konnte der CHA Algorithmus als der für die Aufgabe der Ermittlung von Matchpfaden am besten geeignete identifiziert werden. Eine Abwägung von Faktoren, die für die Erstellung der Matchpfade in einem Werkzeug relevant sind, hat zu diesem Ergebnis geführt.

In dem folgenden Abschnitt wurden Verfahren zur Ermittlung der Matchpfade anhand des Aufrufgraphen vorgestellt, die der Repräsentation gemäss den Anforderungen aus Kapitel 3 gerecht werden.

Abschließend ist ein Verfahren zur Auswertung der aufgestellten Matchpfade erläutert worden, dass die Auswirkungen auf eine spezifizierte Eigenschaft *Cflow* identifiziert. Die Verfahren sind mit dem Ziel entwickelt worden, sie in einem Werkzeug zu realisieren, dass als ein Teil eines Refactoringwerkzeuges einen Entwickler bei der Durchführung von Refactorings unterstützen soll. Für eine Bewertung der erarbeiteten Analyseverfahren, hinsichtlich ihrer Anwendbarkeit und ihrem Nutzen, soll so ein Werkzeug realisiert werden.

Kapitel 5

Realisierung eines Verfahrens zur Analyse der Auswirkungen von Quellcodeänderungen auf die Joinpointeigenschaft Cflow

Im vorherigen Kapitel ist ein Analyseverfahren zur Ermittlung der Auswirkungen von Quellcodeänderungen auf die dynamische Joinpointeigenschaft *Cflow* entwickelt worden. An dieser Stelle soll gezeigt werden, wie dieses Verfahren prototypisch realisiert wurde. Die Beschreibung des so entstandenen Werkzeugs wird gefolgt von einer Darstellung des Einsatzes während eines Refactorings. Durch den Einsatz des Werkzeugs in verschiedenen Szenarien, werden die erreichten Ziele und noch nicht gelöste Probleme deutlich.

5.1 Realisierung des Verfahrens

Das Verfahren wurde nicht als allein stehende Lösung realisiert, sondern als Teil einer Entwicklungsumgebung und eines Pointcut-Analyseframeworks. Der Hauptvorteil liegt in der Nutzung von in der Umgebung vorhandenen Programmrepräsentationen, einer Repräsentation von Pointcuts und der Einbettbarkeit in einen bestehendes Refactoringwerkzeug. Im weiteren Verlauf werden zunächst genutzte Eigenschaften der Umgebung erklärt und dann werden Strukturen und Vorgänge in der Verwirklichung der Verfahren gezeigt.

5.1.1 Umgebung

Bei der gewählten Entwicklungsumgebung handelt es sich um Eclipse, eine Java-Entwicklungsumgebung, die über ein Plug-In Konzept besonders gut zu erweitern ist (vgl. des Rivieres u. Beaton 2006). Das *TOPPrax* Forschungsprojekt, in dessen Rahmen diese Arbeit entsteht, beinhaltet als eine der Schwerpunkte die Integration der aspektorientierten Programmiersprache ObjectTeams/Java (vgl. Herrmann 2005) in diese Entwicklungsumgebung. Die intensive Auseinandersetzung während der Projektmitarbeit und die dadurch gesammelten Kenntnisse haben letztlich zu der Entscheidung geführt, das zu entwickelnden Werkzeug ebenfalls in Eclipse zu integrieren.

Eclipse stellt Plug-Ins verschiedene Repräsentationen des Quellcodes zur Verfügung, die für die Realisierung des Verfahrens von Nutzen sind.

- DOM AST – Eine Form des Abstrakten Syntax Baums, die für jede Datei angefordert werden kann. Hier sind alle Quellcodeelemente vertreten bis hin zur Anweisungsebene.
- Java Model – Eine leichtgewichtigere Form des Abstrakten Syntax Baums, deren Elemente über eine Index-Suchmaschine projektweit mit geringem Zeitaufwand angefordert werden können. Quellcodeelemente sind nur bis zur Deklarationsebene verfügbar.
- Typhierarchie – Eine Typhierarchie die von Elementen des Java Model angefordert werden kann.

Für die Integration von ObjectTeams/Java ist auch eine Anpassung der Refactorings vorgesehen. Erste Refactorings sind im Rahmen einer anderen Diplomarbeit bereits realisiert worden (vgl. Brcan (2005)). In dem gegenwärtigen Stadium der Realisierung enthält ObjectTeams/Java noch keine Pointcutsprache. Für eine anstehende Umsetzung wird jedoch auch hierfür eine Berücksichtigung im Rahmen der Refactorings notwendig sein. Vorbereitend ist von der Arbeitsgruppe des *TOPPrax* Projektes ein Analyseframework und eine Anwendung des Frameworks für diese Zwecke als Plug-In für Eclipse geschaffen worden. Dieses Framework mit dem Namen *Soothsayer* ist auch die Umgebung für die realisierten Analyseverfahren. Einen Überblick über die Architektur stellt Abbildung 5.1 dar. Die Abbildung ist aus gemeinsamen Überlegungen in der Arbeitsgruppe entstanden.

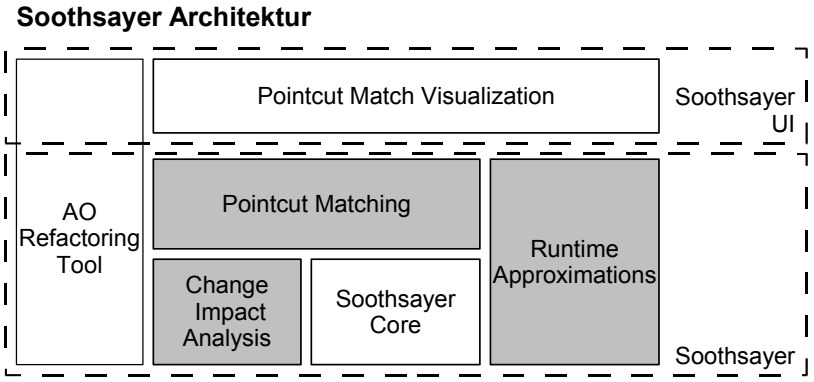


Abbildung 5.1: Architektur des Soothsayer Analyseframeworks in seiner Anwendung als Werkzeug zur Unterstützung von Refactorings in Gegenwart von Pointcuts (graue Bereiche werden Teile der in dieser Arbeit entwickelten Verfahren enthalten)

Aus der Abbildung wird durch die grau markierten Bereiche ersichtlich, an welchen Stelle die Implementierung der hier entwickelten Verfahren untergebracht sind. In dem Bereich *Runtime Approximations* kommen die Verfahren zum Aufbau der Matchpfade zum Einsatz, im Bereich *Pointcut Matching* werden die Verfahren angestoßen und ihre Ergebnisse verarbeitet und im Bereich *Change Impact Analysis* werden die Verfahren zum Vergleich der Matchpfade unterschiedlicher Programmversionen angesiedelt sein.

Bei den anderen Elementen der Übersicht handelt es sich mit *AO Refactoring Tool* um das eigentliche Refactoring-Werkzeug. Die *Pointcut Match Visualization* wird Visualisierungen bereitstellen, wenn eine Interaktion mit dem Entwickler notwendig wird in der der Entwickler Entscheidungen treffen muss, die das Werkzeug nicht treffen kann und auf Informationen der

aufgestellten Analyseergebnisse angewiesen ist (z.B. bei unklaren Situationen, die eine Anzeige der Matchpfade erfordern). *Soothsayer Core* stellt ein Framework für Analyseoperatoren und für die Behandlung unterschiedlicher Repräsentationen zur Verfügung.

Von Soothsayer werden bereits folgende Informationen zur Verfügung gestellt:

- Ein Abstrakter Syntax Graph (ASG), der Java Model und DOM AST in einer Repräsentation zusammenfasst.
- Eine Repräsentation der in Pointcuts spezifizierten Joinpointeigenschaften.
- Starttriggershadows und Endtriggershadows, gemäß den spezifizierten Singletrigger-Joinpointeigenschaften.
- Informationen über getätigte Quellcodänderungen (z.B. Methode von ... nach ... bewegt).

Im gegenwärtigen Stadium der Entwicklung stellt Soothsayer noch keine Klassenhierarchie und keine Kontrollflussanweisungen zur Verfügung, für diese Informationen greift der Prototyp noch direkt auf die Eclipse-Klassenhierarchie und den Eclipse-DOM-AST zu. Mit den von Eclipse und Soothsayer bereitgestellten Informationen sind genau diejenigen vorhanden, die in Abschnitt 4.2.1 als Voraussetzung für die Umsetzung der Verfahren als notwendig identifiziert wurden. Strukturen, die diese Umgebungen nicht zur Verfügung stellen, sind der Aufrufgraph und die Matchpfade.

5.1.2 Aufrufgraph Strukturen

Der Prototyp wird den Aufrufgraph in einer ersten Phase aufbauen und in einer nächsten Phase die Qualifizierung der Kanten vornehmen. In der ersten Phase, in der der Graph aufgebaut wird, Zyklen zusammengefasst werden und überflüssige Teile entfernt werden, reicht eine einfache Struktur aus. Für den einfachen Aufrufgraph ist dabei die in Abbildung 5.2 dargestellte Struktur entwickelt worden.

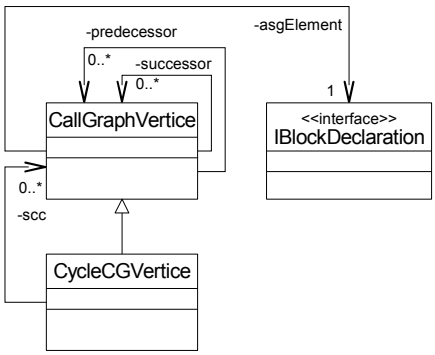


Abbildung 5.2: Struktur des einfachen Aufrufgraphen

Im erweiterten Aufrufgraph, der in der zweiten Phase entsteht, sind zusätzlich die qualifizierten Kanten als Strukturelemente vorhanden. Sie sind entsprechend Abbildung 5.3 umgesetzt.

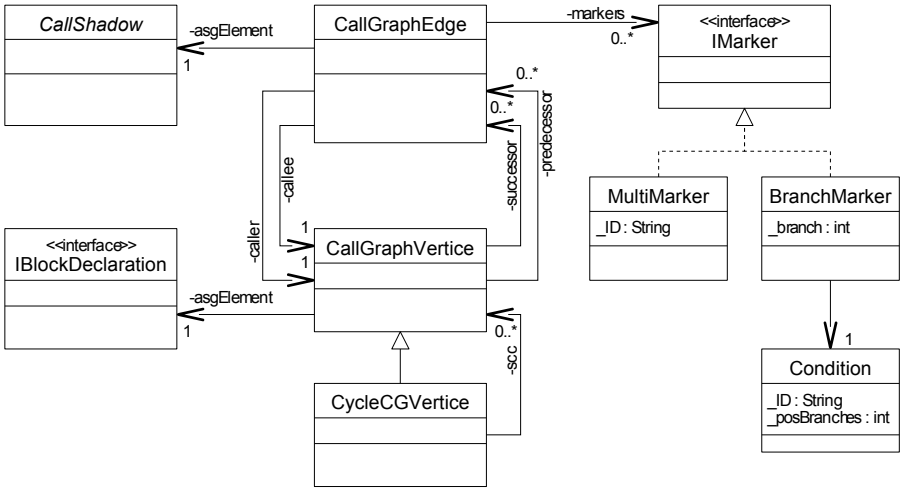


Abbildung 5.3: Struktur des erweiterten Aufrufgraphen

5.1.3 Einbindung in den Prozess des Refactoringwerkzeugs

In der Beschreibung der Einbindung des Prototyps in die Architektur von Soothsayer, ist durch die verschiedenen Architekturelemente bereits deutlich geworden, dass Soothsayer auch einen Analyseprozess vorgeben wird. Der Prozess, der bei Durchführung eines Refactorings vorgenommen wird, hat gegenwärtig folgende Form:

1. Aufbau der ASG Repräsentation für das Programm vor dem Refactoring A.
2. Bestimmung der von Pointcuts referenzierten Strukturen, ausgehend von A.
3. Durchführung des Refactorings.
4. Aufbau der ASG Repräsentation für das Programm nach dem Refactoring A'.
5. Bestimmung der von Pointcuts referenzierten Strukturen, ausgehend von A'.
6. Berechnung einer Struktur, die die Änderungen repräsentiert.
7. Ermittlung der Änderungen in den von den Pointcuts referenzierten Strukturen.
8. Untersuchung der Auswirkungen und Ableitung einer Entscheidung, ob eine Anpassung notwendig ist.
9. Weitere Schritte (z.B. Interaktion mit dem Entwickler).

Der Prozess ist nicht vollständig wiedergegeben, es handelt sich um den für den Prototypen relevanten Ausschnitt. Der entwickelte Prototyp ist in folgender Weise in den Prozess eingebunden. In Schritt 2 wird vor dem Aufbau der Matchpfade (von der Eigenschaft *Cflow* referenzierte Struktur) festgestellt, ob es sich um ein Refactoring handelt, das einen Einfluss auf die Eigenschaft *Cflow* haben kann. Wenn das der Fall ist, werden die Matchpfade für den ASG A aufgebaut. In Schritt 5 werden, wiederum in Abhängigkeit von dem angewendeten Refactoring, Matchpfade aufgebaut, diesmal auf dem ASG A'. Das Verfahren zu Ermittlung der Auswirkungen kommt in den Schritten 7 bis 9 zum Einsatz. Ist das Ergebnis zu dem der Prototyp gekommen ist unklar

oder ein Refactoring nicht durchführbar, dann wird eine Interaktion mit dem Benutzer erforderlich sein (Weiter Schritte). Der Prozess und die Punkte der Einbindung des Prototypen sind der Übersichtlichkeit halber noch einmal in Abbildung 5.4 dargestellt.

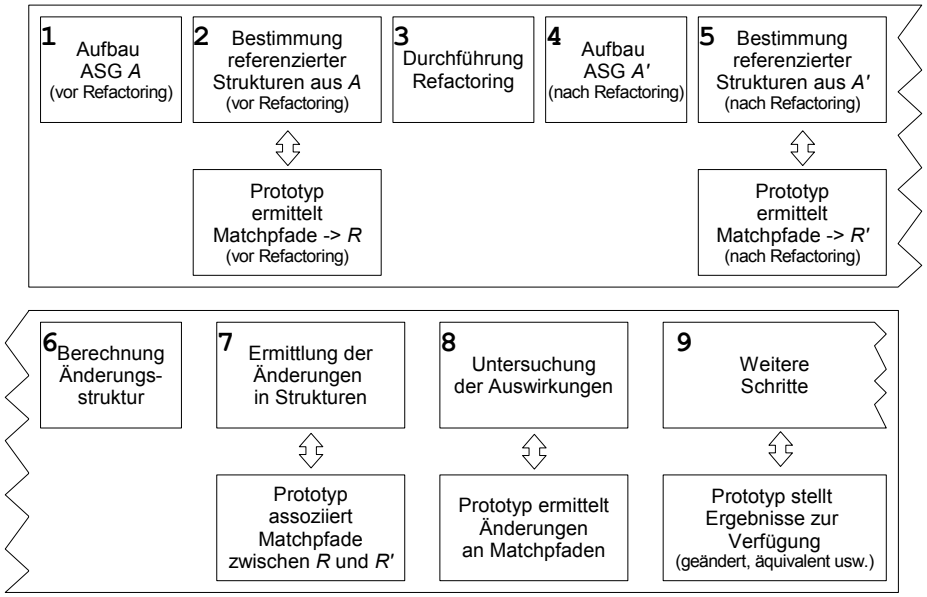


Abbildung 5.4: Die Einbindung des Prototypen in den Refactoringprozess der Soothsayer-Anwendung

Im gegenwärtigen Entwicklungsstatus des Prototyps ist es möglich die Auswirkung eines Refactorings auf eine spezifizierte Eigenschaft *Cflow* zu ermitteln. Der Prototyp ist in der Lage zu bestimmen, ob der Pointcut so belassen werden kann (keine Auswirkung oder äquivalente Matchpfade) oder ob die vorliegenden Auswirkungen des Refactorings eine Rücknahme des Refactorings erforderlich machen (unterschiedliche, nicht äquivalent ineinander überführbare Matchpfade). In bestimmten Fällen (besondere Formen äquivalenter Matchpfade) stellt der Prototyp eine nicht bewertbare Situation fest. Die noch nicht ausreichend vollständige Integration in Soothsayer erfordert, dass die Analyseergebnisse des Prototypen auf der Kommandozeile präsentiert werden.

5.2 Anwendung des Werkzeugs

Mit der prototypischen Implementierung der Verfahren konnten erste Versuche durchgeführt werden, die zeigen sollten welche Analyseergebnisse der Prototyp ermittelt. So konnte in Erfahrung gebracht werden, ob die erwarteten Analyseergebnisse tatsächlich berechnet werden oder ob es unerwartete Abweichungen gibt.

5.2.1 Szenario zusätzliche Pfade

Für den Test ist das Refactoring *Inline Temp* auf verschiedene Quellcodeteilen angewandt worden, wodurch unterschiedliche Merkmale der Analyse deutlich werden sollten. Das *Inline Temp*

Refactoring (vgl. Fowler u. a. 2005, Seite 119) führt unter bestimmten Voraussetzungen dazu, dass sich etwas an den Aufrufabhängigkeiten ändert. Dadurch ist es geeignet die Auswirkungen der verursachten Quellcodeänderungen zu untersuchen.

```
1  public class SomeClass
2  {
3    // ...
4    // starttriggershadow
5    public double
         computeSpecialSelectionPrices(
6      Product product1,
7      Product product2){
8
9      return product1.getPrice()
10       + product2.getPrice();
11   }
12   // ...
13 }
14
15 class Product {
16   private int _tax;
17
18   public double getPrice(){
19     int basePrice = computeBasePrice();
20     double discountFactor;
21     if(basePrice > 1000)
22     {
23       discountFactor = 0.95;
24     } else {
25       discountFactor = 0.98;
26     }
27     return basePrice *
28       discountFactor;
29   }
30
31   public int computeBasePrice(){
32     int result;
33     // ...
34     result = result + getSpecificTax();
35     // ...
36     return result;
37   }
38
39   // endtriggershadow
40   public int getSpecificTax() {
41     return _tax;
42   }
43 }
```

(a) Programm vor dem Refactoring

```
1  public class SomeClass
2  {
3    // ...
4    // starttriggershadow
5    public double
         computeSpecialSelectionPrices(
6      Product product1,
7      Product product2){
8
9      return product1.getPrice()
10       + product2.getPrice();
11   }
12   // ...
13 }
14
15 class Product {
16   private int _tax;
17
18   public double getPrice(){
19
20     double discountFactor;
21     if(computeBasePrice() > 1000)
22     {
23       discountFactor = 0.95;
24     } else {
25       discountFactor = 0.98;
26     }
27     return computeBasePrice() *
28       discountFactor;
29   }
30
31   public int computeBasePrice(){
32     int result;
33     // ...
34     result = result + getSpecificTax();
35     // ...
36     return result;
37   }
38
39   // endtriggershadow
40   public int getSpecificTax() {
41     return _tax;
42   }
43 }
```

(b) Programm nach dem Refactoring

Abbildung 5.5: Refactoring, das zu einem zusätzlichen Matchpfad führt

In einem ersten Szenario wird die in Abbildung 5.5 dargestellte Änderung des Programms in 5.5a zu dem Programm in 5.5b vorgenommen[1], als Resultat eines *Inline Temp*-Refactorings der temporären Variable `basePrice` (Zeile 21 in Abbildung 5.5a).

Der Prototyp ermittelt zunächst für die Version in 5.5a den Aufrufgraph, ausgehend von dem ihm übermittelten Starttriggershadow `double computeSpecialSelectionPrice()`. In dem Graphen befindet sich der ebenfalls in der Eingabe vorliegende Endtriggershadow `public int getSpecificTax()`. Die Ermittlung der SCC führt nicht zu Knoten die zusammengefasst werden

[1]Das Beispiel stellt eine abgewandelte Form des Beispiels aus (Fowler u. a. 2005, Seite 120) dar.

müssen und die Bereinigung des Aufrufgraphen, entfernt keine Knoten und Kanten. Bedingte Kanten oder mehrfache Kanten werden ebenso wenig ermittelt.

Bei der Berechnung der Matchpfade entstehen zwei sichere Pfade. Die zu tätigenden Änderungen führen zu einer entsprechenden Kennzeichnung betroffener Elemente. Für die Testzwecke wurde eine Ausgabe auf der Konsole implementiert, die folgendes Ergebnis zeigt:

```
Before change
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (r) -> computeBasePrice() -> getSpecificTax()]
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (r) -> computeBasePrice() -> getSpecificTax()]
```

Das `d` vor dem Matchpfad zeigt, dass es sich um einen sicheren (engl. definite) Pfad handelt, das `(c)` bedeutet eine Änderung an dem Element (engl. change) und das `(r)`, dass das Element entfernt wurde[2] (engl. removed). Die Methode `getPrice()` hat sich geändert als Folge der Entfernung des Aufrufs zu `computeBasePrice()`.

Die vorläufige Ausführung der Quellcodeänderungen stößt daraufhin erneut die Analyse des Prototypen an. Der Prototyp berechnet wieder die Matchpfade, diesmal auf der vorläufig geänderten Version des Programms und kommt zu folgendem Ergebnis:

```
After change
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (a) -> computeBasePrice() -> getSpecificTax()]
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (a) -> computeBasePrice() -> getSpecificTax()]
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (a) -> computeBasePrice() -> getSpecificTax()]
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (a) -> computeBasePrice() -> getSpecificTax()]
```

Zuletzt wird die Analyse zur Identifizierung der Auswirkungen der Quellcodeänderung (engl. *Change Impact Analysis*) auch für die dynamische Eigenschaft *Cflow* angestoßen. Hier kommt das Verfahren zum Vergleich der Repräsentation der dynamischen Joinpointeigenschaft *Cflow* zum Einsatz. Der Prototyp vergleicht die beiden Teile der Matchpfadmengen, die von Änderungen betroffen sind, in diesem Beispiel sind das alle. Er kommt zu dem Ergebnis, dass eine unterschiedliche Anzahl von Matchpfaden vorliegt, der Entwickler wäre an diese Stelle also zu warnen. Das Werkzeug wird das Refactoring nicht verhindern müssen. Ob sich das Verhalten wirklich geändert hat, hängt zusätzlich von dem durch den Pointcut gebundenen Advice ab. Die Ausgabe des Prototyps ist die folgende[3]:

```
Number of definite paths has changed: from 2 to 4
```

Der Prototyp hat in diesem Szenario das erwartete und auch das offensichtlich richtige Ergebnis ermitteln können.

5.2.2 Szenario äquivalente Pfade

Ein weniger einfach zu ermittelndes Resultat ergibt sich durch das nächste Beispiel. Auch hier wird wieder das *Inline Temp*-Refactoring angewandt. Durch den anderen Ausgangsquellcode werden hier für die Berechnung des richtigen Ergebnisses die Auswertung der Kanteninformation eine Rolle spielen. In dem Beispiel in Abbildung 5.6 sind nur die Quellcodeteile abgebildet, die sich von denen aus Abbildung 5.5 unterscheiden. Es wird das selbe Refactoring an derselben Variablen durchgeführt.

[2] Die Ausgabe wurde aus Gründen der Übersichtlichkeit gekürzt.

[3] Der Ausgabe folgt noch die Angabe der betroffenen Pfade vor und nach der Änderung, die mit den schon gezeigten Ausgaben identisch ist.

```
1  class Product {
2    //... omitted
3
4    public double getPrice(){
5      int basePrice = computeBasePrice();
6      double discountFactor;
7      double result;
8
9      if(computeWeight() > 1000) {
10       discountFactor = 0.98;
11       result = basePrice
12         * discountFactor;
13     } else {
14       discountFactor = 0.95;
15       result = basePrice
16         * discountFactor;
17     }
18     return result;
19   }
20   //... omitted
21 }
```

(a) Programm vor dem Refactoring

```
1  class Product {
2    //... omitted
3
4    public double getPrice(){
5      double discountFactor;
6      double result;
7
8      if(computeWeight() > 1000) {
9        discountFactor = 0.98;
10       result = computeBasePrice()
11         * discountFactor;
12
13     } else {
14       discountFactor = 0.95;
15       result = computeBasePrice()
16         * discountFactor;
17     }
18     return result;
19   }
20   //... omitted
21 }
```

(b) Programm nach dem Refactoring

Abbildung 5.6: Refactoring, das zu äquivalenten Matchpfaden führt

Als erstes wird wieder der Aufrufgraph berechnet. Auch in diesem Beispiel liegen keine Rekursionen vor. Die Bereinigung des von `double computeSpecialSelectionPrice()` ausgehenden Aufrufgraphen beseitigt den Aufruf zu `int computeWeight()`, bedingte oder mehrfache Kanten werden nicht ermittelt. Das Ergebnis besteht, wie im vorigen Beispiel, aus zwei Matchpfaden, mit entsprechender Ausgabe:

```
Before change
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (r) -> computeBasePrice() -> getSpecificTax()]
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (r) -> computeBasePrice() -> getSpecificTax()]
```

Wieder wird der Quellcode vorläufig der Änderung unterzogen und der Aufrufgraph auf dieser Version aufgebaut. Der Aufrufgraph wird in dieser Version ebenso von dem Aufruf zu `computeWeight()` bereinigt und es entstehen ebenso wenig mehrfach Kanten. Dagegen liegen in dieser Version zwei bedingte Kanten vor. Die folgende Ausgabe kommt zustande:

```
After change
c [compute..Prices(..) -> (c) getPrice() (a) [c1:1/2]-> computeBasePrice() -> getSpecificTax()]
c [compute..Prices(..) -> (c) getPrice() (a) [c1:2/2]-> computeBasePrice() -> getSpecificTax()]
c [compute..Prices(..) -> (c) getPrice() (a) [c1:1/2]-> computeBasePrice() -> getSpecificTax()]
c [compute..Prices(..) -> (c) getPrice() (a) [c1:2/2]-> computeBasePrice() -> getSpecificTax()]
```

Das `c` vor den Pfaden zeigt an, dass es sich um bedingte Pfade handelt (engl. conditioned). Markierungen, wie z.B. [c1:1/2], zeigen die Identifizierung der Verzweigungsanweisungen, um welche der möglichen Verzweigungen es sich handelt und wie viele Wege an der identifizierten Verzweigung genommen werden können.

Die *Change Impact Analysis* kann anhand der Information über die Quellcodeänderungen feststellen, dass der erste Pfad der Repräsentation vor der Änderung mit den beiden ersten Pfaden nach der Änderung in Beziehung steht und der zweite Pfad vor der Änderung mit dem dritten und vierten Pfad nach der Änderung. Die Zuordnung basiert auch auf der, in der Ausgabe nicht dargestellten, Unterscheidung zwischen verschiedenen Aufruforten (engl. call sites). Andernfalls wären parallele Kanten, wie sie hier auftreten, nicht zuordenbar.

Die Analyse der Kanteninformation erkennt, dass die jeweils zwei bedingten Pfade die mit einem sicheren Pfad zueinander in Beziehung stehen, im gegenseitigen Ausschluss liegen. Deswegen werden die bedingten Pfade als äquivalente Pfade zu den sicheren Pfaden eingestuft und es wird folgende Ausgabe produziert:

```
Detected equivalent Paths:
```

Der Rest der Ausgabe ist wiederum weggelassen worden, er entspricht der bereits ausgegebenen Information. Das Ergebnis der Analyse entsprach, wie im Beispiel davor, den Erwartungen und kann als richtig bewertet werden. Unabhängig davon, was in dem durch den Pointcut gebundenen Advice an Quelltext vorhanden ist, wird dieses Refactoring das Verhalten bezogen auf die spezifizierte Eigenschaft *Cflow* nicht geändert haben.

5.2.3 Szenario falsch bewertete Situation

Das folgende Beispiel hat hingegen zu einem falschen Ergebnis geführt. Abbildung 5.7 zeigt das Beispiel, das fast äquivalent zu dem vorherigen ist. Statt dem `if-else`-Konstrukt wurden hier nur `if`-Anweisungen benutzt und das Verhalten durch die Negation der ursprünglichen Bedingung in der zweiten `if`-Anweisung ersetzt (siehe in Abbildung 5.7a Zeile 14).

```
1  class Product {
2    //... omitted
3
4    public double getPrice(){
5      int basePrice = computeBasePrice();
6      double discountFactor;
7      double result;
8
9      if(computeWeight() > 1000) {
10       discountFactor = 0.98;
11       result = basePrice
12         * discountFactor;
13     }
14     if(!(computeWeight() > 1000)) {
15       discountFactor = 0.95;
16       result = basePrice
17         * discountFactor;
18     }
19     return result;
20   }
21   //... omitted
22 }
```

(a) Programm vor dem Refactoring

```
1  class Product {
2    //... omitted
3
4    public double getPrice(){
5      double discountFactor;
6      double result;
7
8      if(computeWeight() > 1000) {
9        discountFactor = 0.98;
10       result = computeBasePrice()
11         * discountFactor;
12
13     }
14     if(!(computeWeight() > 1000)) {
15       discountFactor = 0.95;
16       result = computeBasePrice()
17         * discountFactor;
18     }
19     return result;
20   }
21   //... omitted
22 }
```

(b) Programm nach dem Refactoring

Abbildung 5.7: Refactoring, das zu äquivalenten Matchpfaden führt, die nicht als solche identifiziert werden können

Für dieses Beispiel berechnet der Prototyp für den Quellcode vor der Änderung dieselbe Repräsentation in Form der Matchpfade, wie für das Beispiel aus Abbildung 5.6:

```
Before change
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (r) -> computeBasePrice() -> getSpecificTax()]
d [computeSpecialSelectionPrices(..) -> (c) getPrice() (r) -> computeBasePrice() -> getSpecificTax()]
```

Für die Berechnung der Repräsentation auf der vorläufig geänderten Quellcodeversion kommt der Prototyp zu einem anderen Ergebnis. Die Ermittlung der Pfade unterscheidet sich durch andere Markierungen an den Kanten. Hier gehen die Aufrufe nicht von derselben Verzweigungsanweisung als verschiedene Verzweigungen aus. Im vorherigen Beispiel war dies an den Markierungen mit derselben Identifizierung erkennbar. Das Ergebnis enthält stattdessen Aufrufe, wie durch die Ausgabe ersichtlich, die aus verschiedenen Verzweigungsanweisungen hervorgehen (Identifizierungen `c1` und `c2`). Die bedingten Aufrufe werden in den Kanten dadurch gekennzeichnet, dass die Verzweigung als die erste von zwei möglichen Verzweigungen einer Verzweigungsanweisung angegeben sind.

```
After change
c [compute..Prices(..) -> (c) getPrice() (a) [c1:1/2]-> computeBasePrice() -> getSpecificTax()]
c [compute..Prices(..) -> (c) getPrice() (a) [c2:1/2]-> computeBasePrice() -> getSpecificTax()]
c [compute..Prices(..) -> (c) getPrice() (a) [c1:1/2]-> computeBasePrice() -> getSpecificTax()]
c [compute..Prices(..) -> (c) getPrice() (a) [c2:1/2]-> computeBasePrice() -> getSpecificTax()]
```

In diesem Szenario stellt die *Change Impact Analysis*, wie im vorherigen Beispiel, die Verbindungen zwischen den Matchpfaden vor und nach der Änderung fest. Die Analyse der Kanten kommt in hier zu dem Ergebnis, dass sich aus den zwei sicheren Matchpfaden jeweils zwei bedingte Matchpfade ergeben haben. Im Gegensatz zum vorherigen Beispiel wurden die Matchpfade nicht als solche erkannt, die im gegenseitigen Ausschluss zueinander liegen — es liegt weder ein Pfad mit `[c1:2/2]` noch mit `[c2:2/2]` vor – und es konnten so nicht die äquivalenten Matchpfade erkannt werden.

Der Prototyp meldet für dieses Refactoring folgende Veränderung und hat damit das zu erwartende Ergebnis ermittelt:

```
Number and type of paths has changed: from 2 definite to 4 conditioned:
```

An dieser Stelle wird deutlich, dass das Verfahren durch das zugrunde liegende Konzept nicht jede Situation richtig bewerten kann und so Refactorings verhindert, die eigentlich durchgeführt werden könnten. Das konzeptbedingte Auftreten von Refactorings, die fälschlicherweise als nicht durchführbar bewertet werden, erfordert deswegen immer eine Interaktion mit dem Entwickler. Der kann z.B. anhand von entsprechend visualisierten Matchpfaden entscheiden, ob das Refactoring nicht doch vollzogen werden soll.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Jede Software unterliegt während ihres Lebenszyklus einem Evolutionsprozess. Dies ist unabhängig vom verwendeten Programmiermodell und trifft deswegen auch auf aspektorientierte Programme zu. Refaktorisieren wurde als Weg identifiziert, der mit Hilfe von Anpassungen der Struktur, den Umgang mit Änderungen an einem Programm im fortlaufenden Entwicklungsprozess ermöglicht. Das Refaktorisieren aspektorientierter Programme, die ebenso Änderungen im Evolutionsprozess unterliegen, ist die naheliegende Konsequenz.

Die aspektorientierte Programmierung bietet mit dem Aspekt eine neue Moduleinheit, die es erlaubt diejenigen Teile des Quellcodes, die sich im Programm nicht gut modularisiert implementieren lassen, im Aspekt zu kapseln. Dazu lässt sich einerseits die Struktur erweitern und andererseits das Verhalten anpassen. Beide Konzepte erfordern beim Refaktorisieren von Elementen der Basisprache die Berücksichtigung der eingesetzten aspektorientierten Sprachelemente.

In dieser Arbeit wurde untersucht, wie mit Sprachelementen zur Verhaltensanpassung – Advices und Pointcuts – umgegegangen werden kann, sodass bei Durchführung von objektorientierten Refactorings das beobachtbare Verhalten erhalten bleibt. Die von Opdyke aufgestellten Voraussetzungen für verhaltenserhaltende Quellcodeänderungen wurden dazu mit Spezifikationen von Joinpointeigenschaften in Pointcuts in Zusammenhang gebracht. Es hat sich gezeigt, dass die Erhaltung von Verhalten gewährleistet werden kann, wenn ein Pointcut die Joinpointeigenschaften, die er spezifiziert, in äquivalenten Strukturen vor und nach einem Refactoring referenziert.

Dabei haben sich Unterschiede in der Identifizierung äquivalent referenzierter Strukturen für verschiedene Joinpointeigenschaften herausgestellt. Für statische Joinpointeigenschaften können die Strukturen auf denen sie gelten problemlos ermittelt werden. Dynamische Joinpointeigenschaften gelten dagegen in dynamischen Strukturen, die bei der Durchführung eines Refactorings nicht vorliegen. Das besondere der dynamischen Strukturen ist vor allem, dass sie von der Ausführung des Programms abhängig sind und Programme auf unterschiedlichste Weise ausgeführt werden können.

Es wäre also theoretisch notwendig für jede mögliche Ausführung jeweils für das Programm vor und nach dem Refactoring die entsprechende dynamische Struktur zu ermitteln und auf dieser Grundlage zu bestimmen, ob von einem Pointcut äquivalente Joinpointeigenschaften in den Strukturen referenziert werden. Offensichtlich ist die Ermittlung unter dem Gesichtspunkt dessen, dass Programme bis zu unendliche viele Möglichkeiten haben ausgeführt zu werden, nicht sinnvoll.

Die Vermutung wurde aufgestellt, dass eine Repräsentation, die von für eine bestimmte Joinpointeigenschaft irrelevanten Informationen der Programmausführung abstrahiert und sich statisch für ein Programm approximieren lässt, dafür geeignet ist konservative Aussagen darüber zu machen, ob ein Refactoring Änderungen in dem was durch einen Pointcut referenziert wird zur Folge hat.

Am Beispiel der dynamischen Joinpointeigenschaft *Cflow* ist eine solche Repräsentation entwickelt worden, um zu zeigen, dass eine Repräsentation tatsächlich zu Aussagen über die Erhaltung des Verhaltens führen kann. Dazu ist ein Verfahren beschrieben worden, mit dem die Approximation der Repräsentation aus dem Quellcode erstellt werden kann.

Da die Repräsenation an sich noch keine Aussage über eventuelle Änderungen im Verhalten bereitstellt, wurde ein Verfahren zum Vergleich von Repräsentationen für das Programm vor und nach der Änderung entwickelt. Dieses Verfahren erlaubt es die Auswirkungen von Quellcodeänderungen so zu bewerten, dass eine Aussage möglich ist, ob die spezifizierte Eigenschaft *Cflow* genauso referenziert wird, äquivalent referenziert wird oder anders referenziert wird. In den ersten beiden Fällen wird sich das Verhalten nicht geändert haben, im letzten Fall wird es sich geändert haben. Es wurde aber auch festgestellt, dass es Situationen gibt, in denen das Verfahren zu keinem eindeutigen Ergebnis kommen kann, dort wird ein Eingriff des Entwicklers notwendig sein.

Die Verfahren können von einem Entwickler nicht ohne Unterstützung eingesetzt werden. Dazu sind Aufgaben wie die Ermittlung eines Aufrufgraphen schon bei Programmen mittlerer Größe zu komplex. Um trotzdem Nutzen aus den Verfahren ziehen zu können ist prototypisch ein Werkzeug entwickelt worden, das die entsprechenden Berechnungen vornimmt.

Mittels verschiedener Szenarien für die Durchführung eines Refactorings in denen der Prototyp zur Anwendung kam, konnten Möglichkeiten und Limitationen des Werkzeugs illustriert werden. So hat sich gezeigt, dass die entwickelte Repräsentation für die dynamische Joinpointeigenschaft *Cflow* zusammen mit den Verfahren zum Aufbau und zum Vergleich dieser Repräsentation, bei der Durchführung von Refactorings Aussagen bereitstellen kann, aus denen sich schließen lässt, ob das Refactoring wirklich vollzogen werden kann, oder ob die spezifizierte Eigenschaft in einer Weise beeinträchtigen wird, die das Erhalten des Verhaltens unmöglich macht.

6.2 Einschränkungen

Einige Punkte sind in dieser Arbeit noch offen geblieben.

- Die Ausnahmebehandlung von Java in Form des `try - catch - finally`-Konstrukts und der `throws`-Anweisung ist nicht berücksichtigt worden. Dies würde einerseits eine Erweiterung der Repräsentation erfordern und andererseits die Realisierung von entsprechenden Algorithmen für den Aufrufgraph.

- Der *Reflection*-Mechanismus von Java ist nicht beachtet worden. Die Verfahren werden für Programmen die *Reflection* einsetzen keine sinnvollen Ergebnisse ermitteln können.

- Fälle in denen in einer Methodendeklaration mehrere Start- oder Endtriggershadows vorliegen werden nicht richtig behandelt.

- Ein verwendeter Aufrufgraph wird in einem Javaprojekt auch Teile der Standard- oder anderer genutzter Bibliotheken enthalten müssen, die üblicherweise nicht im Quellcode vorliegen. Der realisierte Prototyp benötigt gegenwärtig das vollständige Programm als Quellcode, um den Aufrufgraphen aufbauen zu können.

- Mehr als zwei Trigger für einen *Cflow* sind noch nicht spezifizierbar: `cflow(t1, t2, t3)` mit `t1`, `t2`, `t3` Pointcuts von Singletrigger-Joinpointeigenschaften, ist ein Beispiel für einen Pointcut der noch nicht verarbeitet werden kann.

- Die Spezifikation einer dynamischen Joinpointeigenschaft *Cflow* kann auch Variablenbindungen ähnlich wie bei der Spezifikation einer Eigenschaft *Ereignissequenz* beinhalten (vgl. *Wormhole Pattern* beschrieben in Laddad (2003)). Variablenbindungen werden in der Repräsentation nicht berücksichtigt.

6.3 Ausblick

Fallstudien

Fallstudien wären in verschiedener Hinsicht hilfreich gewesen. Mit ihnen hätten neben dem *Inline Temp*-Refactoring auch andere objektorientierte Refactorings in ihren Auswirkungen auf die Repräsentation untersucht werden können, um so zu ermitteln, ob die entwickelten Verfahren das richtige Ergebnis berechnen.

Refactorings die zu Situationen führen, in denen die Verfahren nicht zu einem klaren Ergebnis kommen, hätten zeigen können, ob die ermittelten Repräsentationen ausreichende Anhaltspunkte für weitere Entscheidungen sind (Refactoring durchführen, nicht durchführen) oder ob der Entwickler eventuell mit unübersichtliche Mengen an Inforamationen konfrontiert wird, die er nicht auswerten kann.

In umfangreicheren Fallstudien wäre auch eine Bewertung der Performanz im Sinne der aufgewendeten Zeit und der genutzten Speicherresourcen möglich gewesen. Davon hängt letztlich die Einsetzbarkeit eines solchen Werkzeugs in einem Refactoringprozess ab.

Es konnte noch nicht geklärt werden, welche Form eine berechnete Repräsentation in einem realen Softwareprojekt hat. Studien könnten zeigen, wie umfangreich Repräsentationen werden. Es könnte sich ergeben, dass es sinnvoller ist einen kompletten Aufrufgraph eines Programms aufzubauen, statt der Teilgraphen.

Erarbeitung anderer dynamischer Joinpointeigenschaften

Die Erarbeitung weitere Repräsentationen dynamischer Joinpointeigenschaften, Aufbauend auf den Erkenntnissen und den entwickelten Analyeverfahren, würde den Einsatz auch anderer dynamischer Joinpointeigenschaften in einer dem Evolutionsprozess unterliegenden aspektorientierten Software ermöglichen.

Nutzung weiterer Programmrepräsentationen

In dieser Arbeit wurden zwei statischen Repräsentationen von Laufzeitverhalten zu Analysezwecken untersucht bzw. eingesetzt, der Aufrufgraph und der Kontrollflussgraph. Im Gebiet der Programmanalyse werden auch andere Programmrepräsentationen, wie zum Beispiel der *Program-Dependence-Graph* und Datenflussgraphen eingesetzt (vgl. Ferrante u. a. 1987). Es ist anzunehmen, dass sich mit diesen Programmrepräsentationen auch für weitere dynamische Joinpointeigenschaften Repräsentationen entwickeln lassen.

Fazit

In dieser Arbeit wurden Vorteile beleuchtet, die der Einsatz von dynamischen Pointcuts in aspektorientierten Programmen bietet (siehe Abschnitt 2.2.1). Dynamische Pointcuts sind zwar robuster als statische Pointcuts, dennoch gibt es Refactorings, die sie beeinträchtigen. Die Entwicklung statischer Analysen dynamischer Programmeigenschaften hat gezeigt, wie mit solchen Situationen umgegangen werden kann. Für die Eigenschaft *Cflow* hat sich gezeigt, dass solche Analysen durchführbar sind, für die Eigenschaft *Ereignissequenz* ist offen geblieben, ob hier eine Analyse realisierbar ist. Unter der Voraussetzung, dass auch aspektorientierte Programme refaktorisiert werden müssen, wirft sich an dieser Stelle die Frage auf, ob dynamische Joinpointeigenschaften zur Spezifikation in Pointcuts eingesetzt werden sollten, für die keine entsprechende Analyse realisierbar ist.

Es gibt eine Vielzahl von Wegen, die zu dem Ziel führen, das automatisierte Refactoring aspektorientierter Programme in einer Weise zu ermöglichen, wie sie schon bei objekorientierten Refactorings bekannt ist. Mit dieser Arbeit ist ein Schritt getan worden, um sich diesem Ziel weiter anzunähern, indem an einem Beispiel gezeigt wurde, wie Refactorings auch in Gegenwart von dynamischen Pointcuts durchgeführt werden können.

Anhang A

Auswahl von Aufrufgraphen

Bei der Erarbeitung der Faktoren für die Auswahl eines Algorithmus, der sich für die Analyse der Auswirkungen von Quellcodeänderungen auf die Joinpoiteigenschaft *Cflow* eignet, wurden verschiedene Algorithmen untersucht. Eine Auswahl wird hier vorgestellt.

A.1 Aufrufgraphen zur Abgrenzung

Diese Aufrufgraphen dienen zur Abgrenzung des Feldes der Aufrufgraphen bezüglicher der Präzision.

A.1.1 $G_{\perp}$

Dieser Graph lässt sich auf einfachste Weise erzeugen, indem von jedem Knoten eine Kante zu jedem anderen Knoten erzeugt wird. Das Bedeutet jede Methode ruft jede andere auf. $G_{\perp}$ ist der konservativste Graph, der sound ist und stellt damit die untere Grenze hinsichtlich der Präzision von sound Aufrufgraphen dar.

A.1.2 $G_{\top}$

Dieser Graph ist der optimistischste Graph, er enthält keine Kanten und ist unsound. $G_{\top}$ stellt die obere Grenze für unsound Aufrufgraphen dar.

A.1.3 G_{ideal}

$G_{\mathbf{ideal}}$ bildet *genau* alle möglichen Ausführungen eines Programmes ab. In diesem Graphen würde es weder Kanten geben, die nicht aufgerufen werden, noch würde es welche nicht geben die aufgerufen werden könnten. Leider lässt sich dieser Graph nicht unbedingt berechnen, da es für die meisten Programme unendlich viele Möglichkeiten gibt sie auszuführen (vgl. Grove u. a. 1997). $G_{\mathbf{ideal}}$ ist der optimistischste Graph der noch sound ist und stellt die obere Grenze für sound und die untere Grenze für unsound Aufrufgraphen dar.

A.2 Basis Algorithmen

Diese Algorithmen werden häufig als Ausgangsbasis für die Berechnung eines Aufrufgraphen mit komplexeren Algorithmen verwendet.

A.2.1 $G_{selector}$

Schränkt $G_{\perp}$ auf sinnvolle Kanten ein, indem nur diejenigen Kanten gültig sind, bei denen sowohl Methodennamen als auch Anzahl der übergebenen Parameter übereinstimmen.

A.2.2 CHA - Call Hierarchy Analysis

Wird an einer Referenz `a` des deklarierten Typs `A` eine Methode `m()` aufgerufen, so muss angenommen werden, dass neben `A.m()` auch jede andere Methode `m()`, die in einem Subtyp von `A` deklariert ist, einen potentielles Aufrufziel darstellt, oder die Methode in einem der Supertypen implementiert ist. Methoden aus Klassen, die nicht in der Hierarchie des deklarierten Empfängerobjektes stehen, kommen als Aufrufziel nicht in Betracht. Für streng typisierte objektorientierte Sprachen würden konservativere Algorithmen keinen sinnvollen Aufrufgraph mehr ergeben, weiter Kanten würden Aufrufe darstellen, die nicht typkonform sind. Dieser Algorithmus wurde in (Dean u. a. 1995) vorgestellt.

A.2.3 RTA - Rapid Type Analysis

Eine verbesserte Form eines mittels CHA erstellten Callgraphen, der in zwei unterschiedlichen Varianten existiert.

A.2.3.1 pessimistisch

Wenn an einem Knoten ein Aufruf an einem Objekt `a` vom deklarierten Typ `A` gemacht wird, so werden von den durch die CHA aufgestellten möglichen Aufrufzielen nur diejenigen beibehalten, deren Empfängertyp irgendwo im Programm instantiiert wurde. Methoden die in `A` oder Subtypen davon deklariert wurden, die im Programm gar keine Instantiierung erfahren, können auch nicht aufgerufen werden.

A.2.3.2 optimistisch

Der Unterschied zwischen der pessimistischen und der optimistischen Variante ist, dass bei der Optimistischen nicht alle irgendwo instantiierten Typen in Frage kommen, sondern nur die bis zu dem gerade zu analysierenden Knoten im Graphen instantiierten. Eine sinnvolle Annahme, da ein Typ der erst zu einem späteren Zeitpunkt instantiiert wird (also in einem dem untersuchten Knoten nachfolgenden Knoten) kein Aufrufempfänger sein kann. Aufgrund von möglichen Zyklen im Graphen muss bei der optimistischen Variante solange iteriert werden bis sich an einem Knoten kein Änderung mehr feststellen lässt (vgl. Bacon u. Sweeney 1996).

Die Rapid Type Analysis (sowohl pessimistisch als auch optimistisch) kann einen Callgraph mit nahezu demselben Aufwand wie CHA erzeugen. Der Ergebnisgraph ist optimistischer als der durch einen CHA Algorithmus erstellten.

A.3 Kontextsensitive Algorithmen

Kontextsensitive Algorithmen berücksichtigen den Aufrufkontext, unter Umständen auch über mehrere vorangegangene Ebenen. Verschiedene Elemente können dabei als Kontext zum Einsatz kommen.

A.3.1 Call Strings

Für den *Call Strings* Algorithmus wird als Kontext die aufrufende Methode gesetzt. (vgl. Sharir u. Pnueli 1981, zitiert nach Ryder (2003)).

A.3.2 Object Sense

Der Object Sense Algorithmus setzt für jedes unterscheidbare Empfängerobjekt einen jeweiligen Kontext (vgl. Milanova u. a. 2002).

A.3.2.1 CPA - Agesen's Cartesian Product Algorithm

Dieser Algorithmus erweitert zunächst die Menge der vorhandenen Knoten, die jeweils für eine bestimmte Methode stehen, zu einer Menge von Konturen in der es für jede Methode eine Gruppe von Knoten gibt. Die Knoten einer solchen Gruppe unterscheiden sich durch eine eigene Festlegung der Parameter und Empfängertypen. Für das Beispiel aus Abbildung 4.1 würde das bedeuten, dass für die Methode `max(Number, Number)` eine Menge von Knoten: `max(Integer, Integer)`, `max(Integer, Float)`, `max(Float, Integer)`, `max(Float, Float)` erzeugt wird[1]. Leicht lässt sich erkennen, dass dieser Algorithmus in seinem Aufwand recht hoch sein kann, wenn die durchschittliche Anzahl von Parametern in einem Programm hoch ist.

A.4 Propagation Based Algorithmen

Algorithmen die Informationen beim Aufbau des Graphen von einem Knoten zum nächsten propagieren. Meist handelt es sich um Typinformationen.

[1] Unter der optimierten Annahme, dass Number selbst abstrakt ist, `Integer` und `Float` die einzigen Subtypen von `Number` sind und die Klasse `AClass` keine Subklassen hat

A.4.1 XTA - (Codename für diesen Algorithmus)

Zunächst wird für jedes Feld und jede Methode des zu analysierenden Programms eine leere Typmenge angelegt. Diese Mengen werden folgendermassen beeinflusst:

- Wird ein Objekt instanziiert und einer lokalen Variablen zugewiesen, so wird der Objekttyp der zur Methode zugehörigen Menge (Methodenmenge) hinzugefügt.
- Wird in einer Methode aus einem Feld gelesen, so werden die Typen der entsprechenden Feldmenge der Methodenmenge hinzugefügt.
- Wird in einer Methode in ein Feld geschrieben, so werden die Typen aus der Methodenmenge der dem Feld zugehörigen Menge (Feldmenge) hinzugefügt. Dabei werden die Typen gefiltert, in dem Sinne, dass nur die Typen hinzugefügt werden, die dem Feltypen oder einem Subtypen entsprechen.
- Wird in einer Methode `m(...)` eine Methode `n(...)` aufgerufen, so werden die folgenden zwei Aktionen durchgeführt:
  1. Alle Typen der zu `m(...)` gehörigen Methodenmenge werden der Methodenmenge die zu `n(...)` gehört hinzugefügt. Die Typen werden (entsprechend dem Filtern beim Schreiben in ein Feld), durch die Parametertypen gefiltert.
  2. Alle Typen der zu `n(...)` gehörigen Methodenmenge werden der zu `m(...)` gehörigen Menge hinzugefügt. In dieser Richtung findet die Filterung entsprechend dem Rückgabetyp von `n(...)` statt.

In Tip u. Palsberg (2000) wird dieser Algorithmus und die drei folgenden beschrieben. Ausserdem werden die Algorithmen bezüglich der Anzahl der verwendeten Mengen zur Approximation der Laufzeitwerte von Expressions mit anderen Algorithmen verglichen. In Tip u. a. (2002) ist in Abschnitt 3. (Call Graph Construction) eine etwas besser verständliche Beschreibung anhand eines Beispiels zu finden.

A.4.2 CTA

Der CTA Algorithmus ist eine vereinfachte Form des XTA Algorithmus. Statt für jede Methode und jedes Feld eines Programmes, existiert nur für jede Klasse eine eigene Typmenge. Der Algorithmus verhält sich analog zu dem oben beschriebenen. Der Unterschied besteht darin, dass statt dem Schreiben in die oder Lesen aus der Methodenmenge oder Feldmenge, ein Schreiben in die oder Lesen aus der Klassenmenge stattfindet, in der die Methode oder das Feld deklariert ist.

A.4.3 MTA

Der MTA Algorithmus ist ebenfalls eine vereinfachte Form des XTA Algorithmus, bzw. eine erweiterte des CTA Algorithmus. Hier gibt es für jede Klasse und jedes Feld eine eigene Menge. Feldmengen werden wie in XTA behandelt. In die Klassenmengen werden Typen geschrieben oder gelesen, wenn in XTA in eine in der Klasse deklarierten Methodenmenge geschrieben oder aus ihr gelesen würde.

A.4.4 FTA

Entsprechend dem MTA Algorithmus nur ist die Rolle der Methoden- und Feldmengen vertauscht.

A.4.5 0-CFA (CFA - Control Flow Analysis)

Dieser Algorithmus wurde ursprünglich am Beispiel der Programmiersprache Scheme im Rahmen einer Doktorarbeit entwickelt (Shivers (1991)). Er kann als erweiterter XTA Algorithmus verstanden werden (siehe auch: Tip u. Palsberg (2000)). In 0-CFA gäbe es dann nicht nur für jede Methode nur eine Methodenmenge. Stattdessen würde für jedes Argument und jeden Ausdruck das/der zu einem Objekt evaluieren kann, jeweils eine dazugehörige Menge existieren. Das würde auch Objektreferenzen auf dem Runtimestack miteinbeziehen. Offensichtlich wäre die Anzahl der zu berücksichtigten Mengen um ein vielfaches höher als bei dem XTA Algorithmus.

A.5 Points-To Analysen

Die folgenden Algorithmen stellen eine sehr kleine Auswahl dar. Eine etwas weitere, allerdings nicht mehr ganz aktuelle Auswahl (z.B. fehlen die BDD-Algorithmen), wird z.B. in Hind (2001) beschrieben.

A.5.1 Equality Based

Algorithmen, die Typmengen gleichsetzen. Diese Art von Algorithmen ist für streng typisierte Sprachen wie Java nicht sinnvoll, da durch die Gleichsetzung von Typmengen, Typen für Variablen ermittelt werden, die wegen der Typisierung nicht zugelassen sind.

A.5.1.1 Steensgards Algorithmus

Erster Equality Based Ansatz. Fast linear im Aufwand, erzeugt aber einen recht ungenauen Graphen. Der Algorithmus wird von Steensgard in Steensgaard (1996) für eine C-ähnliche Sprache beschrieben.

A.5.2 Subset Based

A.5.2.1 Andersens Subset-Based Points-To Analyse

Der erste Subset Based Ansatz wurde in Andersen (1994) vorgestellt. Im ungünstigsten Fall hat er einen kubischen Aufwand.

A.5.2.2 BDD - Binary Decision Diagrams

Binary Decision Diagrams sind, wie der Name schon erkennen lässt, nicht selbst ein Algorithmus sondern eine Datenstruktur. Mit dieser lassen sich große Mengen sehr effizient darstellen und traversieren. In einer relativ jungen Entwicklung (2003) der Sable Gruppe (auch verantwortlich für das Soot Compilerframework) wurde ein Algorithmus implementiert der auf Binary Decision Diagrams aufbaut. In Berndl u. a. (2003) stellen M. Berndl et al. einen Points-To Analysealgorithmus vor, der zur Gruppe der Subset Based Algorithmen gehört. Obwohl Subset Based Algorithmen im Ruf stehen nicht besonders gut zu skalieren, haben es M. Berndl et al. nach eigenen Angaben geschafft, mittels der sehr kompakten Repräsentation der Points-to-Informationen, in Form von Binary Decision Diagrams, gute Ergebnise hinsichtlich der Performanz (Verbrauch von Speicher/Zeit) zu erreichen. Ausserdem existiert auch eine Implementierung in Zusammenhang mit einer Callgraphanalyse als Teil des öffentlich und im Quellcode verfügbaren Soot Compilerframeworks. Darüberhinaus wird dieser Algorithmus entsprechend einer Angabe in Avgustinov u. a. (2005) auch für einen aspektorientierten Übersetzer (ABC Compiler) zu dessen Optimierung angewandt.

Literaturverzeichnis

Aho u. a. 1987
Aho, Alfred V. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1987

Allan u. a. 2005
Allan, Chris ; Avgustinov, Pavel ; Christensen, Aske S. ; Hendren, Laurie ; Kuzins, Sascha ; Lhoták, Ondřej ; Moor, Oege de ; Sereni, Damien ; Sittampalam, Ganesh ; Tibble, Julian: Adding trace matching with free variables to AspectJ. In: *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, ACM Press, Oktober 2005, S. 345–364

Andersen 1994
Andersen, L. O.: *Program Analysis and Specialization for the C Programming Language*, DIKU, University of Copenhagen, Diss., Mai 1994. – (DIKU report 94/19)

Avgustinov u. a. 2005
Avgustinov, P. ; Christensen, A. ; Hendren, L. ; Kuzins, S. ; Lhoták, J. ; Lhoták, O. ; Moor, O. de ; Sereni, D. ; Sittampalam, G. ; Tibble, J.: *Optimising Aspectj*. 2005. – To appear

Bacon u. Sweeney 1996
Bacon, David F. ; Sweeney, Peter F.: Fast Static Analysis of C++ Virtual Function Calls. In: *OOPSLA*, 1996, S. 324–341

Berndl u. a. 2003
Berndl, M. ; Lhotak, O. ; Qian, F. ; Hendren, L. ; Umanee, N.: Points-to analysis using BDDs. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Inplementation*, ACM Press, 2003, S. 103–114

Brcan 2005
Brcan, Gregor: *Erweiterung von objektorientiertem Refactoring für die aspektorientierte Sprache ObjectTeams/Java*, Technische Universität Berlin, Diplomarbeit, Oktober 2005

Chen u. Wagner 2002
Chen, Hao ; Wagner, David: MOPS: an infrastructure for examining security properties of software. In: *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*. New York, NY, USA : ACM Press, 2002, S. 235–244

Cormen u. a. 2001
Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to Algorithms, Second Edition*. Cambridge, MA, USA : The MIT Press, 2001

De Fraine u. a. 2005
De Fraine, Bruno ; Vanderperren, Wim ; Suvée, Davy ; Brichau, Johan: Jumping Aspects Revisited. In: Filman, Robert E. (Hrsg.) ; Haupt, Michael (Hrsg.) ; Hirschfeld, Robert (Hrsg.): *Dynamic Aspects Workshop*, 2005, 77–86

Dean u. a. 1995
Dean, Jeffrey ; Grove, David ; Chambers, Craig: Optimization of Object-Oriented Programs Using Class Hierarchy Analysis. In: Olthoff, W. (Hrsg.): *ECOOP '95 - Object-Oriented Programming: 9th European Conference* Bd. 952. London, UK : Springer-Verlag, August 1995 (Lecture Notes in Computer Science), S. 77–101

Dijkstra 1974
Dijkstra, Edsger W.: *On the role of scientific thought.* August 1974. – published as Dijkstra (1982)

Dijkstra 1982
Dijkstra, Edsger W.: On the role of scientific thought. In: *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, 1982, S. 60–66

Dijkstra 1997
Dijkstra, Edsger W.: *A Discipline of Programming.* Upper Saddle River, NJ, USA : Prentice Hall PTR, 1997

Douence u. a. 2001
Douence, Rémi ; Motelet, Olivier ; Südholt, Mario: A Formal Definition of Crosscuts. In: Yonezawa, A. (Hrsg.) ; Matsuoka, S. (Hrsg.): *Metalevel Architectures and Separation of Crosscutting Concerns 3rd International Conference (Reflection 2001), LNCS 2192*, Springer-Verlag, September 2001, S. 170–186

Eichberg u. a. 2004
Eichberg, Michael ; Mezini, Mira ; Ostermann, Klaus: Pointcuts as Functional Queries. In: Chin, Wei-Ngan (Hrsg.): *Programming Languages and Systems: Second Asian Symposium, APLAS 2004.* Taipei, Taiwan : Springer-Verlag Heidelberg, November 2004 (Lecture Notes in Computer Science), S. 366–382

Ferrante u. a. 1987
Ferrante, Jeanne ; Ottenstein, Karl J. ; Warren, Joe D.: The program dependence graph and its use in optimization. In: *ACM Trans. Program. Lang. Syst.* 9 (1987), Nr. 3, S. 319–349

Fowler u. a. 1999
Fowler, M. ; Beck, K. ; Brant, J. ; Opdyke, W. ; Roberts, D.: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999

Fowler 2006
Fowler, Martin: *Refactoring Tools.* 2006. – Abrufdatum Januar 2006, URL: `http://www.refactoring.com/tools.html`

Fowler u. a. 2005
Fowler, Martin ; Beck, Kent ; Brant, John ; Opdyke, William ; Roberts, Don ; Übers., Bernd K.: *Refactoring: Wie Sie das Design vorhandener Software verbessern.* München; Boston u.a. : Addison-Wesley, 2005

Frederick P. Brooks 1975
Frederick P. Brooks, Jr.: *The Mythical Man-Month: Essays on Software Engineering.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1975

Gamma u. a. 1995
GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995

Gosling u. a. 2005
GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *Java(TM) Language Specification, Third Edition*. Addison-Wesley Professional, 2005

Grove u. a. 1997
GROVE, David ; DEFOUW, Greg ; DEAN, Jeffrey ; CHAMBERS, Craig: Call graph construction in object-oriented languages. In: *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA : ACM Press, 1997, S. 108–124

Gybels 2001
GYBELS, Kris: *Aspect-Oriented Programming using a Logic Meta Programming language to express cross-cutting through a dynamic joinpoint structure*, Vrije Universiteit Brussel, Licentiate thesis, August 2001

Gybels 2002
GYBELS, Kris: Using a logic language to express cross-cutting through dynamic joinpoints. In: COSTANZA, Pascal (Hrsg.) ; KNIESEL, Günter (Hrsg.) ; MEHNER, Katharina (Hrsg.) ; PULVERMÜLLER, Elke (Hrsg.) ; SPECK, Andreas (Hrsg.): *Second Workshop on Aspect-Oriented Software Development of the German Information Society*, Institut für Informatik III, Universität Bonn, Februar 2002. – Technical report IAI-TR-2002-1

Gybels u. Brichau 2003
GYBELS, Kris ; BRICHAU, Johan: Arranging language features for more robust pattern-based crosscuts. In: *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. New York, NY, USA : ACM Press, 2003, S. 60–69

Hanenberg u. a. 2003
HANENBERG, Stefan ; OBERSCHULTE, Christian ; UNLAND, Rainer: Refactoring of Aspect-Oriented Software. In: *Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World, Net.ObjectDays 2003 (NODe 2003)*, Springer-Verlag, September 2003, S. 19–35

Hannemann u. a. 2005
HANNEMANN, Jan ; MURPHY, Gail C. ; KICZALES, Gregor: Role-based refactoring of crosscutting concerns. In: *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, USA : ACM Press, 2005, S. 135–146

Herrmann 2002
HERRMANN, Stephan: Object Teams: Improving Modularity for Crosscutting Collaborations. In: AKŞIT, Mehmet (Hrsg.) ; MEZINI, Mira (Hrsg.): *Net.Object Days 2002*, 2002

Herrmann 2005
HERRMANN, Stephan: *Object Teams / TOPPrax Projektseite*. 2005. – Abrufdatum April 2006, URL: `http://www.objectteams.org`

Hilsdale u. Hugunin 2004
HILSDALE, Erik ; HUGUNIN, Jim: Advice weaving in AspectJ. In: LIEBERHERR, Karl (Hrsg.):

Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004), ACM Press, März 2004, S. 26–35

Hind 2001
HIND, Michael: Pointer analysis: haven't we solved this problem yet? In: *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA : ACM Press, 2001, S. 54–61

Kiczales u. a. 1997
KICZALES, Gregor ; LAMPING, John ; MENHDHEKAR, Anurag ; MAEDA, Chris ; LOPES, Cristina ; LOINGTIER, Jean-Marc ; IRWIN, John: Aspect-Oriented Programming. In: AKŞIT, Mehmet (Hrsg.) ; MATSUOKA, Satoshi (Hrsg.): *Proceedings European Conference on Object-Oriented Programming* Bd. 1241. Berlin, Heidelberg, New York : Springer-Verlag, 1997, S. 220–242

Koppen u. Störzer 2004
KOPPEN, Christian ; STÖRZER, Maximilian: PCDiff: Attacking the fragile Pointcut Problem. In: *EIWAS 2004, European Interactive Workshop on Aspects in Software*, 2004

Laddad 2003
LADDAD, Ramnivas: *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, CT, USA : Manning Publications Co., 2003

Lehman u. Belady 1985
LEHMAN, M. M. (Hrsg.) ; BELADY, L. A. (Hrsg.): *Program evolution: processes of software change*. San Diego, CA, USA : Academic Press Professional, Inc., 1985

Masuhara u. a. 2002
MASUHARA, Hidehiko ; KICZALES, Gregor ; DUTCHYN, Chris: Compilation Semantics of Aspect-Oriented Programs. In: CYTRON, Ron (Hrsg.) ; LEAVENS, Gary T. (Hrsg.): *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, 2002, 17-26

Mens u. Tourwé 2004
MENS, Tom ; TOURWÉ, Tom: A Survey of Software Refactoring. In: *IEEE Transactions on Software Engineering* 30 (2004), Februar, Nr. 2, S. 126–139

Milanova u. a. 2002
MILANOVA, Ana ; ROUNTEV, Atanas ; RYDER, Barbara G.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA : ACM Press, 2002, S. 1–11

Opdyke 1992
OPDYKE, William F.: *Refactoring object-oriented frameworks*. Champaign, IL, USA, University of Illinois at Urbana-Champaign, Diss., 1992

Ostermann u. a. 2005
OSTERMANN, Klaus ; MEZINI, Mira ; BOCKISCH, Christoph: Expressive Pointcuts for Increased Modularity. In: *ECOOP '05: Proceedings of the European Conference on Object-Oriented Programming*, Springer LNCS, 2005

des Rivieres u. Beaton 2006
RIVIERES, Jim des ; BEATON, Wayne: *Eclipse Platform Technical Overview*. April 2006. – Abrufdatum April 2006, Version Februar 2003 `http://www.eclipse.org/whitepapers/`

eclipse-overview.pdf und aktuelle Version April 2006 http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html

Rura 2003
Rura, Shimon: *Refactoring Aspect-Oriented Software.* Undergraduate Thesis, Mai 2003

Ryder 2003
Ryder, Barbara G.: Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In: Hedin, G. (Hrsg.): *Compiler Construction: 12th International Conference* Bd. 2622, Springer, Januar 2003 (Lecture Notes in Computer Science), S. 126 – 137

Sharir u. Pnueli 1981
Sharir, M. ; Pnueli, A.: Two Approaches to Inter-Procedural Data-Flow Analysis. In: Jones, Neil D. (Hrsg.) ; Muchnick, Steven S. (Hrsg.): *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981, S. 189 – 234

Shivers 1991
Shivers, Olin G.: *Control-flow analysis of higher-order languages or taming lambda.* Pittsburgh, PA, USA, Diss., 1991

Steensgaard 1996
Steensgaard, Bjarne: Points-to analysis in almost linear time. In: *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* New York, NY, USA : ACM Press, 1996, S. 32–41

Streckenbach u. Snelting 2000
Streckenbach, Mirko ; Snelting, Gregor: Points-to for java: A general framework and an empirical comparison / University Passau. 2000. – Forschungsbericht. – Eingereicht zur Veröffentlichung

Sundaresan u. a. 2000
Sundaresan, V. ; Hendren, L. ; Razafimahefa, C. ; Vallée-Rai, R. ; Lam, P. ; Gagnon, E. ; Godin, C.: Practical virtual method call resolution for Java. In: *ACM SIGPLAN Notices* 35 (2000), Nr. 10, S. 264–280

Tarjan 1972
Tarjan, Robert E.: Depth-First Search and Linear Graph Algorithms. In: *Siam Journal on Computing* 1 (1972), Juni, Nr. 2, S. 146 – 160

Tip u. a. 2002
Tip, F. ; Sweeney, P. F. ; Laffra, C. ; Eisma, A. ; Streeter, D.: Practical extraction techniques for Java. In: *ACM Transactions on Programming Languages and Systems* 24 (2002), Nr. 6, S. 625–666

Tip u. Palsberg 2000
Tip, Frank ; Palsberg, Jens: Scalable propagation-based call graph construction algorithms, 2000, S. 281–293

Volder u. D'Hondt 1999
Volder, Kris D. ; D'Hondt, Theo: Aspect-Oriented Logic Meta Programming. In: *Meta-Level Architectures and Reflection: Second International Conference, Reflection'99* Bd. 1616/1999, Springer LNCS Berlin/Heidelberg, Juli 1999, S. 250

von Flach G. Chavez u. a. 2005

von Flach G. Chavez, Christina ; Garcia, Alessandro ; Kulesza, Uirá ; Anna, Cláudio S. ; Lucena, Carlos: Taming Heterogeneous Aspects with Crosscutting Interfaces. In: *Journal of the Brazilian Computer Society* (2005), Dezember

Wloka 2005

Wloka, Jan: Aspect-aware Refactoring tool support. In: Tourwé, Tom (Hrsg.) ; Kellens, Andy (Hrsg.) ; Ceccato, Mariano (Hrsg.) ; Shepherd, David (Hrsg.): *Linking Aspect Technology and Evolution*, 2005