Entwicklung eines Debuggers für aspektorientierte Programme unter Berücksichtigung der Aspektwebetechniken

zur Erlangung des akademischen Grades
Diplom-Informatiker

vorgelegt dem
Fachbereich Informatik Fakultät IV der Technischen Universität Berlin

Ralf Anklam
Matrikelnr. 193363, ikeman@cs.tu-berlin.de

Gutachter: Prof. Dr.-Ing. Stefan Jähnichen
2.Gutachter: Dr. Ing. Stephan Hermann
Betreuung: Dipl.-Inf. Carsten Pfeiffer

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, 6. Juli 2006

_____

Unterschrift

Zusammenfassung

Bei fast jeder Phase der Softwareentwicklung kommt es zu Fehlern. Der Prozess der Fehlersuche und dessen Beseitigung wird als Debugging bezeichnet. Dabei stehen dem Entwickler Werkzeuge (Debugger) zu Verfügung, die ihn bei der Fehlersuche und dem Verstehen eines Programms unterstützen. Die Werkzeuge für prozedurale und objektorientierte Sprachen erfüllen die an sie gestellten Anforderungen und erleichtern die Fehlersuche in einem Programm. Welche Anforderungen für Debugger von aspektorientierten Programmen gelten und welche Probleme sich bei der Entwicklung eines solchen Werkzeugs ergeben, beleuchtet diese Diplomarbeit. Dabei werden die Webemechanismen verschiedener Sprachen berücksichtigt.

Die Sprache Object Teams/Java ist ein Vertreter des aspektorientierten Paradigmas. Für diese Sprache wird ein Debugger entwickelt, der die erarbeiteten Anforderungen erfüllt. Der Debugger ist Bestandteil der Entwicklungsumgebung *Object Teams Development Tooling*, welches auf der Eclipse Tool Platform basiert.

Inhaltsverzeichnis

Danksagung

An dieser Stelle bedanke ich mich bei all jenen, die mir beim Erstellen dieser Arbeit durch ihre Unterstützungen zur Seite standen.

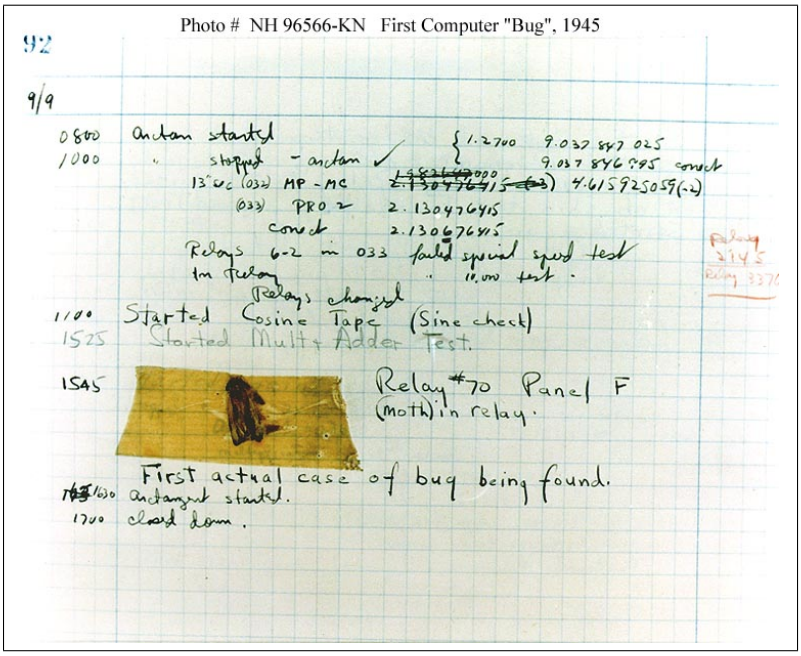
Ich danke Carsten Pfeiffer für die Betreuung meiner Diplomarbeit. Seine Unterstützung in Rat und Tat gab mir Selbstvertrauen und seine Kommentare und Hilfestellungen nahmen mir den wohl üblichen Druck, der beim Erstellen einer Diplomarbeit auftritt. Seine Implementierungsarbeiten flossen im Rahmen dieser Arbeit ein und wurden vom mir, mit seiner freundlichen Unterstützung, dokumentiert.

Weiterhin danke ich Dr.-Ing. Stephan Herrmann und Dipl.-Inform. Christine Hundt für ihre Hilfsbereitschaft und ihre Implementierungsarbeiten, die mir meine Implementierungsarbeiten erleichterten oder sogar erst ermöglichten.

Ich möchte meiner zukünftigen Frau und werdenden Mutter meines Kindes, Nadin Musick, danken. Ihr Verständnis, ihre Geduld und Ausdauer wirken wie ein Ruhepol. Durch ihre Unterstützung konnten viele Fehler in dieser Arbeit behoben werden.

Schließlich möchte ich mich noch bei meinen Eltern, Gabriele und Norbert Anklam, bedanken. Sie haben mein Studium erst möglich gemacht und waren meinen Plänen und Wünschen gegenüber immer offen.

Der Begriff "Bug" (engl. Wanze) wurde durch die Compilerbaupionierin Prof. Grace Hopper geprägt. Das Insekt, was durch die glühenden Kathodenstrahlröhren angelockt worden war, setzte ihren Mark2 Computer außer Betrieb. Seit diesem Vorfall werden in der Softwareentwicklung Fehler "Bugs" genannt.



Der wahrscheinlich erste Bug.

1 Einleitung

In der Softwareentwicklung gehört das Auffinden und Beheben von Fehlern zu den zeitraubensten Tätigkeiten. Das während des Programmierens Fehler entstehen, ist kaum zu verhindern. Um sie aber schneller aufspüren und ausbessern zu können, wurden Debugger entwickelt. Debugger sind Werkzeuge, die den Entwickler bei der Fehlersuche und beim Verstehen eines Programms unterstützen.

> "Debugging typically happens during three activities in software development, and the level of granularity of the analysis required for locating the defect differs in these three." [33]

Diese im Zitat erwähnten drei Tätigkeiten sind Coding, Testing und Production/Deployment. Das Debugging, also das Auffinden und Beheben von Fehlern mit einem Werkzeug, findet sich in wichtigen Bereichen innerhalb des Softwareentwicklungsprozesses wieder.

Diese Diplomarbeit befasst sich mit der Ausarbeitung von Anforderungen (Kapitel 3.1) und der Umsetzung eines Debuggers für Applikationen, die mit der aspektorientierten Sprache Object Teams/Java (kurz: OT/J) implementiert wurden. Das Konzept der Aspektorientierung ist eine Weiterentwicklung der objektorientierten Konzepts und erweitert die Modularisierbarkeit von Software um eine Dimension. Das Kapitel 3.4 wird das näher erläutern.

Object Teams ermöglicht die Modularisierung entlang dieser Dimension und bietet darüber hinaus ein kollaborationsbasiertes Rollenmodell. OT/J ist eine auf Java basierende Programmiersprache, die die Konzepte von Object Teams unterstützt (siehe Kapitel 5.1).

Es gibt verschiedene Ansätze sich dem Fehlerfinden und dem Beheben dieser Fehler zu nähern. In Kapitel 2 dieser Diplomarbeit wird erläutert, welche Ansätze es gibt und welche sich durchgesetzt haben. Kapitel 3 wird Grundanforderungen darstellen, die jeder Debugger erfüllen sollte und genau betrachten, welche Ansätze sich im Laufe der Paradigmenwechsel der Programmiersprachen als State of the Art herausgestellt haben und wie Debugger für prozedurale und objektorientierte Programmiersprachen diese Grundanforderungen erfüllt haben.

Ziel dieser Diplomarbeit ist die Entwicklung einer Debuggers für die aspektorientierte Sprache Object Teams/Java. Im Kapitel 4 wird das dafür notwendige Grundlagenwissen vermittelt und konzeptionell erklärt, was bei der Entwicklung des Debuggers zu beachten ist.

Kapitel 5 stellt zwei aspektorientierte Sprachen und deren Sprachfeatures konzeptionell und technisch vor und beschreibt die Einflüsse, die sich bei der Entwicklung eines Debuggers für Programme dieser Sprachen ergeben. Wie diesen Einflüssen entgegengewirkt werden kann, wird in diesem Kapitel konzeptionell erläutert.

Letztendlich werden diese konzeptionell ausgearbeiteten Lösungen exemplarisch für einen Debugger für die aspektorientierte Sprache Object Teams/Java implementiert. Dabei wird auf

den Eclipse JDT Debugger zurückgegriffen. Dieser wird angepasst und erweitert, damit das Werkzeug den Anforderungen gerecht wird. In Kapitel 6 wird das dokumentiert.

2 Debugging - was ist das?

In einem Glossar mit dem Thema Debugging von Computer Software [40], wird der Begriff wie folgt definiert:

> "debugging (dbg, debug): 1)n. the process of isolating and correcting mistakes in computer programms. 2) adj. having to do with debugging"

Diese Definition reicht nicht aus, denn darüber hinaus wird während des Debuggings das dynamische Verhalten eines Programms illuminiert. Durch Debugging kann das Verständnis für ein Programm genau so gut aufgebaut werden, wie damit Fehler des Programms aufgefunden werden können.[49]

Debugger sind die Werkzeuge, die den Entwickler bei dieser Arbeit unterstützen. Mit Hilfe von Debuggern kann ein Programm kontrolliert zur Ausführung gebracht werden. An einem gewünschten Punkt kann das Programm angehalten werden, um den zu diesem Zeitpunkt vorliegenden Programmzustand auf Korrektheit zu verifizieren.

Bevor es Debugger als Werkzeuge gab, wurden "print statements" eingesetzt, um Informationen über den Zustand eines Programms zu erfahren. Bei großen Softwareprojekten war diese Vorgehensweise nicht geeignet, da bei der Untersuchung eines Abschnitts im Programm, bei dem viele Module (Prozeduren, Methoden oder Klassen) beteiligt waren, eine Vielzahl "print statements" zusammen kamen, die schnell unübersichtlich werden konnten. Da nun viel Zeit für das Suchen und Untersuchen der Ausgabe verwendet werden musste, war dieser Ansatz für schnelle Fehlersuche nicht dienlich. Wie man Textausgaben strukturiert und besser aufbereitet ist Aufgabe des Loggings und wird hier nicht weiter betrachtet. Das Debugging mittels "print statements" war nicht skalierbar, da jede Fehlersuche und die damit notwendige Darstellung des Systemzustandes mit hohem Aufwand verbunden war.

Es existieren andere Ansätze sich der Fehlerfindung und den Programmverstehen zu nähern. Diese Ansätze werden in diesem Kapitel erläutert und deren Prinzipien erklärt.

2.1 Klassisches Debugging

Das Debugging im klassischen Sinn, besteht aus einen vordefinierten Ablauf. Dieser Ablauf ist in Abbildung 2.1 illustriert. Systematisches Vorgehen in Kombination mit geeigneter Werkzeugunterstüzung, meist interaktive Debugger, ergeben Klassisches Debugging. Dieses Kapitel stellt die genaue Vorgehensweise und das zur Verfügung stehende Werkzeug vor:

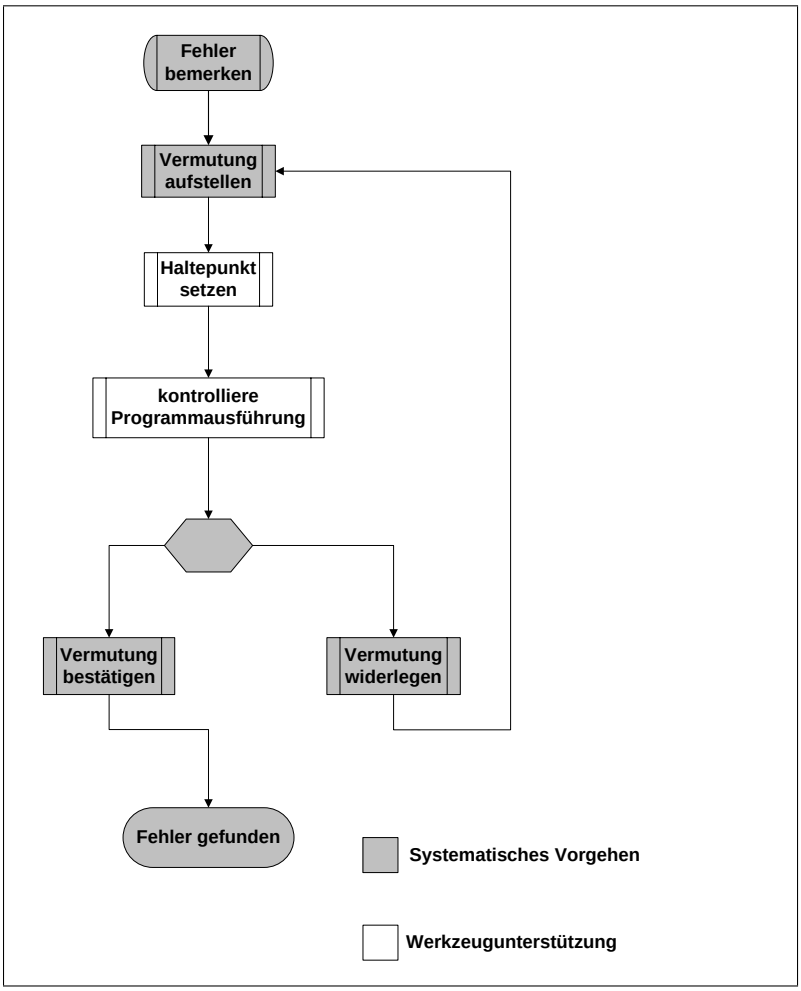


Abbildung 2.1: Ablauf beim Klassischen Debugging

2.1.1 Systematisches Vorgehen

Die Idee hinter dem systematischen Vorgehen während des Debuggings unterliegt dem Ansatz von Versuch und Irrtum. Der Systematische Debugging Prozess lässt sich genauer in drei Schritte unterteilen:

1. **Fehlermanifestation** - Die Beobachtbarkeit/Entdeckung eines Fehlers passiert im Schritt der *Fehlermanifestation*. Beobachtbar werden Fehler u.a. durch kontinuierliches Testen. Durchläuft ein Programm die Tests ohne Fehler, ist damit die Korrektheit des Programms

nicht automatisch gegeben. Beweisen lässt sich die Korrektheit nur durch mathematische Programmverifikationsverfahren, wie das Hoarekalkül.

2. **Fehlerlokalisierung** - Hat sich ein Fehler manifestiert, folgt das systematische Eingrenzen möglicher Fehlerursachen. Durch Aufstellung von Hypothesen und deren Überprüfung wird der Fehlerraum sukzessiv kleiner, bis die Fehlerursache gefunden wird (siehe Abbildung 2.2). Die Wahl der Hypothesen ist für das gezielte Eingrenzen der Fehlerursachen ausschlaggebend. Eine Hypothese ist effektiv, wenn nach deren Bestätigung wenige Fehlerursachen übrig bleiben oder wenn nach deren Widerlegung viele Fehlerursachen ausgeschlossen werden können. Um die Hypothesen zu überprüfen, muss der Programmzustand ersichtlich sein. Dafür eignen sich vor allem interaktive Debugger.

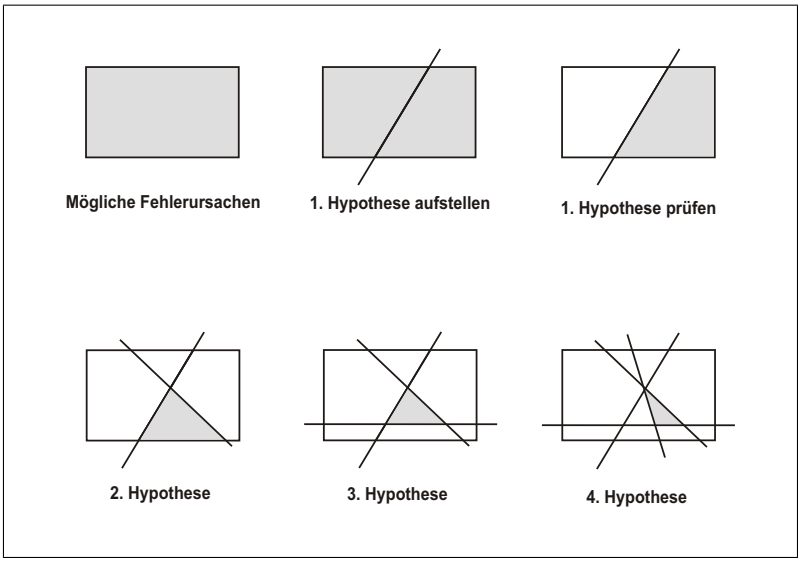


Abbildung 2.2: Systematisches Eingrenzen möglicher Fehlerursachen

3. **Fehlerkorrektur** - Nachdem der Fehler lokalisiert wurde, kann er korrigiert werden. Das kann das Ändern einer Variablen bedeuten oder den Umbau eines ganzen Systems.

Diese wissenschaftliche Methode, Fehler zu finden bedient sich verschiedenster Debuggertechnologien. Der Interaktive Debugger ist hierbei das gängigste Werkzeug, da es mit dessen Hilfe ohne größeren Aufwand möglich ist, aufgestellte Hypothesen zu überprüfen.

2.1.2 Source Level Debugger

In der Softwareentwicklung werden vorwiegend Source Level Debugger eingesetzt. Programme werden in Hochsprachen formuliert und werden danach in Maschinensprache transformiert. Dieser Maschinencode hat mit dem Quellcode, den der Entwickler verfasst hat, kaum noch Gemeinsamkeiten. Damit ein Entwickler Programme in Maschinensprache nachvollziehen kann, wurden Source Level Debugger entwickelt. Sie schaffen für den Benutzer die Illusion, dass das Programm in seiner Hochsprache ausgeführt wird, indem der Debugger den ursprünglichen Quellcode und nicht den Maschinencode anzeigt. Dafür muss der Debugger in der Lage

sein, die Transformation zwischen Quellcode und Maschinencode bidirektional nachzuvollziehen. Quellcodebefehle müssen auf Maschinencodebefehle abbildbar sein, um die Ausführung des *Debuggee*[1] überhaupt kontrollieren zu können. Die Namen der Variablen müssen aus der Hochsprache entnommen werden können und auf den entsprechenden Speicherbereich verweisen. Nur dadurch kann der Inhalt von Variablen inspiziert und verändert werden. Durch Optimierungen des Compilers können überflüssige Befehle und nicht verwendete Variablen eliminiert werden, die das Nachvollziehen der Codetransformation erschweren. Sämtliche Transformationen des Codes können aber durch Tabellen nachvollzogen werden, die der Compiler für den Debugger zusätzlich zur Verfügung stellt. Diese Tabellen enthalten Einträge für alle Befehle und Variablen des Programms mit Verweisen auf die entsprechende Adresse im Speicher.

Der Debugger befindet sich als Bindeglied zwischen dem Benutzer und dem ausgeführten Programm. Seine Aufgabe ist es also, zwischen den Funktionen, die das Betriebssystem dem Debugger bereitstellt, und den Funktionen und Ansichten, die der Debugger dem Benutzer bietet, zu vermitteln.

Interaktive Debugger sind Source Level Debugger, die diese Aufgabe erfüllen:

2.1.3 Interaktive Debugger

Interaktive Debugger sind Universalwerkzeuge, um das Programm in einer definierten Umgebung auszuführen und um es unter bestimmten Bedingungen anhalten zu lassen. Bei einem Halt kann der Zustand des Programms überprüft werden. Es wird erkennbar welche Belegungen die Variablen haben und es wird verständlich wie der Programmfluss des ausgeführten Programm ist. Interaktive Debugger folgen dabei immer zwei Grundprinzipien [49]:

1. **Heisenbergprinzip** - Um ein System untersuchen zu können, muss man in den internen Ablauf mehr oder weniger stark eingreifen. Dabei kann es zur Verfälschung der Messergebnisse kommen. Nach einem der Begründer der Quantenmechanik, dem Physiker Werner Heisenberg, wird das Auftreten dieser Eigenschaft Heisenberg-Prinzip genannt. Für den Bereich der Softwaretechnik, speziell den Bereich des Debuggings bedeutet das, dass ein Debugger das Laufzeitverhalten eines Programms in keiner Weise verändern darf. Wenn ein Programm ohne Debugger ausgeführt wird, also keine Untersuchung stattfindet, muss es sich so verhalten, als wenn ein Programm im Debugmodus ausgeführt werden würde und anders herum. Das erscheint schwierig, wenn man bedenkt, dass der Maschinencode mit Debuginformationen angereichert wird, damit Debugging überhaupt möglich wird. Wenn der Debugger aber Auswirkungen auf das Programm hat, kann es sein, dass Fehler nur in einer der beiden Ausführungsarten auftreten. Deshalb sind die Einflüsse des Debuggers auf das Programm so gering wie möglich zu halten.[31]

2. **Wahrheitsprinzip** - Das zweite Prinzip, was bei der Entwicklung von Debuggern unabdingbar ist, ist das Wahrheitsprinzip. Hierbei sollen alle Informationen, die dem Benutzer präsentiert werden, richtig sein, also der Wahrheit entsprechen. Wenn ein Debugger dem Benutzer falsche Informationen liefert, ist die Benutzung eines Debuggers sinnlos.

Ein interaktiver Debugger ist ein Bestandteil fast jeder modernen Entwicklungsumgebung. Eine Entwicklungsumgebung (IDE) vereint alle Werkzeuge, die ein Entwickler zur Erstellung eines

[1] Ein Debuggee ist das durch einen Debugger kontrollierte, ausgeführte Programm.

Programms benötigt. Eine IDE stellt oft sogar Werkzeuge für Design- *und* Implementierungsphase zur Verfügung und der typische Zyklus des Editierens, Kompilierens und **Debuggings** wird sehr vereinfacht.

2.1.3.1 Benutzerschnittstelle

Ein interaktiver Debugger bietet verschiedene Kontrollmöglichkeiten, in welcher Weise ein Programm ausgeführt werden kann. Um den Programmstatus zu überprüfen, stehen dem Benutzer einige Sichten auf das Programm zur Verfügung. Die Benutzerschnittstellen der interaktiven Debugger variieren zwar, beinhalten aber im Allgemeinen folgende Sichten und Kontrollmöglichkeiten eines Programms.

Sichten

Die verschiedenen Sichten stellen immer einen bestimmten Bereich des Debuggee in den Vordergrund. So kann der Benutzer detaillierte Informationen über den Bereich des ausgeführten Programms erfahren, der ihn gerade interessiert.

Quellcodesicht - Für den Benutzer eines interaktiven Debuggers ist der Quellcode die wichtigste Ansicht auf das ausgeführte Programm. Der Entwickler findet seinen eigenen, gut vertrauten Quellcode vor. Die Quellcodesicht ist die Hauptansicht bei schrittweisen Debugging und viele Informationen werden hier abgebildet. Zu diesen Informationen gehört zum Beispiel in welcher Zeile sich ein Haltepunkt befindet und an welcher Stelle das Programm gerade hält.

Stack Trace-/Stack Frame-Sicht - Diese Sicht zeigt die Position des Programms in der Hierarchie der Funktionen und Unterfunktionen an.

Variablensicht - Die Variablensicht stellt die Werte von Variablen und Strukturen dar, auf die an diesem Ausführungspunkt Zugriff besteht.

Breakpointsicht - In der Breakpointsicht werden alle Haltepunkte aufgelistet, die gesetzt worden sind. Es ist ersichtlich, um was für eine Art von Haltepunkt es sich handelt und an welche Stelle er gesetzt worden ist. Außerdem soll der Benutzer aktive Haltepunkte selektieren und löschen , sowie neue Haltepunkte setzen können.

CPU-Sicht - Die CPU-Sicht zeigt die Registerwerte der CPU.

Evaluatorsicht/Expressionsicht - In dieser Sicht kann während des Debuggens noch Code in das laufende Programm einfügt werden. Dieser Code wird ausgewertet und die Ausgabe des Codes wird in dieser Sicht ebenfalls dargestellt. Dieses Feature ist sehr umstritten, da es möglich wird, das Programm so zu verändern, das es nichts mehr mit dem Ursprünglichen zu tun hat.

Kontrollmöglichkeiten

Um Kontrolle über die Ausführung des Debuggee zu erlangen, stellt ein interaktiver Debugger Funktionen zur Verfügung, die das Programm starten, stoppen, unterbrechen und fortsetzen lassen. Außerdem kann mit Haltepunkten und verschiedensten Schrittmodi Kontrolle erlangt werden.

1. **Schrittmodus** - Um ein Programm kontrolliert auszuführen, muss man es schrittweise ablaufen lassen können. Nach jedem Schritt kann der Programmstatus durch die verschiedenen Sichten in Erfahrung gebracht werden. Die Granularität der Ausführungsschritte wird durch den Schrittmodus angegeben, wobei ein interaktiver Debugger meist folgende Schrittmodi unterstützt:

   Einzelschritt-Modus - Der Einzelschritt-Modus erlaubt dem Debugger den Prozessor so zu kontrollieren, dass entweder nach der Ausführung eines Maschinenbefehls oder eines Quellcodebefehls das Programm gestoppt wird. Es gibt zwei verschiedene Einzelschrittarten(engl. single step modi):

   Step Into - Mit diesem Befehl springt man in die Unterfunktion/Methode, die gerade ausgeführt werden sollte. Es wird zur ersten Instruktion dieser Methode gesprungen und das Programm wird angehalten.

   Step Over/Next - Die Zeile auf der sich der Ausführungspunkt gerade befindet, wird ausgeführt und danach wird zur nächsten Zeile der gerade ausgeführten Methode gesprungen. Die Programmausführung stoppt dort.

   Step Out/Step Return - Mit Step Out/Step Return wird aus der aktuellen Methode gesprungen und an der Zeile gehalten, die danach ausgeführt werden würde. Dieser Schrittmodus gehört aber nicht zum Einzelschritt-Modus.

2. **Haltepunkte** - Die Ausführung eines Programms kann an bestimmten Stellen durch Haltepunkte unterbrochen werden. An jedem Haltepunkt kann der Programmstatus überprüft werden. Dabei werden verschiedene Arten von Haltepunkten unterschieden:

   Zeilenhaltepunkt (engl. Linebreakpoint) - Meist kann durch einfache Selektion einer Zeile in der Quellcodesicht und dem Befehl *Set Breakpoint* ein Haltepunkt an diese Zeile gesetzt werden. Das Programm hält, wenn diese Stelle im Code erreicht wird. Es ist möglich den Haltepunkt mit Bedingungen zu versehen. Nur wenn die Bedingungen erfüllt sind, ist der Haltepunkt aktiv. Zu diesen Bedingungen zählen die Codebedingung und die Zählbedingung. Bei der Codebedingung übergibt man dem Haltepunkt eine Bedingung in der Syntax der Programmiersprache. Die Codebedingung wird jedes Mal ausgewertet, wenn der Haltepunkt erreicht wird. Der Haltepunkt ist aktiv, wenn die Codebedingung als *wahr* evaluiert wird. Man spricht dabei auch von einem bedingten Haltepunkt. Die Zählbedingung (auch Ignore Count/Hit Count) lässt einen Haltepunkt nur aktiv werden, wenn er eine gewisse Anzahl passiert wurde. Diese Anzahl wird dem Haltepunkt als Bedingung übergeben.

   Überwachungspunkt (engl. Watchpoint) - Überwachungspunkte überwachen den Status einer selektierten Variable. Wenn der Wert der Variablen verändert wird, oder wenn lesend oder schreibend auf die Variable zugegriffen wird, hält das Programm an. Ein Überwachungspunkt kann aktiviert und deaktiviert werden. Außerdem ist es möglich dem Überwachungspunkt die gleichen Bedingungen zu übergeben, wie dem Zeilenhaltepunkt.

2.2 Erweiterte Debuggingtechniken

Nachdem nun der interaktive Debugger mit seinen Basisfunktionalitäten erläutert wurde, werden jetzt erweiterte Debuggingtechniken vorgestellt. Zum Einen erstreckt sich die Erweiterung

auf den interaktiven Debugger um die Möglichkeit, den Debuggee rückwärts auszuführen. Zum Anderen soll durch automatisches Fehlereingrenzen, die Interaktion zwischen Benutzer und Debugger verringert und somit Zeit gespart werden. Dieses Kapitel stellt diese erweiterten Debuggingtechniken näher vor.

2.2.1 Bidirektionales Debugging

Bei der Entwicklung eines interaktives Debugger steht die Frage "Welche Informationen kann man dem Benutzer zur Verfügung stellen, während das Programm läuft?" im Mittelpunkt. Bei der Entwicklung eines bidirektionalen Debuggers liegt der Fokus auf der Frage "Welche Informationen helfen dem Benutzer am meisten?". [44]

Ein bidirektionaler Debugger protokolliert wichtige Ereignisse während der Programmausführung. Nach Beendigung eines Programms, stehen dem Benutzer alle Sichten des Debuggers zur Verfügung, um den Status des Programms zu validieren. Der Benutzer hat aber nun mehr Kontrollmöglichkeiten. Ein Einzelschritt kann vorwärts und rückwärts (engl. step backward) ausgeführt werden. Das Auftreten eines Fehlers und das Finden des Ursprungs kann durch "step backward" in einem Ablauf erfolgen. Das ist bei schwer reproduzierbaren Fehlern besonders vorteilhaft. Betrachtet der Benutzer Code, der ihm vorher unbekannt war, so kann mittels bidirektionalem Debugging schnell Verständnis für den Programmablauf geschaffen werden, da unter anderem sehr einfach ersichtlich ist, wo im Programmfluss welche Variablen verändert wurden.

Ein bidirektionaler Debugger ist ein interaktiver Debugger, der um die Fähigkeit der "Rückwärts" Debuggings erweitert wurde. Für die Realisierung eines bidirektionalen Debuggers existieren verschiedene Ansätze:

Logging - Beim Logging werden alle Daten während der Programmausführung gespeichert, die für die Umkehrung der einzelnen Instruktionen ("step backward") benötigt werden. Diese Daten beschreiben alle lesenden und schreibenden Zugriffe auf Variablen und alle Methodenaufrufe. Damit ist gewährleistet, dass wirklich jede Instruktion umkehrbar ist. Damit ist auch das Problem bei dieser Art der Realisierung erkennbar - ein immens hoher Speicherbedarf.

Checkpointing - Wird ein bidirektionaler Debugger durch Checkpointing realisiert, dann bedeutet das, dass beim Erreichen von definierten Überwachungspunkten (engl. checkpoints) während des Programmablaufs regelmäßig ein Abbild des momentanen Programmzustandes gespeichert wird. Das "step backward" bedeutet hierbei, dass der Programmzustand des letzten passierten Überwachungspunkts wiederhergestellt wird, und ab da der Debuggee in Vorwärtsrichtung ausgeführt wird, bis die geforderte Instruktion erreicht wird.

Reverse Instruction - Der Ansatz bei Reverse Instruction ist das Rückgängig machen jeder einzelnen Instruktion. Aus jeder Instruktion wird eine direkt umkehrbare Instruktion erstellt. Bei einem "step backward" wird dann die umgekehrte Instruktion ausgeführt. Dieser Ansatz ist sehr speicherintensiv und das "step backward" ist deshalb auch durch den Speicher begrenzt.

Durch bidirektionales Debugging kann eine Vielzahl von Fehlern schnell gefunden werden, da die systematische Fehlersuche (siehe Abbildung 2.2) erleichtert wird. Der Benutzer kann nun direkt

zum Ursprung des Fehlers gelangen, ohne fortwährend neue Hypothesen aufstellen und validieren zu müssen. Das schrittweise Ausführen des Debuggee in umgekehrter Programmausführung ist aber durch den hohen Speicherverbrauch des bidirektionalen Debuggers begrenzt. Diese Begrenzung ist das größte Problem bei der Umsetzung eines bidirektionalen Debuggers.

2.2.2 Automatisiertes Debugging

Im Fokus des Automatisierten Debuggings steht das Eingrenzen der Fehlerursachen ohne Benutzerinteraktion. Bisher wurde Techniken vorgestellt, die dem Benutzer bei der Validierung seiner aufgestellte Thesen (siehe Abbildung 2.2) unterstützen. Automatisiertes Debugging nutzt verschiedene Techniken, um den Fehler eines Programms automatisch zu lokalisieren. Durch Analyse des Programmcodes können fehlerhafte Teile des Programms identifiziert werden. Das hilft dem Benutzer beim Aufstellen effektiver Fehlerthesen. Diese Techniken wurden bislang nur unter Laborbedingungen geprüft; es ist unsicher, ob die Verfahren auch auf große Programme anwendbar sind.[19]

Program Slicing - Mittels *Program Slicing* wird versucht, die Fehlersuche auf den Teil des Quellcodes zu beschränken, der den Fehler verursacht. Durch die automatische Berechnung relevanter Teile soll die Fehlersuche und das Programmverstehen beschleunigt werden. Dazu werden Methoden der Programmanalyse verwendet, wobei die Analyse statisch oder dynamisch erfolgen kann. Bei statischer Analyse (statisches Slicing) wird nur der Programmquelltext verwendet, ohne mögliche Eingabewerte zu betrachten. Bei dynamischer Analyse (dynamisches Slicing) werden Eingabewerte als bekannt vorausgesetzt, wobei somit die Menge möglicher Ausführungspfade reduziert wird. Wird beim Debugging eine fehlerhafte Variable erkannt, werden durch Slicing nur die Programmteile "herausgeschnitten", die diese Variable beeinflussen. Auf diesen Slice (Ausschnitt eines Programms) können dann die Fehlerthesen validiert werden, wobei der Rest des Programms nicht betrachtet werden muss. Der zu untersuchende Programmausschnitt kann noch durch Subtraktion eines Slices von einem anderen (Programm Dicing) verkleinert werden. Für das obige Beispiel bedeutet das, dass aus dem Programmausschnitt (Slice) der inkorrekten Variable, die Anweisungen eines zweiten Slice entfernt werden, wobei der zweite Slice ein Programmausschnitt für eine korrekte Variable sein muss. Der Fehlerraum wird durch diese gebildete Mengendifferenz weiter beschränkt.

Algorithmisches Debugging - Algorithmisches Debugging versucht, einen Fehler im ausgeführten Programm durch Interaktion mit dem Benutzer zu lokalisieren. Dabei kann durch den Benutzer spezifiziert werden, ob bestimmte Programmeinheiten, wie ein Berechnungsschritt oder eine Prozedur, korrekt ausgeführt worden sind. Das algorithmische Debugging wurde initial für Prolog-Programme realisiert und arbeitete auf einem Berechnungsbaum, der als Darstellung des Programmablaufs gewählt wurde. Der Programmablauf wird dabei von den Antworten auf die Fragen bezüglich der Korrektheit der Ausführung einzelner Prozeduren gesteuert. Die Anzahl der Fragen, die gestellt werden müssen, kann durch Slicing reduziert werden. Dem Benutzer kann das Beantworten der Fragen teilweise abgenommen werden, indem Testdaten und Zusicherungen (Assertions[2]) ausgewertet werden.

[2] Eine Zusicherung oder Assertion (lat./engl. für Aussage, Behauptung) ist eine Aussage über die Variablen eines Computer-Programms.

Delta Debugging - Bei *Delta Debugging* wird systematisches Testen als Debugging-Technik eingesetzt. Regressionstests stellen dabei die Basis dar. Die Testdaten der Regressionstests, also der Programmquelltext, werden minimal variiert und danach werden die Tests ausgeführt. Das Delta (Differenz) zwischen fehlschlagenden und erfolgreichen Tests der beiden Programmversionen weist den Weg zur Fehlerursache.

Weitere Debuggingtechniken wie *Relative Debugging* [22] und *Modell-basiertes Debugging* [20] sollen hiermit Erwähnung finden, werden aber in dieser Diplomarbeit nicht weiter erläutert.

Für die hier vorgestellten Debuggingtechniken existieren oft auch Referenzimplementierungen, die sich aber nicht als State of the Art durchsetzen konnten. Was sich genau durchsetzen konnte beleuchtet das nächste Kapitel.

3 State of the Art of Debugging

In diesem Kapitel werden Grundanforderungen definiert, die jeder interaktive Debugger, unabhängig von der Programmiersprache, erfüllen sollte. Jedes Programmierparadigma hat Einfluss auf die Umsetzbarkeit der definierten Grundanforderungen. Im Laufe der Paradigmenwechsel mussten auch die Grundanforderungen überprüft und teilweise neu erfüllt werden. Dementsprechend sehen auch die Werkzeuge aus. Inhalt dieses Kapitel ist die Untersuchung verschiedener interaktiver Debugger und die Darstellung, was State of the Art beim Debugging von strukturierten prozeduralen Programmen (SPP) und von objektorientieren Programmen (OOP) ist. Dabei steht die Umsetzung der allgemein gültigen Grundanforderungen im Mittelpunkt. Das aspektorientierte Paradigma (AOP) ist noch vergleichsweise jung und so lässt sich noch nicht mit Sicherheit sagen, was einen Debugger für aspektorientierte Programme auszeichnet.

3.1 Grundanforderungen an einen interaktiven Debugger

Debugger sind Hilfswerkzeuge, die Funktionen zur Überprüfung und Kontrolle der Programmausführung und des Programmzustandes zur Verfügung stellen. Die Benutzbarkeit eines Debuggers hängt dabei von einigen Faktoren ab. Zum Einen sollte die Aufmerksamkeit des Benutzers auf das Verhalten des Programms und nicht auf das Verhalten des Debuggers gelenkt werden, und zum Anderen sollte der Debugger *intuitiv* benutzbar sein. Im folgenden werden diese und weitere Faktoren, die in dieser Arbeit als **Grundanforderungen an einen interaktiven Debugger** bezeichnet werden, genauer erläutert. Jeder interaktive Debugger, unabhängig für welche Sprache konzipiert, sollte diese Anforderungen erfüllen.

Quellcodedarstellung - Ein Debugger muss eine Verbindung zwischen ausgeführten Maschinen- und angezeigten Quellcode herstellen. Das Setzen von Haltepunkten geschieht üblicherweise im Quellcode. Wird das Programm an dem gesetzten Haltepunkt angehalten, so muss auch der Quellcode im Editor dargestellt werden, der gerade ausgeführt wird.

Kontrolle durch :

Haltepunkte - Ein Debugger muss es ermöglichen Haltepunkte zu setzen. Mindestens ein Haltepunkttyp, der Zeilenhaltepunkt, muss vom Debugger unterstützt werden.

Schrittmodus - Der Benutzer muss festlegen können, wann das Programm halten soll. Ein Debugger unterstützt z.B. mehrere Schrittmodi[1] , zwischen den ein Benutzer wählen kann. Die Wahl des Modus legt fest, in welcher Weise das Programm weiter

[1] Erklärung der Schrittmodi in Kapitel 2.1.3.1

ausgeführt wird, wann und wo es wieder anhalten soll. Ohne Schrittmodi wäre keine kontrollierte Programmausführung möglich.

Visualisierung - Ein Debugger muss Visualisierungen besitzen, die Informationen über den Programmzustand darstellen[2]. Verschiedenste Sichten auf ein Debuggee sollten unterstützt werden. Dabei kann auch auf Methoden der Softwarevisualisierung zurückgegriffen werden, die die Fehlersuche unterstützen und den Programmverlauf veranschaulichen.

Programmmanipulation - Der Debugger muss dem Benutzer die Möglichkeit zu Verfügung stellen, das Programm vor und während dessen Ausführung zu verändern. Das bedeutet die Manipulation des Zustandes und die Manipulation des Programmflusses.

Mit der Weiterentwicklung von Programmierparadigmen muss jedes Mal auch das Debugging auf den Prüfstand gestellt werden. Ein Debugger für objektorientierte Programme setzt die zu erfüllenden Anforderungen anders um, als ein Debugger für prozedurale Programme. In der Vergangenheit änderte sich also die Umsetzung der zu erfüllenden Anforderungen, als Werkzeuge für das damals neue Programmierparadigma Objektorientierung entstanden. Das Gleiche muss nun für Werkzeuge für aspektorientierte Sprachen geschehen.

3.2 Prozedurales Paradigma

Das prozedurale Paradigma liegt dem imperativen Programmiermodell zu Grunde. Dieses Programmiermodell beruht auf Befehlen wie Wertzuweisungen und Verzweigungen. Es können arithmetische und logische Ausdrücke ausgewertet werden. Imperative Programmiersprachen zeichnen sich durch eine enge Anlehnung an die *von Neumann-Rechnerarchitektur* aus, die auf der Idee eines Speichers mit Daten und Instruktionen, einer Steuer- und einer Verarbeitungseinheit basiert. Die Anlehnung beruht vor allem auf folgenden Merkmalen:

Ablauforientierung - Instruktionen (Befehle, Anweisungen) werden sequentiell und schrittweise ausgeführt.

Speicher-Wert-Zuordnung - Speicherbereiche sind über Namen ansprechbar und ihnen können Werte zugewiesen werden. Diese Werte können dann modifiziert und abgerufen werden. Eine Variable repräsentiert den Inhalt einer Speicherzelle.

Dieses imperative Konzept findet sich im prozeduralen, objektorientierten und aspektorientierten Paradigma wieder.

concern - In der Softwareentwicklung sind *concerns* "Dinge von Interesse", die in einem Software-System modelliert werden sollen. Um wartbare und wiederverwendbare Software zu schreiben, ist es unabdingbar von der Software zu leistende Anforderungen, zu identifizieren und zu kapseln. Der Begriff *concern* wurde in einem Buch [24] von Edsger W. Dijkstra erstmals erwähnt.

separation of concerns - Eines der wichtigsten Prinzipien der Softwareentwicklung ist das Prinzip von *Separation of Concerns*. Es besagt, dass concerns durch die Programmiersprache in expliziter und/oder deklarativer Weise dargestellt und örtlich gebunden werden sollten.[3]

[2] Die Visualisierungen sind u.a. Sichten, wie sie in Kapitel 2.1.3.1 erklärt wurden.

[3] The principle ... also states that important issues should be represented in programs intentionally (explicitly, declaratively) and well localized. Zitiert nach [21]

Separation of Concerns wird bei prozeduralem Programmierstil erreicht, indem wiederkehrende Teilalgorithmen als Unterprogramme/Prozeduren definiert werden, die durch elementare Anweisungen aufgerufen werden können. Das prozedurale Konzept ist die Strukturierung von Programmen in Unterprogramme. Diese können sich selbst aufrufen; rekursive Algorithmen sind also möglich. Abstrakt betrachtet werden größere Aufgaben in Kleinere auf gespaltet. Damit ist Quellcode wiederverwendbar, universeller und einfacher zu schreiben, da die zu lösenden Probleme verkleinert werden.

Sprachen wie C, PASCAL, BASIC und FORTRAN folgen dem prozeduralen Ansatz, beinhalten also das imperative und prozedurale Konzept.

3.2.1 Ein Debugger für prozedurale Programme

Der bekannteste graphische Debugger für prozedurale Programme ist der Data Display Debugger (DDD). Er benutzt Kommandozeilendebugger wie GDB (GNU Debugger), DBX, und JDB (Java Debugger). Der DDD unterstützt die Programmiersprachen, die die jeweiligen Kommandozeilendebugger unterstützen. Der Data Display Debugger wird meistens in Kombination mit dem GNU Debugger verwendet und unterstützt demnach Sprachen wie C, Pascal und Objective-C. Aber auch die objektorientierte Sprache C++ wird unterstützt. Der DDD wird als State of the Art Debugger für prozedurale Sprachen herangezogen.

Quellcodedarstellung - Das Finden der Quellcodedatei für das gerade ausgeführte Programm stellt kein Problem dar. Ein Compiler bietet oft auch die Möglichkeit der Optimierung des Maschinencodes. Wenn der Compiler Maschinencode erstellt, der im Debugmodus ausgeführt werden soll, wird auf Optimierungen verzichtet, damit der Quelltext dem Kompilat entspricht. Somit lässt sich jede Quellcodezeile einfach auf Maschinencode abbilden. Die Darstellung in der Quellcodesicht ist somit unproblematisch.

Kontrolle durch Schrittmodi - Während des Debuggings kann der Benutzer die Ausführung des Programms kontrollieren. Dabei werden alle in Kapitel 2.1.3.1 erläuterten Schrittmodi und auch Weitere unterstützt:

Step - Mit dem *Step* Befehl läuft die Programmausführung für eine Programmzeile weiter. Dabei werden Unterprogrammzeilen mit angezeigt. Dieser Befehl entspricht also dem erläuterten *Step Into* Befehl.

Next - Der *Next* Befehl entspricht dem *Step Over* Befehl. Das Programm läuft also eine Programmzeile weiter, wobei nicht in Unterprogrammaufrufe gesprungen wird.

Interrupt - *Interrupt* unterbricht Programme auch ohne Haltepunkt (z.B. Endlosschleifen).

Cont - Der *Cont* Befehl startet das Programm ab der aktuellen Zeile.

Up - *Up* entspricht dem *Step Return*. Damit springt man also aus Unterprogrammen zum voherigen Unterprogrammaufruf.

Down - Mit *Down* springt man vom Unterprogrammaufruf zum Unterprogramm.

Stepi/Nexti - *Step/Next* setzt Programmausführung auf Maschinencodeebene und nicht auf Quellcodeebene fort. Dieser Schrittmodus wurde bis jetzt noch nicht erwähnt und erweitert die Kontrollmöglichkeit des Programms auf Maschinencodeebene.

Undo Execution - Durch *Undo Execution* können vergangene Werte in die Datadisplaysicht(Abbildung 3.2) zurückgeholt werden, d.h. der Displayzustand von vergangenen Haltepunkten wird wiederhergestellt. Es wird zu einer früheren Position gesprungen und Variablen und Plots bekommen ihre früheren Werte. Der Programmzustand wird dabei nicht verändert. DDD zeigt nur einen früheren Programmstatus, der durch den DDD aufgezeichnet wurde. Variablen können also nicht verändert oder andere Dinge modifiziert werden. Undo Execution darf nicht mit bidirektionalen Debugging verwechselt werden (siehe Kapitel 2.2.1).

Kontrolle durch Haltepunkte - In Kapitel 2.1.3.1 wurden einige Haltepunkttypen erläutert. Diese und weitere Haltepunkttypen werden unterstützt:

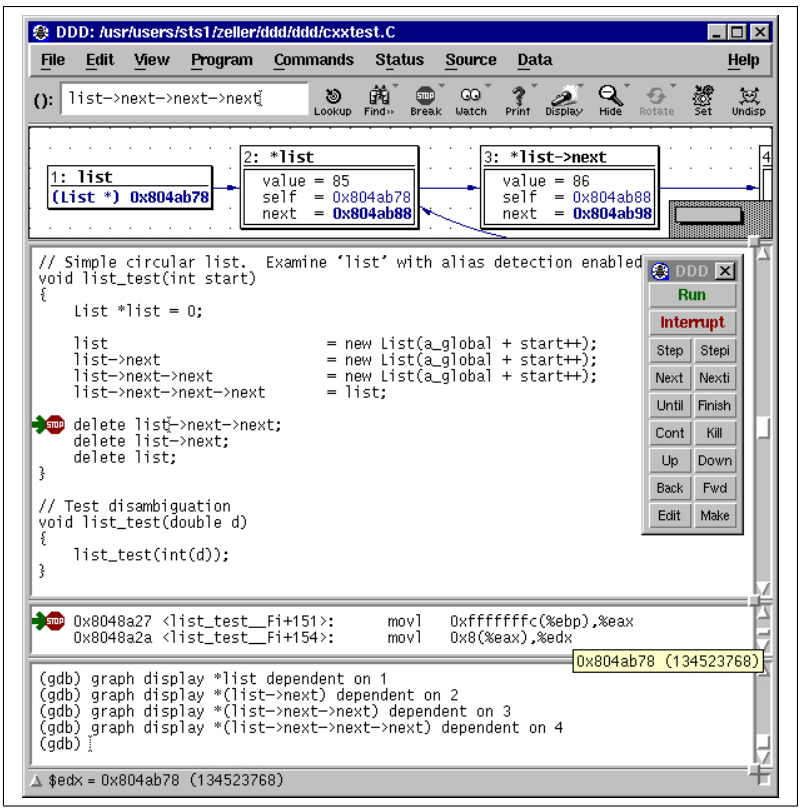


Abbildung 3.1: Data Display Debugger, Hauptfenster mit Funktionsübersicht

Zeilenhaltepunkt - Ein Zeilenhaltepunkt kann mit den in Kapitel 2.1.3.1 erläuterten Bedingungen versehen werden. Der DDD unterstützt weitere Befehle, die beim Passieren eines Haltepunktes ausgeführt werden können. Solche Kommandos können zum Beispiel das Ausgeben eines Wertes einer bestimmen Expression[4] oder das Setzen eines anderen Haltepunktes sein.

Temporärer Haltepunkt - Ein temporärer Haltepunkt ist ein Haltepunkt, der sofort gelöscht wird, wenn er erreicht wurde. Das Kommando *Run Until Here*, des Popup-Menüs in der Quellcodesicht setzt an der selektierten Stelle einen temporären Hal-

[4] Expression (dt. Ausdruck) bedeutet hier ein Stück ausführbarer Quelltext.

tepunkt und das Programm wird bis zu dieser Stelle ausgeführt. Diese Funktion wird nicht vom JDB unterstützt.

Continue Haltepunkt - Continue Haltepunkte sind Haltepunkte, die so konfiguriert sind, dass der DDD bei deren Erreichen das Kommando *cont* ausführt. Das bedeutet, dass der Programmlauf unmittelbar fortgesetzt wird, nachdem alle Displays und Plotfenster aktualisiert worden sind.

Überwachungspunkt - Überwachungspunkte werden unterstützt und können mit den in Kapitel 2.1.3.1 erläuterten Bedingungen versehen werden.

Visualisierung - Der DDD besitzt fünf verschiedene Sichten auf ein Programm. Im Source Window (Quellcodesicht) wird der gerade ausgeführte Quellcode angezeigt. Das Data Window (Variablensicht) ist das Fenster für Variablen und Strukturen. Im Machine Code Window wird der Quellcode auf Maschinensprachebene angezeigt. Das Execution Window ist das Fenster für das eigene textorientierte Programm (Aus-/Eingabe). Eine Besonderheit ist die Visualisierungmöglichkeit von Daten in Form von Plots, die mit dem Werkzeug *Gnuplot*[5] erstellt werden (siehe Abbildung 3.2 auf Seite 17). Große Datenmengen können durch Plots sehr gut dargestellt werden. Datentypen wie numerische Werte (int, float, double...), Arrays (ein- oder zweidimensional) und Historien (Werteverläufe über die Programmausführung) werden unterstützt. Plot-Windows werden bei jedem Haltepunkt aktualisiert. Somit ergibt sich unter Verwendung von Continue Haltepunkten die Möglichkeit zur Erstellung von Animationen. Hierzu werden die durch GDB-Kommandos extrahierten Daten an Gnuplot geschickt und die von Gnuplot zurückgelieferten Plot-Kommandos in einem eigenen DDD-Fenster dargestellt.

Programmmanipulation - Mit Hilfe eines externen Editors, der aus dem DDD heraus gestartet werden muss, kann der gerade ausgeführte Code bearbeitet werden. Nachdem das Editieren beendet ist, wird der Debuggee rekompiliert und Quellcodesicht wird aktualisiert.

Die Benutzerschnittstelle des Data Display Debugger nutzt alle Funktionalitäten des Backenddebuggers (GDB) und bietet darüber hinaus eigene Visualisierungen an. Der Data Display Debugger (+ GNU Debugger) erfüllt somit alle an einen Debugger gestellten Grundanforderungen.

3.3 Objektorientiertes Paradigma

Im Gegensatz zu SPP werden bei OOP Objekte und nicht Algorithmen als fundamentale Bausteine benutzt. Objekte haben drei charakteristische Eigenschaften:

1. Eine einzigartige Identität.
2. Einen Zustand, also eine Belegung der internen Attribute[6].
3. Ein Verhalten, also erlaubte Operationen, Funktionen oder Methoden.

Objektorientierung basiert dabei auf vier wichtigen Konzepten:

[5] Gnuplot ist ein skript- bzw. kommandozeilengesteuertes Computerprogramm zur grafischen Darstellung von Funktionen und Daten. Gnuplot kann sowohl Kurven (x/y-Datenpaare) als auch 3D-Objekte (Flächen) abbilden.

[6] Ein Attribut ist die Zuordnung von einem Merkmal, wobei sich die Merkmale auch ändern können.

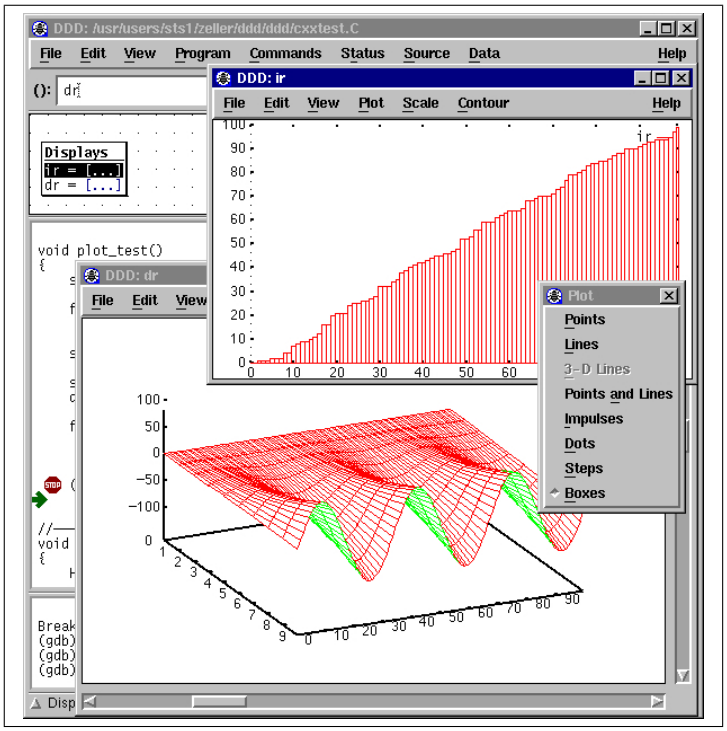


Abbildung 3.2: Data Display Debugger, Plot Ansichtsfenster

Kapselung - Objekte können den internen Zustand anderer Objekte nicht in unerwarteter Weise lesen oder ändern. Ein Objekt hat eine Schnittstelle, die darüber bestimmt, auf welche Weise mit dem Objekt interagiert werden kann.

Abstraktion - Jedes Objekt im System kann als ein abstraktes Modell eines Akteurs betrachtet werden, der Aufträge erledigen, seinen Zustand berichten, ändern und mit den anderen Objekten im System kommunizieren kann, ohne offen legen zu müssen, wie diese Fähigkeiten implementiert sind.

Vererbung - Neue Arten von Objekten können auf der Basis bereits vorhandener Objektdefinitionen festgelegt werden. Es können neue Bestandteile hinzugenommen werden oder vorhandene überlagert werden.

Polymorphie - Verschiedene Objekte in einer Vererbungshierarchie können auf die gleiche Nachricht unterschiedlich reagieren. Wird die Zuordnung einer Nachricht zur Reaktion auf die Nachricht erst zur Laufzeit aufgelöst, dann wird dies auch späte Bindung genannt.

Objekte kommunizieren miteinander, indem sie Nachrichten mittels eines definierten Protokolls untereinander verschicken. Das Konzept der Klasse wird benutzt, um die Struktur und das Verhalten für eine Menge von ähnlichen Objekten zu beschreiben. Eine Klasse ist ein Template für das Erzeugen von Objekten. Objekte sind also Instanzen einer Klasse. Die Klasse entspricht in etwa einem komplexen Datentyp wie in der prozeduralen Programmierung, geht aber darüber hinaus: Sie legt nicht nur die Datentypen fest, aus denen die mit Hilfe der Klassen erzeugten Objekte bestehen, sie definiert zudem die Algorithmen, die auf diesen Daten operieren. Während also zur Laufzeit eines Programms einzelne Objekte miteinander interagieren, wird das Grundmuster dieser Interaktion durch die Definition der einzelnen Klassen festgelegt.

Mit Objektorientierung ist es möglich eine Modularisierung zu erreichen, die über Unterprogramme hinausgeht. Bestimmte Concerns können durch Kapselung und Vererbung strukturiert werden. Diese Modularisierung nach der Struktur wird durch Objektorientierung erreicht.

Bekannte Vertreter dieses Paradigmas sind JAVA, C++, C#, SMALLTALK und EIFFEL.

3.3.1 Ein Debugger für objektorientierte Programme

Das Debugging von SS-Programmen unterscheidet sich vom Debugging von OO-Programmen [41]. Alle vier erwähnten Konzepte des objektorientierten Paradigmas, besonders aber Polymorphie (dynamisches Binden) und Kapselung beeinflussten die Umsetzbarkeit der in Kapitel 3.1 erwähnten Grundanforderungen.

Stand-alone Debugger wie der Data Display Debugger rücken immer mehr in den Hintergrund, da die Produktivität eines Entwicklers mit Entwicklungsumgebungen nachweislich gesteigert wird. Der Fokus der Betrachtung liegt also auf Debugger in Entwicklungsumgebungen wie Eclipse, Visual Studio .NET oder IDEA. Diese unterscheiden sich aber kaum voneinander. Am Beispiel des Eclipse Debuggers für das Java Development Tooling werden die Umsetzungen der Anforderungen erklärt, die beim Debugging von objektorientierten JAVA-Programmen, als State of the Art gelten. Der JDT Debugger verwendet keinen Backenddebugger und benutzt die Visualisierungsmöglichkeiten des Eclipse Frameworks.

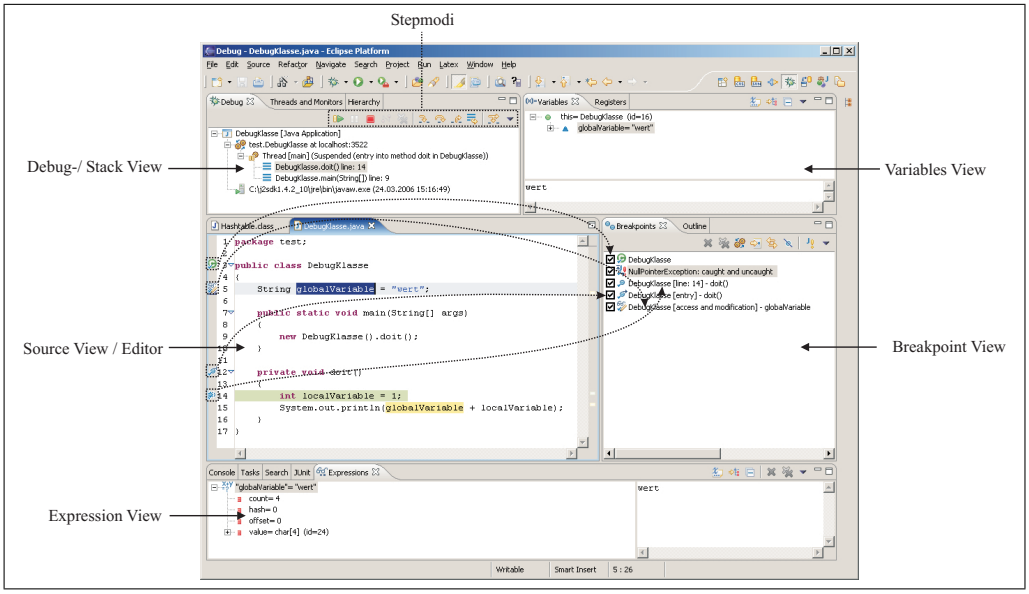


Abbildung 3.3: Eclipse JDT Debugger Perspektive

Quellcodedarstellung - Der Quelltext entspricht dem Maschinencode des Programms, es liegt somit ein 1:1 Matching vor.

Kontrolle durch Schrittmodi - Step Into, Step Over und Step Out/Step Return werden unterstützt(Erklärung siehe 2.1.3.1 auf Seite 8). Des weiteren werden noch folgende Kontrollmöglichkeiten geboten:

Run To Cursor - Dieser Modus unterbricht die Ausführung erst, wenn die Quellcodeinstruktion in der angegebenen Zeile ausgeführt werden soll.

Drop to Frame - Mit *Drop to Frame* ist es möglich an den Beginn der Methodenausführung zurückzuspulen. Allerdings werden Variablen davon nicht beeinflusst und sie behalten ihre aktuellen Werte. Dieses Feature steht nur zur Verfügung, wenn es die darunter liegende Virtuelle Maschine auch unterstützt.

Stepfilter - Durch Konfiguration des Debuggers ist es möglich, bestimmte Module der Objektorientierung (Packages und Klassen) während des Debuggings nicht zu betrachten. Die Ausführung wird dabei nicht beeinflusst; trifft der Debugger aber auf ein Modul was gefiltert werden soll, so wird es nicht in den Sichten (Quelltexteditor, Variablen View und Stackframe) angezeigt. Dadurch kann der Bereich, der während des Debugging betrachtet werden soll, stark eingegrenzt werden. Standardpakete und benötigte Pakete anderer Entwickler können somit gefiltert werden, damit der betrachtete Bereich nur den eigenen Code umfasst.

Kontrolle durch Haltepunkte - Durch die hinzugekommenen Konzepte wurden neue Haltepunkttypen realisiert. Existierende Haltepunkttypen wurden angepasst:

Zeilenhaltepunkt - Wie beim Data Display Debugger (siehe Kapitel 3.2.1) bietet auch der JDT-Debugger die Möglichkeit einen Zeilenhaltepunkt mit Bedingungen zu versehen, die erfüllt sein müssen, damit der Haltepunkt aktiviert wird. Außerdem kann mit *Depends on* die Abhängigkeit geschaffen werden, dass ein Haltepunkt nur aktiviert wird, wenn vorher die aufgelisteten Haltepunkte passiert wurden.[7]

Klassenladehaltepunkt (engl. Class Load Breakpoint) - Während ein Zeilenhaltepunkt das Programm beim Auftreten einer bestimmten Quellcodezeile anhält, stoppt ein Klassenladehaltepunkt das Programm, wenn eine selektierte Klasse in die Virtuelle Maschine geladen wird.

Methodenhaltepunkt (engl. Method Breakpoint) - Die Ausführung wird beim Methodenhaltepunkt durch den Aufruf einer selektierten Methode unterbrochen. Mit *Method Entry* und *Method Exit* kann noch weiter eingeschränkt werden, ob die Programmausführung am Anfang und/oder am Ende der Methode gestoppt werden soll. Ansonsten kann der *Methodenhaltepunkt* mit den gleichen Bedingungen versehen werden, wie der Zeilenhaltepunkt.

Fehlerhaltepunkt (engl. Exception Breakpoint) - Ein Haltepunkt vom Typ Fehlerhaltepunkt hält das Programm beim Auftreten einer selektierten Exception (dt. Fehler, Ausnahme). Die Selektion kann noch verfeinert werden, indem zwischen abgefangener und/oder nicht abgefangener Exception gewählt werden kann. Der Fehlerhaltepunkt kann nur mit der Zählbedingung versehen werden. Die Codebedingung ist nicht möglich.

Visualisierung - Neben den in der Abbildung 3.3 erklärten Fenstern (Sichten), verfügt der JAVA Debugger von Eclipse über keine weiteren Visualisierungen. Sichten wie die Callhierarchy[8]

[7] Die Möglichkeit Abhängigkeiten zwischen Haltepunkten zu schaffen, wird nicht von JDT-Debugger unterstützt.

[8] Die Callhierarchy ist eine Sicht, die zeigt von welchen Methoden eine selektiere Methode aufgerufen wird und welche Methoden sie selbst aufruft.

und die Typhierarchy[9], machen den Programmfluss erkenntlicher, gehören aber nicht explizit zum den Debuggervisualisierungen.

Programmmanipulation - Das Editieren des Quelltextes ist auch während der Debug Phase möglich. Durch die Möglichkeit des *Hot Code Replacement* muss nach Änderungen des Codes die Programmausführung und damit der Vorgang des Debuggings nicht neu gestartet werden. Die Änderungen werden auch während der Debug Phase übernommen. Strukturänderungen einer Klasse sind aber nicht möglich. So kann zum Beispiel keine neue Methode zu einer Klasse hinzugefügt werden, wenn der Debuggee Instanzen dieser Klasse verwendet.

Die hinzugekommenen Haltepunkttypen decken die OO-Modualisierungskonzepte, Methode und Klasse ab. Die kontrollierte Programmausführung wurde um Filterungsmöglichkeiten erweitert, um die Fehlersuche effektiver zu gestalten. Alle Grundanforderungen wurden erfüllt. Die Umsetzung ist damit erfolgreich.

3.4 Aspektorientiertes Paradigma

Das aspektorientierte Paradigma erweitert das objektorientierte Paradigma, indem die Modularisierbarkeit um eine Dimension erweitert wird. Logisch unabhängige Concerns können mit AOP auch physisch voneinander getrennt werden.

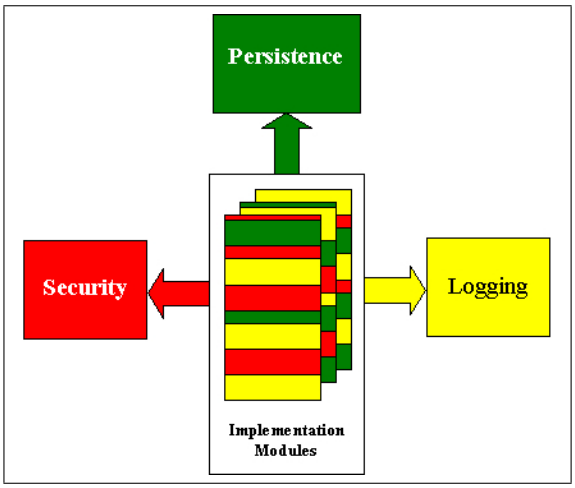


Abbildung 3.4: Mixing of concerns, Copyright 2002, Ramnivas Laddad. I want my AOP!

crosscutting concerns/Aspekt - Funktionalitäten oder System-Eigenschaften, die über sehr viele Klassen verteilt realisiert werden müssen, werden *crosscutting concerns* genannt. Die Logging-, Persistenz- und Securityfunktionalität eines System sind Paradebeispiele.

[9] Die Typhierarchy zeigt die Vererbungsstruktur mit Super- und Subklassen an. Dabei werden auch implementierte Interfaces dargestellt.

scattering - *scattering* ist eine Art der Verteilung eines crosscutting concerns. Kann ein Concern der im klassischen objektorientierten Design nur durch Zerstreuung in mehrere Module/Klassen erreicht werden, wird das *scattering* genannt. Der Quellcode für das Logging aller Methodenaufrufe würde mit OOP-Mitteln zum Beispiel in jeder einzelnen Methode auftauchen.

tangling - Die Vermischung verschiedener Concerns an einem Ort, wird *tangling* genannt. Code, der unterschiedlichste Funktionalität an einer Stelle innerhalb einer Methode oder Klasse realisiert, wird als tangling Code bezeichnet.

Mit Aspektorientierung ist es möglich "crosscutting concerns" zusammenzufassen. Ein Aspekt definiert hierbei ein "crosscutting concern" und bündelt damit die Funktionalität an einem Ort. Probleme wie scattering und tangling werden dadurch vermieden und es entsteht dabei Code, der besser wartbar und wiederverwendbar ist.

Um AOP praktisch umsetzen zu können, bedarf es zum Einen einer Aspektsprache, in der die Aspekte implementiert werden. Zum Anderen müssen diese Aspekte mit dem restlichen System kombiniert werden. Das geschieht durch Angabe von Join Points. Ein Aspektweber webt nun die implementierten Aspekte an definierte Webepunkte der Basisapplikation. Ausgeführt wird dann eine Komposition aus Basisapplikation und Aspekten.

Join Points - *Join Points* sind eindeutig definierte Stellen/Ereignisse im Programm an denen der implementierte Aspektcode eingefügt werden kann. Eine aspektorientierte Sprache bietet die Möglichkeit der Selektion der Join Points. Oft beinhaltet die aspektorientierte Sprache eine Join Point Sprache (JPL), die eine genaue Selektion der Join Points erst möglich macht. Typische Join Points sind der Ein- oder Austritt aus einer Methode oder das Auftreten einer Exception.

Pointcut - Ein *Pointcut* selektiert eine Menge von Join Points. Ein Pointcut ist also eine Auswahl aus der Menge aller Join Points eines Programms. Durch die JPL werden die Join Points eingegrenzt, an denen ein *Advice* gewoben werden sollen. Die resultierende Menge der eingegrenzten Join Points werden Pointcut genannt.

Advice - *Advices* sind Anweisungen, die zur Ausführung kommen, wenn bestimme *Join Points* erreicht werden. Advices können in vielen aspektorientierten Sprachen auf Laufzeitinformationen der gewählten Join Points zurückgreifen. Im Allgemeinen gibt es drei Formen wie ein Advice mit einem Join Point assoziiert werden kann: after, before und replace (auch around). Ein before Advice würde dann direkt vor dem spezifizierten Join Point aufgerufen werden, ein after Advice dirket danach. Ein replace Advice umschließt einen Join Point und kontrolliert die ursprüngliche Ausführung an diesem Join Point. Im Advice können zum Beispiel Argumente verändert werden und die Ausführung würde dann mit diesen veränderten Argumenten fortgesetzt werden.

Weber/weaver - Ein Weber ist das Werkzeug, was die Basisapplikation und die Aspekte zusammen webt. Je nach Programmiersprache kann der Weber Bestandteil des Compilers, der Laufzeitumgebung oder Beidem sein. Ein Weber webt die Aspekte an die durch die JPL definierten Punkte(*Pointcuts*) der Basisapplikation.

weben - Die Integration des Aspektcodes in die Basisapplikation wird *weben* genannt. Das *weben* ist ein automatischer Vorgang und es wird zwischen statischen und dynamischen Weben unterschieden(siehe Kapitel 4.1.3).

Die Modularisierbarkeit, die durch AOP erreicht wird, ermöglicht eine vollständige Trennung verschiedenster Funktionalitäten eines Softwaresystems (separation of concerns). Aspektorientierung bietet weiterhin Funktionen zur strukturellen Erweiterung und Verhaltenserweiterung.

Object Teams/Java, CaesarJ, AspectJ, Aspect C++, Hyper J und LOOM .NET sind Vertreter dieses Paradigmas.

3.4.1 Ein Debugger für aspektorientierte Programme

Die Werkzeugunterstützung für Sprachen des aspektorientieren Paradigmas ist noch nicht so umfangreich wie die Werkzeugunterstützung objektorientierter Sprachen. AspectJ ist die verbreitetste aspektorientierte Sprache und die Unterstützung für diese Sprache durch geeignete Werkzeuge ist im Gegensatz zu anderen aspektorientierten Sprachen am umfangreichsten. Deshalb wird der Debugger hier als State of the Art Debugger für AOP Sprachen herangezogen. Der Debugger ist Bestandteil des AJDT (AspectJ Development Tool).

Quellcodedarstellung - Die Umsetzung bei der Erfüllung der Anforderung der Quellcodedarstellung ist beim AJDT-Debugger unvollständig. Eine detaillierte Betrachtung ist im Kapitel 5.2.2.3 zufinden.

Kontrolle durch Schrittmodi - Der AJDT-Debugger bietet die gleichen Schrittmodi wie der JDT-Debugger. Darüber hinaus werden keine weiteren Schrittmodi unterstützt.

Kontrolle durch Haltepunkte - Das Setzen von Zeilenhaltepunkten in Klassen der Basisapplikation und im Aspektcode wird vollständig unterstützt. Klassenladehaltepunkte, Methodenhaltepunkte, Fehlerhaltepunkte und Überwachungspunkte können in Basiscode in gewohnter Weise benutzt werden. Das Setzen dieser Haltepunkttypen in Aspektcode wird nicht unterstützt.

Visualisierung :

Aspect Visualizer - Der *Aspect Visualizer* ist Bestandteil des AJDT und visualisiert, wie und wo Klassen von Aspekten adaptiert werden. Er ist kein Bestandteil der Debug Unterstützung, hilft aber dem Benutzer beim Verstehen des Programms und dessen Ausführung und wird deshalb hier erwähnt. Die Auswirkungen der Aspekte auf die Basisapplikation werden visualisiert, indem die Klassen als Balken dargestellt werden. Aspektcode wird farbig als Streifen innerhalb des Balkens abgebildet. Adaptieren mehrere Aspekte die gleiche Klasse, so werden die unterschiedlichen Aspekte anders farbig innerhalb des Balkens (Klasse) dargestellt. Die Streifen stellen auch dar, wo der Aspekt den Basiscode beeinflusst. Durch Filterung ist es möglich nicht adaptierte Klassen auszublenden, beziehungsweise bei adaptierten Klassen einzelne Aspekte der Visualisierung hinzuzufügen oder zu verstecken.

Anpassung der Sichten - Die Stack Frame-Sicht (Debug View) zeigt im Stackframe Aufrufe von Methoden, die nicht im Quellcode zu finden sind. Die Variablensicht zeigt Variablen an die nicht im Quellcode zu finden sind.

Codemanipulation - Das Ändern des Programmzustandes und das Ändern des Programmflusses ist möglich.

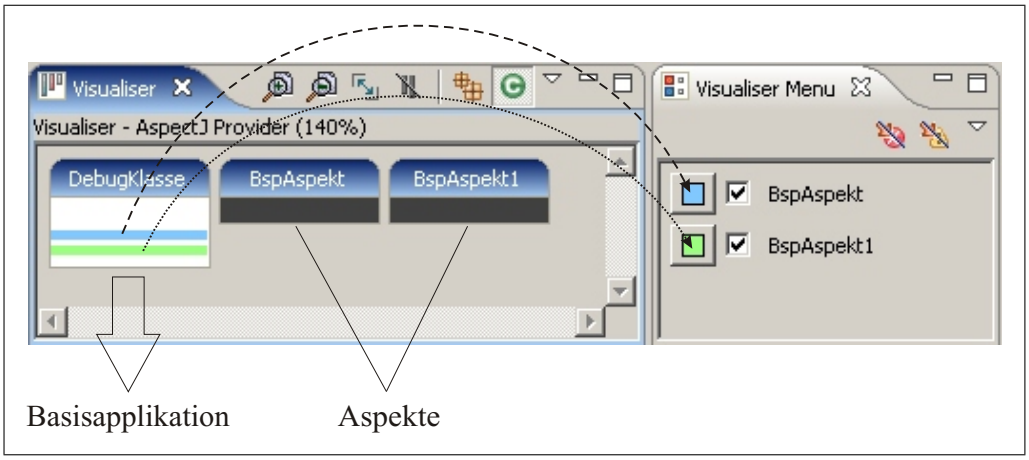


Abbildung 3.5: AspectJ Visualizer aus dem AspectJ Development Tool

Fazit

Derzeitige Debugger für aspektorientierte Sprachen erfüllen die an sie gestellten Anforderungen nicht. Einfache Grundanforderungen wie das Quellcodedarstellung stellen ein Problem dar. Die Umsetzungen in den Anforderungsbereichen *Kontrolle durch Schrittmodi* und *Kontrolle durch Haltepunkte* sind unvollständig. Die Debugunterstützung für aspektorientierte Programme ist zur Zeit unzureichend.

Worin liegt die Schwierigkeit bei der Erfüllung der Grundanforderungen an einen interaktiven Debugger für aspektorientierte Programme? Um diese Frage beantworten zu können, muss ein Blick auf die technischen und konzeptionellen Grundlagen aspektorientierter Sprachen geworfen werden. Nur dadurch werden die Probleme verständlich, die die Entwicklung eines Debugger für aspektorientierte Programme erschweren. Im nächsten Kapitel wird der Grundstein für das Verständnis der auftretenden Probleme gelegt.

4 Grundlagenwissen - AOP und Debugging

Eine Vielzahl von Programmiersprachen realisieren die in Kapitel 3.4 erläuterten Konzepte des aspektorientiertes Programmierparadigmas. Jede Sprache bietet ihre Interpretation der Konzepte und stellt fundamentale Werkzeugunterstützung, wie ein Übersetzungswerkzeug, und wenn benötigt, eine Laufzeitumgebung zur Verfügung. Die meisten aspektorientierten Sprachen sind Erweiterungen der objektorientierten Sprache Java. Solche Sprachen stehen in dieser Diplomarbeit im Mittelpunkt.

Im vorherigen Kapitel wurde unter anderem auch ein Debugger für aspektorientierte Programme vorgestellt. Dieses Werkzeug musste als *kaum benutzbar* eingestuft werden, da die gestellten Anforderungen, die jeder interaktive Debugger erfüllen sollte, nicht vollständig erfüllt werden. In diesem Kapitel werden allgemeine technische Grundlagen zu Webetechniken und zur Debugunterstützung für Java-basierende AOP-Sprachen erläutert. Außerdem werden Grundlagen der Aktivierung der Aspekte erklärt. Resultierend aus dem Grundlagenwissen werden die Schwierigkeiten bei der Realisierung eines Debuggers für aspektorientierte Programme offensichtlich.

4.1 Grundlagen der Aspektorientierung

4.1.1 Generative Programmierung

Es gibt viele Möglichkeiten das Konzept der Aspektorientierung technisch umzusetzen. Ein Ansatz nutzt das Generatorenprinzip des generativen Programmierens. Realisierungen durch Veränderung des Interpreters bei interpretierten Sprachen werden hier nicht betrachtet.

Eine Definition des Generativen Programmierens lautet:

> "Generatives Programmieren modelliert Software-Systemfamilien so, dass ausgehend von einer Anforderungsspezifikation mittels Konfigurationswissen aus elementaren, wiederverwendbaren Implementierungskomponenten ein hoch angepasstes und optimiertes Zwischen- oder Endprodukt nach Bedarf automatisch erzeugt werden kann." [51]

Generatives Programmieren definiert zum Beispiel Anforderungen auf einer abstrakten Ebene. Aus der abstrakten Beschreibung wird durch einen Automatismus das Endprodukt, zum Beispiel ein Softwaresystem, erzeugt. Die Technik hinter dem Automatismus sind Generatoren.

Generatoren - *Generatoren* können aus abstrakten Eingaben weniger abstrakte Ausgaben erstellen. Zu den Aufgaben eines Generators gehört das Prüfen und das Vervollständigen der Spezifikation, die Durchführung von Optimierungen und das Generieren der Implementierung.

Ein Generator transformiert etwas Ursprüngliches zu etwas Anderem. Diese Transformation macht aber nur Sinn, wenn sich entweder die Struktur oder die Abstraktionsebene zwischen dem Ursprünglichen und dem Generierten verändert. Diese Transformation lässt deshalb klassifizieren:

Horizontale Transformation - Die Transformation passiert hier auf der horizontalen Ebene. Bei dieser Transformationsklasse wird die Struktur des Ursprünglichen, aber nicht die Abstraktionsebene verändert.

Vertikale Transformation - Generatoren dieser Klasse verändern die Abstraktionsebene. Das abstrakte Ursprüngliche wird bei der vertikalen Transformation zu etwas Konkreterem umgeformt. Dabei wird die Struktur nicht verändert.

Ausgehend von der Klassifizierung der Transformationen lassen sich Kategorien von Generatoren ableiten. Ein Postprozessor, der Debuginformationen aus Maschinencode-Dateien entfernt, um die Dateigrösse zu verringern oder ein Postprozessor, der Optimierungen am Maschinencode[1] vornimmt und dabei wieder Maschinencode erzeugt, sind Generatoren, die eine horizontale Transformation vornehmen. Einfache Compiler, die eine Hochsprache in eine Low-Level-Sprache überführen oder CASE-Werkzeuge, die eine abstrakte UML-Beschreibung in eine Hochsprache transformieren, gehören zu der Klasse von Generatoren, die eine vertikale Transformation vornehmen. Führt ein Generator eine horizontale und vertikale Transformation durch, wird der Generator *oblique* genannt. Er gehört also beiden Transformationsklassen an. Ein Compiler, der neben der Transformation der Sprache auch Optimierungen an der Low-Level-Sprache vornimmt, ist ein Beispiel für einen oblique Generator.

In der Aspektorientierung werden diese Techniken genutzt, um die Konzepte von AOP umzusetzen. Dabei wird der Generator in zweifacher Hinsicht verwendet. Zum Einen findet mit dessen Hilfe eine Überführung der abstrakt beschriebenen Aspekte in eine Low-Level-Sprache statt, und zum Anderen realisiert der Generator die Komposition des Aspektcodes mit dem Basiscode, also das Weben. Die bei den aspektorientierten Sprachen Object Teams/Java und AspectJ eingesetzten Generatoren gehören zu den oblique Generatoren, da sie eine horizontale Transformation und vertikale Transformation des ursprünglichen Codes vornehmen. Auf diese Klasse der Generatoren und auf deren Arbeitsweise wird im Weiteren eingegangen.

4.1.2 Abstraktion für Aspektbeschreibung

Das Beschreiben der Aspekte auf Quellcodeebene muss durch eine Sprache unterstützt werden. Diese Aspektsprache wird als gekapselte Spracherweiterung einer vorhandenen Sprache realisiert. Dabei wird die ursprüngliche Programmiersprache um Konzepte der Aspektorientierung erweitert. Object Teams/Java und AspectJ sind Vertreter für AOP-Sprachen, die die objektorientierte Programmiersprache Java um die jeweiligen AOP-Konzepte erweitern. So lassen sich neue Sprachkonstrukte und Schlüsselwörter der aspektorientierten Sprache und Standard Java Konstrukte miteinander kombinieren.

[1] Eine Optimierung ist zum Beispiel *Inlining*. Beim Inlining wird der Methodenrumpf an die Stelle kopiert, an der die Methode aufgerufen wird. Dadurch wird ein Methodenaufruf eingespart und die Laufzeit-Performance gesteigert.

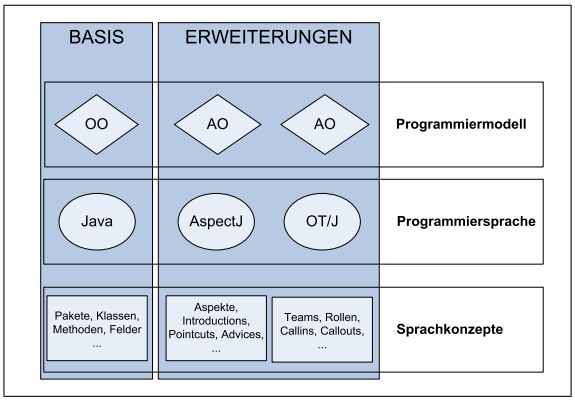


Abbildung 4.1: Erweiterung der Basissprache um Aspektsprachkonzepte

Was sich auf abstrakter Ebene, also im Quellcode beschreiben lässt, muss auch auf technischer Ebene umgesetzt werden. Das bedeutet, dass sich die abstrakte Beschreibung der Aspekte in der Hochsprache in eine Low-Level-Sprache überführen lassen muss. Für die AOP-Sprachen Object Teams/Java und AspectJ heißt das, dass sich die Spracherweiterungen, die sich auf der abstrakten Ebene befinden, durch einen Übersetzer in eine weniger abstrakte Sprache, hier die Maschinensprache, überführen lassen müssen. Dieser Übersetzer ist im Fall Object Teams/Java und AspectJ ein Standard Java Compiler, der erweitert wurde, um aus den durch die Spracherweiterung hinzugekommenen Sprachelementen, ausführbaren Maschinencode erzeugen zu können. Dieser erweiterte Java Compiler ist im Kontext der Generativen Programmierung ein oblique Generator, da er Transformationen auf vertikaler und horizontaler Ebene vornimmt. Welche Transformationen der oblique Generator bei der AOP-Sprache Object Teams/Java vornimmt wird in Kapitel 5.1 bzw. 5.2 näher erläutert.

4.1.3 Webetechniken

Nachdem der Bereich des Generators erklärt wurde, der für die Überführung von abstrakt definierten Aspektbeschreibungen in eine Low-Level-Maschinencodesprache verantwortlich ist, liegt jetzt der Fokus auf der Komposition von Basis- und Aspektcode, also dem Weben. Wie schon in Kapitel 3.4 konzeptionell erklärt wurde, werden Basiscode und Aspektcode unter Hilfenahme eines Webers miteinander verwoben.

Webemechanismen lassen sich im Kontext des Generativen Programmierens mit zwei verschiedenen Techniken realisieren: **Code Transformation** und **dynamische Reflektion** [21]. Beide Techniken sind Beispiele aus der Metaprogrammierung.

Metaprogrammierung - *Metaprogrammierung* wird in der Literatur oft als das "Manipulieren von Programmen als Daten" charakterisiert. Das Metaprogramm kann die Struktur und/oder das Verhalten des Basisprogramms verändern. Der Metacode steht dabei "über" dem Basiscode, sowie der Aspektcode "über" dem Basiscode steht.

Code Transformation ist ein Beispiel für *Statisches Metaprogrammieren*. Die Analyse der Komponenten[2] zum Zeitpunkt der Komponentenkonstruktion, die darauf aufbauende Transforma-

[2] Basis- und Aspektkomponente

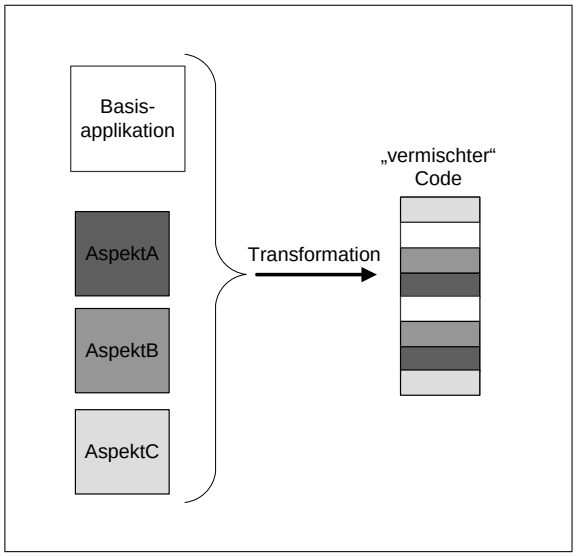


Abbildung 4.2: Transformation von Basis- und Aspektcode

tion der Komponenten und die automatische Erzeugung von *Infrastrukturcode* (in [30] Klebecode genannt), ordnen besonders das *Weben zur Übersetzungszeit* (siehe unten) dem Statischen Metaprogrammieren zu. Die Analyse der Komponenten zur Laufzeit und die automatische, dynamische Anpassung sind Charakteristika des *Dynamischen Metaprogrammierens*. Die dynamische Reflektion ist ein Beispiel dafür [13].

Die beiden erwähnten Techniken werden zur Komposition von Basis- und Aspektcode eingesetzt und werden im folgenden Abschnitt genauer erläutert.

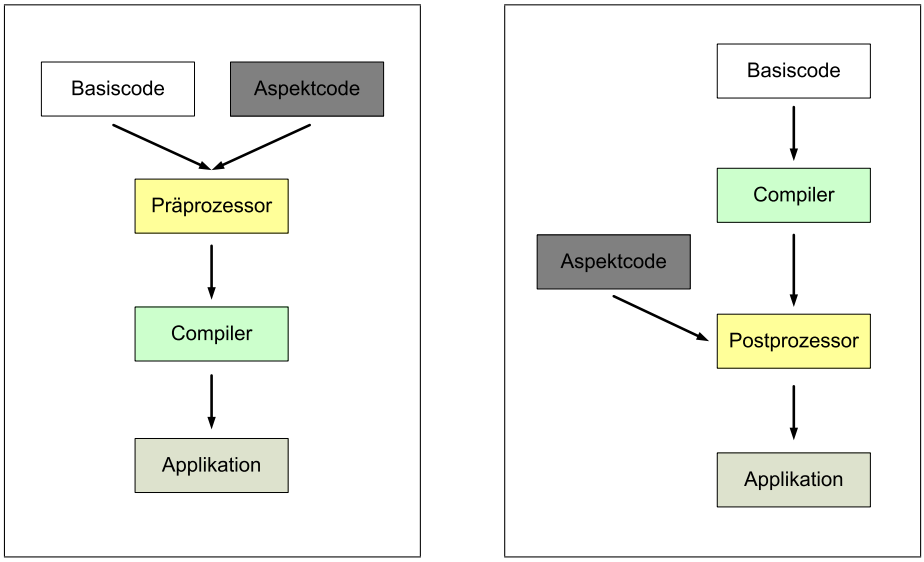
4.1.3.1 Code Transformation

Code Transformation wird im Bereich der Aspektorientierung zum Weben der Aspekte genutzt. Code kann auf Quellcode- oder Maschinencodeebene zu "vermischtem" Code transformiert werden, wie er in Abbildung 4.2 erkennbar ist. Der Zeitpunkt der Code Transformation entscheidet darüber, ob es sich um *Übersetzungszeit* oder *Ladezeit Weben* handelt. Das Resultat ist dabei aber immer vermischter Code.

Weben zur Übersetzungszeit (engl. Compiletime Weaving)

Die Code Transformation beim Weben zur Übersetzungszeit kann zum Einen auf Quellcodeebene(Sourcecode Weaving) und zum Anderen auf Maschinencodeebene(Bytecode Weaving) geschehen.

Die Transformation auf Quellcodeebene übernimmt in vielen Fällen ein Präprozessor, der Basis- und Aspektquellcode analysiert und als Ergebnis Quellcode generiert, der von einem Standard Compiler übersetzt werden kann. Der resultierende Maschinencode wird danach nicht weiter verändert und kann in dieser Form ausgeführt werden. Das Weben findet bei dieser Art der Realisierung auf Quellcodeebene statt. Das Sourcecode Weaving wird bei der aspektorientierten Sprache AspectC++ für die Komposition von Basisapplikation und Aspekt eingesetzt. Für die Transformation wird ein Präprozessor eingesetzt, der die Mächtigkeit eines Compilers besitzt



(a) Sourcecode Weaving (b) Bytecode Weaving

Abbildung 4.3: Weben zur Übersetzungszeit

und aus mehreren Komponenten wie Scanner, Parser und Weber besteht. Zum Anderen kann die Transformation auf Maschinencodeebene passieren. Der Basisquellcode wird dabei zuerst in Maschinencode übersetzt, dann konsumiert ein Postprozessor Basismaschinen- und Aspektcode und produziert letztendlich "vermischten" Maschinencode. Manchmal ist der Postprozessor als Teil des Compilers implementiert. AspectJ unterstützt das Weben zur Übersetzungszeit.

Weben zur Ladezeit

Anders als beim Weben zur Übersetzungszeit, passiert die Komposition von Aspekt und Basis hier zur Ladezeit. Das Weben zur Ladezeit (engl. Load Time Weaving) transformiert eine Applikation während des Ladeprozesses [48]. Dafür gibt es eine Vielzahl von Frameworks, die das Laden eines Programms durch einen eigenen Mechanismus ersetzen. Dieser Mechanismus lädt das Programm und initialisiert den Webevorgang. Der Weber ist bei dieser Form des Webens Teil der Laufzeitumgebung und verändert die Applikation auf Maschinencodeebene. Während dieses Vorgangs wird der Maschinencode nur im virtuellen Speicher transformiert. Die Programmdateien auf der Festplatte werden nicht verändert. Nach Beendigung des Webens wird das Programm ausgeführt. ObjectTeams/Java und AspectJ[3] realisieren den Webemechanismus zur Ladezeit.

Pseudo-Laufzeit-Weben

Pseudo-Laufzeit-Weben passiert zur Laufzeit des Programms und auf Maschinencodeebene. Es gibt zwei verschiedene technische Umsetzungen:

Die **erste** Umsetzung realisiert das Pseudo-Laufzeit-Weben, indem die Applikation im Debug-Modus ausgeführt wird. Mit Hilfe von Debugging-Schnittstellen wird in den Ablauf des Pro-

[3] AspectJ v1.5.1a war zum Zeitpunkt der Entstehung der Diplomarbeit die aktuelle AspectJ Version. Diese Version unterstützt das Weben zur Übersetzungszeit und das Weben zur Ladezeit.

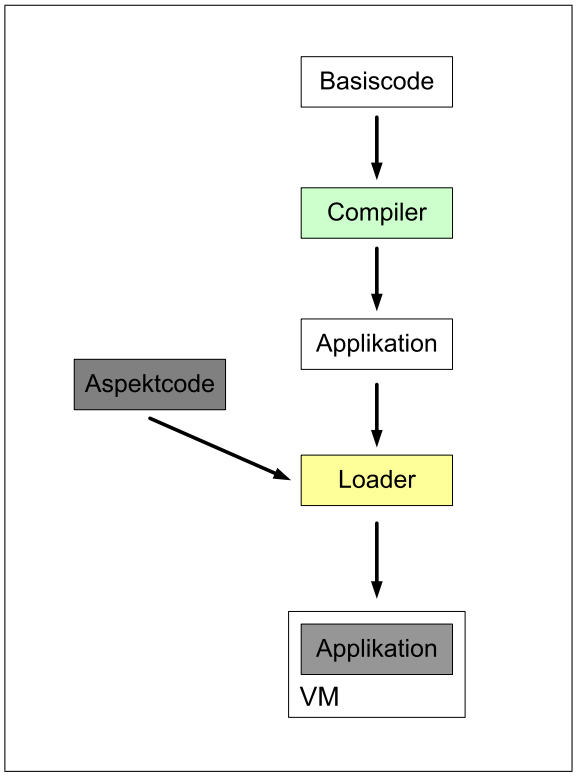


Abbildung 4.4: Weben zur Ladezeit

gramms eingegriffen. Eine spezielle "Registry" verwaltet dabei die Aspekte. Der Bytecode wird innerhalb der Virtual Machine mit Hilfe der *HotSwap-Technik*[4] verändert. Die AOP-Sprache AspectWerkz [16] unterstützt neben dem Weben zur Übersetzungszeit, auch diese Art des Pseudo-Laufzeit-Webens.

Die **zweite** Umsetzung wird *Just-in-time Compiler Weaving* genannt. Dabei wird unveränderter Bytecode in die Virtual Machine geladen und ein modifizierter *Just-in-time Compiler*[5] fügt zusätzlichen Code in die Applikation ein. Der zusätzliche Code sorgt später für die Ausführung des Aspektcodes. PROSE [10] und Steamloom [12] sind modifizierte Virtual Machines, die diese Art des Webens unterstützen. PROSE liefert neben der Virtual Machine auch eine Aspekt-Sprache, wo Aspekte in Standard Java Syntax definiert werden.

4.1.3.2 Dynamische Reflektion

Dynamische Reflektion setzt nicht die Transformation des Codes, ob nun in kompilierter oder in rein textueller Form, für die Komposition von Aspekt und Basisapplikation ein. Bei der dynamischen Reflektion wird nicht "vermischter" Code ausgeführt, sondern der Code wird zur

[4] Die HotSwap-Technik wird in Kapitel 4.2.1 näher erläutert.
[5] Ein Just-in-time Compiler (JIT-Compiler) erzeugt zur Laufzeit eines Programms nativen, auf die Basismaschine optimierten Code aus einem beliebigen Zwischencode [3].

Laufzeit interpretiert. Teile des Codes werden während der Laufzeit als eine Art Metacode[6] erkannt und die Programmkontrolle wird an die Aspekte transferiert. Dabei kann der Transfer so häufig stattfinden, dass der Eindruck entsteht, es würde "vermischter" Code ausgeführt werden.

Das Weben bei der dynamischen Reflektion passiert zur Laufzeit und gehört damit zu den dynamischen Webetechniken. Auf technischer Seite wird das Weben zur Laufzeit auch generisches Laufzeit-Weben genannt.

Generisches Laufzeit-Weben

Beim Generischen Laufzeit-Weben (engl. generic runtime weaving) werden im Java-Kontext dynamische Proxy eingesetzt. Das erfordert keine Transformation des Codes, weil nur Standard Mechanismen[7] der Sprache Java verwendet werden. Ein dynamischer Proxy ist ein dynamisches Objekt, welches zur Laufzeit von der Java Virtual Machine erzeugt wird und ein oder mehrere Interfaces implementiert. Beim proxy-basierten Ansatz werden die Aspekte mittels dynamischen Proxy auf die Zielobjekte angewendet. Ein Aufruf einer Proxymethode wird an ein Objekt weitergeleitet, das sich dafür registriert hat. Das Spring AOP Framework [11] und das Nanning Aspects AOP Framework [38] benutzen proxy-basiertes, generisches Laufzeit-Weben.

4.1.3.3 Statisches/Dynamisches Weben

Webetechniken lassen sich nicht nur nach Zeitpunkt des Webens und der technischen Realisierung einteilen, sondern auch nach deren Konzept.

Statisches Weben

Das Konzept des statischen Webens wird durch Code Transformation technisch realisiert. Das Weben passiert genau einmal und immer *bevor* das Programm ausgeführt wird. Das Programm wird während der Laufzeit nicht mehr verändert und somit hat das Weben auf die Laufzeit-Performance auch keinen Einfluss. Verletzungen der Typkonformität werden vor der Ausführung ersichtlich. Nach der Code Transformation hat man keinen Einfluss auf die Aspekte mehr. Sie sind mit dem Basiscode vereint und gewobene Aspekte lassen sich zur Laufzeit nicht mehr entfernen, noch Neue hinzufügen. Bei Änderungen der Aspekte muss die komplette Applikation neu kompiliert werden. Passiert die Code Transformation eine Zeiteinheit später, also zur Ladezeit, wird das Problem der Neukompilierung der Applikation umgangen. Welcher Aspekt gewoben werden soll, kann jetzt auch später, zur Ladezeit, entschieden werden. Diese Features gehen zu Lasten der Performance. Das Laden des Programms verzögert sich durch das Weben und ein spezieller Lader[8] ist unabdingbar.

Das Statische Weben bietet den Vorteil, dass das Weben nur einmal passiert und Optimierungen des Maschinencodes vor der Ausführung möglich sind. Nachteilig ist die fehlende Flexibilität. Die einmal gemachten Verknüpfungen können nachträglich nicht mehr geändert werden. Dynamisches Entfernen und Hinzufügen von Aspekten kann mit statischen Weben nicht realisiert werden.

[6] K. Czarnecki [21] betitelt diesen Code auf abstrakter Ebene als Metapräsentation einiger Elemente von Programmiersprachen. Metapräsentation können Klassen, Methoden, Message sends, Methodenausführungen etc. sein, deren Bedeutung/Verhalten zur Laufzeit verändert werden kann.

[7] Das Reflection-Modell erlaubt es, Klassen und Objekte, die zur Laufzeit von der JVM im Speicher gehalten werden, zu untersuchen und in begrenztem Umfang zu modifizieren.

[8] Ein Lader (engl. Loader) ist ein Programm, was das Programm lädt und zur Ausführung bringt.

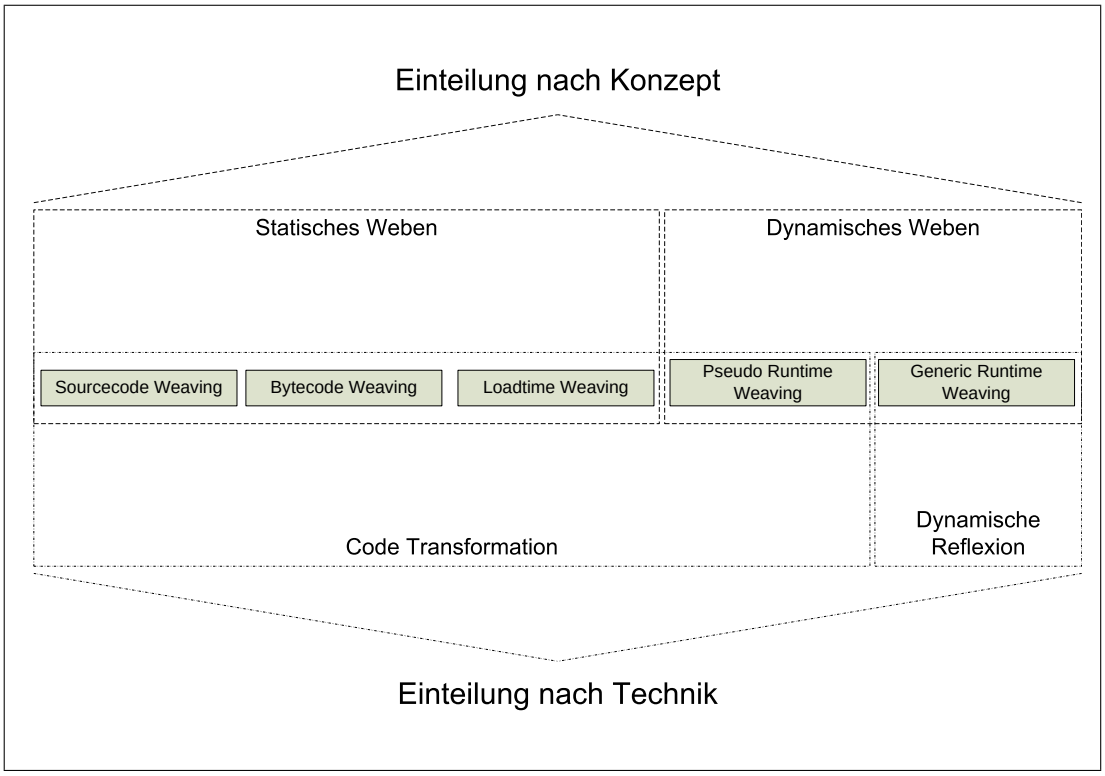


Abbildung 4.5: Einteilung von Webetechnikrealisierungen

Dynamisches Weben

Das Weben zur Laufzeit wird als Dynamisches Weben bezeichnet und wird technisch durch Pseudo-Laufzeit-Weben und Generisches Laufzeit-Weben realisiert. Es passiert während das Programm ausgeführt wird und das flexible Anpassen der Aspekte gibt diesen Webemechanismen den dynamischen Charakter. Verschiedene Aspektvarianten können somit innerhalb einer Ausführung getestet werden. Die Flexibilität des dynamischen Webens geht aber zu Lasten der Laufzeit-Performance und erfordert einen erhöhten Speicherverbrauch. Das Weben passiert hier nicht nur einmal, sondern immer wenn ein Zugriff auf die Basisfunktionen[9] stattfindet.

4.1.4 Aspektaktivierung

Wenn ein Aspekt in eine Basisapplikation gewoben wird, bedeutet das nicht unbedingt, dass er auch *aktiv* ist. Aktiv bedeutet in diesem Fall, dass der Aspekt wirkt und das Laufzeitverhalten des adaptierten Programm beeinflusst. Einige aspektorientierte Sprachen besitzen einen eigenen Aktivierungsmechanismus, wie die Sprache Object Teams/Java (siehe Kapitel 5.1.3). Aspekte können hier aktiviert und deaktiviert werden. Andere Sprachen bieten nur das explizite Ausprogrammieren von Bedingungen und Alternativen, um Aspekte zur Laufzeit wirken zu lassen oder nicht. Zwar durchzieht diese Art der Aspektaktivierung die gesamte Struktur eines Programms und kann sich negativ auf Verständlichkeit und Wartbarkeit des Programms

[9] Basisfunktionen sind zum Beispiel Methoden einer Klasse der Basisapplikation. Ein Zugriff, also ein Methodenaufruf, wird erkannt und der Webevorgang eingeleitet.

auswirken, aber einen Einfluss auf die Funktionsweise des Aktivierungsmechanismus hat dies nicht - ein Aspekt kann zur Laufzeit dynamisch aktiviert und deaktiviert werden.

Die Aspektaktivierung kann unabhängig vom Webemechanismus betrachtet werden, denn egal ob Aspekte nun statisch oder dynamisch gewoben wurden, deren Ausführung wird bei den meisten aspektorientierten Sprachen erst zur Laufzeit entschieden. Aspektorientierte Programme können somit durch dynamische Aspektaktivierung hohe Komplexitätslevel erreichen, die das Nachvollziehen solch eines Programms ohne weitere Werkzeugunterstützung unmöglich machen können.

4.2 Grundlagen des Java Debugging

Wie schon am Anfang des Kapitels erwähnt, liegt der Fokus auf aspektorientierten Sprachen, die zum Einen Erweiterungen der Sprache Java sind und deren Compiler und Aspektweber zum Anderen Standard Java Maschinencode erzeugen. Das bedeutet, dass der Fokus auf Sprachen liegt, deren Programme Standard Java Programme sind, die auch in einer Standard Java Virtual Machine laufen können. Das aspektorientierte Programm[10] ist auf Maschinencodeebene ein Standard Java Programm, das in einer Standard JVM ausführbar ist und das die Standard Java Debuginformationen enthält und diese nur über Standard Debugschnittstellen der Java Debug Architektur ausgelesen werden können.

Dieser Abschnitt erläutert die Java Debug Architektur und die Debuginformationen, mit denen ein Standard Java Programm angereichert werden kann.

4.2.1 Java Platform Debugger Architecture (JPDA)

In Java stellt die Java Platform Architecture (JPDA) die Debug-Schnittstelle für Debugger dar. Das JPDA unterstützt das Setzen von Halte- und Überwachungspunkten, die Steuerung des Programms mit verschiedenen Einzelschrittmodi und weitere Debuggingfunktionen. Der Zugriff auf die Java Virtual Machine und dessen Steuerung wird durch Schnittstellen innerhalb des JPDA realisiert. Mit Hilfe des JPDA ist die Entwicklung von Debuggern für Java Programme möglich, ohne auf die Besonderheiten der Plattform Rücksicht nehmen zu müssen. Hardware, Betriebssystem und die Java Virtual Machine-Implementierung müssen nicht gesondert betrachtet werden, solange all die Komponenten die geforderten Schnittstellen des JPDA anbieten oder implementieren [6].

Das JPDA besteht aus zwei Softwarekomponenten, zwei Schnittstellen und einem Protokoll (siehe Abbildung 4.6).

4.2.1.1 JPDA - Komponenten

debuggee

Der Debuggee ist ein Prozess/Programm, der vom Debugger kontrolliert wird. Er besteht aus der zu debuggenden Applikation, der *VM* (Virtual Machine) auf der die Applikation läuft und

[10] Der Fokus liegt auf AOP-Sprachen, die Standard Java Bytecode erzeugen.

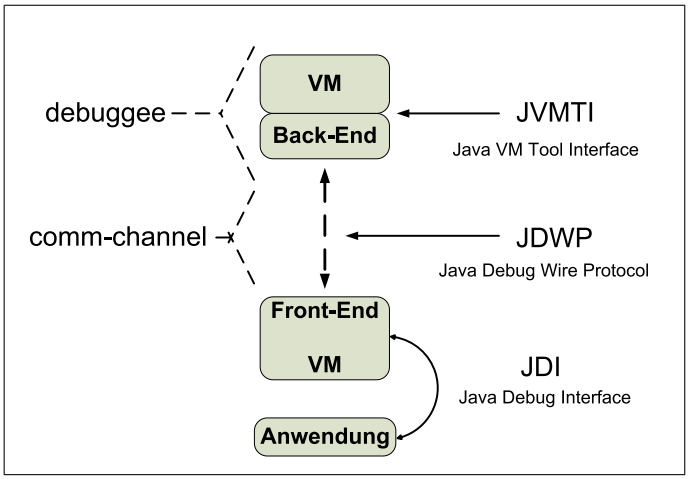


Abbildung 4.6: Java Platform Debugger Architecture (JPDA)

dem *Back-End*. Die zu *debuggende VM* implementiert das JVMTI, ein native-Interface, welches zur Kommunikation mit dem Back-End dient.

comm-channel

Der Kommunikationskanal (engl. comm-channel) stellt die Verbindung zwischen Front-End und Back-End dar. Die Art des Kanals ist nicht festgelegt und kann zum Beispiel über *Sockets*, einer *seriellen Verbindung* oder *shared memory* realisert werden.

Back-End und Front-End

Das Back-End ist für die Kommunikation zwischen der VM des Debuggee und dem Front-End zuständig. Das Back-End kommuniziert zur einen Seite mittels JVMTI- Schnittstelle mit der VM des Debuggee und zur anderen Seite mittels JDWP mit dem Front-End.

Das Front-End implementiert die JDI-Schnittstelle. Mit dessen Hilfe können Applikationen das Debugging durchführen.

4.2.1.2 JPDA - Interfaces

JVMTI-Schnittstelle

Die JVMTI-Schnittstelle ist eine native Schnittstelle, die von der Virtual Machine(VM) implementiert wird. Über die Schnittstelle können Dienste der VM benutzt werden, die die VM für Applikation wie Debugger oder Monitoring-Programme anbietet. Außerdem können Anfragen für Informationen[11], für Benachrichtigungen [12] oder für Aktionen [13] an die Schnittstelle gestellt werden. Ein Stack Frame wird durch die JVM bei jedem Methodenaufruf erstellt und enthält unter anderem Speicherplatz für die lokalen Variablen einer Methode. Das JVMTI steht dabei

[11] Eine Information wäre zum Beispiel der aktuelle Stack Frame.
[12] Wenn ein Haltepunkt erreicht wird, wird zum Beispiel eine Nachricht an die an der Schnittstelle registrierten Applikationen gesendet.
[13] Eine Aktion ist zum Beispiel das Setzen eines Breakpoints.

zwischen der JVM des Debuggee und dem Back-End, das für die Kommunikation mit der JVM des Debuggers nötig ist.

Das JVMTI ist seit Java 1.5 Teil der Java 2 Platform, Standard Edition (J2SE) und ersetzt das Java Virtual Machine Debug Interface (JVMDI) .

JDI-Schnittstelle

Das Java Debug Interface (JDI) ist eine High-Level Java Schnittstelle und bietet Zugriff auf die Funktionen, die ein Debugger für das Debugging von Java Programmen benötigt. Das JDI bietet Funktionen zur Kontrolle und zum Zugriff auf interne Zustände des Debuggee und darüber hinaus explizite Kontrolle über die Ausführung der JVM. Das Anhalten und Wiederaufnehmen von Threads, das Setzen von Halte- und Überwachungspunkten, die Benachrichtigung bei Ereignissen, wie das Auftreten einer Exception oder das Laden einer Klasse, die Erzeugung eines neuen Threads und das Inspizieren eines angehaltenen Threads oder der lokalen Variablen, sind Funktionen, die das JDI bietet und die von Debuggerapplikationen genutzt werden können. Das JDI ist die höchste Ebene des JPDA und es wird durch das Front-End implementiert.

JDWP

Das Java Debug Wire Protocol (JDWP) definiert das Format der Daten und der Anfragen, die bei der Kommunikation der JVM des Debuggee und der JVM des Debuggers ausgetauscht werden. Für die Kommunikation zuständig sind dabei die beiden Softwarekomponenten, auf der Debuggee JVM das Back-End und auf der Debugger JVM das Front-End. Das JDWP trennt die Prozesse des Debuggee und des Debugger Front-Ends. Dies erlaubt das Ausführen der beiden Prozesse auf verschiedenen JVMs. Auch muss das Front-End nicht in Java geschrieben sein.

Weitere Features

Die Java Virtual Machine unterstützt das Feature, dass während der Ausführung eines Programms, dessen Bytecode aktualisiert werden kann. Es ist also möglich die Instanz einer Klasse, während das Programm ausgeführt wird, mit einer modifizierten, neu kompilierten Version der Klasse auszutauschen. Bei Java wird diese Technik als *HotSwap* bezeichnet. Das JPDA wurde um entsprechende Funktionen und Spezifikationen erweitert, so dass auf allen drei Ebenen (JVMTI, JDWP, JDI) auf die HotSwap-Technik zugegriffen werden kann.

Das HotSwap eröffnet eine neue Möglichkeit, um Hypothesen zur Fehlerursache zu überprüfen. Nach gefundener Ursache kann diese, während die Ausführung unterbrochen ist, eventuell korrigiert werden. Durch Fortsetzen der Ausführung ist eine direkte Validierung möglich. Erneutes Starten des Debuggees ist nicht notwendig [47].

Eine Debugger-Applikation kann an jeder der drei Ebenen ansetzen, jedoch bietet das JDI, für in Java geschriebene Debugger den einfachsten und komfortabelsten Zugriff auf die benötigten Funktionen. Für Debugger in anderen Sprachen ist das Aufsetzen auf dem JDWP die bessere Wahl.

4.2.1.3 Haltepunkte und Schrittmodi in Java

Das Setzen und Entfernen von Haltepunkten ist mit Hilfe des JPDA in einfacher Weise möglich. Dabei unterstützt das JPDA folgende Haltepunkttypen: Zeilenhaltepunkt, Exceptionhaltepunkt, Methodenhaltepunkt, Classpreparehaltepunkt und Classunloadhaltepunkt (genauere

Informationen in Kapitel 3.3.1). Überwachungspunkte (engl. watchpoints) werden auch unterstützt. In Kapitel 2.1.3.1 wurde erwähnt, dass Haltepunkte und Überwachungspunkte mit Bedingungen verknüpft werden können. Dabei wird nur die Zählbedingung direkt vom JPDA unterstützt. Die von JPDA unterstützten Halte- und Überwachungspunkte können mit einem Zähler versehen werden, der um eins reduziert wird, wenn der Halte- oder Überwachungspunkt passiert wurde. Erst wenn der Zähler Null ist, wird der Debugger mittels eines Breakpoint Events informiert und das ausgeführte Programm wird angehalten. Die Codebedingung wird nicht direkt vom JPDA unterstützt, sondern muss von der Debuggerapplikation realisiert werden.

Das JPDA unterstützt mehrere Schrittmodi, die sich in Schritttiefe und Schrittgröße unterschieden werden.. Die Schrittgröße kann STEP_LINE oder STEP_MIN sein. Die Schrittgröße STEP_LINE umfasst eine Zeile des Quellcodes. Erst wenn sich mit der ausgeführten Bytecodeinstruktion (Maschinencodebefehl) auch die Zeilennummer ändert wird der Debugger durch ein Step Event[14] benachrichtigt und der Debuggee wird angehalten. Die zweite Schrittgröße ist STEP_MIN. Soll der Debuggee mit der Schrittgröße STEP_MIN ausgeführt werden, so wird nach jeder Bytecodeinstruktion ein Step Event von der JVM erzeugt und an den Debugger versendet. Die Schrittgröße beträgt hier also eine Maschinencodebefehl-Länge (Bytecodeinstruktion).

Bei den Schritttiefen unterscheidet das JPDA in STEP_INTO, STEP_OVER und STEP_OUT. Diese Schritttiefen entsprechen den erläuterten Schrittmodi in Kapitel 2.1.3.1

4.2.2 Debuginformationen

Das Debugging von Programmen ist nur möglich, wenn das zu debuggende Programm bestimmte Zusatzinformationen enthält. So lässt sich die Grundanforderung **Quellcodedarstellung** nur erfüllen, wenn sich der generierte Maschinencode, hier Bytecodeinstruktionen, auf Quellcode zuordnen und abbilden lässt (siehe Kapitel 2.1.2). Außerdem muss von der Belegung des Speichers in einem Ausführungspunkt auf die Werte der Variablen des Quellcodes rückgeschlossen werden können. In Java wird das realisiert, indem in Java Class-Dateien zusätzliche Attribute installiert werden, die die benötigten Informationen enthalten. Die Attribute werden nur vom Java Compiler generiert, wenn er mit dem Parameter "-g" gestartet wird und ist somit optional. Die Attribute sind `Sourcefile` Attribut, `Linenumbertable` Attribut und `Localvariabletable` Attribut.

Sourcefile Attribut

Das Sourcefile Attribut dient dazu, die Bytecodeinstruktionen dem Ursprung, also der Quellcodedatei zuordnen zu können. Dieses Attribut existiert einmal in einer Class-Datei und enthält den Namen der Quelldatei, aus der die Class-Datei erstellt wurde. Der Name ist der einfache Name der Quellcodedatei (z.B. Main.java) und enthält keine Pfadangaben. Diese sind laut Spezifikation nicht erlaubt. Der Debugger kann diese Information nutzen, um die Maschinencodeversion und die Quellcodeversion eines Programm in Verbindung zu bringen.

[14] Step Events sind Benachrichtigungen, die die JVM erstellt, wenn ein Ausführungsschritt (engl. step) beendet wurde. Ein Step Event wird generiert bevor der Code ausgeführt wird.

Linenumbertable Attribut

Das Linenumbertable Attribut enthält eine Symboltabelle, die ein Zuordnen von Bytecodeinstruktionen und Quellcodezeilen ermöglicht. Die Zeilennummer der Quellcodezeile und die erste aus ihr generierte Bytecodeinstruktion ergeben einen Eintrag in der Symboltabelle. Ein Compiler erstellt für jede Methode einer Klasse ein Linenumbertable Attribut.

```
1 public class Main
2 {
3   public void test
4   {
5     int i = 0;
6     i = i + 3;
7   }
8 }
```

(a) Java Quellcode

```
public void test()
Code(max_stack = 1, max_locals = 2, code_length = 6)
0:    iconst_0
1:    istore_1
2:    iinc          %1      3
5:    return
```

(b) Java Bytecode

Bytecodeinstruktion	Quellcodezeile
0	5
2	6
5	7

(c) Symboltabelle

Abbildung 4.7: Beispiel von Java Quellcode (4.7a), dessen generierten Maschinencode (4.7b) und der Symboltabelle des Linenumbertable Attribut (4.7c)

Anhand des Beispiels in Abbildung 4.7 lässt sich die Zuordnung von Byte- und Quellcode nachvollziehen. In der Tabelle 4.7c ist zu erkennen, dass die Bytecodeinstruktionen 0 und 1 der Quellcodezeile 5 zu zuordnen sind. Bytecodeinstruktionen 2 bis 4 sind der Zeile 6 und Bytecodeinstruktionen 5 der Quellcodezeile 7 zu zuordnen.

Um die Bytecodeinstruktionen den Zeilen in der Quellcodedatei zuordnen zu können, werden das Sourcefile Attribut und das Linenumbertable Attribut verwendet. Die Zuordnung ermöglicht einerseits das Setzen von Zeilenhaltepunkten im Quellcode und andererseits die Einzelschrittausführung auf dem Quellcode.

Localvariabletable Attribut

Die Zuordnung von Variablen auf ihren Quellcode geschieht durch das Localvariabletable Attribut. Jede Methode hat ein Localvariabletable Attribut, welches den Quellcodenamen der lokalen Variablen und deren Gültigkeitsbereich enthält. Nur dadurch ist es möglich den Zustand der Variablen zu betrachten und den Programmzustand während des Debuggings zu manipulieren. Die Darstellung von Variablen in den verschiedenen **Sichten** auf ein Programm ist in Java nur möglich, weil über die Referenzen in der Localvariabletable auf die jeweiligen Speicherorte der aktuelle Werte der Variablen zugegriffen werden kann.

Codemarkierung

Für das Debugging ebenfalls von Bedeutung ist die Möglichkeit, Bytecode als generiert zu markieren. Das *synthetic*-Attribut, das Code als generiert markiert, kann Klassen, Methoden

und Felder markieren. Die Markierung einzelner Instruktionen oder lokaler Variablen ist nicht möglich [47].

4.2.3 Weitere Java Debuginformationen - SourceDebugExtension Attribut

Der Java Specification Request 45 (JSR-045) sieht vor, Debugging von Programmen zu unterstützen, die nicht in Java geschrieben wurden, aber die Java Virtual Machine als Ausführungsumgebung nutzen. Im November 2003 wurde der Java Specification Request 45 vom Standardisierungskomitee Java Community Process (JCP) [5] fertig gestellt und ist seit Java Version 1.4 fester Bestandteil der JVM und des JPDA. Java Server Pages, Jython aber auch aspektorientierte Sprachen wie Object Teams/Java, AspectJ oder CaesarJ sind Programmiersprachen, deren Übersetzungswerkzeuge[15] Java Maschinencode (Bytecode) erstellen, der in der JVM ausgeführt werden kann. Die Zuordnung von Bytecode- und Quellcodeinstruktion ist bei solchen Sprachen nur mit Hilfe vom Sourcefile und Linenumbertable Attribut nicht möglich. Die Erfüllung der Grundanforderungen **Quellcodedarstellung** und **Kontrolle durch Haltepunkte** ist nur mit Bereitstellung einer weiteren Debuginformation in der Class-Datei möglich - dem SourceDebugExtension Attribut. [7]

Der Standard, der aus dem Java Specification Request 45 entstand, nimmt Einfluss auf das Übersetzungswerkzeug der Sprache, auf die vom Übersetzungswerkzeug erzeugte Class-Datei, auf die Java Virtual Machine, auf die Java Platform Debugger Architecture und nimmt Einfluss auf die Debuggerapplikation an sich.

Die JSR-045 Spezifikation fokussiert ausschließlich die Zuordnung von Quellcodeinstruktionen zu Bytecodeinstruktionen. Das Problem der Zuordnung von Variablen wird explizit nicht betrachtet und auf einen eventuell folgenden JSR verschoben.

In der JSR-045 Spezifikation werden drei Szenarien erläutert, wie eine Übersetzung vom ursprünglichen Quellcode zum Java Bytecode geschehen kann. Diese werden jetzt erläutert, vorher werden aber einige Begriffe eingeführt.

language processor - Ein *language processor* übersetzt *translated Source* in *final source* oder zu eine anderen translated Source.

translated Source - *Translated Source* bezeichnet den Quellcode, der durch einen language processor oder ein anderes Übersetzungswerkzeug in eine andere Sprache oder Form übersetzt wird.

final Source - Die **final source** bezeichnet den Quellcode (meist Java Quellcode), der von einem Compiler in Maschinencode übersetzt wird.

Source Map/SMAP - Die **Source Map** enthält die Zuordnungsinformationen vom ursprünglichen Quellcode zum übersetzten Maschinencode.

Stratum - Bei mehreren Übersetzungsschritten (siehe Szenario III) werden mehrere Quellcodeebenen erzeugt, die Strata genannt werden. Ein *Stratum* ist eine Ebene aus der Menge der verschiedenen Quellcodeebenen.

[15] Übersetzungswerkzeuge können Compiler oder eine Kombination aus Präprozessor, Compiler und Postprozessor sein.

Szenario I: Direktes Erstellen der Class-Dateien - Beim *Direkten Erstellen der Class-Dateien* übersetzt ein Übersetzungswerkzeug den Quellcode direkt in eine Class-Datei. Dabei erstellt das Übersetzungswerkzeug gleichzeitig eine Source Map (siehe Abbildung 4.8). Das Übersetzungswerkzeug installiert ein zusätzliches Attribut (SourceDebugExtension Attribut) in die Class-Datei, welches wiederum die Source Map enthält. Die Class-Datei (Java Bytecode) wird nun in die JVM geladen und ausgeführt. Wenn das Programm im Debug Modus ausgeführt wird, kommuniziert der Debugger über Schnittstellen des JPDAs mit der JVM und liest die Source Map aus dem SourceDebugExtension Attribut der Class-Dateien aus, um sie auszuwerten.

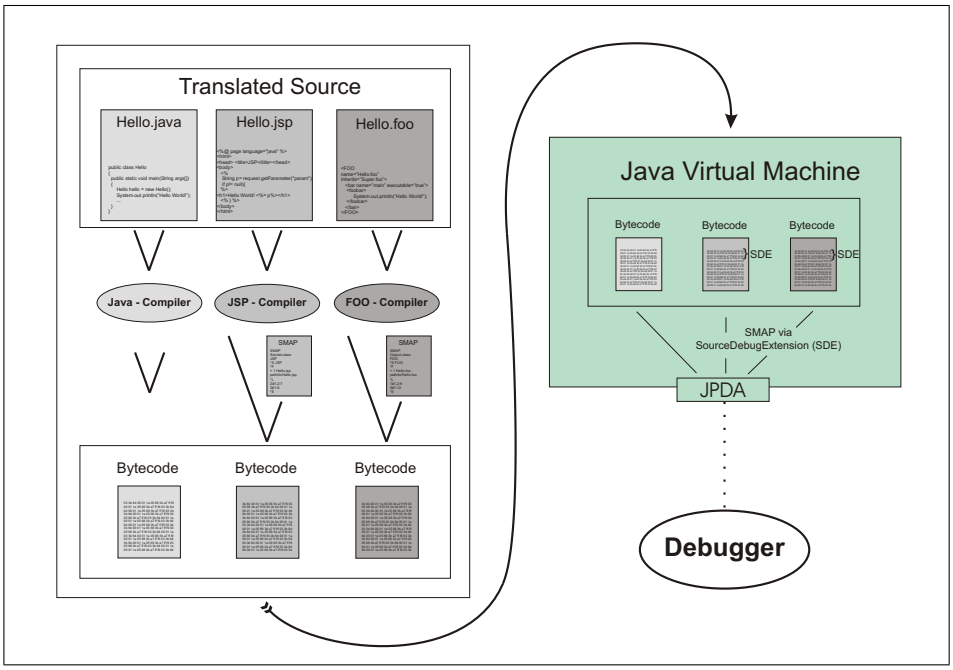
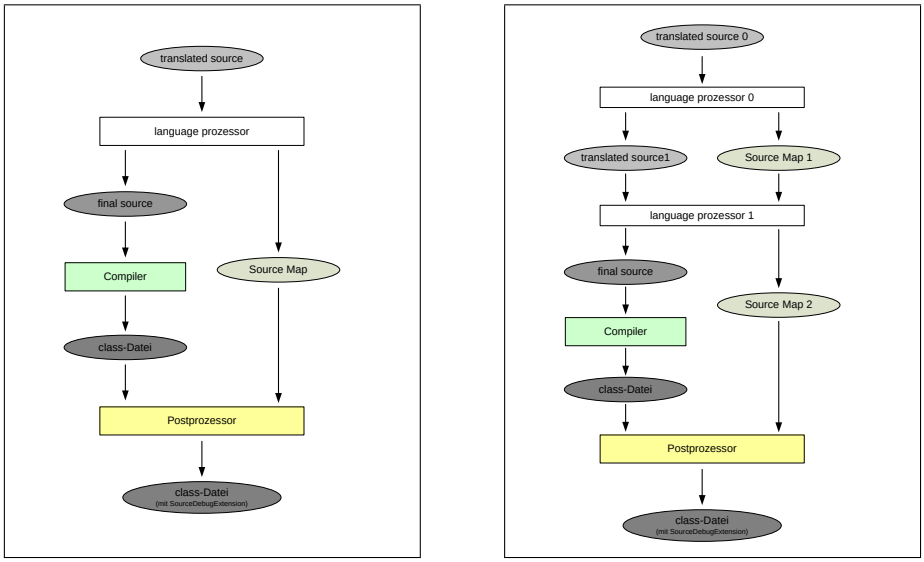


Abbildung 4.8: Direktes Erstellen der Class-Dateien und deren Ausführung

Szenario II: Einmaliges Übersetzen - Beim *Einmaligen Übersetzen* wird der Bytecode nicht direkt aus dem ursprünglichen Quellcode erzeugt, sondern vorher durch einen language processor in einen anderen Quellcode (final Source) transformiert. Der language processor hat neben der Transformation der transled Source auch die Aufgabe der Erzeugung der Source Map in eine separate Datei. Nachdem die final Source in Maschinencode übersetzt wurde, installiert ein Postprozessor den Inhalt der Source Map Datei in den Maschinencode. Dabei wird das SourceDebugExtension Attribut in die Class-Datei hinzugefügt.

Szenario III: Mehrmaliges Übersetzen - Beim *Mehrmaligen Übersetzen* sind beim Erstellen der final Source mehrere Übersetzungschritte nötig. Die translated Source wird zum Beispiel vom language processor1 in einen Quellcode der Sprache Foo übersetzt. Dabei erzeugt der language processor1 auch eine Source Map. Der Quellcode der Sprache Foo wird nun von einem language processor2 in Quellcode der Sprache Bar transformiert und ebenfalls erzeugt der language processor2 eine Source Map für diesen Quellcode. Der Quellcode der Sprache Bar wird nun in die final Source übersetzt und der daraus resultierende Maschinencode erzeugt. Das Debugging eines Programms in diesem Szenario muss



(a) Einmaliges Übersetzen (b) Mehrmaliges Übersetzen

Abbildung 4.9: Einmaliges und mehrmaliges Übersetzen

nicht nur auf der ursprünglichen Quellcodeebene möglich sein, sondern auch auf jedem Quellcode der einzelnen Übersetzungsschritte. Um das Debugging auf den verschiedenen Ebenen zu gewährleisten, müssen die verschiedenen Source Maps vereint und aufgelöst werden, da nur eine Source Map in die Class-Datei geschrieben werden darf. Die Source Map enthält in diesem Szenario vier Strata, das Stratum der translated Source, das Stratum der Sprache Foo, das Stratum der Sprache Bar und das Stratum der final Source.

SourceDebugExtension Attribut

Das SourceDebugExtension Attribut wurde durch JSR-045 eingeführt und beinhaltet die Source Map, die komplexere Zuordnungen von Bytecode zu Quellcode erlaubt, als sie mit dem Sourcefile und dem Linenumbertable Attribut möglich wären. Die Source Map erlaubt die Zuordnung von einer Maschinencodeebene zu mehreren Quellcodeebenen (Strata). Eine Bytecodeinstruktion kann dadurch einmal pro Stratum zugeordnet werden. Das bedeutet, dass diese Bytecodeinstruktion n Quellcodezeilen zugeordnet werden kann. Des weiteren ist es durch die Source Map möglich Quellcodedateien anzusteuern, die keine Javadateien sein müssen. Die Beschränkungen des Sourcefile Attributs gelten bei der Source Map nicht und so können Quellcodedateien angesteuert werden, die sich in einem beliebigen Verzeichnis befinden können. Durch die Source Map können zu einer Class-Datei auch mehrere Quelldateien pro Quellcodeebene zugeordnet werden. Eine Class-Datei kann also pro Stratum auf mehrere tranlated source-Dateien verweisen.

Die Beschränkungen von Sourcefile und Linenumbertable Attribut, die Zuordnungen nur für Java Quellcode zu Java Bytecode erlauben, wurden vom SourceDebugExtension Attribute hinreichend erweitert. Dass es trotz SourceDebugExtension Attribut noch zu Problemen bei der Zuordnung von Quellcode und Java Maschinencode kommen kann, wird in folgenden Kapiteln noch erläutert.

4.3 Warum ist das Debugging von aspektorientierten Programmen schwierig?

Um diese Frage hinreichend beantworten zu können, muss sie aus allen Blickwinkeln betrachtet werden. Der Entwickler eines Debuggers für aspektorientierte Programme wird die Frage wahrscheinlich anders beantworten, als ein Benutzer dieses Debuggers. Beide Betrachtungsweisen geben aber Auskunft über mögliche Schwierigkeiten.

4.3.1 Entwicklersicht

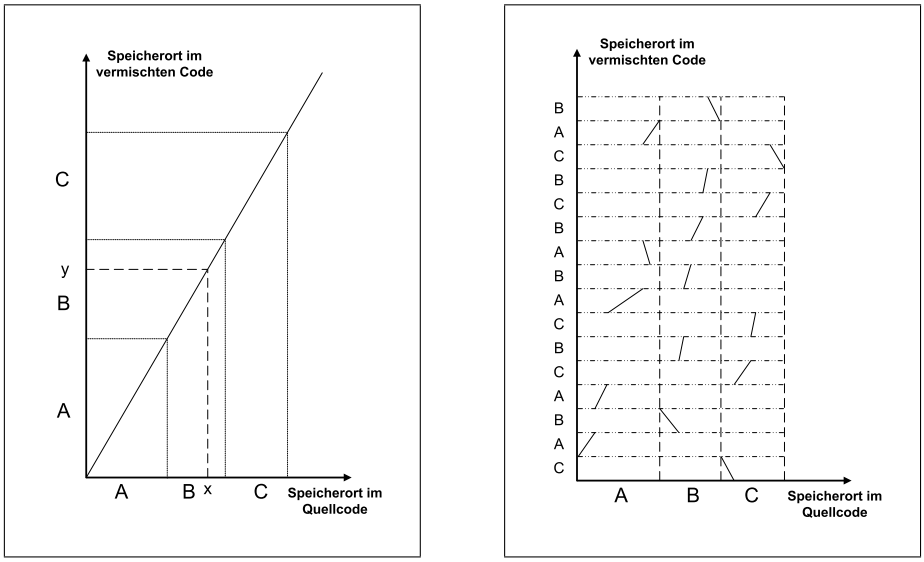
Bei der Entwicklung eines interaktiven Debuggers für aspektorientierte Programme ist die Erfüllung der in Kapitel 3.1 definierten Grundanforderungen unabdingbar. Um die Grundanforderungen für einen AOP-Debugger umsetzen zu können, muss besonders auf die technischen Besonderheiten der Aspektorientierung eingegangen werden. Nach Betrachtung der Kapitel 4.1.2 und 4.1.3 werden die Probleme ersichtlich, die besonders bei der Grundanforderung **Quellcodedarstellung** auftreten - vermischter Code.

Vermischter Code

Durch die Sprachkonzepte der einzelnen aspektorientierten Sprachen und den Webetechniken kommt es fast immer zu transformiertem[16] Maschinencode. Die Zuordnung des transformierten Maschinencodes zu dem entsprechenden Quellcode kann sich sehr schwierig gestalten. In einer Dissertation über generatives Programmieren [21] wird ein zweidimensionales kartesisches Koordinatensystem (siehe Abbildung 4.10) des Autors Vandevoorde [52] vorgestellt, um die Komplexität visuell darzustellen. Vandevoordes Modell zeigt Zuordnungsbeziehungen zwischen den Speicherorten (engl. locations) im High-Level-Code (in unserem Fall Quellcode) und den Speicherorten im Low-Level-Code (in unserem Fall Bytecode). Die horizontale Achse des Koordinatensystems stellt die Speicherorte (zum Beispiel Zeilennummern) im High-Level-Code dar und die vertikale Achse repräsentiert die Speicherorte im Low-Level-Code (vermischter Code). Das Diagramm 4.10a zeigt eine lineare Transformation zwischen den Speicherorten im Quellcode und den des transformierten Codes. Wären A, B und C zum Beispiel Methoden in Standard Java implementiert (horizontale Achse) und würden sie in Maschinencode übersetzt werden (vertikale Achse), so würde sich der Quellcode an Stelle x der Stelle y im Maschinencode zuordnen lassen. Der Quellcode an Stelle x+1 würde sich dann der Maschinencodestelle y+1[17] zuordnen lassen und so weiter. Das Diagramm 4.10b verdeutlicht das Problem des vermischten Codes. Wären A, B und C abstrakte Aspektbeschreibungen in einer AOP-Sprache und würden sie durch ein Übersetzungswerkzeug transformiert und durch einen Weber gewoben werden, so könnte Maschinencode entstehen, wie er in der vertikalen Achse der Abbildung 4.10b zu sehen ist. Quellcode der Stelle x würde sich nun der Maschinencodestelle y zuordnen lassen. Durch die nonlineare Transformation müsste sich der Quellcode der Stelle x+1 nicht der Maschinencodestelle y+1, sondern der Stelle y-10 oder der Stelle y+5 im vermischten Code zuordnen lassen.

[16] In der Literatur wird oft nur vom transformierten Code gesprochen. Eigentlich ist vermischter Code gemeint. In den Erklärungen wird beides synonym verwendet.

[17] x+1 stellt hier jetzt nicht die nächst folgende Zeilennummer dar und y+1 nicht die nächst folgende Bytecodeinstruktion. Es soll nur verdeutlichen, das sich Quellcode und Maschinencode linear abbilden lassen.



(a) Lineare Transformation (b) Nicht lineare Transformation

Abbildung 4.10: Zuordnungsbeziehungen zwischen Quellcode und transformierten Code

Das Problem des vermischten Codes beschränkt sich aber nicht nur auf die Grundanforderung der **Quellcodedarstellung**, sondern betrifft auch die Grundanforderungen **Kontrolle durch Haltepunkte und Schrittmodi**. Zeilenhaltepunkte werden im Quellcode gesetzt. Kann die Stelle des Quellcode nicht der richtigen Maschinencodestelle zugeordnet werden, so hält das Programm gar nicht oder an einer falschen Stelle. Auch ist das schrittweise Ausführen des Programms nur möglich, wenn auch die Quellcodedarstellung funktioniert.

Infrastrukturcode

Wird das Problem des vermischten Code nicht als Problem der Zuordenbarkeit, sondern unter Einbeziehung der in Kapitel 4.1.1 erläuterten Generatorentechniken betrachtet, so wird ein neues Problem ersichtlich - Infrastrukturcode. Infrastrukturcode ist Maschinencode, der keine Quellcodeentsprechung besitzt. Klassen, Methoden, einzelne Statements, Felder oder lokale Variablen - alles kann Infrastrukturcode sein, dessen Aufgabe die Zusammenarbeit von Basiscode und Aspektcode ist. Das Übersetzungswerkzeug und/oder der Weber können Infrastrukturcode erzeugen, für den es keine Quellcodezuordnung in der Ebene der AOP-Sprachen gibt. Der Infrastrukturcode stellt also bei der Entwicklung eines Debuggers für aspektorientierte Programme ein Problem dar. Dieser Code beeinflusst besonders die Erfüllung der Grundanforderung **Visualisierung**. Alle Sichten auf ein Programm, besonders aber die Quellcodesicht und die Variablensicht, müssen den Infrastrukturcode filtern und ihn für den Benutzer transparent machen[18].

[18] Das Ermöglichen des Debuggings auf verschiedenen Codeebenen wird im Kapitel 7 näher beschrieben. Das Debugging auf einer der Transformationsebenen würde den Infrastrukturcode nicht filtern, sondern ihn in den Sichten anzeigen. Bei dem Debugging auf einer Transformationsebene würde dann nicht das aspektorientierte Programm, sondern der Codegenerator (Übersetzungswerkzeug oder Weber) im Vordergrund stehen.

Antwort des Entwicklers

Die Probleme *vermischter Code* und *Infrastrukturcode* treten bei AOP-Sprachen auf, die generative Übersetzungswerkzeuge bzw. Codetransformation als Webetechnik einsetzen. Bei diesen Sprachen gestaltet sich die Entwicklung eines Debuggers also schwieriger als bei AOP-Sprachen, die dynamische Webetechniken nutzen, also keinen vermischten Code oder Infrastrukturcode erzeugen. Die Frage lässt sich also nicht global für alle aspektorientierten Sprachen einheitlich beantworten und wird in Kapitel 5.3 näher untersucht.

4.3.2 Benutzersicht

Das es bei der Entwicklung eines Debuggers für aspektorientierte Programme zu Schwierigkeiten kommen kann, wird in der Entwicklersicht beschrieben. Geht man davon aus, dass die Probleme von vermischten Code, Infrastrukturcode und Zuordenbarkeit gelöst seien und die Grundanforderungen durch den entstandenen AOP-Debugger erfüllt wären, so stellt sich jetzt die Frage, ob ein Benutzer mit diesem Werkzeug in der Lage wäre, entwickelte aspektorientierte Programme nachzuvollziehen und Fehler zu finden.

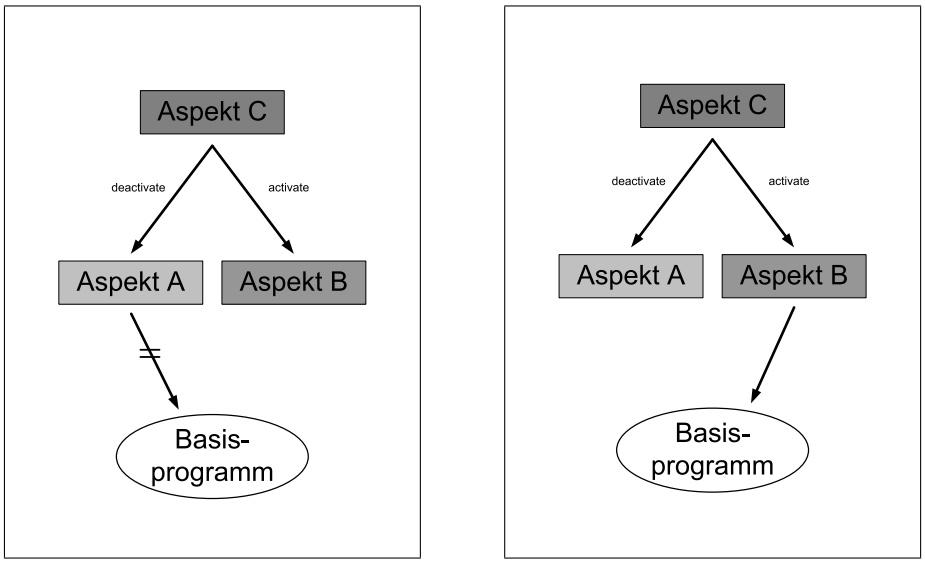


Abbildung 4.11: Gleichzeitige Aspektaktivierung und -deaktivierung

Die Abbildung 4.11 zeigt gleich zwei Sprachfeatures, die die Nachvollziehbarkeit eines Programms und somit auch das Debugging aus Benutzersicht erschweren, die *Dynamische Aspektaktivierung* und das Feature *Aspekte von Aspekten*.

Dynamische Aspektaktivierung

In Kapitel 4.1.4 wurde erläutert, dass aspektorientierte Sprachen entweder durch eigene Aktivierungsmechanismen oder durch explizites Ausprogrammieren von Bedingungen dynamische Aspektaktivierung zur Laufzeit unterstützen. In Abbildung 4.11 wird der Fall illustriert, dass

zwei verschiedene Aspekte (Aspekt A und Aspekt B) in die Basisapplikation gewoben wurden und der zuvor aktive Aspekt A deaktiviert und der zuvor inaktive Aspekt B zur Laufzeit aktiviert wird. Dieses Beispiel zeigt, dass sich das Verhalten der adaptierten Basisapplikation während der Laufzeit vollkommen ändern kann. Ohne Zusatzinformationen über Aktivierung und Deaktivierung der Aspekte ist es für den Benutzer kompliziert das Verhalten der Applikation nachzuvollziehen. Und da das Fehlerfinden immer das Verstehen des Programms voraussetzt, müssen in den Bereichen **Visualisierung** und **Kontrolle durch Haltepunkte** Erweiterungen vorgenommen werden. Der Bereich der Visualisierung muss um eine Sicht erweitert werden, die die gewobenen Aspekten und deren Aktivierungszustand[19] darstellt. Dadurch sieht der Benutzer, welche Aspekte die Basisapplikation beeinflussen. Des weiteren sollte der Bereich der **Kontrolle durch Haltepunkte** um einen weiteren Haltepunkttyp erweitert werden - den Aktivierungshaltepunkt. Durch Setzen dieses Haltepunkts, hält das Programm, immer wenn ein Aspekt aktiviert und/oder programmatisch deaktiviert wurde.

Diese Erweiterungen helfen bei der Erfüllung der Grundanforderungen und damit dem Benutzer, um der Komplexitätsteigerung durch dynamische Aspektaktivierung entgegenzuwirken.

Aspekt von Aspekten

Wie in Abbildung 4.11 zu sehen ist, hat Aspekt C Einfluss auf Aspekt A und Aspekt B. In diesem Beispiel aktiviert und deaktiviert der Aspekt C die beiden Aspekte A und B. Würde der Aspekt C nicht nur den Aspekt B aktivieren, sondern auch die vorhandenen Bindungen zwischen Aspekt B und der Basisapplikation verändern, so spricht man von einem Aspekt eines Aspekts. Behandelt ein Aspekt einen anderen Aspekt wie ein Basisprogramm, so bezeichnet man das den Aspekt eines Aspekts. Wenn eine AOP-Sprache diese Konstellation erlaubt, so erreicht eine Applikation mit Aspekten von Aspekten ein weiteres Komplexitätslevel. Auch hier muss der Bereich der Visualisierung erweitert werden. Die Sicht, die der Komplexitätsteigerung bei der dynamischen Aspektaktivierung entgegenwirkt, lässt sich auch hier verwenden. Dabei werden die von einem Aspekt adaptierten Aspekte speziell gekennzeichnet, um sie für den Benutzer visuell hervorzuheben.

Antwort des Benutzers

Um die Benutzbarkeit eines interaktiven Debuggers für aspektorientierte Programme zu gewährleisten, müssen in den einzelnen Bereichen der Grundanforderungen Erweiterungen umgesetzt werden. Die Information über Aspektaktivierung und -deaktivierung bedarf dabei besonderer Darstellung.

4.3.3 Fazit

Die Schwierigkeiten bei der Entwicklung eines AOP-Debugger variieren sehr stark zwischen den aspektorientierten Sprachen. Dabei sind die Techniken der Übersetzungswerkzeuge und der Weber von besonderer Bedeutung. Aber auch die Konzepte des aspektorientierten Programmierparadigmas beeinflussen die Entwicklung eines AOP-Debugger. Allgemeine Konzepte, wie Aspektaktivierung zur Laufzeit müssen in den Bereichen der Grundanforderungen behandelt werden. Wie die Einflüsse auf die Grundanforderungen bei zwei aspektorientierten Sprachen aussieht, beschreibt das nächste Kapitel.

[19] Der Aktivierungszustand eines Aspekts ist aktiv oder inaktiv. Einige Sprache bieten noch feinere Unterscheidungen der Aspektaktivierung.

5 Einflüsse auf die Entwicklung von Debuggern für AOP-Sprachen

Im vorherigen Kapitel wurden technische Grundlagen bei der Umsetzung der aspektorientierten Konzepte erläutert und Schwierigkeiten dargestellt, die sich bei der Entwicklung eines Debuggers für aspektorientierte Programme ergeben. Auf Besonderheiten der Programmiersprache wurde dabei nur am Rand eingegangen. In diesem Kapitel werden die Sprachkonzepte und deren technische Umsetzung, sowie der verwendete Webemechanismus zweier aspektorientierter Vertreter näher beleuchtet und deren Einfluss auf die Entwicklung eines Debuggers in den Bereichen der Grundanforderungen (siehe Kapitel 3.1) genau dargestellt. Am Ende des Kapitels wird eine Übersicht über die bekanntesten Webetechniken und deren Auswirkungen auf Debuggerentwicklungen gegeben.

5.1 Object Teams/Java

Die Sprache Object Teams/Java gehört zu den Sprachen, die in dieser Diplomarbeit im Fokus stehen. Zum Einen erweitert sie die Sprache Java um die Konzepte der Aspektorientierung und zum Anderen erzeugen Übersetzungstechnik und Webetechnik Standard Java Maschinencode, der in einer Standard Virtual Machine ausgeführt werden kann. Diese Sprache wird jetzt näher vorgestellt:

In einer für die *Net.Objectdays*[1] eingereichten Veröffentlichung [34], wird das Programmiermodell *Object Teams* vorgestellt. Es erweitert OO um ein rollenbasiertes Kollaborationsmodell und führt Sprachkonzepte ein, die die Entwicklung von wiederverwendbaren, durch Vererbung erweiterbaren Modulen (Teams) ermöglichen, die größer als Klassen sind. Teams sind Container für Rollenklassen, in denen die Bindungen an die Basisapplikation auf Typ- und Methodenebene deklariert werden.

5.1.1 Object Teams/Java Sprachkonzepte

Die Sprachkonzepte, die Object Teams/Java bietet, werden im Folgenden näher erläutert. Dabei wird hauptsächlich auf die Konzepte eingegangen, die für diese Arbeit vor Bedeutung sind.

[1] Net.Objectdays ist eine jährlich stattfindende Softwaretechnik-Konferenz.

5.1.1.1 Team

Teams stellen bei Object Teams das Kernkonzept dar und ermöglichen die Kollaboration[2] mehrerer Klassen. Diese Klassen sind *Rollen* und leben nur im Kontext ihres Teams. Team-Klassen werden mit dem Schlüsselwort `team` deklariert und erben automatisch von der root-Klasse aller Teams - dem `org.objectteams.Team` . Teams sind Java-Klassen, bieten aber zusätzliche Funktionen und Features. Die Aufgabe eines Teams ist die Verwaltung und Koordination seiner Rollen. Dabei sind die Rollen von "außen" nur über Teamfelder und -methoden ansprechbar. Da die Interaktion zwischen den Rollen eines Teams auch über Teamfelder und -methoden passiert, vereinigt ein Team die Funktionen der Design-Pattern Facade und Mediator [37].

Listing 5.1: Beispiel eines Teams mit zwei Rollen

```
1 public team class SampleTeam
2 {
3         public class SampleRole playedBy SampleBase{ ... }
4         public class AnotherSampleRole { ... }
5
6         // Beispielmethode mit deklarativen Lifting
7         public void sampleMethod(SampleBase as SampleRole sampleRole) { ... }
8 }
```

Deklaratives Lifting

Im Listing 5.1 ist in der Zeile 7 eine Teammethode zu sehen, die einen Rollentyp und einen Basistyp als Parameter besitzt. Die Methode `sampleMethod` besitzt eigentlich nur eine Instanz des Basistyps als Parameter und beim Aufruf der Methode wird die Instanz des Basistyps zu einer Instanz des Rollentyps *geliftet*. Diesen Vorgang wird in Object Teams *Deklaratives Lifting* genannt (siehe Kapitel 5.1.1.2). Innerhalb der Methode kann in diesem Beispiel auf die durch das Lifting entstandene Rolleninstanz mittels `sampleRole` zugegriffen werden.

Teamaktivierung

Object Teams/Java bietet einen Aktivierungsmechanismus der Aspekte an. Dieser erlaubt es Teams zu aktivieren und zu deaktivieren. Es gibt implizite und explizite Möglichkeiten der Aktivierung, die im Kapitel 5.1.3 im Kontext des Debuggings näher erläutert werden.

nested Teams

Teams und Rollen können in Object Teams/Java auch verschachtelt sein. Eine Rolle, die den Modifier `team` erhält, ist gleichzeitig ein Team und kann wiederum selbst Rollen enthalten. Eine solche gemischte Klasse, die Rollen- und Teamfeatures in sich vereint heißt *nested Team*.

5.1.1.2 Rolle

Rollen sind innere Klassen von Teams und können auf die Referenz ihres Teams mittels `Team.this` zugreifen. Jede in einem Team deklarierte Klasse ist automatisch eine Rolle dieses Teams. Rollen können durch das Schlüsselwort `playedBy` an eine andere Klasse, im Folgenden Basisklasse genannt, gebunden werden (siehe Listing 5.1).

[2] Arbeiten mehrere Klassen zusammen und dienen dabei einem gemeinsamen Zweck wird das als Kollaboration bezeichnet.

Rollendateien

Rollen können auch in separaten Dateien deklariert werden. Das Listing 5.2 zeigt eine Deklaration einer Rolle in einem *Role File*. Wie in Zeile 1 zu sehen ist, muss die `package`-Deklaration das Schlüsselwort `team` und den Namen des Teams enthalten, damit die deklarierte Rolle dem angegebenen Team zugeordnet werden kann.

Listing 5.2: Beispiel einer externen Rolle

```
1 team package SampleTeam;
2
3 public class SampleRole playedBy SampleBase { ... }
```

Rolle-Basis-Beziehung

Um die Funktionalität eines Teams mit einer Anwendung zu verknüpfen, werden seine Rollen an Basisklassen gebunden. Basisklassen sind Klassen der adaptierten Anwendung, deren Verhalten durch die Rollen, die sie adaptieren, verändert oder erweitert werden kann. Das Binden einer Rollenklasse an eine Basisklasse passiert mit dem Schlüsselwort `playedBy` (siehe Zeile 3 des Listings 5.1). Zur Laufzeit werden Instanzen der Basisklasse zu Instanzen der Rollenklasse geliftet. Aber auch die Umkehrung - *das Lowering* ist möglich. Die Umwandlung von Rolle zu Basis und Basis zu Rolle passiert auf Objektebene und implizit. Das bedeutet, dass die Umwandlung nicht explizit durch den Programmierer vorgenommen wird, sondern, dass das Typsystem die Umwandlung zu gegebener Zeit automatisch durchführt. Um das Verhalten einer Basis verändern oder erweitern zu können, muss eine Bindung auf Methodenebene möglich sein. Dabei wird bei Object Teams zwischen zwei Methodenbindungen, abhängig von der Bindungsrichtung unterschieden:

Listing 5.3: Beispiel einer Callout-Methodenbindung

```
1  public class SampleRole playedBy SampleBase
2  {
3    ...
4    public abstract void roleMethod1();
5    roleMethod1 → baseMethod1;
6
7    //Kurzform einer Callout-Deklaration
8    void roleMethod2() → void baseMethod2();
9    ...
10 }
```

Callout - Aspekt-Basis-Richtung

Eine Callout-Methodenbindung bewirkt einen Aufruf der Basismethode aus der Rolle heraus. Ein Rollenmethodenaufruf *delegiert* den Methodenaufruf dabei an die Methode der Basis.

Callin - Basis-Aspekt-Richtung

Die Aufrufrichtung beim Callin ist gegenüber dem Callout umgekehrt - ein Aufruf der Basismethode bewirkt einen Aufruf der Rollenmethode. Ein oder mehrere Basismethoden werden an eine Rollenmethode gebunden und ein *Modifier* gibt dabei den Zeitpunkt des Rollenmethodenaufrufs an:

- before - Der Rollenmethodenaufruf soll vor der Ausführung der Basismethode passieren.
- after - Nach der Ausführung der Basismethode wird die Methode der gebundenen Rolle aufgerufen.
- replace - Die Ausführung der Basismethode wird durch die Ausführung der Rollenmethode ersetzt.

Eine Rollenmethode, die mit `replace` an eine Basismethode gebunden wird und sie damit überschreibt, wird mit dem Schlüsselwort `callin` gekennzeichnet. Innerhalb dieser Methode sind *base calls* möglich, die die ursprüngliche Basismethode aufrufen.

Listing 5.4: Beispiel von Callin-Methodenbindungen

```
1  public class SampleRole playedBy SampleBase
2  {
3    ...
4    public void roleMethod3() {...}
5    roleMethod3 ← before baseMethod3;
6
7    public void roleMethod4() {...}
8    roleMethod4 ← after baseMethod4, baseMethod6;
9
10   public void roleMethod5()
11   { ...
12     base.roleMethod5();
13     ...
14   }
15   roleMethod5 ← replace baseMethod5;
16   ...
17 }
```

Parametermapping

Um bei Methodenbindungen Parameter und Rückgabewerte der Rollen- und Basismethoden aufeinander abbilden zu können, gibt es Parametermappings.

5.1.1.3 Vererbungbeziehungen

Rollen und Teams sind Java-Klassen[3] und können deshalb Vererbungsbeziehungen durch das Java Schlüsselwort `extends` eingehen. Teams aber dürfen nur von Klassen erben, die ebenfalls Teams sind. Hat ein Team keine explizite Vererbungsbeziehung, so erbt es automatisch von `org.objectteams.Team`. Durch Object Teams wird aber noch eine neue Vererbungsbeziehung eingeführt - die *implizite Vererbung* (Erklärung siehe unten).

Die Vererbungsbeziehung bei Rollen[4] unterliegt dabei keinen Beschränkungen. Durch implizite Vererbung und objektbasiertes Erben der Basisklassen, kann eine Rolle Features aus drei Richtungen erben. Damit eine Rolle auch alle drei Richtungen referenzieren kann, existieren die Schlüsselworte `super`, also die Referenz auf die reguläre Superklasse, `tsuper`, die Referenz auf die implizite geerbt Superrolle und `base`, die Referenz auf die Basisklasse.

[3] Java-Klassen mit zusätzlichen Eigenschaften, wie aus den bisherigen Sprachkonzepterklärungen ersichtlich ist.
[4] Hier sind einfache Rollen und keine nested Teams gemeint.

Implizite Vererbung

Geht ein Team eine Vererbungsbeziehung mit einem anderen Team ein, so werden die Rollen des Oberteams mit geerbt. Die implizite Vererbung basiert dabei auf Namensgleichheit. Existiert im Oberteam eine Rolle `SampleRole` und wird nun im erbenden Team ein Rolle mit dem selben Namen definiert, so erbt die Rolle des erbenden Teams alle Features[5] der gleichnamigen Rolle des Oberteams.

5.1.2 ObjectTeams/Java - Übersetzungstechnik, Webetechnik und deren Einfluss auf die Entwicklung eines Debuggers

Wie im vorangegangenen Kapitel ersichtlich ist, bietet Object Teams eine Vielzahl von Sprachkonzepten, die die Sprache Java erweitern. Das Übersetzungswerkzeug, das diese Konzepte in ausführbaren Code transformiert ist der Object Teams Compiler. Dieser generiert aus OT/J-Quellcode direkt Java Bytecode, ohne vorher Java Quellcode zu generieren. Die Realisierung der Sprachfeatures in Java und ihre Auswirkungen beim Benutzen eines interaktiven Java-Debuggers liegen nun im Fokus.

5.1.2.1 Der Object Teams/Java Compiler

Der Object Teams Compiler generiert zusätzlichen Code, um die Sprachfeatures zu realisieren und gehört damit zu den oblique Generatoren (siehe Kapitel 4.1.1). Zu diesem Code gehört unter anderem Infrastrukturcode, wie er im Kapitel 4.3.1 schon beschrieben wurde und führt zu Beeinträchtigungen beim Debuggen durch interaktive Java-Debugger. Um das Debuggen zu ermöglichen, müssen die Sprachfeatures einzeln betrachtet und Codetypen[6], bei deren Realisierung in Java identifiziert werden. Der Compiler muss den Code nach bestimmten Kriterien markieren, damit die Debuggerapplikation den Code in geeigneter Weise behandeln kann.

Um den vom Compiler generierten Bytecode für Erklärungszwecke menschenlesbar darzustellen, wird er mittels Standard Java Decompiler zurück in Quellcode transformiert. Dieser Quellcode ist jetzt aber nicht mehr OT/J-Quellcode, sondern Standard Java Quellcode. Die Object Teams Laufzeitumgebung verändert den Bytecode noch einmal. Die Veränderung fließen bei der jetzigen Betrachtung mit ein. Das Weben und die damit verbundene Codetransformation werden erst im Kapitel 5.1.3 näher erläutert.

Sprachfeatures und deren Realisierung

Das Aufzeigen der Realisierungen aller Sprachfeatures würde den Rahmen dieser Diplomarbeit sprengen und war auch schon Thema in den anderen Diplomarbeiten ([14], [54], [37]). Deshalb wird nur **eine** Transformation des Compilers (Callout-Methodenbindung) näher erläutert und steht beispielhaft für alle anderen. In der Tabelle 5.2 wird ein Überblick über die Realisierungen der Object Teams Sprachfeatures gegeben.

Realisierung der Sprachfeatures am Beispiel der Callout-Methodenbindung

Im Listing 5.5 ist ein Team dargestellt, dessen Rolle (`SampleRole`) eine Callout-Methodenbindung (Zeile 5) zur gebundenen Basis (`SampleBase`) deklariert. Der OT/J-Ausdruck

[5] Zu den Features gehören Methoden, Felder, Bindungen, extends-Beziehungen usw.
[6] Codetypen sind unter anderem Infrastrukturcode und vermischter Code.

`void roleMethod() -> void baseMethod();` wird vom OT-Compiler in Java Bytecode transformiert, dessen Java Quellcode Repräsentation in Abbildung 5.1 dargestellt ist.

Listing 5.5: Rolle mit Callout-Methodenbindung (OT/J Quellcode)

```
1 public team class SampleTeam
2 {
3   public class SampleRole playedBy SampleBase
4   {
5     void roleMethod() → void baseMethod();
6   }
7 }
```

Aus der Callout-Methodenbindung wird eine Standard Java Methode generiert, deren Methodenrumpf den Aufruf der Basismethode auslöst. Außerdem enthält die generierte Java Methode Infrastrukturcode. Die Abbildung 5.1 zeigt die Einteilung der einzelnen Instruktionen in die verschiedenen Codetypen. Die dunkelgrau markierten Bereiche stellen Infrastrukturcode auf Statementebene dar. Die hellgrau markierten Bereiche kennzeichnen den Java Code, der semantisch äquivalent zum OT/J-Code ist. Der in der abstrakten AOP-Sprache verfasste Quellcode hat

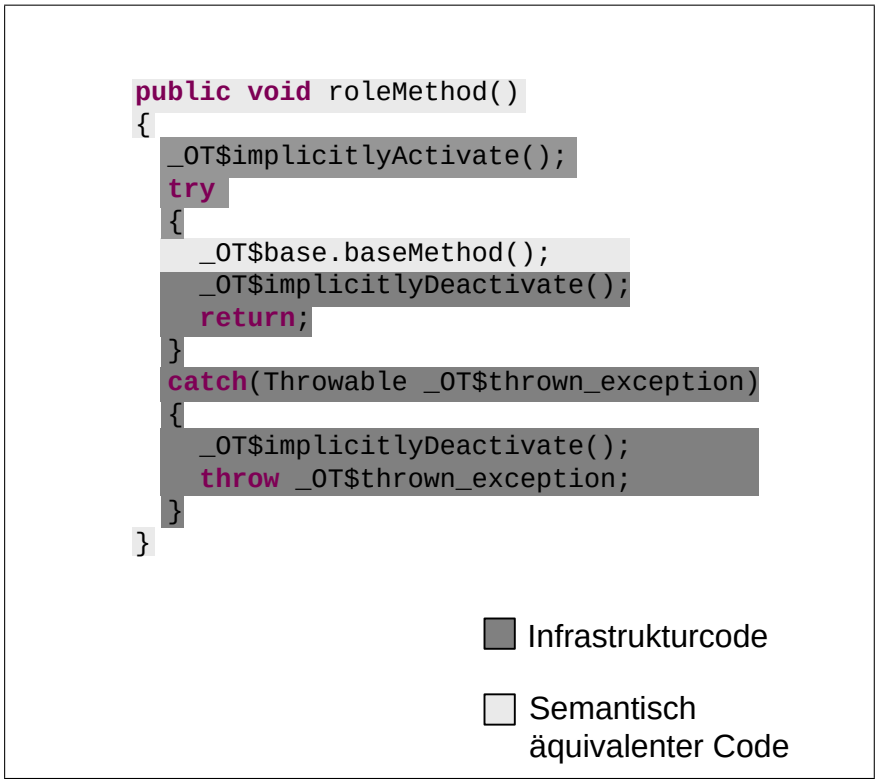


Abbildung 5.1: Compilertransformation der Callout-Methodenbindung (Java Quellcode)

nach der Transformation eine Standard Java Entsprechung. Aus diesem Standard Java Code können Codeteile identifiziert werden, die die Bedeutung des in der AOP-Sprache verfassten Quellcodes widerspiegeln. Bei der Callout-Methodenbindung ist die generierte Methodendeklaration und der Basismethodenaufruf (`_OT$base.baseMethod();`) semantisch äquivalent zum OT/J-Quellcode (`void roleMethod() -> void baseMethod();`).

Überblick Sprachfeature - Realisierung

Die Tabelle in Abbildung 5.2 fasst die Mechanismen zusammen, die der OT/J-Compiler für die Umsetzung der einzelnen Sprachfeatures verwendet. Neben dem Infrastrukturcode, der nach

Realisierung/Sprachfeature	Team	Team - Rolle	Rolle - Basis	Lifting/ Lowering	Role File	Implizite Vererbung	Callin (Vorbereitung)	Callout	Parameter-mapping	Aspekt-aktivierung
Infrastrukturcode (Klassen und Interface)	-	x	-	-	-	x	-	-	-	-
Infrastrukturcode (Methode)	x	x	x	x	-	-	x	x	-	-
Infrastrukturcode (einzelne Instruktionen)	x	x	x	x	-	-	x	x	x	x
Duplizierter/Kopierter Code (Klassen, Methoden, einzelne Instruktionen)	-	-	-	-	-	x	-	-	x	-
Umbennen von Klassen	-	x	-		x	x	-	-	-	-
Umbennen von Methoden	-	-	-	-	-	-	x	-	-	-
Non Java konforme Quelldatei Classdatei-Benennung	-	x	-	-	x	-	-	-	-	-

Abbildung 5.2: Überblick über die Sprachfeatures und ihre Realisierung durch den Compiler

generierten Klassen, Interfacen, Methode und einzelnen Instruktionen (Statements) unterschieden wird, werden noch weitere Einflüsse auf das Debugging gezeigt. Durch das Object Teams Konzept der impliziten Vererbung entsteht noch ein weiterer Codetyp - kopierter Code. Dieser Bytecode besitzt eine einzige Entsprechung auf der Quellcodeebene - die Quelle des kopierten, duplizierten Codes. Der kopierte Code ist Bestandteil der Klasse und ist nicht mit Mechanismen wie dynamischen Binden zu verwechseln. Abbildung 5.3 illustriert die Implizite Vererbung und führt den dritten Codetyp, den kopierten Code, ein. Die Rolle `SampleRole` des `SubTeams` wird durch Kopieren des Bytecodes von `SuperTeam.SampleRole` zur inneren Klasse des SubTeams und muss während des Debugging besonders behandelt werden. Diese Vererbung durch Kopieren wird aus Sicht des Compilers als *Copy-Inheritance* bezeichnet.

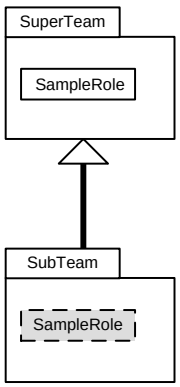


Abbildung 5.3: Implizite Vererbung

Das Umbenennen von Klassen und Methoden sind weitere Mechanismen, um einige Sprachfea-

tures zu realisieren. Abbildung 5.4[7] zeigt die Transformation der Rollen, die durch den Compiler durchgeführt wird. Aus Rolle `Team1.Role1` werden durch den OT-Compiler zwei Bytecodedateien erzeugt - `Team1$Role1.class` und `Team1$__OT__Role1.class`.
`Team1$Role1.class` ist die Repräsentation der Schnittstelle von `Team1.Role1` und `Team1$__OT__Role1.class` enthält dessen Implementierung. Ein Standard Java Compiler erzeugt aus einer inneren Klasse MyClass.InnerClass die Bytecodedatei MyClass$InnerClass.class, die die Implementierung enthält. Der OT-Compiler tut das nicht und die Zuordnung von Bytecode- und Quelldatei birgt Probleme, die vom Debugger zu lösen sind.

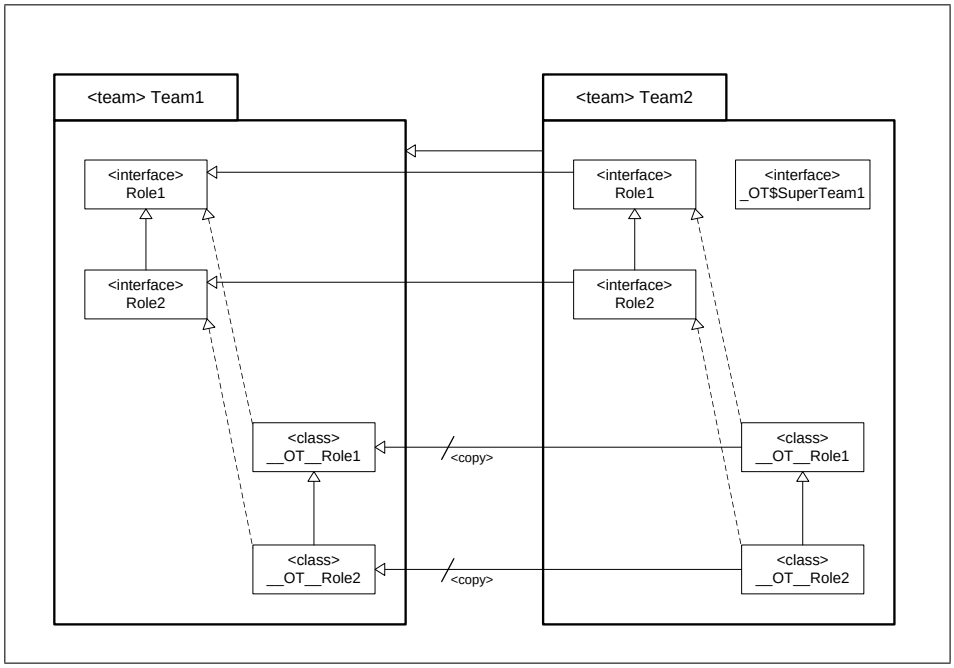


Abbildung 5.4: Copy-Inheritance

5.1.2.2 Der Object Teams Weber

Bei Object Teams/Java geschieht das Weben zur Ladezeit (siehe Kapitel 4.1.3.1). Die Abbildung 5.5 illustriert das Laden der Java Class-Dateien in die Java Virtual Machine. Eine spezielles Framework, das *JMangler Framework*, sorgt dafür, dass die Java Class-Dateien, bevor sie zur Ausführung kommen, also der *Execution Engine* übergeben werden, an den *Transformations Manager* umgeleitet werden. Dieser transferiert sie an den *Composition Algorithm* weiter, der für die Aktivierung und Koordination der Transformer verantwortlich ist. Am Vorgang des Webens sind mehrere Code-Transformer[8] beteiligt. Die Transformer greifen auf ein Framework zurück, was die Erzeugung und Manipulation von Java Bytecode vereinfacht - BCEL (Byte Code Engineering Library). Nach der Transformation werden die veränderten Class-Dateien zurück an das *Classloader System* transferiert. Die *Execution Engine* führt die veränderten Class-Dateien dann aus.

[7] Die Abbildung wurde aus der Diplomarbeit [54] entnommen.
[8] Nähere Information dazu sind in der Diplomarbeit [37] zu finden.

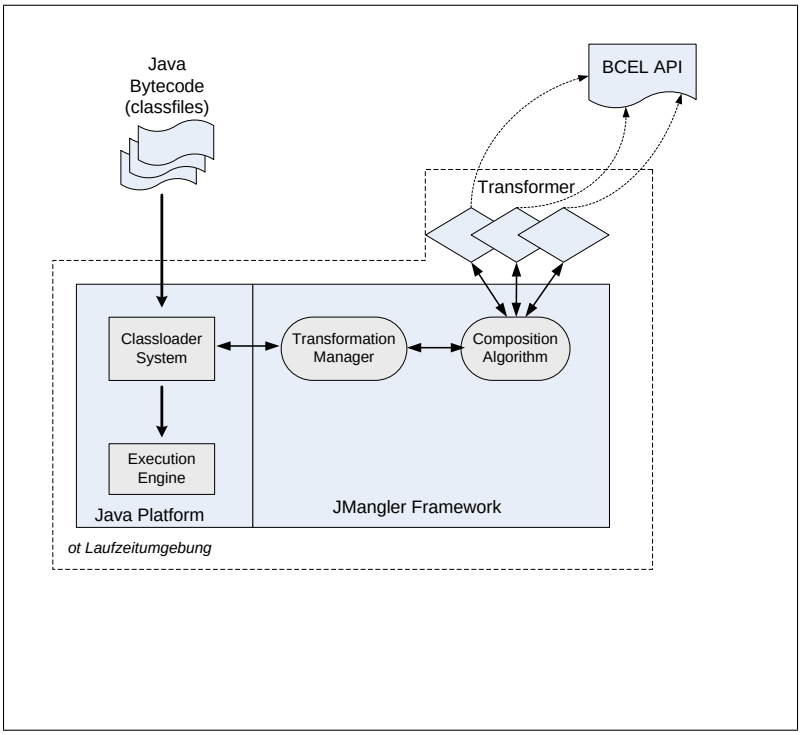


Abbildung 5.5: Webemechanismus bei Object Teams/Java

Das Weben passiert bei Object Teams/Java zur Ladezeit und auf Bytecodeebene. Damit das überhaupt möglich ist, muss der OT-Compiler auf der Seite des Aspektcodes Vorbereitungen treffen. Callin-Methodenbindungen bewirken die Transformation der gebundenen Basismethoden, also Transformationen auf Basiscodeseite, die durch den Weber vorgenommen werden. Der Weber ist bei Object Teams/Java Bestandteil der Laufzeitumgebung. Die Laufzeit benötigt dazu Informationen über Teams, Rollen und Rollenmethoden, um daraus Bytecode zu generieren, der mit dem Bytecode der Basisklasse vermischt wird. Der vermischte Bytecode beinhaltet dann Codeteile des ursprünglichen Basiscodes und Methodenaufrufe von Rollenmethoden. Da die Laufzeitumgebung nicht alle benötigten Informationen sammeln kann, muss das einen Schritt früher, während des Kompiliervorgangs passieren. Der OT-Compiler sammelt also die Informationen während der einzelnen Compilerphasen und speichert sie im Bytecode. Die Informationen werden in Attribute der Class-Datei geschrieben. Class-Dateien beinhalten viele Attribute (zum Beispiel das Sourcefile- oder Linenumbertable Attribut). Der OT-Compiler erzeugt neue, ot-spezifische Attribute, die von der Object Teams Laufzeitumgebung ausgelesen werden. Deuten die Informationen auf eine Bindung einer Rollenmethode an eine Basismethode, wird der Bytecode der Basisklasse transformiert.

Transformation der Basisklasse durch eine Callin-Methodenbindung

Der Methodenrumpf der adaptierten Basismethode wird während der Transformation überschrieben und enthält danach generierten Code, der das Verhalten der Aspekte, also die Rollenmethodenaufrufe initialisiert. Die Methode wird *initial wrapper* genannt. Damit der Originalcode der adaptierten Basismethode nicht verloren geht und aufgerufen werden kann, wurde vorher eine neue Methode generiert und der Code dorthin verschoben, was einem Umbennen der Basismethode gleichkommt. Aus dem anfangs generierten Code (initial wrapper) wird der

chaining wrapper aufgerufen. Der *chaining wrapper* ist eine generierte Methode, die sich selbst rekursiv und dabei alle Rollenmethoden aufruft, die die Basismethode adaptierten. Mehrere Aspekte können eine Basisapplikation und somit eine Basismethode adaptieren. Bei Object Teams/Java können somit mehrere Rollenmethoden an eine Basismethode gebunden werden. Bei einem Aufruf der Basismethode müssen dann alle durch ein Callin-Methodenbindung gebundene Rollenmethoden aufgerufen werden. Wann die Rollenmethode aufgefufen wird, legt die Art des Callin-Methodenbindungs (before, after, replace) fest. Durch die Transformation

	Callin-Methodenbindung
Infrastrukturcode (Klassen und Interface)	-
Infrastrukturcode (Methode)	x
Infrastrukturcode (einzelne Instruktionen)	x
Duplizierter/Kopierter Code (Klassen, Methoden, einzelne Instruktionen)	-
Umbennen von Klassen	-
Umbennen von Methoden	x
Non Java konforme Quelldatei-Classdatei-Benennung	-

Abbildung 5.6: Transformationsauswirkungen bei Callin-Methodenbindungen

der Basisklasse stößt die Debuggerapplikation während des Debuggings auf Infrastruktur- und vermischten Code, den die Applikation erkennen und besonders behandeln muss.

5.1.2.3 Erfüllung der Grundanforderungen

Im Kapitel 3.1 wurden Grundanforderungen aufgestellt, die jeder interaktive Debugger erfüllen sollte. Der ausgeführte Maschinencode bei Object Teams/Java ist Standard Java Bytecode. Der zugrunde liegende Debugger für diese Art des Maschinencodes ist ein Standard Java Debugger. Die Sprache Java wurde um OT-Sprachkonzepte erweitert, die durch den OT-Compiler und den OT-Weber umgesetzt werden. Ein interaktiver Debugger für OT-Programme muss die aufgestellen Grundanforderungen ebenfalls erfüllen. Wie OT-Compiler und OT-Weber die Erfüllung der Grundanforderungen erschweren, beschreibt dieser Abschnitt:

Der durch den OT-Compiler erzeugte Aspektbytecode und der durch den Weber transformierte Basisbytecode besteht auf dieser Bytecodeebene aus folgenden Codetypen:

- **Originalcode** - Bytecode der unangetastet blieb und in keiner Weise transformiert wurde, ist Originalcode. Der OT-Compiler und der Weber erzeugen aus Quellcode A den

Bytecode, der nach Dekompilierung durch einen Java Decompiler Quellcode B ergibt. Entspricht Quellcode A dem Quellcode B, so handelt es sich um *Originalcode* auf Bytecodebene.

- **Infrastrukturcode** - Infrastrukturcode wurde schon in Kapitel 4.3.1 vorgestellt.
- **Semantisch äquivalenter Code** - Semantisch äquivalenter Code wurde im Rahmen der Realisierung der Sprachfeatures im Kapitel 5.1.2.1 schon vorgestellt.
- **Kopierter Code** - Kopierter Code wurde ebenfalls im Kapitel 5.1.2.1 eingeführt.
- **Vermischter Code** - Vermischter Code wurde im Kapitel 4.3.1 vorgestellt.

Diese Codetypen haben einen großen Einfluss auf die Erfüllung der Grundanforderungen. Eine Debuggerapplikation für Object Teams/Java Programme muss diese Codetypen gesondert behandeln. Auch haben Sprachfeatures wie *Implizite Vererbung* und die Möglichkeit Rollen in *Rollendateien* zu deklarieren einen Einfluss auf die Erfüllbarkeit. Im Folgenden werden die Probleme und deren Lösungen bei der Erfüllung der einzelnen Grundanforderungen erklärt. Die Probleme bei der Erfüllung der Grundanforderung **Programmmanipulation** werden hier nicht näher betrachtet, da der Umfang der Betrachtung Thema einer separaten Diplomarbeit sein könnte.

Quellcodedarstellung

Um die Quellcodedarstellung zu ermöglichen, müssen die einzelnen Bytecodetypen während des Debuggings, also während das Programm ausgeführt wird, erkannt und angezeigt/nicht angezeigt werden. Dabei muss jeder Codetyp anders behandelt werden:

- **Behandlung von Originalcode** - Originalcode muss während des Debuggings nicht als besonderer Bytecode erkannt werden und benötigt auch keine gesonderte Behandlung. Die Debuginformationen Source File und Linenumbertable reichen aus, um diesen Bytecode dem entsprechenden Quellcode zuordnen zu können.
- **Behandlung von Infrastrukturcode** - Klassen, Interface, Methoden, Felder und einzelne Statements können Infrastrukturcode darstellen. Dieser Bytecode besitzt, wie schon in Kapitel 4.3.1 erläutert, keine Entsprechung im Quellcode. Um diesen Codetyp während des Debuggings zu erkennen, muss er bei der Generierung markiert werden. Klassen, Methoden und Felder können mit dem synthetic-Attribut (siehe Kapitel 4.2.2) versehen und so markiert werden. Die Markierung einzelner Statements wird nicht unterstützt. Der Compiler könnte zwar ein eigenes Class-Datei-Attribut erzeugen, das die Zeilennummern der Statements speichert, die Infrastrukturcode darstellen. Da diese Information aber durch die Einschränkungen der *Java Platform Debugger Architecture* für die Debuggerapplikation unerreichbar ist, ist das kein brauchbarer Ansatz. Debuginformationen, die durch einen Compiler/Weber für Statements geschrieben werden können sind Zeilennummern, also Einträge in der Linenumbertable. Diese können auch von einem Debugger ausgelesen werden. Um also Statements markieren zu können, müssen einige Zeilennummern eine besondere Bedeutung bekommen und den Statements (Bytecodeinstruktionen) zugeordnet werden. Diese speziellen Zeilennummern muss auch die Debuggerapplikation kennen. Wird Bytecode als Infrastrukturcode erkannt, kann er von der Debuggerapplikation behandelt (zum Beispiel gefiltert) werden. Der Eclipse JDT-Debugger ermöglicht das Filtern von Paketen, Klassen und Methoden (synthetic-Attribut). Der Debugger für Object Teams muss jetzt auch die Funktion der Filterung der markierten Infrastrukturcodestatements bieten (siehe Kapitel 6.4.3.2).

- **Behandlung von Semantisch äquivalenten Code** - Die Hauptarbeit der Markierung und der Behandlung von *semantisch äquivalenten Code* liegt auf der Seite des Compilers. Der Debugger nimmt dabei den konsumierenden Part ein, da alle Informationen in den durch den Compiler generierten Debuginformationen stecken. Das durch die *Java Platform Debugger Architecture* unterstützte SourceDebugExtension Attribut(siehe Kapitel 4.2.3) ist der Träger der Informationen. Betrachtet man die Compiler- und Webetechniken von Object Teams, so lassen sie sich dem Szenario des **Direkten Erstellen der Class-Dateien** (siehe Abbildung 4.8) zuordnen. Der OT-Compiler generiert zusätzlich zum Bytecode eine Source Map, die den semantisch äquivalenten Code auf ihren Ursprung, dem Quellcode, abbildet. Im Beispiel des in Abbildung 5.1 gezeigten Java Codes, würde der Bytecode des Statements `_OT$base.baseMethod();` dem Quellcode `void roleMethod() -> void baseMethod();` zugeordnet werden. Der Debugger wertet die Source Map während der Programmausführung aus und zeigt im Fall der Callout-Methodenbindung die Rollendatei[9] an, die die Callout-Methodenbindung deklariert.

- **Behandlung von Kopierten Code** - Um kopierten Bytecode Quellcode zuordnen zu können, wird die gleiche Vorgehensweise wie bei der Behandlung von semantisch äquivalenten Code verwendet. Der Compiler merkt sich darüber hinaus beim Kopieren des Bytecodes den Ursprung des Bytecodes und dessen Zuordnung zum Quellcode. Daraus wird eine Source Map generiert, die dem kopierten Bytecode den Quellcode des Ursprungbytecodes zuordnet. Eine solch komplexe Zuordnung wäre mit den Debuginformationen Sourcefile und Linenumbertable allein nicht möglich.

- **Behandlung von Vermischten Code** - Vermischter Code kommt bei Object Teams/Java nur auf der Seite des Basiscodes vor. Der vom Object Teams Weber erstellte vermischte Code besteht im Einzelnen aus Infrastrukturcode, Codefragmenten des Rollen- und Basiscodes und Methodenaufrufen von Rollen- und Basismethoden. Wie Infrastrukturcode markiert und behandelt wird, wurde bereits erläutert. Das Zuordnen von Rollenbytecode zu Rollenquellcode und Basisbytecode zu Basisquellcode geschieht wieder mittels Source Map. Bei Object Teams entsteht Vermischter Code nur durch die OT-Laufzeitumgebung. Diese muss auch die Source Map erstellen. Für die Erstellung der Source Map benötigt die OT-Laufzeitumgebung aber Informationen, die nur während des Kompiliervorgangs erhältlich sind. Um diese Informationen zu bekommen, muss der Compiler die Informationen in Class-Dateien der Aspekte speichern, da der OT-Compiler den Basiscode in keiner Weise verändert. Die Laufzeitumgebung liest diese Informationen aus, transformiert den Basiscode und generiert die Source Map um die komplexen Zuordnungen möglich zu machen.

Durch die vorgestellten Markierungs- und Behandlungmöglichkeiten, kann ein Debugger für Object Teams Programme die Grundanforderung der Quellcodedarstellung erfüllen.

Kontrolle durch Haltepunkte/Schrittmodi

Die Sprachfeatures von Object Teams beeinträchtigen das Setzen von Haltepunkten.

- **Implizite Vererbung** - Durch das Konzept der impliziten Vererbung und deren Umsetzung durch den Compiler, entsteht eine 1:n-Beziehung von Quellcode zu Bytecode. Eine Rolle kann auf Bytecodeebene n-mal kopiert werden, obwohl sie nur einmal als Quellcode existiert. Das Diagramm in Abbildung 5.3 veranschaulicht das Problem. Der

[9] Ist die Rolle in der gleichen Quelldatei wie das Team deklariert, dann wird die Teamquelldatei angezeigt. Ist die Rolle ein externe Rolle, so wird die Rollendatei angezeigt.

Bytecode von `SuperTeam.SampleRole` und der Bytecode von `Subteam.SampleRole` werden dem Quellcode von `SuperTeam.SampleRole` zugeordnet, da `Subteam.SampleRole` in diesem Beispiel durch keinen "eigenen" Quellcode repräsentiert wird. Um einen Zeilenhaltepunkt in `Subteam.SampleRole` setzen zu können, muss er im Quellcode von `SuperTeam.SampleRole` gesetzt werden. Damit das Programm auch bei der Ausführung von `Subteam.SampleRole` hält, muss der gesetzte Haltepunkt auch in den kopierten Bytecode gesetzt werden. Ein Setzen eines Zeilenhaltepunktes in eine Rolle muss also ein Setzen des Zeilenhaltepunktes in alle kopierten Bytecodeversionen bewirken.

- **Rollendateien** - Rollen dürfen, obwohl sie innere Klassen von Teams sind, in separaten Rollendateien deklariert werden (siehe Listing 5.2). Der Compiler sorgt dafür, dass die externe Rolle auf Bytecodeebene als innere Klasse des Teams umgesetzt wird. Das Setzen eines Zeilenhaltepunktes passiert im Quelltext der Rolle. Das Zuordnen zum entsprechenden Bytecode stellt bei der **Kontrolle durch Haltepunkte** ein Problem dar. Um das Setzen eines Zeilenhaltepunktes innerhalb von Rollendateien zu ermöglichen, muss der Haltepunkt in der entsprechenden Rollen-Class-Datei, analog der Umsetzung des Compilers, installiert werden.

Die Kontrolle durch Schrittmodi wird bei vollständiger Erfüllung der Grundanforderung der Quellcodedarstellung nicht beeinträchtigt. Der ausgeführte Maschinencode ist Standard Java Bytecode. Alle Schrittmodi sind deshalb ohne Probleme anwendbar. Die Grundanforderung der Quellcodedarstellung sollte aber erfüllt sein, da beim interaktiven Debugging der Quellcode gekennzeichnet wird, dessen Bytecodezuordnung gerade ausgeführt wird. Das, was der Benutzer sieht, beeinflusst die Wahl des nächsten Schrittmodus. Ist die Quellcodedarstellung fehlerhaft, werden funktionierende Schrittmodi unbrauchbar.

Visualisierung

Die Visualisierungen, also die verschiedenen Sichten auf den Programmzustand eines Debuggee, illustrieren je nach Sicht den Methodenaufrufstack, die Werte der Variablen, die gesetzten Haltepunkte und mehr. Die Implementierungen hinter den Sichten interpretieren den Java Bytecode und leiten die Informationen an die jeweilige Sicht weiter. Bei Object Teams/Java ist der Java Bytecode transformiert, vermischt und kopiert. Ist die Grundanforderung der Quellcodedarstellung erfüllt, so ist auch die Quellcodesicht angepasst. Alle anderen Sichten benötigen ebenfalls Anpassung.

- **Anpassung Stack Trace-/Stack Frame-Sicht** - Im Laufe der Transformationen durch den Compiler und den Weber werden Methoden umbenannt und zusätzliche Methoden generiert. Der Programmfluss stimmt nicht mehr mit dem aus dem Quellcode ableitbaren Programmfluss überein, da sich zum Einen die Namen der Methoden und zum Anderen die Zahl der aufgerufenen Methoden innerhalb eines Stack Traces ändern können. Der Benutzer erwartet einen Methodenaufrufstack, den er aus seinem verfassten Quellcode ableiten kann. Das bedingt die Transparenz der Compiler- und Weberrealisierungen. Die Anpassung der Stack Trace-/Stack Frame-Sicht bedeutet das Anzeigen der Methodennamen des Quellcodes und die Anzahl der Methodenaufrufe so wie sie im Quellcode vorkommen.

- **Anpassung Variablensicht** - Die Variablensicht zeigt den Variablennamen und dessen Wert bei einem Halt des Debuggee an. Wie in Kapitel 4.3.1 erläutert wurde, können Felder und lokale Variablen Teil des Infrastrukturcodes sein. Diese Anpassung der Variablensicht bedeutet ein Nichtanzeigen von Infrastrukturcode-Variablen.

- **Anpassung Breakpointsicht** - Die Object Teams Features *Implizite Vererbung* und *Rollendeklaration in Rollendateien* bewirkten Erweiterungen bei der **Kontrolle durch Haltepunkte**. Diese Änderungen müssen für den Benutzer des Object Teams Debuggers transparent sein. Bei *Impliziter Vererbung* wird ein Zeilenhaltepunkt einmal in der Superrolle und damit in jede Bytecodekopie gesetzt. In der Breakpointsicht, also der Sicht, die alle gesetzten Haltepunkt anzeigt, darf der gesetzte Zeilenhaltepunkt nur einmal vorkommen, obwohl er n-mal in den Bytecodekopien gesetzt wurde.

5.1.3 Object Teams/Java - Aspektaktivierung und dessen Einfluss auf die Entwicklung eines Debuggers

Object Teams bietet einen eigenen Aktivierungsmechanismus für Basis-Aspektbindungen. Nachdem der OT-Compiler den Aspektquellcode in Java Bytecode transformiert und damit auch Vorbereitungen für das Weben getroffen hat, webt die OT-Laufzeitumgebung durch Codetransformation Basis- und Aspektcode zusammen. Damit sind die Aspekte verwoben. Über deren Ausführung entscheidet aber der Aspektaktivierungsstatus.

Bei Object Teams/Java legt der Teamaktivierungstatus das Ausführen der an die Basismethode gebundenen Rollenmethoden fest. Wurde ein Team nicht aktiviert, so haben die Callin-Methodenbindungen keinen Effekt. Um ein Team zu aktivieren stehen dem Programmierer mehrere Möglichkeiten zur Verfügung:

- **activate()/deactivate()-Methoden**
  Jedes Team erbt automatisch von `org.objectteams.Team` und damit auch die `activate()`- und `deactivate()`-Methode, die auf einer Team-Instanz aufgerufen werden können. Das Ein- und Ausschalten der Teamfunktionalität kann gezielt zu bestimmten Zeitpunkten und darüber hinaus für bestimmte Threads passieren. Sollen Callin-Methodenbindungen für alle Threads aktiviert werden, so muss die Konstante `ALL_THREADS` als Parameter übergeben werden und das Team ist global aktiviert. Wenn eine bestimme Thread-Instanz als Parameter übergeben wird, wird das Team nur für diesen Thread aktiviert. Der Aufruf der `activate()`-Methode ohne Parameter bewirkt eine Aktivierung des Teams für den aktuellen Thread.

- **within-Konstrukt**
  Mit dem Block-Konstrukt `within(teamInstance){...stmts...}` kann eine Team-Instanz kurzzeitig für den aktuellen Thread aktiviert werden. Callin-Methodenbindungen sind innerhalb des Blocks aktiv.

- **Guards**
  Ein weiteres Mittel um Callin-Methodenbindungen zu kontrollieren, sind **Guards**. Sie können an verschiedenen Stellen innerhalb von Object Teama/Java Programmen auftauchen: direkt an Callin-Methodenbindungen, an Rollenmethoden, an Rollendeklarationen und an Teamdeklarationen. So kann die Aktivierung für einzelne bis hin zu allen innerhalb eines Teams (und Rollen) deklarierten Callin-Methodenbindungen kontrolliert werden.

Wird ein Team nicht durch oben genannte Sprachkonstrukte aktiviert, kann es trotzdem in einigen Fällen zur *impliziten Aktivierung* des Teams kommen. Werden Team- oder Rollenmethoden aufgerufen, so wird das Team automatisch für diese Aufrufe aktiviert.

5.1.3.1 Erfüllung der Grundanforderungen

Im Kapitel 4.3.2 wurde erklärt, warum die Aspektaktivierung das Debugging von AOP-Programmen erschwert und, dass die Bereiche **Kontrolle durch Haltepunkte** und **Visualisierung** erweitert werden müssten, um der Komplexitätsteigerung durch Aspektaktivierung entgegenzuwirken. Die Mannigfaltigkeit der Aktivierungmöglichkeiten bei Object Teams und besonders die *implizite Aktivierung* manifestieren die getroffenen Aussagen. Ohne Erweiterung in den Grundanforderungsbereichen ist ein Überblicken über den Aktivierungsort[10] und der Aktivierungdauer[11] unmöglich.

Die Grundanforderungsbereiche **Quellcodedarstellung** und **Kontrolle durch Schrittmodi** sind von der Aspektaktivierung nicht betroffen.

Kontrolle durch Haltepunkte

Ein Team kann durch die verschiedensten Methoden und Mechanismen aktiviert werden. Um alle Aktivierungmöglichkeiten abzudecken steht ein neuer Haltepunkttyp zur Verfügung - der Aktivierungshaltepunkt. Dieser Haltepunkt wird nicht über eine Zeile im Quellcode gesetzt, sondern ist an das Ereignis der Teamaktivierung gebunden; ähnlich dem Fehlerhaltepunkt (Exceptionbreakpoint), der an das Ereignis des Auftretens einer Exception gebunden ist.

Visualisierung

Das Ereignis der Teamaktivierung ist auch für eigene Visualisierungen interessant. Diese neue Sicht (Team Monitor) visualisiert, welche Teams zur Laufzeit instantiiert und welche aktiviert sind. Alle Aspekte die das Basisprogramm gerade adaptieren, werden innerhalb einer Sicht repräsentiert. Von der im Quellcode oder im Bytecode[12] verstreuten Teamaktivierung und -deaktivierung kann abstrahiert werden, da alle instantiierten Teams und deren Zustand auf einem Blick ersichtlich sind.

5.2 AspectJ

Im vorigen Abschnitt wurde auf Object Teams als ein Vertreter von AOP-Sprachen näher eingegangen und die Arbeitsweisen des Compilers und Webers vorgestellt. Dabei wurden die Einflüsse ersichtlich, die der von ihnen produzierte Java Bytecode auf die Entwicklung eines Debuggers hat. In diesem Kapitel wird die bekannteste AOP-Sprache unter den AOP-Sprachen vorgestellt - AspectJ. Auch sie gehört zu den Sprachen, deren Werkzeuge Standard Java Bytecode erstellen, der in einer Standard Java Virtual Machine ausgeführt werden kann. Die Sprachkonzepte, Übersetzungstechnik und Webetechnik von AspectJ werden in diesem Abschnitt erläutert und deren Einflüsse auf die Entwicklung eines Debuggers dargestellt.

[10] Der Aktivierungsort ist die Stelle im Quellcode, an der das Team aktiviert wurde.
[11] Die Aktivierungdauer kann von der Dauer eines Methodenaufrufes bis hin zur Ausführungsdauer eines ganzen Programms reichen.
[12] Um die implizite Aktivierung zu ermöglichen, generiert die OT-Laufzeitumgebung zusätzlich Infrastrukturcode auf Bytecodeebene. Dieser hat keine Quellcodezuordnung.

5.2.1 AspectJ Sprachkonzepte

Die Mächtigkeit von AspectJ liegt in den vielfältigen Beschreibungsmöglichkeiten von Pointcuts. Diese Beschreibungsmöglichkeiten und weitere Konzepte werden im Folgenden erläutert.

Join Point Model

Join Points sind identifizierbare Ausführungspunkte in einem System und wurden in Kapitel 3.4 schon allgemein erklärt. Sie sind unabhängig vom Weben zu betrachten. In AspectJ gibt es ein Join Points Modell, dass Join Points in Kategorien einteilt:

Method Join Point - Diese Kategorie wird in Methodenaufruf (method call) und Methodenausführung (method execution) unterschieden. Die *Method Call*-Join Points identifizieren Orte, wo Methoden aufgerufen werden. Ein *Method Execution*-Join Point identifiziert den Methodenrumpf einer Methode als potentiellen Webepunkt.

Constructor Join Point - Der *Constructor Join Point* ähnelt dem Method Join Point. Anstatt einer Methode ist hier der Konstruktor betroffen. Der *Execution Contructor*-Join Point betrifft den Rumpf des Konstruktors einer Klasse. Der *Call Constructor*-Join Point entspricht dem Punkt in einer Methode, an dem die Erzeugung eines Objekts veranlasst wird.

Field Access Join Point - *Field Access Join Points* repräsentieren die Lese- und Schreibzugriffe auf Felder eines Objektes, einer Klasse oder eines Interfaces. Nicht einbezogen werden Zugriffe auf lokale Variablen in einer Methode.

Exception Handling Join Point - Die Kategorie steht für Orte in einem System, wo die Behandlung von Exceptions stattfindet.

Class Initialization Join Point - *Class Initialization Join Points* decken das Ereignis des Ladens von Klassen ab.

Object Initialization Join Point - Diese Kategorie ist das Äquivalent zu den Class Initialization Join Points für Objekte. Typischerweise wird diese Art von Join Point genutzt, um zusätzliche Initialisierungen zu erzielen.

Weiterhin existieren noch Object Pre-Initialization und Advice Execution Join Points.

Pointcut

In AspectJ wählen *Pointcuts* bestimmte Join Points innerhalb eines Programmflusses aus. Für diese Selektion stehen bei AspectJ verschiedene *Pointcut Designators* zur Verfügung. Es existieren *call*, *execution*, *get*, *set*, *preinitialization*, *initialization*, *staticinitialization*, *handler* und *adviceexecution* Designators. Auf Designators wird in dieser Diplomarbeit nicht weiter eingegangen, da der generierte Bytecode und dessen Einfluss auf die Entwicklung eines Debuggers im Vordergrund liegen. Mit welchen Hilfsmitteln die Webepunkte (Join Points) beschrieben werden können, ist dabei nebensächlich.

Advice

Ein Advice stellt das aktive Element in AspectJ dar. Es definiert, was an einem Join Point, der von einem Pointcut selektiert wird, geschehen soll. In Object Teams sind das die Rollenmethoden. Advices können vor und nach einem Join Point ausgeführt werden. Außerdem kann mittels around-Advice eine Ausführung umgangen werden, wobei trotzdem die Möglichkeit

besteht, das umgangene Original auszuführen. Um innerhalb eines around-Advices den normalen Programmfluss an dem aktuellen Join Point ablaufen zu lassen, muss das Schlüsselwort `proceed();` genutzt werden. In einem Advice kann auf den Kontext des Join Points zugegrif-

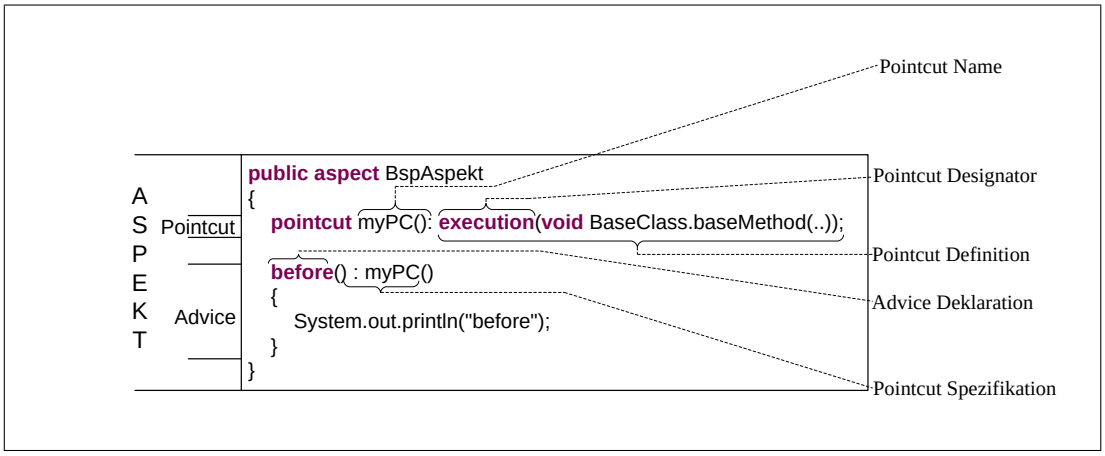


Abbildung 5.7: Ein Aspekt Deklaration in AspectJ

fen werden. Dafür steht zum Einen das Schlüsselwort *thisJoinPoint* zur Verfügung und zum Anderen kann mittels *Execution-* und *Argument-JoinPoints* darauf zugegriffen werden:

thisJoinPoint - AspectJ stellt eine spezielle Referenz-Variable zur Verfügung, die Informationen, über den aktuellen Join Point beinhaltet. *thisJoinPoint* kann nur innerhalb des Kontextes eines Advices verwendet werden und enthält Informationen wie die Art des Join Point, dessen Signatur und dessen Quellcodeposition.

Execution- und Argument-Join Points - Mit bestimmten Pointcut-Schlüsselwörtern können Kontextinformationen dem Advice als Argumente übergeben werden. So kann zum Beispiel auf Argumente der gebundenen Basismethode innerhalb eines Advices zugegriffen werden, wenn das Schlüsselwort `args(typ)` verwendet wird.

Statisches Crosscutting

Durch *Statisches Crosscutting* kann eine Basisapplikation strukturell erweitert werden. Die statische Klassenstruktur eines Basis-Objektes wird durch Hinzufügen und Ändern von Methoden (Member Introduction) und Attributen umgeformt. Es ist möglich, die Vererbunghierarchie einer Klasse(Type-hierarchy Modification) zu verändern. Diese Art des statischen Crosscuttings wird *Inter-type Declaration* genannt. Des weiteren existiert noch die *Compile-Time Declaration*, durch die zusätzliche Warnungen und Fehlermeldungen in Basiscode eingefügt werden. Das kann dazu genutzt werden, um unzulässige Struktur- und Anwendungsmuster zur Übersetzungszeit festzustellen

Aspekt

Aspekte sind klassenähnlich und stellen das Konstrukt zur Kapselung von Crosscutting Concerns in AspectJ dar. Aspekt-Deklarationen können zum Einen Pointcuts und Advices und zum Anderen Methoden und Felder enthalten. Aspekte können wie Klassen mit speziellen Zugriffsrechten wie private und public versehen sein, als abstrakt deklariert werden und Klassen,

abstrakte Aspekte sowie Interfaces erweitern bzw. implementieren. Dabei kommt es aber zu Einschränkungen. Aspekte dürfen keine inneren Klassen von Standard Javaklassen sein und nicht direkt instantiiert werden.

5.2.2 AspectJ - Übersetzungstechnik, Webetechnik und deren Einfluss auf die Entwicklung eines Debuggers

Der AspectJ Compiler ist wie der Object Teams Compiler ein erweiterter Java Compiler. Er akzeptiert AspectJ Quell- und Bytecode und generiert daraus Standard Java Bytecode. Intern besteht der Compiler aus zwei Ebenen - einem Front-End und einem Back-End. Das Front-End kompiliert AspectJ- und Java Quellcode (Aspekt- und Basisquellcode) in Java Bytecode. Die AspectJ Pointcut-Deklarationen, Aspekte und Advices werden in Java Bytecode mit speziellen AspectJ-Bytecodeattributen transformiert. Das Back-End des Compilers ist der Teil der den Bytecode zusammen webt und wird in Kapitel 5.2.2.2 näher vorgestellt.

5.2.2.1 Der AspectJ Compiler

Das Front-End des AspectJ Compilers ist eine Erweiterung eines Standard Java Compilers und transformiert Aspektquellcode in Standard Java Bytecode. Die Transformation ist relativ einfach - ein Aspekt wird in eine Javaklasse und ein Advice in eine Javamethode transformiert. Das Front-End des Compilers generiert zusätzliche AspectJ-spezifische Bytecodeattribute, die vom Back-End ausgelesen und für die weitere Bytecodetransformation (Weben) verwendet werden.

Advice Transformation

Aus einem Advice wird eine Standard Javamethode erstellt. Die Parameter dieser Methode sind die Parameter des Advices. Wurde innerhalb des Advices auf Kontextinformationen des Join Point (`thisJoinPoint`) zugegriffen, wird die Signatur der Javamethode um `JoinPoint thisJoinPoint`, `JoinPoint.StaticPart thisJoinPointStaticPart` und `JoinPoint.StaticPart thisEnclosingJoinPointStaticPart` erweitert. Der Advice-Rumpf entspricht nach der Transformation dem Rumpf der erstellten Methode. In Abbildung 5.8 ist die Transformation eines Advices illustriert. Listing 5.8a zeigt den Quellcode des Advices. Der Compiler übersetzt den Advice in Bytecode(siehe Listing 5.8c). Der entsprechende Java Quellcode ist in Listing 5.8b dargestellt.

Ein Besonderheit stellt die Transformation eines around-Advices dar, in dem ein `proceed`-Aufruf vorkommt. `proceed` bewirkt eine Transformation auf Aspekt- und Basiscodeseite. An dieser Stelle wird nur die Transformation auf Aspektcodeseite beschrieben, da die Transformation auf Basiscodeseite vom Weber vorgenommen wird. Wie beim after- und before-Advice wird aus dem around-Advice eine Javamethode generiert, deren Methodenrumpf der Advice-Rumpf ist. Der `proceed`-Aufruf im around-Advice bewirkt, dass eine zusätzliche Methode im Aspekt generiert wird, deren Aufgabe die Ausführung des Original-Basiscodes ist. Die generierte Methode wird aus dem transformierten Advice aufgerufen.

Durch die Advice-Transformation in eine Javamethode, entsteht kein zusätzlicher Infrastrukturecode auf Statementlevel. Der Compiler generiert Bytecode, dessen Typ fast dem *Originalcode* entspricht. Bei einem around-Advice mit `proceed`-Anweisung entsteht zusätzlich eine Infrastrukturmethode und ein Infrastrukturstatement.

```
before(): myPC()
{
  System.out.println("before");
}
```

(a) Advice - AspectJ Quellcode

```
public void ajc$before$test_BspAspekt$1$c0ce()
{
  System.out.println("before");
}
```

(b) Advice - dekompilierter Java Bytecode

```
public void ajc$before$test_BspAspekt$1$c0ce()
Code(max_stack = 2, max_locals = 1, code_length = 9)
0:  getstatic  java.lang.System.out Ljava/io/PrintStream; (43)
3:  ldc  "before" (45)
5:  invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V (51)
8:  return
```

(c) Advice - kompilierter Java Bytecode

Abbildung 5.8: Advice Transformation

Aspekt Transformation

Ein Aspekt wird vom AspectJ Compiler zu einer Standard Java Klasse transformiert. Im Unterschied zu Object Teams findet bei AspectJ kein Splitting des Aspekts in Schnittstelle und Implementierung und keine Umbenennung statt. Es werden aber zusätzliche Methoden generiert, die keinem Quellcode zugeordnet werden können. Diese Methoden werden dem Bytecodetyp Infrastrukturcode auf Methodenlevel zugeordnet 4.3.

Statisches Crosscutting

Mit *Statischem Crosscutting* ist es möglich, Basisklassen mit Methoden und Feldern aus dem Aspekt anzureichern. Der Compiler tut dies, indem in der Basisklasse eine Methode und im Aspekt zwei Methoden generiert werden. Die erste Methode im Aspekt ist die im Aspektquellcode deklarierte Methode. Diese Methode hat einen anderen Namen, enthält aber den Methodenrumpf der deklarierten Methode. Die zweite generierte Methode im Aspekt ist Infrastrukturcode (Dispatchercode), der die erste Methode aufruft. Die generierte Methode in der Basisklasse ist ebenfalls Infrastrukturcode und ruft den Infrastrukturcode im Aspekt auf. Sie trägt aber den Namen der im Aspekt deklarierten Methode. Das Generieren der Methode auf Basiscodeseite wird vom Weber übernommen. Bei Feldern wird das Feld durch den Compiler/ Weber einfach in die Basisklasse kopiert.

5.2.2.2 Der AspectJ Weber

Im vorigen Abschnitt wurde erwähnt, dass der AspectJ Weber im Back-End des AspectJ Compilers enthalten ist. Das ist so nicht ganz richtig, denn in der jetzigen Version[13] kann zwischen Weben zur Übersetzungszeit und Weben zur Ladezeit gewählt werden. Am Beispiel Object Teams wurde das Weben zur Ladezeit schon genau erklärt, deshalb gewidmet sich dieser Abschnitt dem Weben zur Übersetzungszeit.

[13] AspectJ Version: 1.5.1a

```
void baseMethod()
{
}
```

(a) Basismethode - Java Quellcode

```
void baseMethod()
{
  BspAspekt.aspectOf().ajc$before$test_BspAspekt$1$c0ce();
}
```

(b) Adaptierte Basismethode - dekompilierter Java Bytecode

```
void baseMethod()
Code(max_stack = 1, max_locals = 1, code_length = 7)
0:  invokestatic test.BspAspekt.aspectOf ()Ltest/BspAspekt; (23)
3:  invokevirtual       test.BspAspekt.ajc$before$test_BspAspekt$1$c0ce ()V (26)
6:  return
```

(c) Adaptierte Basismethode - kompilierter Java Bytecode

Abbildung 5.9: Basismethoden Transformation

Das Back-End des AspectJ Compilers transformiert den Basiscode, indem an den entsprechenden Stellen Aufrufe der vorkompilierten Advice-Methoden eingefügt werden. An bestimmten Stellen im Bytecode, die mögliche Join Points darstellen, wird überprüft, ob eine Advice Pointcutbeschreibung den Join Point beinhaltet. Ist das der Fall, wird ein Aufruf der implementierten Methode (transformierter Advice) an dieser Stelle eingefügt. Manchmal muss an dieser Stelle noch zusätzlicher Code eingefügt werden, der zur Laufzeit dynamisch überprüft, ob der Join Point der Pointcutbeschreibung entsprach. In Abbildung 5.9 ist die Transformation einer Basismethode dargestellt. Listing 5.9a zeigt den Java Quellcode der Basismethode; den leeren Methodenrumpf. Listing 5.9c zeigt die Bytecodeinstruktionen die der Weber, also das Back-End des Compilers erstellt hat. Der in Listing 5.9c dargestellte Standard Java Quellcode entspricht den generierten Bytecodeinstruktionen und es wird ersichtlich, dass der Aufruf der Aspektmethode in die Basismethode hinein gewoben wurde.

Das Weben eines around-Advices mit einem `proceed`-Aufruf bewirkt größere Transformationen auf Basiscodeseite. In die Basisklasse wird eine zusätzliche Methode generiert, die die Statements der adaptierten Basismethode beinhaltet. Außerdem wird eine Klasse (`AroundClosure`) innerhalb der Basisklasse erstellt. Mit dieser Klasse kommuniziert der Aspekt. Wird auf Aspektseite `proceed` aufgerufen, sorgt die `AroundClosure`-Klasse für den Aufruf der Basisklassenmethode, die die Statements der ursprünglich adaptierten Basismethode beinhaltet. Die `AroundClosure`-Klasse und die zusätzliche Basismethode kapseln den normalen Programmfluss, der vom around-Advice unterbrochen wurde.

Der Weber produziert innerhalb der Basisklasse Infrastrukturcode (Methoden- und Klassenlevel) und semantisch äquvalenten Bytecode. Infrastrukturcode wird während des Debuggings keinem Quellcode und der semantisch äquvalente Bytecode Aspektquellcode zugeordnet.

5.2.2.3 Erfüllung der Grundanforderungen

In den vorigen Abschnitten wurden die Sprachfeatures von AspectJ und deren Realisierungen mittels Compiler und Weber vorgestellt. In diesem Abschnitt sollen, analog zum Kapitel *Object Teams/Java und die Erfüllung der Grundanforderungen*, die Einflüsse der Sprachfeature-Realisierungen auf die Erfüllung der Grundanforderungen gezeigt werden. AspectJ bietet eine Entwicklungsumgebung mit integrierten, interaktiven Debugger. Wie dieser die Grundanforderung erfüllt, wird in den folgenden Abschnitten miteinbezogen.

Quellcodedarstellung

Während der Analyse der Arbeitsweisen vom AspectJ-Compiler und AspectJ-Weber, wurden Originalcode, Infrastrukturcode, semantisch äquivalenter Code und vermischter Code als Bytecodetypen identifiziert. Kopierter Code, wie er bei Object Teams vor kam, gibt es bei AspectJ nicht[14].

Die Behandlung der einzelnen Bytecodetypen wurden in Kapitel 5.1.2.3 erklärt, als generierter Bytecode von Object Teams/Java-Programmen analysiert wurde. Diese Behandlungsmethoden lassen sich eins zu eins auf AspectJ abbilden und werden an dieser Stelle nicht noch einmal wiederholt. Anstatt dessen wird die Arbeitsweise des AspectJ Debuggers bei der Quellcodedarstellung aufgezeigt.

Um Bytecode Quellcode zuordnen zu können, existieren nur die Debuginformationen Sourcefile- und Linenumbertable-Attribut. Komplexe Zuordnungen sind somit nicht möglich. Bei AspectJ wird eine Quellcodedarstellungsstrategie[15] in Basisklassen verfolgt, die sich von der beim Object Teams/Java Debugger umgesetzten Strategie unterscheidet. Diese Quellcodedarstellungsstrategie filtert nicht nur Infrastrukturcode, sondern auch alle Aspektanteile des vermischten Bytecodes. Beim Debugging von adaptierten Basisprogrammen wird somit Aspektbytecode ausgeführt aber kein Aspektquellcode angezeigt. Das wird erreicht, indem der Compiler/Weber für den Bytecode von before-Advicemethodenaufrufen keine Einträge in die Linenumbertable einfügt. Bei Aufrufen von after-Advicemethoden, wird die Zeilennummer der schließenden Klammer der adaptierten Basismethode genutzt und in die Linenumbertable eingetragen. Beim Debugging wird somit auf Quellcodeseite die Klammer angezeigt, obwohl auf Bytecodeseite die Bytecodeinstruktion ausgeführt wird, die den Aufruf der after-Advicemethode bewirkt. Würde der Benutzer an dieser Stelle ein *StepInto* veranlassen, so würde in die Aspektmethode gesprungen und Aspektquellcode angezeigt werden. Der around-Advice erfordert die meisten Transformationen des Basiscodes. Infrastrukturcode wird beim around-Advice nicht komplett gefiltert. Anstatt dessen wird für einige Bytecodeinstruktionen die Zeilennummer "1" in die Linenumbertable der Methode eingetragen. Somit wird während des Debuggings eines around-Advices ab und zu, für den Benutzer unverständlich, die erste Zeile der Basisklasse hervorgehoben.

Der AspectJ Compiler (Front-End und Back-End) unterstützt keine Source Map Generierung. Ohne Source Map und Erweiterung des Standard Java Debuggers, lassen sich die Probleme bei der Quellcodedarstellung nicht beheben.

Die im AspectJ-Debugger konzipierte Quellcodedarstellungsstrategie ist auch für den Object Teams/Java Debugger denkbar. Der Object Teams/Java Debugger würde den Aspektcode auf

[14] Der AspectJ Compiler kann *inlined* Code erzeugen. Das heißt, dass bei einem around-Advice, dessen Rumpf in die Basisklasse kopiert wird. Es würde kopierter Code entstehen. Der Compiler kann aber so konfiguriert werden, dass er keinen inlined Code erzeugt. Der generierte Bytecode sieht dann so aus, wie er in Kapitel 5.2.2.2 dargestellt wurde.

[15] Ob nun mangels fehlender Source Map Generierung oder gewollt.

Basiscodeseite komplett filtern und somit beide Quellcodedarstellungsstrategien unterstützen (siehe Kapitel 7.2.1.

Kontrolle durch Haltepunkte/Schrittmodi

Die Realisierungen der Sprachfeatures von AspektJ forcieren nur bedingte Änderungen des Standard JDT-Debuggers beim Setzen von Haltepunkten. Das Vervielfältigen von Haltepunkten ist bei AspectJ nicht notwendig, da kein Bytecode, wie es bei Object Teams/Java der Fall ist, kopiert wird. Durch die Möglichkeit des Statischen Crosscuttings wird aber ein Umlenken von Überwachungspunkten notwendig. Fügt ein Aspekt einer Basisklasse ein Feld durch Statisches Crosscuttings hinzu, wird das Feld kopiert. Damit ein Überwachungspunkt auf dieses Feld gesetzt werden kann, muss der Überwachungspunkt auf die Kopie gesetzt werden. Wird das Feld modifiziert, hält das Programm und der Quellcode im Aspekt wird hervorgehoben.

Diese Erweiterungen wurden nicht beim AspectJ Debugger umgesetzt. Es ist nicht möglich überhaupt Überwachungspunkte noch Methodenhaltepunkte in AspectJ Quellcodedateien zu setzen.

Visualisierung

Die verschieden Bytecodetypen erfordern die gleichen Anpassungen der Sichten, wie sie schon im Zuge von Object Teams/Java erklärt wurden (siehe Kapitel 5.1.2.3).

5.2.3 AspectJ - Aspektaktivierung und dessen Einfluss auf die Entwicklung eines Debuggers

AspectJ stellt keinen eigenen Aspektaktivierungsmechanismus zur Verfügung. Nur durch explizites Ausprogrammieren[16] kann über die Ausführung des Aspektes entschieden werden. Bei AspectJ gibt keinen zentralen Punkt wie bei Object Teams, über den der Aktivierungszustand eines Aspektes geändert werden kann. Eine Realisierung des Aspektmonitors und des Aktivierungshaltepunktes ist somit bei AspectJ auch nicht notwendig.

5.3 Überblick über die Einflüsse verschiedener Webetechniken auf die Entwicklung eines Debuggers

In den vorherigen beiden Abschnitten dieses Kapitels wurden schon zwei Webetechniken und deren Einflüsse auf die Debuggerentwicklung sehr genau erklärt. Die Webetechniken der beiden Sprachen setzen Codetranformation ein, um eine Basisapplikation zu adaptieren. In Kapitel 4.1.3 wurden die bekanntesten AOP-Webetechniken vorgestellt. In diesen Abschnitt wird nun eine Übersicht gegeben, wie die vorgestellten Webetechniken die Entwicklungen von Debuggern beeinflussen.

Die bekanntesten AOP-Implementationen sind auf Basis der Codetransformation und der dynamischen Reflexion realisiert. Da hier Java-basierte AOP-Sprachen im Vordergrund stehen,

[16] In einer Pointcut Definition kann der if-Pointcut Designator verwendet werden, um Bedingungen anzugeben, die über die Ausführung der Advices entscheiden.

bedeutet das die Modifikation des Bytecodes und der Einsatz von dynamischen Proxys. AOP-Frameworks wie Spring und Nanning bedienen sich dynamischer Proxys, um Aspekte auf Zielobjekte anzuwenden. Das hat den Vorteil, dass nur reguläre Java-Klassen erzeugt werden. Allerdings werden nur wenige Join Points unterstützt. AOP-Implementationen wie AspectJ, Object Teams/Java[17] und AspectWerkz erreichen durch Modifikation des Bytecodes die Komposition aus Aspekt- und Basiscode. Die Tabelle 5.10 zeigt die einzelnen Webeverfahren, eine

Webetechnik + Beispiel	Einfluss auf Quellcodedarstellung	Einfluss auf Kontrolle durch Haltepunkte	Einfluss auf Kontrolle durch Schrittmodi	Einfluss auf Visualisierung	Weben realisiert durch
Sourcecode Weaving - EAOP	x	x	-	x	Präprozessor
Bytecode Weaving - AspectJ	x	x	-	x	Compiler
Loadtime Weaving - OT/J	x	-	-	x	Classloader + Laufzeitumgebung
Pseudo Runtime Weaving - AspectWerkz	x	-	-	x	HotSwap
JIT Compiler Weaving - Steamloom	x	?	-	x	JVM/JIT-Compiler
Generic Runtime Weaving - Spring AOP	-	-	-	-	Java Laufzeit Bibliothek

Abbildung 5.10: Übersicht über Webetechniken und deren Einflüsse

Sprache, deren Webetechnik mit dem Verfahren umgesetzt wurde und die Einflüsse auf die Bereiche der Grundanforderungen.

EAOP steht für Event-based Aspect-Oriented Programming [26] und bietet einen Präprozessor, der aus Java Basiscode und Java Aspektcode Standard Java Quellcode erzeugt. Das entspricht dem **Sourcecode Weaving**. Würde ein Debugger für die mittels EAOP erstellten Programme realisiert werden, würden die Bereiche Quellcodedarstellung, Kontrolle durch Haltepunkte und Visualisierung betroffen sein. Der Weber wird bei Sourcecode Weaving durch einen Präpozessor realisiert, der Quellcode erzeugt, der wiederum von einem Standard Compiler übersetzt wird. Beim Weben durch Codetransformation werden *wrapper*-Methoden in den Basiscode eingefügt, die die Aspektmethoden aufrufen. Der Weber beim EAOP-Ansatz geht so vor und erzeugt zusätzliche Methoden in der transformierten Basisklasse. Der resultierende Bytecode muss, damit die Grundanforderung der Quellcodedarstellung erfüllt ist, mit zusätzlichen Debuginformationen (Source Map, Szenario II) angereichert werden. Der Benutzer setzt Zeilenhaltepunkte in den ursprünglichen und nicht den transformierten Quellcode. Das Setzen des Zeilenhaltepunktes in die richtige Bytecodestelle stellt deshalb ein Problem dar und beeinflusst die Kontrolle durch Haltepunkte. Spezielle Haltepunkte, die die Source Map-Debuginformation bei deren Setzen miteinbeziehen, sind bei diesem Problem die Lösung. Die *wrapper*-Methoden und die wahrscheinlich zusätzlich in der Basisklasse generierten Felder beeinflussen die Sichten auf das Programm und somit die Grundanforderung der Visualisierung.

Bytecode Weaving und **Loadtime Weaving** und deren Einflüsse auf die Debuggerentwicklung wurden in den vorigen Abschnitten, im Zuge der Beispielsprachen AspectJ und Object Teams/Java hinreichend erläutert.

Basis-Aspekt-Komposition wird bei AspectWerkz unter anderem durch **Pseudo Runtime Weaving** mit HotSwap-Realisierung erreicht. Hierbei wird Bytecode innerhalb der JVM kurz von dessen Ausführung transformiert. Diese Webetechnik nutzt Codetransformation auf Bytecodeebene und beeinflusst dadurch die Bereiche Quellcodedarstellung und die Visualisierung. Durch die Bytecodetransformation entsteht bei AspectWerkz Infrastruktur- und semantisch

[17] In einer der nächsten Versionen wird OT/J eine Pointcut-Sprache unterstützen, die das Binden von Rollenmethoden an beliebige Join Points erlaubt.

äquivalenter Code (wrapper-Methoden und zusätzliche Felder), der durch zusätzliche Debuginformationen und Filter entsprecht behandelt werden kann, um die Grundanforderungen zu erfüllen.

Just-in-time Compiler Weaving muss durch die JVM unterstützt werden. Steamloom ist eine Virtual Machine, die um diese Fähigkeit erweitert wurde. Das Weben initiiert die Bytecodemanipulation der adaptierten Basismethode. Der native Code wird dadurch ungültig und erfordert ein Rekompilieren durch den JIT-Compiler. Der resultierende verwobene Bytecode ähnelt dem AspectJ Bytecode [25]. Die Einflüsse auf die Debuggerentwicklung sind somit die gleichen: Einfluss auf die Bytecode-Quellcodedarstellung und Einfluss auf die Sichten eines Programms. Die Lösungen dafür gestalten sich aber wahrscheinlich etwas schwieriger,da das Framework, das die Bytecodemanipulation veranlasst auch für die Erstellung von zusätzlichen Debuginformationen sorgen muss. Ob alle für die Erstellung benötigten Debuginformationen zu diesem Zeitpunkt vorhanden sind, ist fraglich.

Das Spring AOP Framework bedient sich dynamischer Proxys und ist somit ein Vertreter für das **Generic Runtime Weaving**. Dieses Weben erfordert keine Quellcode- oder Bytecodetransformation und hat somit keinen Einfluss auf die Entwicklung eines Debuggers. Ein unmodifizierter Standard Java Debugger kann verwendet werden. Lediglich ein Hinzufügen von Step-Filtern ist nötig, um die verwendeten Reflection-Bibliotheken während des Debuggings auszublenden.

5.4 Fazit

Debugger werden individuell für einzelne Sprachen entwickelt. Dieses Kapitel hat gezeigt, dass verschiedene AOP-Sprachen die Entwicklung eines Debugger trotzdem in ähnlicher Weise beeinflussen. Die Probleme, die durch die oft verwendete Codetransformation entstehen, können nur mit zusätzlichen Debuginformationen (Source Map) gelöst werden. Wann diese Debuginformationen erstellt werden müssen, hängt zwar von der verwendeten Webetechnik ab, ohne sie ist aber das Debugging aber nicht möglich. Auf besondere Features der Sprache muss bei der Entwicklung eines Debugger aber gesondert eingegangen werden.

6 Realisierung eines Debuggers für Object Teams/Java Programme

Im Kapitel 5.1 wurden Eigenschaften der Sprache Object Teams/Java aufgezeigt, die Einfluss auf die Realisierung eines Debuggers für Programme dieser Sprache nehmen. Anhand der einzelnen Grundanforderungen an interaktive Debugger wurden Lösungen, für die durch die Einflüsse auftretenden Probleme konzeptionell dargestellt. Diese Lösungen umfassen Erweiterungen am Object Teams Compiler, an der Object Teams Laufzeitumgebung und am *Eclipse Java Debugger*. In diesem Kapitel werden die Erweiterungen erläutert.

6.1 Umgebung

Der Debugger für Object Teams/Java Programme reiht sich in das *Object Teams Development Tooling* (OTDT), die Entwicklungsumgebung für Object Teams, ein. Dieses wiederum basiert auf Eclipse [4], einer Tool Platform mit integrierter Entwicklungsumgebung (IDE), welche durch Plug-ins erweiterbar ist. Die Entwicklungsumgebung unterstützt die Sprache Java, da die Eclipse Kern-Platform initial um ein Plug-in Werkzeugset für die Entwicklung von Java-Programmen erweitert wird. Das Plug-in Werkzeugset wird *Java Development Tools* (JDT) genannt und bietet Werkeugunterstützung in den Bereichen Java Debugging und Refactoring von Java Programmen. Der Java Compiler ist ebenfalls Bestandteil des JDT und als Plug-In realisiert. Die IDE bietet verschiedene Sichten auf Java Quellcode an.

Das *Object Teams Development Tooling* baut zum großen Teil auf den Java Development Tools auf und besteht aus mehreren Plug-ins. Object Teams/Java ist eine Erweiterung der Sprache Java und so erweitert das *Object Teams Development Tooling* das JDT. Dem Konzept folgend erweitert der Object Teams Compiler den Java Compiler. Die Realisierung des Object Teams/Java Debuggers wird durch Plug-ins verwirklicht, die den JDT Debugger erweitern werden.

6.2 Erweiterung des Object Teams/Java Compilers

Die Debuggerapplikation für Object Teams/Java Programme benötigt Debuginformationen, die im Bytecode der Class-Dateien existieren müssen. Ohne Debuginformationen ist das Debugging nicht möglich. Für die Erstellung der Debuginformationen ist der Object Teams/Java Compiler[1] zuständig. Der Compiler generiert das Sourcefile-Attribut, das Linenumbertable-Attribut und

[1] Die OT-Laufzeitumgebung muss Debuginformationen für den durch sie erstellten vermischten Code generieren.

das Localvariabletable-Attribut (siehe Kapitel 4.2.2). Die Erstellung des SourceDebugExtension-Attributs wurde im Rahmen dieser Diplomarbeit für den Object Teams/Java Compiler implementiert. Die im SourceDebugExtension-Attribut (siehe 4.2.3) enthaltene Source Map ermöglicht die komplexen Zuordnungen von kopiertem und semantisch äquivalentem Bytecode zu Quellcode, um die Grundanforderung der **Quellcodezuordnung** für den Object Teams/Java Debugger zu erfüllen. Der OT-Compiler sammelt die dafür notwendigen Informationen während der einzelnen Compilerphasen und generiert das SourceDebugExtension-Attribut, um die erstellte Source Map darin zu speichern. Der Object Teams/Java Debugger liest diese und alle anderen Debuginformationen aus, um sie für die Darstellung zu nutzen.

6.2.1 Bytecode-Quellcode-Zuordnung

Das Sourcefile-Attribut, das Linenumbertable-Attribut und das SourceDebugExtension-Attribut enthalten die Debuginformationen, die für die Zuordnung von Bytecode zu Quellcode notwendig sind. Die Informationen werden während des Kompilierungsvorgangs gesammelt und am Ende geschrieben. Dabei wird auf Datenstrukturen und Modelle des OT-Compilers zugegriffen. Der OT-Compiler basiert auf dem JDT-Compiler und so wurden die Modelle des JDT-Compilers genutzt und erweitert.

Abstract Syntax Tree (AST)

Der *abstrakte Syntaxbaum* (AST) ist eine abstrakte Sicht auf eine Java-Quellcodedatei und ist eines dieser Modelle. *Scanner* und *Parser* sind Teile des Compilers und führen die lexikalische und syntaktische Analyse des Quellcodes durch. Der Parser erstellt mit Hilfe des Scanners den AST, in dem eine Java-Quelldatei als Baumstruktur dargestellt wird. Das Wurzelelement des AST ist die `CompilationUnitDeclaration` und enthält wiederum Elemente (AST-Knoten), wie den in der Java-Quelldatei vorkommenden Typ (`TypeDeklaration`) und dessen Methoden (`MethodDeclaration`). Die Implementierung des AST in Eclipse besteht aus über 100 Klassen und deckt alle Java-Sprachelemente vollständig ab. Der AST wurde um die durch Object Teams hinzugekommenen Sprachelemente erweitert. Der Object Teams Compiler benötigt dieses Modell während des Kompilierungsvorgangs.

6.2.1.1 Source Map Generierung

Eine Source Map beschreibt Zuordnungen zwischen Codepositionen einer Eingabesprache und Codepositionen der generierten Ausgabesprache. Eine Sicht auf den generierten Code der Ausgabesprache wird Stratum genannt. Die Übersetzung von Object Teams/Java Quellcode in Java Bytecode wird durch den Compiler realisiert und lässt sich damit dem Szenario I (siehe Kapitel 4.2.3) zuordnen. Der Compiler ist für das Erstellen der Source Map verantwortlich. Es existiert bei Object Teams nur ein Stratum (Quellcodeebene) - das *OTJ Stratum*. Eine Source Map besteht laut Spezifikation aus:

Header - Der *Header* identifiziert die Daten als Source Map, indem in der erste Zeile das Wort "SMAP" geschrieben steht. Die nächste Zeile ist der Name der generierten Datei (z.B. SomeType.class). Die letzte Zeile des Headers gibt das Standard-Stratum an und legt somit fest, welche Quellcodeebene während des Debuggings dargestellt werden soll.

StratumSection - Eine *StratumSection* repräsentiert eine Quellcodeebene und wird mit "*S" und dem Namen der Quellcodeebene eingeleitet. *FileSection* und *LineSection* enthalten die genauen Zuordnungsinformationen und sind Bestandteil der StratumSection.

FileSection - In der *FileSection* werden die Quellcodedateien beschrieben, denen der Bytecode zugeordnet wird. Mehrere Dateinamen können dabei angegeben werden, wobei ein Eintrag aus einer Datei-Identität (Numerischer Wert), dem Dateinamen und einer optionalen Dateipfadangabe besteht. Die FileSection wird mit "*F" eingeleitet.

LineSection - Die *LineSection* assoziiert eine Zeilennummer der Ausgabesprache mit der Zeilennummer der Eingabesprache und wird mit "*L" eingeleitet. Die LineSection beinhaltet Zeileninformationen (*LineInfos*). Eine Zeileninformation ordnet eine bestimmte Zeilennummer der Ausgabesprache einer bestimmten Zeilennummer der Eingabesprache zu, wobei eine Zeileninformation eine Referenz auf eine der angegebenen Dateien der FileSection besitzt.

VendorSection - Die *VendorSection* enthält spezifische Informationen des Vendors (siehe *Unique Package Names* in [45]) und wird durch "*V" eingeleitet.

EndSection - Das Ende und somit die letzte Zeile einer Source Map wird durch die *EndSection* ("*E") angegeben.

In Abbildung 6.1 ist ein Beispiel einer vom Object Teams/Java Compiler erstellten Source Map dargestellt. Der linke Bereich identifiziert die einzelnen Sections der Source Map und der rechte Bereich erklärt die Daten der Sections genauer.

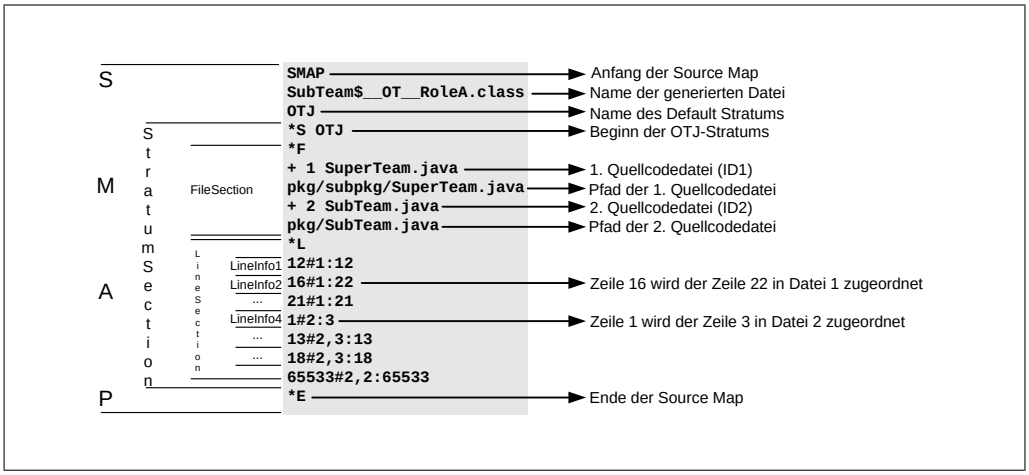


Abbildung 6.1: Beispiel einer Source Map

Der Compiler bezieht die benötigten Informationen während der einzelnen Compilerphasen aus dem AST und speichert sie in den entsprechenden Datenstrukturen ab. Das Klassendiagramm (Abbildung 6.2) zeigt die dazu verwendeten Klassen. Jede Section des Stratums wird durch eine Klasse repräsentiert. Die Klasse `SmapStratum` beschreibt eine Quellcodeebene und erzeugt und speichert FileInfo- und LineInfo-Referenzen. Obwohl Rollen auf Quellcodeebene Bestandteil des Teams sind, werden Rollen in eigene Class-Dateien kompiliert. Die Rollen-Class-Dateien enthalten einen Verweis auf die Team-Class-Datei und darüber hinaus eigene Bytecode-Attribute. Deshalb können Teams und Rollen jeweils eine Source Map enthalten. Die Source Map der Rolle

beschreibt die Zuordnungen des Rollen-Bytecodes zum Rollen-Quellcode und die Source Map der Teams beschreibt dessen Bytecode-Quellcode-Zuordnung. Es wurden deshalb zwei Source Map Generatoren (`TeamSmapGenerator` und `RoleSmapGenerator`) implementiert. Eine Rolle kann auch ein Team sein (*nested Team*). Deshalb erbt `RoleSmapGenerator` vom `TeamSmapGenerator`. Bei *nested Teams* wird die Zuordnungsbeschreibung des Team-Bytecodes zu Teamquellcode um die Zuordnungsbeschreibung des Rollen-Bytecodes zum Rollenquellcode erweitert und in **einer** Source Map gespeichert.

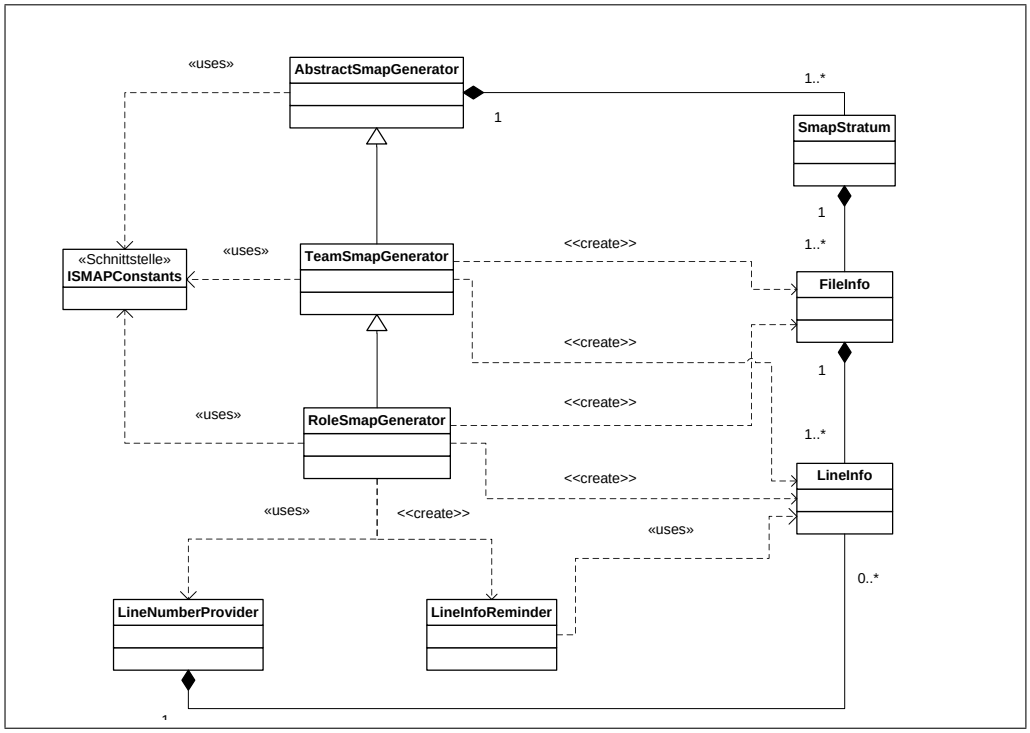


Abbildung 6.2: Klassendiagramm (Übersicht über die Klassen, die an der Generierung der Source Map beteiligt sind.)

Die Zuordnungen innerhalb der Source Map werden mittels Zeilennummern beschrieben. Die Zeilennummer des Bytecodes wird auf eine Zeilennummer des Quellcode abgebildet. Die Zeilennummer des Bytecodes wird dabei der Linenumbertable entnommen. Handelt es sich bei dem Bytecode um kopierten Bytecode, wird auch das Linenumbertable-Attribut kopiert. Zu diesem Zeitpunkt ist die Zuordnung zwischen dem Bytecode der Methode und dem Quellcode fehlerhaft. Die Einträge der Linenumbertable werden deshalb mit Hilfe der Klasse `LineNumberProvider` neu erstellt. In der Linenumbertable von kopierten Bytecode werden Einträge mit gültigen[2] Zeilennummern erzeugt. Mit Hilfe der Source Map, verweisen diese Zeilennummern auf den Ursprungsquellcode.

[2]Gültig bedeutet hier, dass die Zeilennummer noch nicht vergeben wurde und somit keinen anderen Eintrag der Linenumbertables innerhalb einer Class-Datei dupliziert.

6.2.1.2 Erstellung der SourceDebugExtension

Nachdem die Source Map erstellt wurde, muss sie in die Class-Datei geschrieben werden. Dafür erstellt der Compiler das SourceDebugExtension-Attribut. Das SourceDebugExtension-Attribut ist ein Classfile-Attribut und beinhaltet die Source Map als `String` im UTF-8 Format. Bei der Erstellung der Source Map ist besonders auf die Größe zu achten. Das SourceDebugExtension-Attribut darf nicht größer als 65535 Bytes sein, da sonst ein Fehler von der Virual Machine geworfen wird. Um das zu Umgehen, wird ein Optimierungsalgorithmus auf der Source Map ausgeführt, um Einträge zusammenzufassen. Danach wird die Source Map erst geschrieben.

6.2.2 Markierung von Infrastrukturcode

Der Object Teams Compiler markiert Infrastrukturcode mit speziellen Zeilennummern. Infrastrukturcode kann dabei mit zwei verschiedenen Zeilennummern versehen werden - mit der **StepOverLinenumber** und der **StepIntoLinenumber**. Für diese Zeilennummern wurden die höchst möglichen Einträge in der Linenumbertable gewählt - **65534** und **65533**. Beim Auftreten dieser Zeilennummern behandelt die Debuggerapplikation den damit markierten Bytecode. Die verschiedenen Behandlungsarten werden im Kapitel 6.4 erläutert.

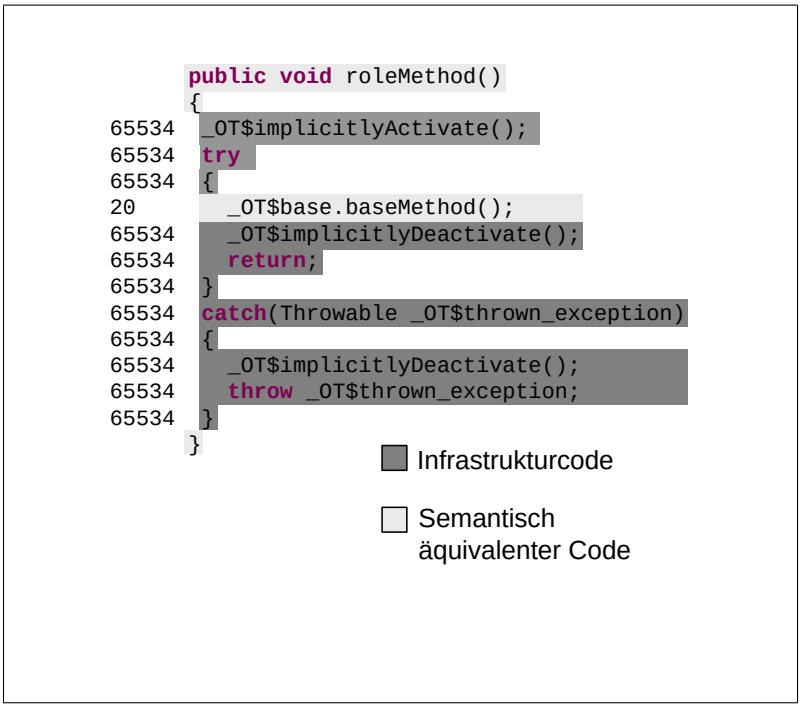


Abbildung 6.3: Transformiertes Callout mit Zeilennummern

Ein durch den Compiler transformiertes Callout wird mit den in Abbildung 6.3 dargestellten Zeilennummern versehen[3]. Der Infrastrukturcode bekommt hier die spezielle Zeilennummer

[3] Das bedeutet, dass Einträge der Linenumbertable erstellt werden, die die generierten Bytecodeinstruktionen mit diesen Zeilennummern verknüpfen.

65534 und der semantisch äquivalente Code bekommt eine "normale" Zeilennummer (in diesem Beispiel die Zeilennummer 20), die mit Hilfe der Source Map auf Quellcode abgebildet wird.

6.3 Erweiterung der Object Teams/Java Laufzeitumgebung

Im Kapitel 5.1.2.2 wurde der Webemechanismus von Object Teams vorgestellt. Es wurde deutlich, dass die Object Teams Laufzeitumgebung mit Hilfe von Informationen (spezielle Class-Datei Attribute), die der Compiler erstellt, Transformationen an der Basisapplikation vornimmt. Es entsteht vermischter Bytecode (Definition siehe Kapitel 5.1.2.3). Um diesen Bytecode während des Debuggings Quellcode zuordnen zu können, muss der Generator des Codes[4] auch die Debuginformationen erstellen. Neben der Erweiterung des Object Teams Compilers um die Fähigkeit der Erstellung und Speicherung der Source Map, muss auch die Object Teams Laufzeitumgebung um diese Fähigkeit erweitert werden.

6.3.1 Bytecode-Quellcode-Zuordnung

Die Informationen, die die Object Teams Laufzeitumgebung für die Erstellung der Source Map benötigt, muss der Compiler als Attribute in den Class-Dateien der Rollen abspeichern. Zum Einen muss der Compiler die benötigen Informationen erstellen, da sie nur während des Kompilierens vorhanden sind und zum Anderen werden sie in den Class-Dateien der Rollen abgespeichert, da die Basisapplikation[5] nicht durch den Compiler verändert werden darf.

6.3.1.1 Source Map Generierung + Erstellung der SourceDebugExtension

Das Erstellen der Source Map übernimmt eine Klasse innerhalb der Object Teams Laufzeitumgebung - die Klasse `SourceMapGeneration`. Die dazu benötigten Informationen wurden durch den Compiler im CallinMethodMappings-Attribut bereitgestellt. Das ist ein Object Teams spezifisches Classfile-Attribut in der Rollen-Class-Datei. Das Attribut enthält zum Einen den Pfad der Quellcodedatei, in der die Callin-Methodenbindung deklariert wurde und zum Anderen die Zeilennummer der Deklaration. Mit Hilfe dieser Informationen wird eine Source Map erstellt, die die Aspektbytecodeteile des Vermischten Codes dem Aspektquellcode und die Basisbytecodeteile dem Basisquellcode zuordnet.

Die Aufgabe der Vergabe von gültigen Zeilennummer für neu generierten und kopierten Bytecode, wird auf Seiten des Compilers von der Klasse `LineNumberProvider` übernommen. In der Laufzeitumgebung stellt die Klasse `SourceMapGeneration` diese Funktionen zur Verfügung und ist ebenfalls für die Generierung der Source Map zuständig.

Nach der Generierung der Source Map wird das SourceDebugExtension-Attribut erstellt und die Source Map in den Class-Dateien der Basisapplikation gespeichert. Die Debuginformationen Localvariabletable-Attribut und Linenumbertable-Attribut werden ebenfalls für den neu generierten und transformierten Bytecode durch die Laufzeitumgebung erstellt.

[4] Der Codegenerator ist hier nicht der Compiler, sondern die Laufzeitumgebung.

[5] Genauer gesagt, dürfen die Class-Dateien der Basisapplikation nicht verändert werden, da es sich beim verwendeten Webemechanismus um *Weben zur Ladezeit* handelt.

6.3.2 Markierung von Infrastrukturcode

Die Laufzeitumgebung generiert Vermischten Code und Infrastrukturcode. Für die Markierung des Infrastrukturcodes auf Statementlevel ist somit die Object Teams Laufzeitumgebung verantwortlich. Diese Markierung verläuft analog zur Markierung durch den Compiler.

6.4 Erweiterung des Java Debuggers

Die Grundlage des Debuggers für Object Teams/Java ist der Eclipse Java Debugger. Der Java Debugger ist Bestandteil des JDT und implementiert basierend auf der *Eclipse Debug Platform* (siehe Kapitel 6.4.1) spezifische Unterstützung für das Debugging von Java Programmen. Der Debugger des JDT implementiert die JDI-Schnittstelle der Java Platform Debugger Architecture (siehe Kapitel 4.2.1) und liest die Debuginformationen der ausgeführten Programme aus. Die Implementierung des JDT-Debuggers beinhaltet eine graphische Benutzerschnittstelle, die die in Kapitel 2.1.3.1 erwähnten Sichten bietet. Diese Sichten werden in der *Debug Perspektive*[6] zusammengefasst. Die Benutzerschnittstelle des JDT-Debuggers ist in Kapitel 3.3.1 genau beschrieben.

6.4.1 Eclipse Debug Platform

Die Eclipse Debug Platform soll die Entwicklung eines Debuggers für Programme beliebiger Sprachen vereinfachen und bietet dafür das *Launching Framework*, das *Debug Model*, das *Source Lookup Framework* und die *Debug Model Presentation*.

Launching Framework

Das Launching Framework ermöglicht das Starten von Programmen aus Eclipse heraus. Ein Programm kann dabei im run-Modus und im debug-Modus ausgeführt werden. Wird das Programm im debug-modus gestartet, so wird das Programm mit entsprechender Debuggingunterstützung ausgeführt. Werden Java-Programme im debug-Modus gestartet, ist die JVMTI-Schnittstelle der Virtual Machine, in der die Java-Programme ausgeführt werden aktiv. Im run-Modus sind die Debug-Schnittstellen inaktiv.

Object Teams/Java stellt einen eigenen Launching-Mechanismus bereit. In diesem ist das JMangler Framework (siehe Kapitel 5.1.2.2) integriert, um das *Weben zur Ladezeit* zu ermöglichen.

Debug Model

Das Debug Model definiert Abstraktionen von Elementen, die während des Debuggings eines Programms einzelnen Programmteilen zugeordnet werden können. Es existieren weiterhin Aktionen, die von diesen Elementen angeboten werden und Ereignisse, die von Aktionen ausgelöst werden können.

Debug Elemente: Das Debug Modell definiert folgende Programmelemente(Debug Elemente):

[6] Eine Perspektive ist in Eclipse eine Sammlung von Views. Views sind Sichten auf Informationen. Die Variablen-Sicht ist ein Beispiel einer View im Debug-Kontext.

Debug Target - Das *Debug Target* stellt einen debugbaren Kontext, wie einen Prozess oder eine Virtual Machine dar und ist das Wurzelelement in der Hierarchie der Debug-Elemente. Das Debug Target enthält die Threads, die zu ihm gehören.

Threads - Innerhalb eines Debug Targets sind *Threads* eine Reihe von Ausführungsflüssen. Ein Thread enthält Stack Frames, auf die nur zugegriffen werden kann, wenn der Thread angehalten hat.

Stack Frame - *Stack Frames* sind Ausführungskontexte eines Threads, der angehalten wurde. Sie enthalten lokale Variablen und Methodenparameter.

Variable/Value - *Variablen* repräsentieren Datenstrukturen innerhalb eines Stack Frames. Values sind die Werte der Variablen.

Expression - Eine *Expression* ist ein Stück Code. Dieser Code wird evaluiert und gibt einen Wert zurück.

Debug Aktionen: Die beschriebenen Debug Elemente unterstützen Aktionen, die während des Debuggings ausgeführt werden können. Verschiedene Debug Elemente implementieren verschiedene Debug Aktionen (Debug Actions). Zu diesen Aktionen gehören:

Disconnect - Ein *Disconnect* beendet eine Ausführung des Ziel-Programms im debug-Modus. Danach wird das Ziel-Programm im run-Modus weiter ausgeführt.

Step - Die Aktion *Step* beschreibt die Möglichkeit, das Programm mit den Schrittmodi Step Into, Step Over oder Step Return vom aktuellen Ausführungspunkt aus, weiter auszuführen.

Filtered Step - Mit der Aktion *Filtered Step* kann das Programm mit oben genannten Schrittmodi ausgeführt werden, wobei in zu filternden Elementen nicht gehalten wird.

Suspend/Resume - Ein Ausführungskontext kann angehalten(*Suspend*) und danach weiter ausgeführt(*Resume*) werden.

Terminate - Ein Ausführungskontext, zum Beispiel ein Thread oder ein Debug Target kann mittels Terminate-Aktion beendet werden.

Value Modification - Die Aktion *Value Modification* ändert den Wert einer Variablen in einem Debug Target.

Debug Ereignisse: Debug Ereignisse(Debug Events) sind Ereignisse, die während des Debuggings eines Programms auftreten können. Debug Ereignisse beinhalten die Debug Elemente, die mit dem Ereignis assoziiert sind. Zu den Ereignissen gehören:

Create - Das *Create*-Ereignis tritt auf, wenn neue Debug Elemente (Debug Target oder Thread) erstellt werden.

Terminate - Das *Terminate*-Ereignis tritt auf, wenn existierende Debug Kontexte (Debug Target oder Thread) beendet werden.

Suspend - Wird ein Debug Target oder ein Thread angehalten, wird ein *Suspend*-Ereignis kreiert. Das Ereignis beinhaltet die Ursache des Auftretens(Erreichen eines Haltepunktes oder das Ende eines Schrittes (Step)).

Resume - Ein *Resume*-Ereignis wird erzeugt, wenn ein Debug Element (Debug Target oder Thread) weiter ausgeführt wird. Das Resume-Ereignis enthält die Ursache der Erzeugung (zum Beispiel eine Step-Aktion).

Sprachspezifische Debugger implementieren und erweitern dieses Modell, um sprachspezifisches Verhalten hinzuzufügen.

Source Lookup Framework

Die Aufgabe des *Source Lookup Frameworks* ist das Finden und Anzeigen der Quellcodestelle, die zum aktuellen Stack Frame gehört. Nach Selektion eines Stack Frames findet das Source Lookup Framework das zum Stack Frame zugehörige Quell-Element und mit Hilfe der Debug Model Presentation wird die entsprechende Stelle im Quellcode-Editor angezeigt.

Debug Model Presentation

Die *Debug Model Presentation* ist ein Modell für die Darstellung der Debug Model Elemente. Die Benutzeroberfläche eines interaktiven Debuggers in Eclipse besteht aus einzelnen Sichten. Diese benutzen das Modell für das Anzeigen der Debug Model Elemente. Die Debug Model Presentation stellt Bilder und Beschriftungen(labels) für die einzelnen Debug Model Elemente zur Verfügung. Des weiteren ermöglicht die Debug Model Presentation die Zuordnung der Debug Model Elemente zu deren entsprechenden Quellcode-Editor.

Haltepunkte in Eclipse

Die Infrastruktur für das Setzen, Verwalten und die Persistenz von Haltepunkten besteht in Eclipse aus folgenden Komponenten:

Breakpoint extension - Mittels *Breakpoint extension* werden die Typen der Haltepunkte in der Eclipse Debug Platform bekannt gemacht.

Marker - *Marker* werden benutzt, um zum Beispiel die Attribute eines Haltepunktes (Typname und Zeilennummer) zu speichern. Marker werden mit Quellcode verknüpft und werden als Meta-Informationen des Quellcodes gespeichert.

Breakpoint Manager - Der Breakpoint Manager ist für die Verwaltung der Haltepunkte zuständig und wird benachrichtigt, wenn Haltepunkte erzeugt oder entfernt werden.

Debug Target - Das Debug Target aus dem Debug Model ist für das Installieren der Haltepunkte in den ausführbaren Programmen zuständig.

6.4.2 Bytecode-Quellcode-Zuordnung

Die für Bytecode-Quellcode-Zuordnung erforderlichen Debuginformationen werden vom Object Teams Compiler und der Object Teams Laufzeitumgebung in den Bytecode der Class-Dateien geschrieben. Das Generieren und Schreiben der Debuginformation SourceDebugExtension wurde genau in vorherigen Kapiteln erklärt. Die Debuginformationen müssen vom Debugger ausgelesen und interpretiert werden. Der Java Debugger des JDT unterstützt das Auslesen und die Interpretation des Sourcefile- und Linenumbertable-Attributs. Darüber hinaus wird auch das Auslesen der optionalen Debuginformation SourceDebugExtension-Attribut unterstützt. Die JDT-Debugger-Implementierung der JDI-Schnittstelle erlaubt das Auslesen der in der Source Map beschriebenen Quellcodedateien und Quellcodepositionen. Anhand einer

Callout-Methodenbindung wird der Ablauf vom Kompilieren bis hin zur Quellcodehervorhebung während des Debuggingvorgangs illustriert (Abbildung 6.4).

1. Nachdem der Quellcode erstellt wurde, wird er kompiliert. Durch den Object Teams Compiler wurde der Bytecode mit Debuginformationen generiert. In der Zeile 7 in `RoleA.java` wurde ein Zeilenhaltepunkt gesetzt und das Programm soll im debug-Modus gestartet werden. Das Launching Framework hat die Informationen, dass es sich bei dem zu startenden Programm um ein Object Teams Programm handelt und das im debug-Modus ausgeführt werden soll.

2. Das Launching Framework startet das Object Teams Programm mit dem speziellen JMangler Framework, um das Weben zur Ladezeit zu ermöglichen. Da das Programm (Class-Datei) im debug-Modus ausgeführt wird, werden bei beiden Virtual Machines (JVM des Front-Ends Debuggee-JVM) die entsprechenden Debug-Schnittstellen (JDI- und JVMTI-Schnittstelle) aktiviert (siehe Kapitel 4.2.1).

3. Im Bytecode (vergrößerte Ansicht) sind unter anderem die dargestellten Debuginformationen enthalten. Der dekompilierte Bytecode des Callouts (`roleMethod`) und die dazugehörige Linenumbertable sind im oberen Bereich der Abbildung dargestellt. Bei Erreichen des Zeilenhaltepunkts, liest die Debuggerapplikation das Linenumbertable-Attribut der `roleMethod` über die JDI-Schnittstelle aus. Die 7. Zeile soll laut Linenumbertable in der Quellcodedatei `RoleA.java` angezeigt werden.

4. Der JDT-Debugger liest die Source Map (SMAP) aus dem Bytecode aus. Die Source Map beschreibt, dass die Zeilennummern von 1 bis 8 und die Zeilennummern 65533 bis 65534 auf die gleichen Zeilennummern in der Quelldatei `RoleA.java`, deren Pfad "TeamA/RoleA.java" ist, abzubilden sind. Die Zeile 7 (Eintrag der Linenumbertable) wird laut Source Map auf die Zeile 7 in der Quellcodedatei `TeamA/RoleA.java` abgebildet. Die FileSection der Source Map setzt das Sourcefile-Attribut außer Kraft und ersetzt die Quelldatei `RoleA.java` durch `TeamA/RoleA.java`.

5. Die Debuggerapplikation übergibt dem Source Lookup Framework den Pfad der anzuzeigenden Quellcodedatei und die Zeilennummer.

6. Das Source Lookup Framework findet die Quellcodedatei (Source Element) und reicht die Daten an die Debug Model Presentation weiter.

7. Die Debug Model Presentation ordnet dem Source Element (`TeamA/RoleA.java`) einen Editor zu und veranlasst die Eclipse Platform den Editor zu öffnen und die Quellcodedatei darzustellen. Die Zeilennummer 7 wird hervorgehoben.

Die Informationen, welcher Quellcode, wann und wo während eines Debuggingvorgangs dargestellt werden soll, befinden sich im Bytecode und steuern dadurch die Debuggerapplikation. Da der JDT-Debugger die vom Object Teams Compiler und der Object Team Laufzeitumgebung erstellten Debuginformationen und vor allen das SourceDebugExtension-Attribut interpretieren kann, sind in diesem Bereich keine Erweiterungen am Debugger notwendig.

6.4.3 Besonderheiten bei der Behandlung von Code

Während des Debuggings von Object Teams Programmen wird Bytecode ausgeführt, der zum Einen keine Quellcodeentsprechung besitzt und zum Anderen aus Bibliotheken stammt, deren

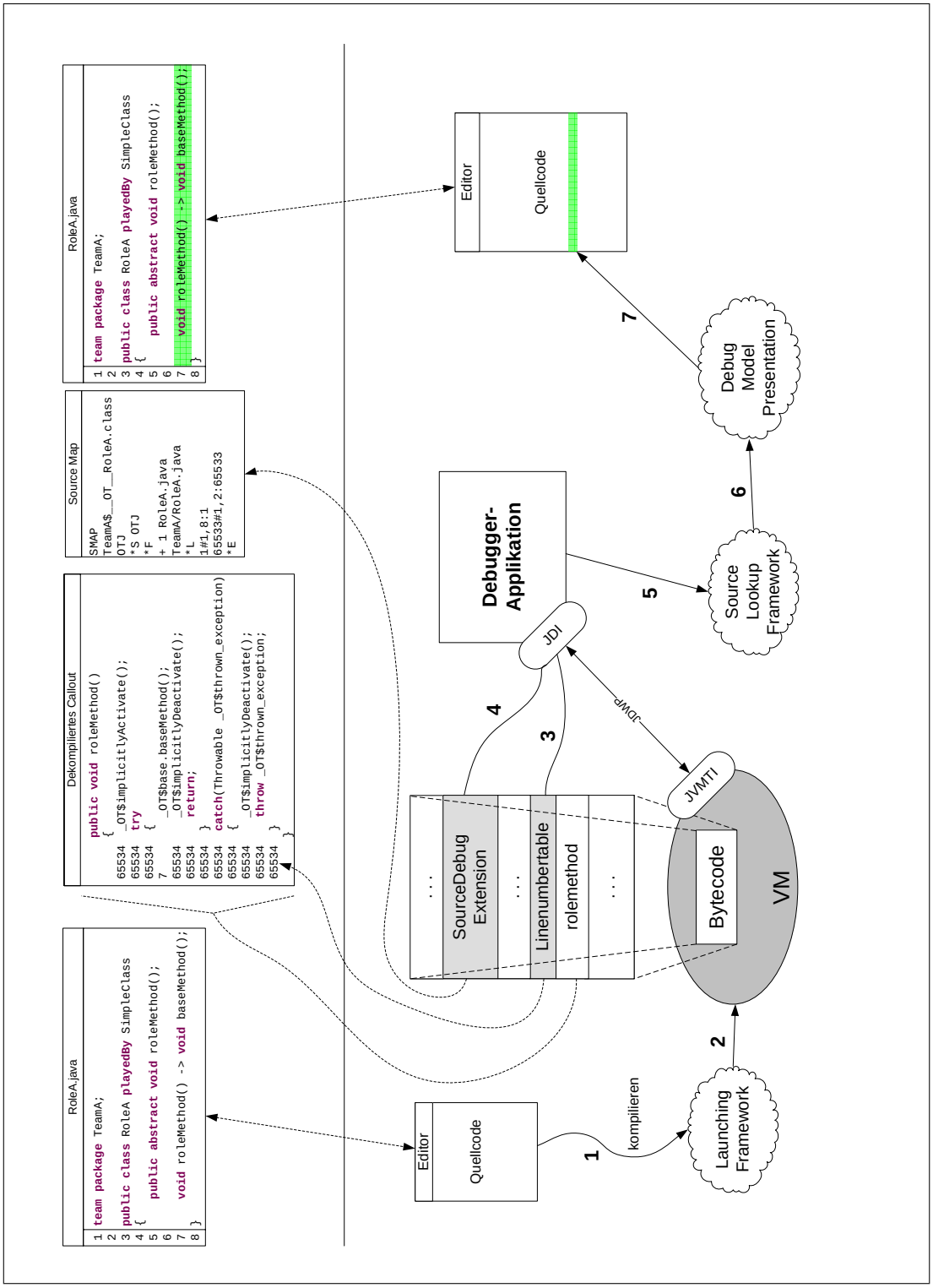


Abbildung 6.4: Bytecode-Quellcode-Zuordnung mit JSR-045 Unterstützung

Existenz für den Entwickler von Object Teams Programmen transparent sein sollte. Dieses Kapitel beschreibt, wie dieser Bytecode vom Object Teams Debugger behandelt wird.

6.4.3.1 Behandlung von Bibliotheksklassen

Um das Weben zur Ladezeit zu ermöglichen, werden das JMangler Framework und das BCEL Framework verwendet. Die Klassen aus diesen Bibliotheken und die Klassen aus der Object Teams Laufzeitumgebung sorgen für die Transformation der Basis- und Rollenklassen zur Ladezeit. Die Transformation soll für den Benutzer transparent sein. Die Pakete werden deshalb auf Seite der Debuggerapplikation gefiltert. Dafür wird die von der Debug Platform angebotene Step-Filter Infrastruktur programmatisch erweitert. Um das Halten des Programms in den Bibliotheksklassen während des Debuggings zu vermeiden, werden die Pakete "de.fub.bytecode.*", "gnu.regexp.*", des BCEL-Frameworks und das "org.cs3.jmangler.*"-Paket des JMangler-Frameworks und all ihre Unterpakete den Step-Filtern hinzugefügt. Das "org.objectteams"-Paket der OT-Laufzeitumgebung enthält Klassen und Unterpakete. Zu den Klassen gehört das `org.objectteams.Team`. Dieses darf während des Debuggings nicht gefiltert werden, um das Anzeigen der geerbten Teammethoden (siehe Kapitel 5.1.1.1) zu ermöglichen. Die Unterpakete von "org.objectteams" und somit alle internen Klassen der Object Teams-Laufzeitumgebung werden gefiltert.

Das Filtern der Klassen und Pakete der verwendeten Frameworks schafft aber ein neues Problem. Wird bei der Entwicklung von Object Teams Programmen auf gefilterte Pakete zugegriffen oder werden sie erweitert, so würden die Filter auch das eigene Programm betreffen. Es wäre nicht möglich, Quellcode der gefilterten Pakete anzeigen zu lassen, obwohl sie Bestandteil des eigenen Programms sind. Dieses Problem lässt sich zwar lösen, indem die Filter ausgeschaltet werden. Es werden nun aber auch Framework-Klassen während des Webens angezeigt.

6.4.3.2 Behandlung von Infrastrukturcode

Die Markierung von Infrastrukturcode, also von zusätzlich generierten Bytecode der keinen Quellcode zugeordnet werden kann, nehmen bei Object Teams/Java der OT-Compiler und die OT-Laufzeitumgebung vor. Die Markierung durch das "synthetic"-Attribut spielt bei Object Teams Bytecode eine untergeordnete Rolle, da Infrastrukturcode selten eine komplette Klasse oder Methode ist. Bei der Transformation durch Compiler und Laufzeitumgebung entsteht Infrastrukturcode auf Statementlevel. Die Bytecodeinstruktionen des Infrastrukturcodes bekommen in der Symboltabelle des Linenumbertable-Attributs die Zeilennummern **65533** und **65534**, also die StepOver- und StepIntoLinenumber. Die Debuggerapplikation liest die Einträge der Symboltabelle aus und wenn der Debugger während der schrittweisen Ausführung auf diese Zeilennummern trifft, initiiert der Debugger je nach Zeilennummer automatisches, benutzerunabhängig ausgeführtes, schrittweises Debugging. In welchen Schrittmodi die Ausführung passiert, entscheidet die Zeilennummer:

StepOverLinenumber -
: Infrastrukturcode, der die StepOverLinenumber besitzt, soll übersprungen werden. Der Object Teams Debugger muss dazu benachrichtigt werden, wenn ein Halten auf Bytecode bevorsteht, dessen Zeilennummer die StepOverLinenumber ist. Dazu implementiert die Klasse `StepFromLinenumberGenerator` das Interface `IDebugEventFilter`. Die Klasse kann so auf Debug-Ereignisse reagieren. Steht ein Halten auf Code mit der StepOverLinenumber bevor, wird das Ereignis gefiltert und eine neue Debug-Aktion initiiert - die

Step Aktion. Die initiierte Step Aktion hat die Schritttiefe Step Over. Die Ausführung des Programms wird somit nicht gestoppt, sondern solange fortgeführt, bis Code erreicht wird, der nicht die StepOver- oder die StepIntoLinenumber besitzt.

StepIntoLinenumber -
Infrastrukturcode mit der StepIntoLinenumber soll nicht übersprungen, sondern in die Ausführung des Programms miteinbezogen werden. Die *chaining wrapper*-Methode, die durch die Transformation der adaptierten Basisklassen entsteht, beinhaltet einen rekursiven Aufruf. Dieser Aufruf besitzt die StepIntoLinenumber. Methodenaufrufe, die einerseits zum Infrastrukturcode gehören, die aber andererseits Methoden aufrufen, die nicht zum Infrastrukturcode gehören, bekommen die StepIntoLinenumber. Die Klasse `StepFromLinenumberGenerator` wartet auf ein Debug-Ereignis, was ein Halten des Programms auf der StepIntoLinenumber bewirken würde und initiiert eine Step Aktion mit der Schritttiefe Step Into. Die Methode wird ausgeführt und hält an der ersten Bytecodeinstruktion, sofern sie nicht der StepOver- oder die StepIntoLinenumber-Zeilennummer zugeordnet ist.

6.4.4 Haltepunkte

Um das Setzen von Haltepunkten in Object Teams Programmen zu ermöglichen, muss in den dafür vorhandenen Mechanismus eingegriffen werden. Die bei der *Kontrolle durch Haltepunkte/Schrittmodi* erwähnten Probleme werden auf Implementierungebene wie folgt gelöst:

Das Setzen von Haltepunkten bei Impliziter Vererbung -
Um das Setzen eines Zeilenhaltepunktes in eine Rolle und somit ein Setzen des Zeilenhaltepunktes in alle kopierten Bytecodeversionen zu ermöglichen, wird eine Typhierarchy für die Rolle aufgebaut, wenn der Haltepunkt gesetzt wird. Mit Hilfe der Hierarchy werden alle Subrollen der Rolle gefunden. Dazu gehören alle durch implizite Vererbung entstandenen Subrollen[7]. Das Kopieren des Bytecodes und die Generierung der Source Map bewirken, dass sich die Linenumbertable-Einträge der Originalmethoden und der kopierten Methoden unterscheiden. Um den Zeilenhaltepunkt in alle Subrollen setzen zu können, muss die zum Original korrespondierende Zeilennummer der kopierten Methode ermittelt werden. Der initial gesetzte Zeilenhaltepunkt wird danach in jede Bytecodekopie an die entsprechende Zeile gesetzt. Wird ein Haltepunkt in einer Bytecodekopie erreicht, wird mit Hilfe der Source Map der Quelltext der Originalrolle angezeigt. Das Setzen der Zeilenhaltepunkte in die Bytecodekopien ist somit für den Benutzer transparent.

Das Setzen von Haltepunkten in Rollendateien -
Für das Setzen von Haltepunkten wird der vollständig qualifizierte Klassenname verwendet. Dabei darf nicht der Klassenname des in der Quellcodedatei enthaltenen Typs, sondern der Name, der durch den Compiler generierten Klasse benutzt werden. Der vollständig qualifizierte Klassenname `pkg.MyTeam.MyRole` wird durch den Namen `pkg.MyTeam$__OT__MyRole` ersetzt und für das Setzen der Haltepunkte verwendet.

Das Setzen von Haltepunkten ist durch die oben erklärten Erweiterungen in gewohnter Weise möglich. Die vom JDT-Debugger angebotenen Haltepunkttypen können somit auch für Object Teams Programme verwendet werden.

[7] Die Hierarchy findet alle Subrollen, egal ob sie nur im Bytecode existieren, oder ob sie im Quellcode redefiniert wurden (siehe Kapitel 5.1).

Aktivierungshaltepunkt

Ein neuer Haltepunkttyp, der nur in Object Teams Programmen, genauer gesagt nur in Teams, gesetzt werden darf, ist der Aktivierungshaltepunkt. Dieser nutzt die *Breakpoint extension*, um innerhalb der Eclipse Debug Platform als Haltepunkt eingeführt zu werden. Um einen Haltepunkt dieses Typs zu setzen, muss ein Team ausgewählt werden. Ob das Object Teams Programm bei Aktivierung oder Deaktivierung oder bei beiden Ereignissen halten soll, kann in den Bedingungen des Haltepunkt angegeben werden.

Die Realisierung des Object Teams-Debuggers basiert auf dem Debug Model des JDT. Das Setzen und Entfernen von Haltepunkten findet somit über Methoden statt, die von Elementen des Debug Models angeboten werden. Die Infrastruktur des Debug Models und des JPDA wird genutzt und nicht verändert, sondern gegebenenfalls erweitert. Das bietet den Vorteil, dass alle Java spezifischen Haltepunkttypen und die Anzeigen (zum Beispiel die Breakpointsicht) durch den Object Teams Debugger genutzt werden können.

6.4.5 Team-Monitor

Der Team-Monitor ist die Umsetzung des Aspekt-Monitors bei Object Teams. Die Aktivierung der Aspekte wird bei Object Teams durch Team-Aktivierung realisiert. Während einer Programmausführung kann es zu wechselnder Aktivierung und Deaktivierung eines Teams kommen. Der Team-Monitor visualisiert die Änderungen, indem Haltepunkte in den Aktivierungs- und Deaktivierungsmethoden des `org.objectteams.Team` gesetzt werden. Beim Erreichen der Haltepunkte erzeugt die Virtual Machine ein Debug-Ereignis, das eine Klasse der Team-Monitor-Implementierung abfängt und behandelt. Das Programm wird kurz angehalten, um benötigte Variablen auszulesen. Der Teamaktivierungstatus wird ermittelt und die Daten an die erweiterte Debug Model Presentation weitergeleitet. Die zur Laufzeit aktivierten Team-Instanzen und deren Teamaktivierungstatus werden in der Team-Monitor-Sicht angezeigt (siehe Abbildung 6.5).

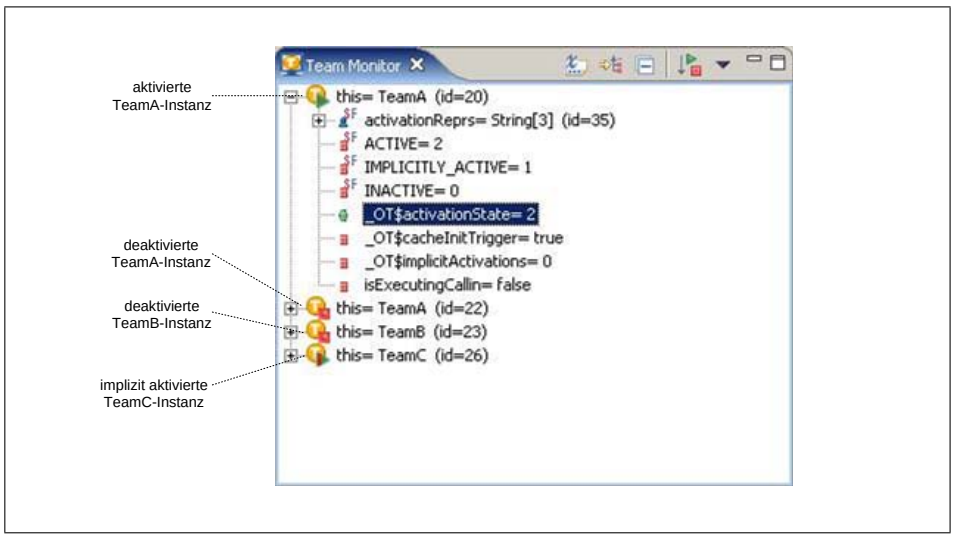


Abbildung 6.5: Der Team Monitor

Abbildung 6.5 zeigt zwei Instanzen des Teams A und jeweils eine Instanz der Teams B und C. Die obere Instanz des Teams A ist zu diesem Zeitpunkt aktiv. Das bedeutet, dass alle Callin-Methoden-Bindungen, die in Rollen dieses Teams deklariert sind, an die Basisapplikation gebunden wurden und aktiv sind. Die zweite TeamA-Instanz und die TeamB-Instanz sind nicht aktiviert. Zur Laufzeit existiert darüber hinaus eine TeamC-Instanz, die implizit, also nicht explizit über die `activate-Methode` aktiviert wurde.

Die Implementierung des Team-Monitors greift auf die Debug Model Platform zurück und die Team-Monitor-Sicht steht damit nur zur Verfügung, wenn das Programm im debug-Modus ausgeführt wird. Um zu deutlichen, dass der Team Monitor nur während des Debuggings zur Verfügung steht, wird er der Debug-Perspektive hinzugefügt.

7 Zusammenfassung und Ausblick

In diesem Kapitel werden die Inhalte, die Vorgehensweise und die Ergebnisse nochmals zusammengefasst. Außerdem erfolgt ein Ausblick auf weiterführende Arbeiten und ein Abschluss-Fazit.

7.1 Zusammenfassung

In der Softwareentwicklung werden Debugger eingesetzt, um werkzeuggestützt Fehler in Programmen zu finden. Interaktive Debugger bieten kontrollierte Programmausführung und verschiedene Visualisierungen, um das Fehlerfinden zu ermöglichen. Debugger werden für bestimmte Sprachen entwickelt. Das von der Sprache unterstützte Programmierparadigma und spezielle Sprachfeatures beeinflussen die Entwicklung eines Debuggers der Sprache.

Ziel dieser Arbeit war es die Auswirkungen des aspektorientierten Programmierparadgmas auf die Entwicklung eines interaktiven Debuggers für AOP-Programme zu erforschen. Nachdem die Einflüsse und dadurch entstehende Probleme identifiziert und konzeptionelle Lösungen dafür gefunden wurden, wurde ein Debugger für die aspektorientierte Sprache Object Teams/Java implementiert und die konzeptionellen Lösungen praktisch umgesetzt.

Einleitend wurde in dieser Arbeit der Begriff Debugging und die gängigsten Debugging-Techniken erklärt. Dabei wurde besonders auf interaktive Debugger eingegangen und es wurden Beispiele für interaktive Debugger verschiedener Programmierpradigmen gegenübergestellt. Als Resultat konnten zum Einen Grundanforderungen identifiziert werden, die von jedem interaktiven Debugger erfüllt werden sollten. Zum Anderen wurde ersichtlich, dass der Debugger für Programme des aspektorientierten Paradigmas, die Grundanforderungen unzureichend erfüllt. Um die Gründe dafür zu erforschen, wurden technische Grundlagen der Aspektorientierung, besonders Webetechniken und das Java Debugging betrachtet. Im Vordergrund der Betrachtungen standen Java-basierte aspektorientierte Sprachen, deren erstellter Maschinencode Standard Java Bytecode ist. Als Ergebnis wurden die Gründe und damit auch die Probleme ersichtlich, die sich bei der Entwicklung von Debuggern für aspektorientierte Programme ergeben. Um die genauen Einflüsse auf die Entwicklung von Debuggern zu zeigen, wurden zwei AOP-Sprachen, Object Teams/Java und AspectJ, in Abhängigkeit der verwendeten Compiler- und Webetechnik dargestellt, die auftretenden Einflüsse und Probleme aufgezeigt und konzeptionelle Lösungen erarbeitet. Die für die Komposition von Basis- und Aspektcode verwendete Webetechnik, hat einen großen Einfluss auf die Erfüllung der Debuggergrundanforderungen. Deshalb wurde eine Übersicht über die verschiedenen Webetechniken und deren Einfluss auf Debuggerentwicklungen gegeben. Abschließend wurde die Realisierung des Debuggers für OT/J-Programme und die dafür notwendigen Erweiterungen am Object Teams-Compiler, an der Object Teams Laufzeitumgebung und am JDT-Debugger dokumentiert.

7.2 Ausblick

7.2.1 Offene Arbeiten

Im Laufe der Diplomarbeit wurden Probleme angeschnitten, deren Lösungen aber noch nicht in der jetzigen Realisierung des AOP-Debuggers aufgenommen wurden. Dieser Abschnitt listet die Probleme nochmals kurz auf und zeigt Lösungen, die in überschaubarer Zeit der Realisierung des OT/J-Debuggers hinzugefügt werden können.

Variablenzuordnung

Während der Transformation des Codes durch den OT/J-Compiler und der OT/J-Laufzeitumgebung kann es zu Variablenumbennung und zum Zerlegen einer Variablen in Mehrere kommen. Außerdem entstehen Variablen auf Bytecodeebene, die auf Quellcodeebene nicht vorkommen. Warum das zu Problemen führt und wie diese gelöst werden können, wird im Folgenden erklärt:

Variablenumbennung - Die Belegungen der Variablen geben Auskunft über den Programmzustand. Werden Variablen umbenannt, heißen sie also auf Bytecodeebene anders als auf Quellcodeebene, ist es schwierig den Programmzustand zu analysieren. Um den ursprünglichen Namen der Variablen in den verschiedenen Sichten auf ein Programm anzeigen zu können, sind zusätzliche Debuginformationen notwendig. Diese Debuginformationen müssten ähnlich einer Source Map aufgebaut sein. Die verschiedenen Namen eines Feldes oder einer lokalen Variablen und deren Zuordnung im Bytecode, müssten vom Übersetzungswerkzeug im Bytecode als Attribut gespeichert werden. Dieses und das Localvariablestable-Attribut können von der Debugger-Applikation genutzt werden, um bei mehreren Quellcodeebenen (Strata), den richtigen Namen der Variablen anzuzeigen. Diese Prinzip ähnelt dem Prinzip der Source Map. In der JSR-045 Spezifikation wurde dieses Problem angesprochen. Die Lösung dafür soll in einem separaten *Java Specification Request* behandelt werden. Eine Zwischenlösung wäre, den ursprünglichen Namen einer Variablen im neuen Namen zu codieren. Die einzelnen Sichten oder im Fall von Eclipse, die Debug Model Presentation, müssen den Variablennamen analysieren und verändern, bevor er angezeigt wird.

Variablenzerlegung - Bei der Codetransformation bestimmter Sprachfeatures kommt es zur Zerlegung eines Methodenparameters in zwei oder mehrere Variablen. Am Beispiel des OT/J-Sprachfeatures *Deklaratives Lifting*, wird das Problem erläutert:

Listing 7.1: Deklaratives Lifting

```
1 public void teamMethod(SimpleClass as MyRole myrole){ ... }
```

Listing 7.2: Deklaratives Lifting (dekompiliert)

```
1 public void teamMethod(SimpleClass simpleClass)
2 {
3   MyRole myrole = _OT$liftToMyRole(simpleClass);
4   ...
5 }
```

Aus den im Listing 7.1 dargestellten OT/J Quellcode erstellt der Compiler den im Listing 7.2 dargestellten Code. Der Parameter der Methode im Listing 7.1 ändert sich durch die Transformation. Es entstehen zwei Variablen. Aus dem Parameter `myrole` (Listing 7.1) wird der Parameter `simpleClass` (Listing 7.2) und zusätzlich wird die lokale Variable `myrole` erzeugt. Im weiteren Methodenrumpf wird dann nicht auf den Parameter der Methode, sondern auf die lokale Variable zugegriffen. In der Variablensicht erscheinen beide Variablen, was zur Verwirrung des Benutzers führen kann.

Im Allgemeinen kann es bei Variablenzerlegung zu verschiedenen Problemen kommen. Ein Parameter einer Methode heißt auf Bytecodeebene zum Beispiel anders, als auf Quellcodeebene und wird darüber hinaus einer neuen lokalen Variablen zugewiesen und dabei verändert. Das Problem dabei ist, dass der Benutzer auf Wertänderungen des ursprünglichen Parameters wartet, sich die Werte aber bei der neuen, veränderten Variable ändern. Die Lösung für das Problem ist zum Einen das Verstecken des ursprünglichen Parameters und zum Anderen das Umbenennen der neuen Variablen in den Namen der ursprünglichen Variable. Für den Benutzer entsteht so der Eindruck, er würde die Werte des ursprünglichen Parameters betrachten. In einigen Fällen ist nicht nur die ursprüngliche Variable, sondern auch einige oder alle durch die Zerlegung entstandenen Variablen interessant. Welche Variablen davon angezeigt oder versteckt werden, muss abhängig vom Sprachfeature entschieden werden.

Neue Variablen - In einigen Fällen werden bei der Codetransformation neue Variablen generiert. Diese Infrastruktur-Variablen können auch für den Benutzer interessant sein. So existiert zum Beispiel in Rollen das Feld `_OT$base`, das auf Quellcodeebene nicht vor kommt, für den Benutzer aber interessant ist, da es die adaptierte Basisklasse referenziert. In diesem Fall würde das Feld in der Variablensicht angezeigt werden. Felder oder lokale Variablen, die zum uninteressanten Infrastrukturcode gehören, werden versteckt und erscheinen somit nicht in den Sichten.

Bei Object Teams enthalten generierte Felder und Variablen Präfixe wie `_OT$`, um sie von den nicht zusätzlich generierten Variablen unterscheiden zu können. Abhängig vom Sprachfeature muss entschieden werden, ob Variablen umbenannt und angezeigt oder versteckt werden. Das Verstecken von Variablen kann erreicht werden, indem die zu versteckenden Variablen aus dem Localvariablestable-Attribut gelöscht oder erst gar nicht eingetragen werden.

Die Variablenzuordnung ist genauso wichtig wie die Quellcodedarstellung. Der OT/J-Compiler und der JDT-Debugger mussten erweitert werden, um während des Debuggings den entsprechenden Quellcode anzuzeigen. Die Werte der Variablen spiegeln den Programmzustand wieder. Damit ein Benutzer den Programmzustand während des Debuggings in gewohnter Weise abfragen kann, sind oben genannte Änderungen vorzunehmen.

Quellcodedarstellungsstrategien

Während des Debuggings von OT/J-Programmen wird Bytecode entsprechenden Quellcode zugeordnet. Durch die Erweiterungen am Compiler und Debugger, besonders aber durch die Erstellung der Source Map, wurde invasiv in die Zuordnungsbeschreibungen zwischen Byte- und Quellcode eingegriffen. Dabei wurde die Strategie verfolgt, dass bei vermischten Code, der durch das Weben entsteht, Basisbytecode dem Basisquellcode und der hinzugefügte Aspektbytecode dem Aspektquellcode zugeordnet wird. Während des Debuggings der Basisapplikation wird in Aspektquellcode gesprungen, wenn die Basisapplikation adaptiert wurde.

Bei AspectJ wird eine andere Quellcodedarstellungsstrategie beim Debugging von Basisapplikationen gewählt, die ein Filtern der Aspekte aus der Basisapplikation-Sicht vorsieht. Treten Advices während des schrittweisen Debuggings von Basisapplikationen in AspectJ auf, so werden sie einfach ausgeführt, ohne das der entsprechende Aspektequellcode dafür angezeigt wird. Nur wenn ein Haltepunkt im Aspektquellcode gesetzt wird, wird Aspektquellcode angezeigt. Die Intention hinter dieser Quellcodedarstellungsstrategie ist wohl die unabhängige Entwicklung von Basis- und Aspektprogramm. Der Entwickler der Basisapplikation soll nichts von der Entwicklung der Aspekte wissen, was illusorisch erscheint, da Aspekte den Programmfluss der Basisapplikation verändern.

Die bei AspectJ verfolgte Quellcodedarstellungsstrategie kann der Jetzigen hinzugefügt werden. Vor und sogar während des Debuggings könnte entschieden werden, mit welcher der beiden Quellcodedarstellungsstrategien debuggt werden soll. Technisch würde es durch das Hinzufügen eines weiteren Stratums gelöst werden, das durch die OT-Laufzeitumgebung erstellt werden müsste. Die OT-Laufzeitumgebung würde somit eine Source Map mit zwei Strata für den vermischten Code erstellen. Das erste Stratum entspräche dem Stratum, wie es jetzt schon erstellt wird. Das zweite Stratum würde sämtlichen Aspektbytecode der StepOverLinenumber zuordnen. Somit würden Aspektbytecodecodeinstruktionen ausgeführt, deren Quellcodezuordnungen dabei aber nicht angezeigt werden.

Source Map für alle Sprachfeatures

Bei der Realisierung des OT/J-Debuggers wurden die wichtigsten Sprachfeatures auf Bytecodeebene analysiert und festgelegt, welcher Quellcode dafür angezeigt werden soll. Die dabei betroffenen Linenumbertables wurden angepasst und entsprechende Einträge in der Source Map erstellt.

Die Source Map-Erstellung wurden aber noch nicht für alle Sprachfeatures von Object Teams/ Java umgesetzt. So wurden Guards, Parametermappings, deklaratives Lifting und Rollenkonstruktoren noch nicht vollständig analysiert. Auch Kombinationen von Features wie zum Beispiel ein nested Team, das in einer Rollendatei deklariert wurde und implizit erbt, sind auch noch zu untersuchen.

7.2.2 Zukünftige Arbeiten

Dieser Abschnitt beschreibt zum Einen Einschränkungen, die während des Debuggings zu beachten und die nicht in absehbarer Zeit aufzuheben sind. Zum Anderen werden denkbare Erweiterungen der Object Teams-Debugunterstützung diskutiert.

7.2.2.1 Einschränkungen

Programmmanipulation

Eine der von interaktiven Debuggern zu erfüllenden Grundanforderungen ist die Programmmanipulation. Programme sollen während des Debuggings durch Wertänderung der Variablen und durch das Einfügen von zusätzlichen Code manipulierbar sein. Der OT/J Debugger unterstützt nur das Ändern von Variablenwerten. Der JDT-Debugger, auf dem der OT/J Debugger basiert, unterstützt das *Hot Code Replacement* für Standard Java Applikationen. Das Ersetzen von Bytecode, während das Programm ausgeführt wird, ist somit möglich. Der OT/J-Compiler

und die OT-Laufzeitumgebung erstellen zwar Standard Java Bytecode, das Ersetzen des alten Bytecodes durch neuen, führt aber zu Problemen. Das Hot Code Replacement erlaubt kein Ändern der Klassenstruktur. Es dürfen also keine neuen Felder oder Methoden hinzugefügt werden. Nur schon vorhandene, dürfen verändert werden. Der Fehler, der beim Ersetzen des alten Bytecodes auftritt, besagt, dass versucht wird, die Klassenstruktur zu ändern. Eine genaue Analyse wurde nicht vorgenommen, aber schaut man sich den OT/J Quellcode und den resultierenden Bytecode an, so ist ersichtlich, dass die Klassenstrukturen nicht übereinstimmen. Ob das die wirkliche Ursache des Fehlers ist, kann nicht gesagt werden.

Conditions und Expressions

Einigen Haltepunkttypen können Codebedingungen (Conditions) übergeben werden (siehe Kapitel 2.1.3.1). Diese Codebedingungen werden zur Laufzeit kompiliert und überprüft und entscheiden, ob ein Haltepunkt aktiv ist. Außerdem ist es während des Debuggings möglich, Code auswerten zu lassen. Dieser Code wird *Expression* genannt. In Eclipse werden Conditions und Expressions mit einer `ASTEvaluationEngine` evaluiert:

Der `ASTEvaluationEngine` wird eine Condition und ein Referenztyp[1] übergeben, um die Condition zu überprüfen. Die Condition wird überprüft, indem eine `CompilationUnit`[2] erstellt wird, die zusätzlich eine Methode mit der Condition enthält. Um aus dem Referenztyp eine `CompilationUnit` zu erstellen, muss dem `ASTParser`[3] Quellcode übergeben werden. Der `ASTParser` erstellt einen DOM-AST[4], des Wurzel die `CompilationUnit` ist. Der `ASTParser` benötigt dazu den Quellcode des Referenztyps. Für die Erstellung des Quellcodes existieren zwei Wege:

1. **SourceBasedSourceGenerator** - Mit Hilfe der Debuginformation Sourcefile-Attribut oder SourceDebugExtension-Attribut (FileSection der Source Map) wird der Name der Quellcodedatei des Referenztyps ermittelt und im entsprechenden Java-Projekt gefunden. Jetzt wird der `ASTParser` genutzt, um aus der Quellcodedatei ein DOM-AST zu erstellen. Der `SourceBasedSourceGenerator` ist ein AST-Visitor und traversiert alle AST-Knoten. Dabei erstellt er Quellcode und fügt die Condition in den Quellcode ein.

2. **BinaryBasedSourceGenerator** - Der `BinaryBasedSourceGenerator` erstellt Quellcode auf Basis der Informationen in der Virtual Machine. Alle Informationen, die der Referenztyp bietet, werden sukzessiv abgefragt und daraus Quellcode gebildet. Auch hier wird die Condition dem Quellcode hinzugefügt.

Nachdem der Quellcode erstellt wurde, erzeugt der `ASTParser` die `CompilationUnit`. Diese wird auf Fehler überprüft. Um zu prüfen, ob die Condition die Ursache des Fehler ist, wird die Sourceposition[5] des Fehlers mit der Sourceposition der Methode, die die Condition enthält verglichen. Stimmen die Sourcepositionen überein, ist die Condition fehlerhaft.

Enthält die `CompilationUnit` keine Fehler, wird die Condition in Instruktionen überführt. Die Condition ist nun eine `ICompiledExpression` und kann beliebig oft innerhalb eines Laufzeit-Kontextes evaluiert werden. Das passiert mit Hilfe eines Interpreter, der die Instruktionen

[1] Der Referenztyp repräsentiert den Typ eines Objektes in einer Virtual Machine.
[2] Eine `Compilation Unit` repr¨asentiert eine .java Quelltextdatei.
[3] Der ASTParser erzeugt aus Quelltext AST-Knoten.
[4] Der DOM-AST (Document Object Model/Abstract Syntax Tree) ist ein Modell des Quelltextes als strukturiertes Dokument. Es repräsentiert ein Programm von `Compilation Units` bis hinunter zu Anweisungen und Ausdrücken. [42]
[5] Sourcepositionen sind Positionen einzelner Quellcodeelemente im Kontext des ganzen Quellcodes.

im Laufzeit-Kontext ausführt. Nach deren Ausführung wird der Interpreter gestoppt und das Ergebnis an registrierte Listener gesendet.

Conditions und Expressions können während des Debuggings von OT/J-Programmen nicht angegeben werden. Das Problem sind die beiden Quellcode-Generatoren.

Beim `SourceBasedSourceGenerator` gibt es gleich mehrere Probleme. Der Quellcodedateiname wird mit Hilfe der Debuginformationen ermittelt. Die in der Source Map angegebene FileSection kann mehrere Quellcodedateinamen enthalten. In Eclipse wird standardmäßig der erste Eintrag übernommen. So kann es sein, dass für den Referenztyp `ClassA` der Quellcodedateiname (ClassB.java) ermittelt wird. Des Weiteren arbeitet der `SourceBasedSourceGenerator` auf Basis des DOM-AST und erstellt aus den einzelnen AST-Knoten Quellcode. Der DOM-AST wurde wie der AST des Compilers (siehe Kapitel 6.2.1) um zusätzliche AST-Knoten erweitert, um die neuen Object Teams Spracheelemente abzudecken. Der `SourceBasedSourceGenerator` kennt nur AST-Knoten der Sprache Java und kann so keinen OT/J-Quellcode generieren.

Die Lösung des ersten Problems wäre entweder dafür zu sorgen, dass der richtige Quellcodedateiname immer an der ersten Stelle der FileSection steht, oder über alle Einträge in der FileSection zu iterieren und durch Namensvergleich, die zum Referenztyp gehörige Quellcodedatei zu ermitteln. Das zweite Problem lässt sich nur Lösen, indem der `SourceBasedSourceGenerator` erweitert wird und auch die OT/J spezifischen AST-Knoten besucht und somit OT/J-Quellcode erstellt.

Schlägt die Quellcodeerzeugung mittels `SourceBasedSourceGenerator` fehl, wird der `BinaryBasedSourceGenerator` eingesetzt. Dieser erstellt aus dem Referenztyp Quellcode. Der Referenztyp ist ein Standard Java Typ und entspricht dem Java Bytecode, der vom OT-Compiler und vom OT-Weber erstellt wurde. Wird also Quellcode ermittelt, indem die Informationen des Referenztyps genutzt werden, entsteht Standard Java Quellcode. Der Standard Java Quellcode enthält Variablen und Methodennamen, die reserviert[6] sind. Und so enthält die vom `ASTParser` erstellte `CompilationUnit` (Wurzel des DOM-AST) Fehler. Diese Fehler führen dazu, dass die Condition als fehlerhaft erkannt wird. In Wahrheit ist aber nicht die Condition fehlerhaft, sondern andere Teile des erstellten Quellcodes. Eine einfache Lösung für dieses Problem, ist nicht erkennbar.

Um Expressions zu evaluieren, wird ebenfalls die `ASTEvaluationEngine` genutzt. Dabei kommt es zu den gleichen Problemen wird bei den Conditions.

7.2.2.2 Erweiterungen

Debugging für OT Compiler/Laufzeit Entwickler

Der OT/J-Debugger unterstützt Quellcodestrategien, um Object Teams Quellcode anzuzeigen. Entwickler von Object Teams Programmen können so ihren verfassten Code debuggen. Wenn die Zielgruppe aber nicht die Entwickler von Object Teams Programmen, sondern die Entwickler des OT-Compilers und der OT-Laufzeitumgebung ist, muss eine völlig andere Strategie verfolgt werden.

[6] Bestimmte Variablen- und Methodennamen dürfen im Quellcode nicht verwendet werden, da sie generiert werden.

Diese Strategie sieht vor, aus dem von OT-Compiler und von der OT Laufzeitumgebung erstellten Bytecode Javaquellcode zu erstellen und diesen Javaquellcode an Stelle des OT/J-Quellcodes während des Debuggings anzuzeigen. Da stellt technisch eine Herausprderung dar, da der von OT/J verwendete Webemechanismus Bytecode in bei Laden der Klassen transformiert. Der Bytecode dürfte erst nach der Transformation dekompiliert werden. Normale Decompiler wären also ungeeignet, da sie auf Datei-Basis arbeiten. Der veränderte Bytecode befindet sich bei OT/J in virtuellen Speicher und so müssten Frameworks eingesetzt werden, die aus dem Bytecode Javaquellcode direkt erstellen können und diesen Javaquellcode dann persistent speichern. Auch eine Kombination aus Framework und Decompiler ist vorstellbar. Das Framework speichert den eben transformierten Bytecode ab und ein Decompiler greift darauf zu und erstellt Javaquellcode. Die dritte Möglichkeit zum Erstellen des Javaquellcodes ist die vom `BinaryBasedSourceGenerator` verwendete Vorgehensweise.

Um während des Debuggings Javaquellcode angezeigen zu lassen, sind Veränderungen am Debugger notwendig. So müssten zum Beispiel Änderungen bei der Verwendung des Source Lookup Frameworks und der Debug Model Presentation vorgenommen werden.

7.3 Fazit

Das aspektorientierte Paradigma hat, wie andere Paradigmen davor, großen Einfluss auf die Entwicklung von Werkzeugen. Die Entwicklung von Debuggern bildet da keine Ausnahme. Technische und konzeptionelle Besonderheiten aspektorientierter Sprachen müssen bei der Debuggerentwicklung besonders betrachtet werden. Hauptsächlich standen in der Arbeit Java-basierte AOP-Sprachen im Mittelpunkt. Sie wurden auf technischer Ebene bezüglich der Einflüsse auf die Debuggerentwicklung untersucht. Dabei hat sich herausgestellt, dass die Einflüsse sehr gross sind, wenn Übersetzungs- und Webetechnik Codetransformation einsetzen, um Aspekte in Bytecode zu übersetzen und Aspekte mit Basisapplikationen zu kombinieren. Damit die Grundanforderungen an einen AOP-Debugger erfüllt werden können, sind nicht nur Erweiterungen und Änderungen am Debugger selbst, sondern auch am Übersetzungswerkzeug und am Aspektweber notwendig. Unabhängig von techischen Umsetzungen, haben auch spezielle Sprachfeatures Einfluss auf die Debuggerentwicklung. Die Bereitstellung von zusätzlichen Informationen, die beim Feature dynamische Aspektaktivierung hilfreich sind, sind ein Beispiel dafür.

Neben der Entwicklung eines Debuggers für eine AOP-Sprache, wurden in dieser Arbeit auch Lösungen erarbeitet, die sich bei der Debuggerentwicklung für andere AOP-Sprache verwenden lassen.

Literaturverzeichnis

[1] Aspectj faq homepage, March 2006. Hompage: http://www.eclipse.org/aspectj/doc/released/faq.html.

[2] Data display debugger, March 2006. DDD-Hompage: http://www.gnu.org/software/ddd/.

[3] Definition: Just-in-time compiler, June 2006. Webseite: http://www.software-kompetenz.de/?21851.

[4] Eclipse, June 2006. Webseite: http://www.eclipse.org.

[5] Java community process (jcp), May 2006. Webseite: http://jcp.org.

[6] Java platform debugger architecture (jpda), May 2006. Webseite: http://java.sun.com/products/jpda/.

[7] Java specification request 45 (jsr-045): Debugging support for other languages, May 2006. Webseite: http://www.jcp.org/en/jsr/detail?id=45.

[8] Object teams homepage, June 2006. Hompage: http://www.objectteams.org.

[9] Ollydbg homepage, March 2006. Hompage: http://www.ollydbg.de/.

[10] Prose website, June 2006. Webseite: http://prose.ethz.ch/Wiki.jsp.

[11] Spring aop framework website, June 2006. Webseite: http://www.springframework.org.

[12] Steamloom website, June 2006. Webseite: http://www.st.informatik.tu-darmstadt.de/steamloom.

[13] Wolfgang Schröder-Preikschat Andreas Gal and Olaf Spinczyk. Aspectc++: Language proposal and prototype implementation, 2001.

[14] Christof Binder. Aspectual collaborations: Erweiterung des java-compilers für verbesserte modularität durch aspekt-orientierte techniken. Master's thesis, Technical University Berlin, August 2002.

[15] Bitan Biswas and R. Mall. Reverse execution of programs, 1999.

[16] J. Boner and A. Vasseur. Aspectwerkz web site, June 2006. Webseite: http://aspectwerkz.codehaus.org.

[17] Matthias Braun. Einführung in die aspektorientierte programmierung, July 2003.

[18] Gregor Brcan. Erweiterung von objektorientiertem refactoring für die aspektorientierte sprache objectteams/java. Master's thesis, Technical University Berlin, October 2005.

[19] Holger Cleve and Andreas Zeller. Automatisches debugging, 2001.

[20] Dominik Wieland Cristinel Mateis, Markus Stumptner and Franz Wotawat. Jade - ai support for debugging java programs, 1996.

[21] Dipl.-Inf. Krzysztof Czarnecki. *Generative Programming*. PhD thesis, Technical University of Ilmenau, October 1998.

[22] John Michalakes David Abramson, Ian Foster and Rok Sosic. Relative debugging: A new methodology for debugging scientific applications, 1996.

[23] Edsger Wybe Dijkstra. *A discipline of programming*. Englewood Cliffs, N.J. : Prentice-Hall, 1976.

[24] Edsger Wybe Dijkstra. On the role of scientific thought, in "'selected writings on computing: A personal perspective"', 1982.

[25] Tom Dinkelaker. Advanced method bytecode management for steamloom. Master's thesis, Technical University Darmstadt, March 2005.

[26] Remi Douence and Mario Südholt. A model and a tool for event-based aspect-oriented programming (eaop), June 2002.

[27] M. Al-A'ali E. H. Khan and M. R. Girgis. Object-oriented programming for structured procedural programmers, October 1995.

[28] Michael Austermann Günter Kniesel, Pascal Costanza. Jmangler - a framework for load-time transformation of java class files. In IEEE Computer Society, editor, *SCAM 2001, Proceedings of IEEE Workshop on Source Code Analysis and Manipulation*, pages 100–110. IEEE Computer Society Press, nov 2001.

[29] Michael Austermann Günter Kniesel, Pascal Costanza. Jmangler - a powerful back-end for aspect-oriented programming. In T. Elrad R. Filman and M. Aksit S. Clarke, editors, *Aspect-oriented Software Development*. Prentice Hall, 2004. To appear.

[30] Prof. Dr. G. Goos. Softwarekomposition und metaprogrammierung, 2004.

[31] Wayne C. Gramlich. Debugging methodology: session summary, 1983.

[32] Alan Grosskurth and Rolando Blanco. Aspect-oriented programming, March 2005.

[33] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification, 2002.

[34] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations., 2002. Download: http://www.objectteams.org/publications/NODe02.pdf.

[35] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[36] Joachim Hänsel. Entwicklung von verfahren zur statischen analyse dynamischer programmeigenschaften für das refactoring aspektorientierter programme. Master's thesis, Technical University Berlin, June 2006.

[37] Christine Hundt. Bytecode-transformation zur laufzeitunterstützung von aspektorientierter modularisierung mit object-teams/java, Februar 2003.

[38] R. Lillsjo J. Lind J. Tirsen, J. Larsson and L. AB. Nanning aspects - a simple yet scaleable aspect-oriented framework for java, June 2006. Webseite: http://nanning.codehaus.org/.

[39] John D. Johnson and Gary W. Kenney. Implementation issues for a source level symbolic debugger. *SIGPLAN Not.*, 18(8):149–151, 1983.

[40] Mark Scott Johnson. A software debugging glossary, 1982.

[41] Joseph R. Kiniry. Idebug: An advanced debugging framework for java, September 1998.

[42] Michael Krüger. Definition und implementierung von metriken für die aspektorientierte programmiersprache objectteams/java. Master's thesis, Technical University Berlin, March 2006.

[43] Ramnivas Laddad. I want my aop!, March 2006. Artikel-Hompage: http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html.

[44] Bil Lewis. Debugging backwards in time, 2003.

[45] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification - Second Edition.* Addison-Wesley Professional, 1999.

[46] Paolo Tonella Mariano Ceccato and Filippo Ricca. Is aop code easier or harder to test than oop code?, 2005.

[47] Christian Meffert. Realisierung eines caesarj-debuggers. Master's thesis, Technical University Darmstadt, March 2006.

[48] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[49] Jonathan B. Rosenberg. How debuggers work: algorithms, data structure, and architecture, 1996.

[50] Wolfgang Schult. Aspektorientierte programmierung, Februar 2003.

[51] Tammo van Lessen. Generatives programmieren, January 2004.

[52] D. Vandevoorde. Hewlett-packard, california language lab, cupertino, california. personal communication, 1998.

[53] A. Vasseur. Dynamic aop and runtime weaving for java - how does aspectwerkz address it? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, 2004.

[54] Markus Witte. Portierung, erweiterung und integration des objectteams/java compilers für die entwicklungsumgebung eclipse. Master's thesis, Technical University Berlin, Dezember 2003.

[55] Dr.-Ing. Andreas Zeller and Dipl.-Inform. Jens Krinke. Programmierwerkzeuge, June 2000.

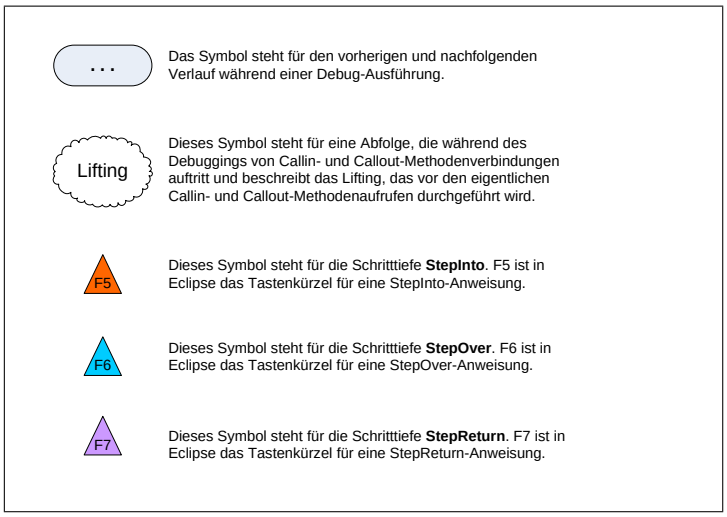
Abbildungsverzeichnis

Schlagwortverzeichnis

A Schrittweises Debugging mit dem OT/J Debugger

Die Benutzung des OT/J Debuggers bedarf zusätzlichen Erklärungen. Während des schrittweisen Debuggings entscheiden die Schritttiefen über die weitere Ausführung. Damit nicht die Vorgehensweise *Learning by Doing* verfolgt werden muss, sind im Folgenden Abbildungen dargestellt, die das mögliche Debug-/Sprungverhalten bei Callin- und Callout-Methodenbindungen illustrieren.

Erklärungen der in den Abbildungen vorkommenden Symbole:



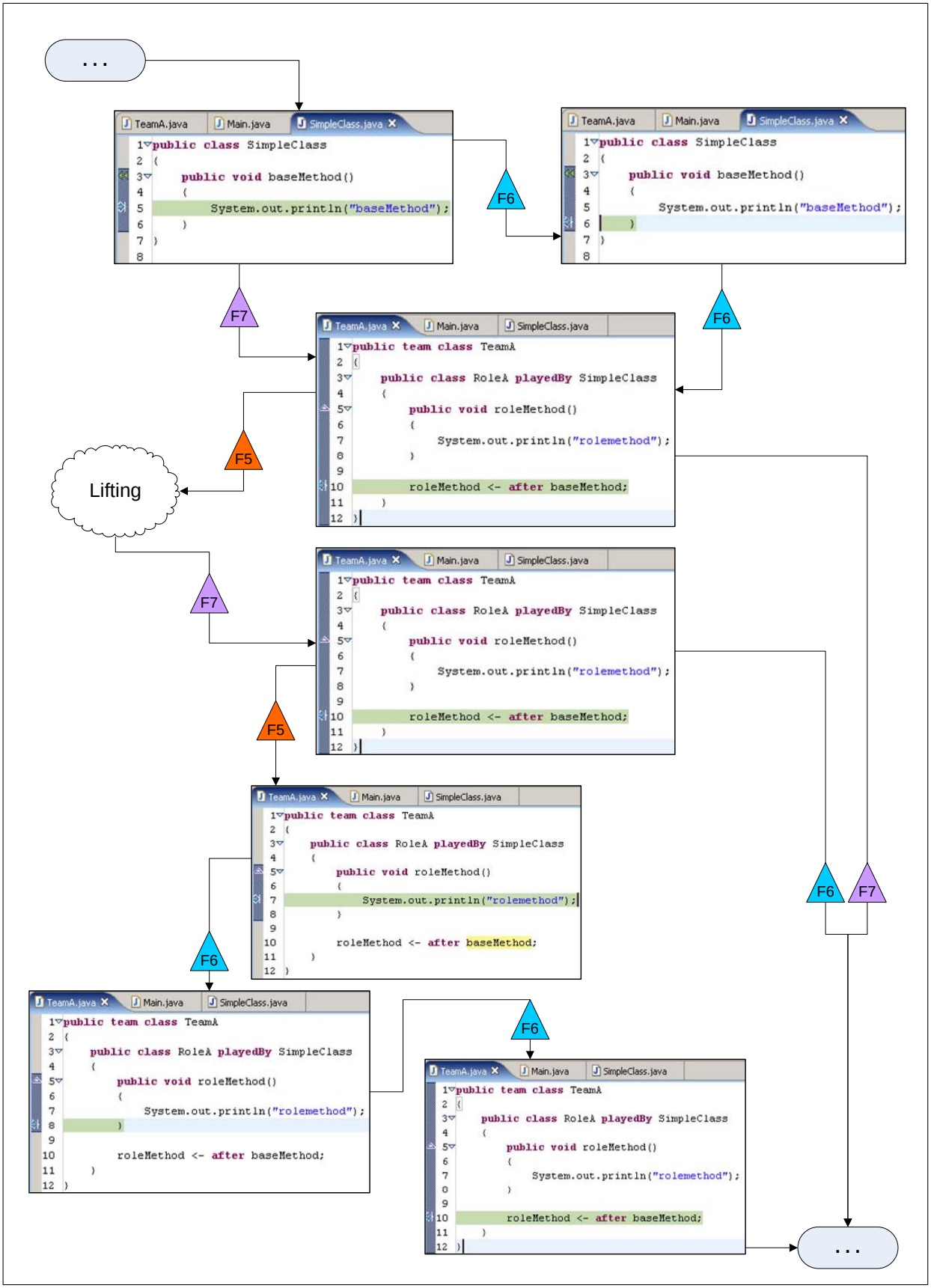


Abbildung A.1: Beispiel für Debug-/Sprungverhalten bei Callout-Methodenbindungen

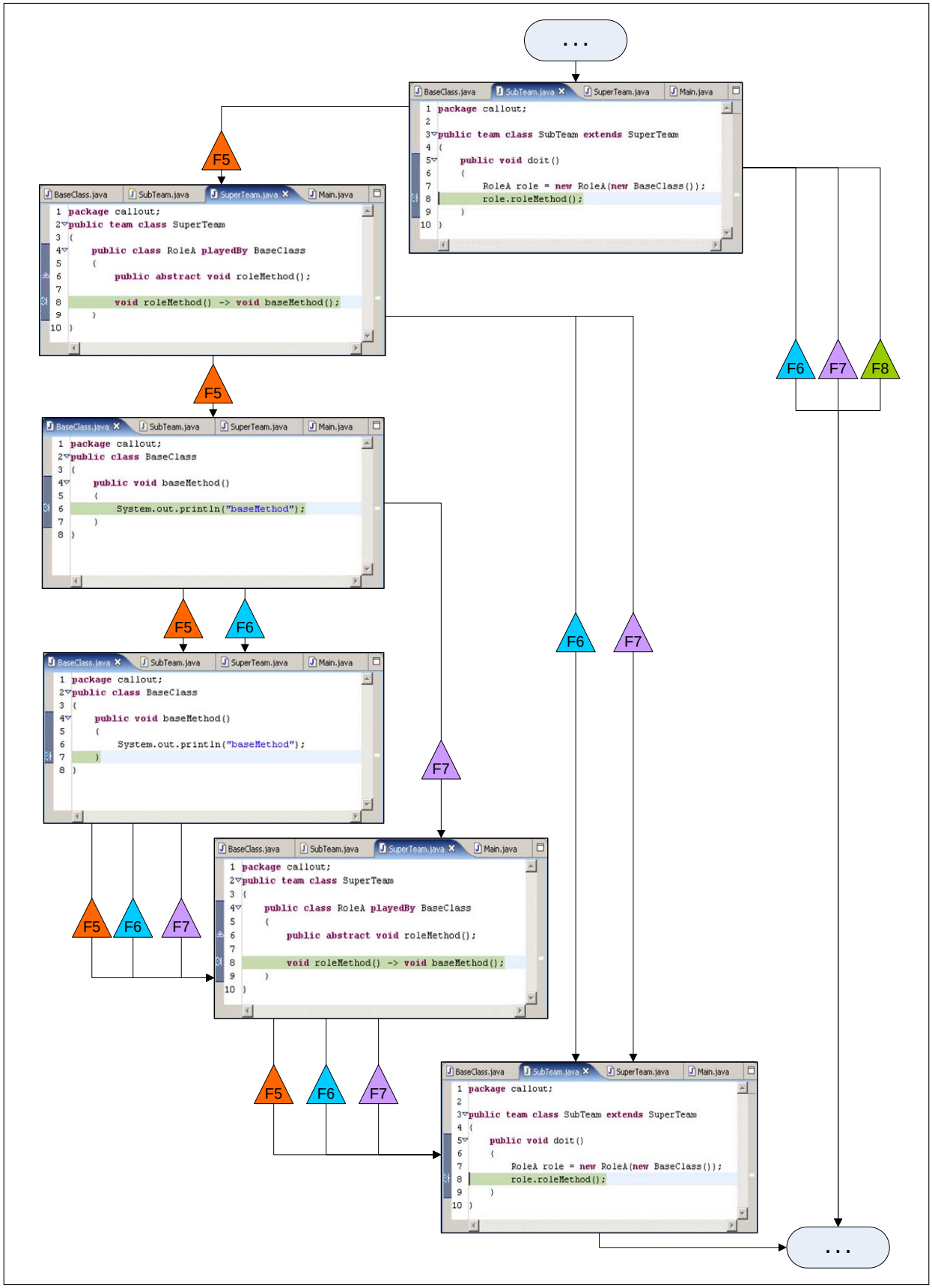


Abbildung A.2: Beispiel für Debug-/Sprungverhalten bei Callout Override-Methodenbindungen

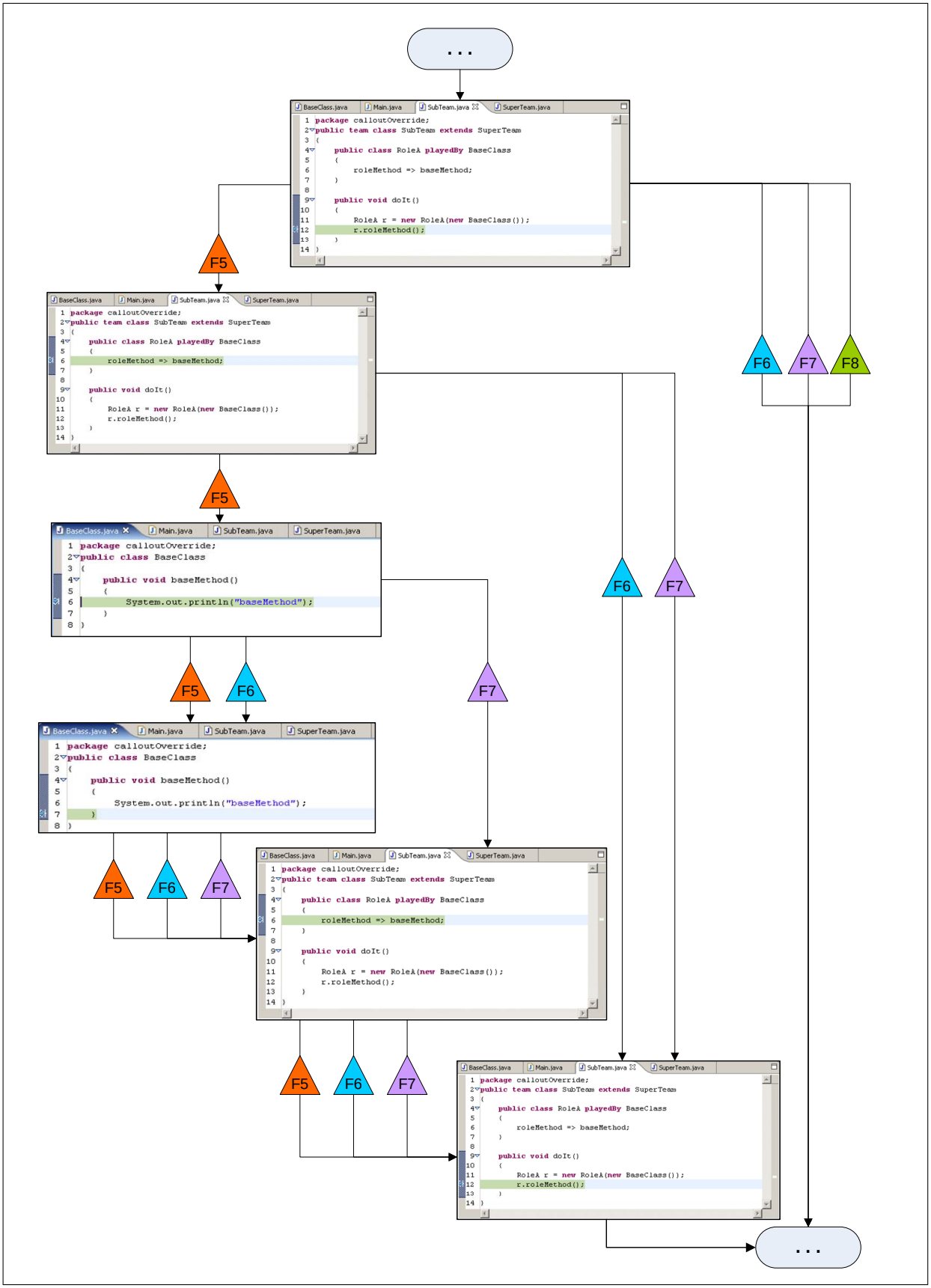


Abbildung A.3: Beispiel für Debug-/Sprungverhalten bei Callin-Methodenbindungen (before)

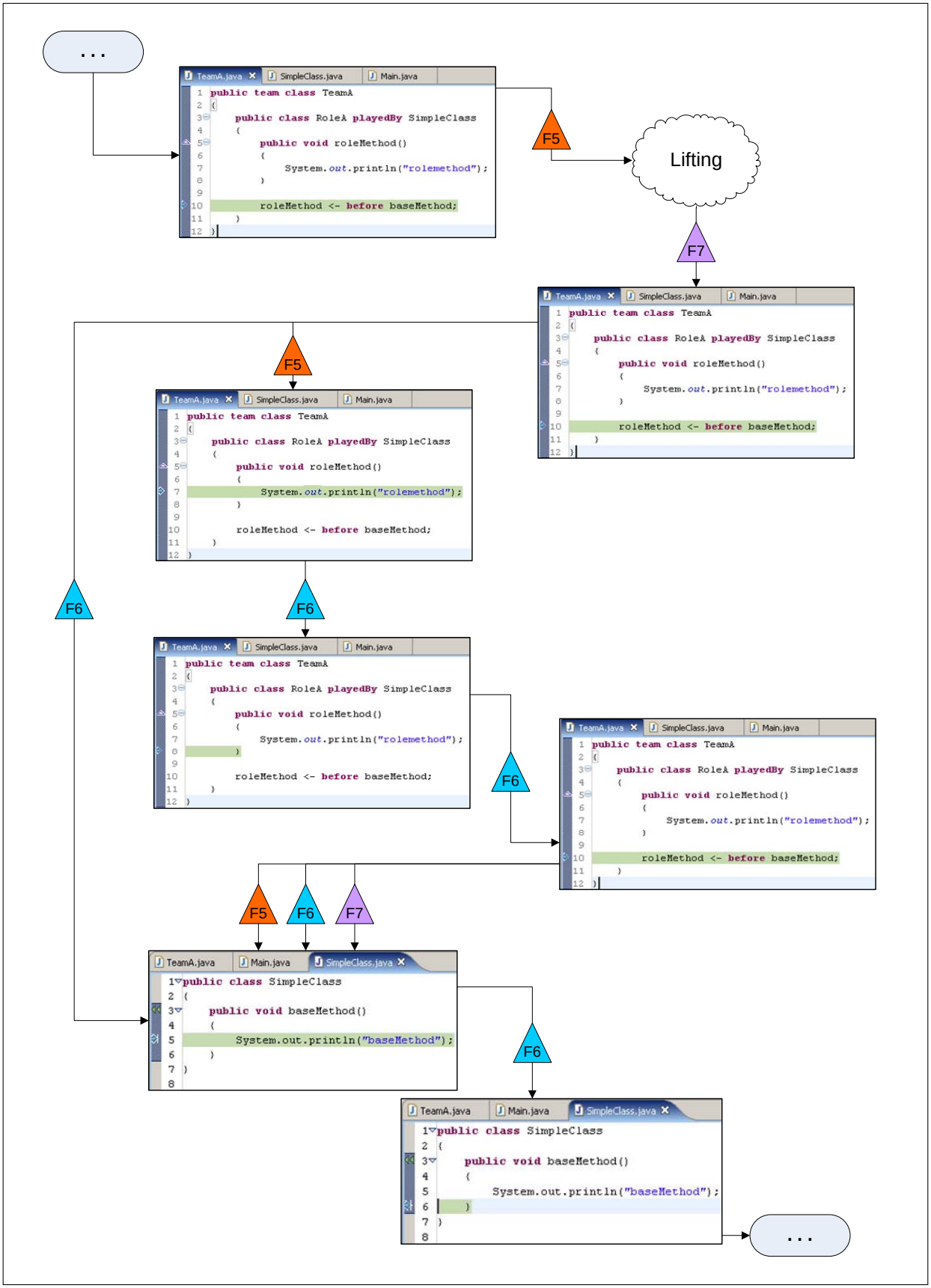


Abbildung A.4: Beispiel für Debug-/Sprungverhalten bei Callin-Methodenbindungen (after)

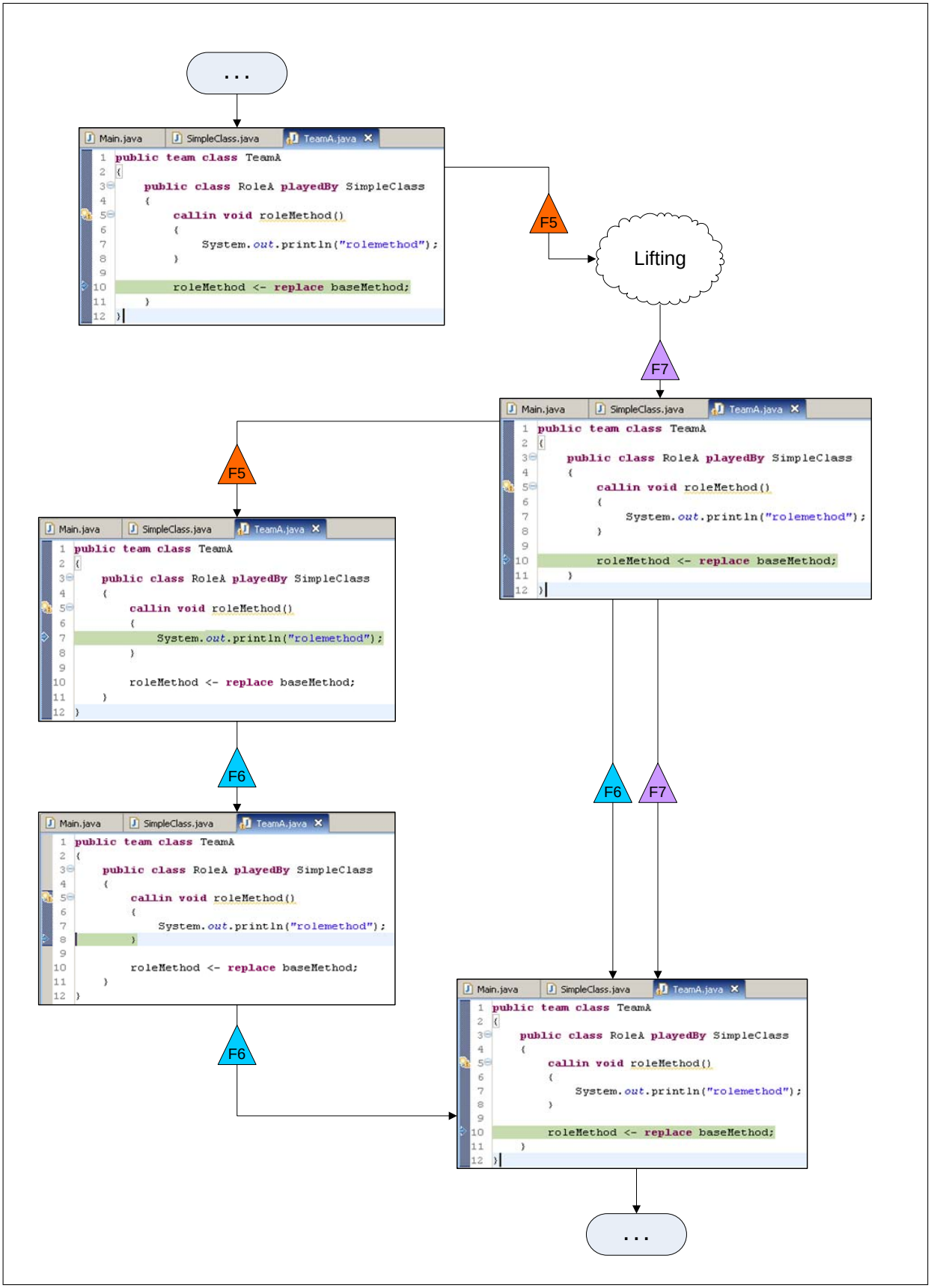


Abbildung A.5: Beispiel für Debug-/Sprungverhalten bei Callin-Methodenbindungen (replace)

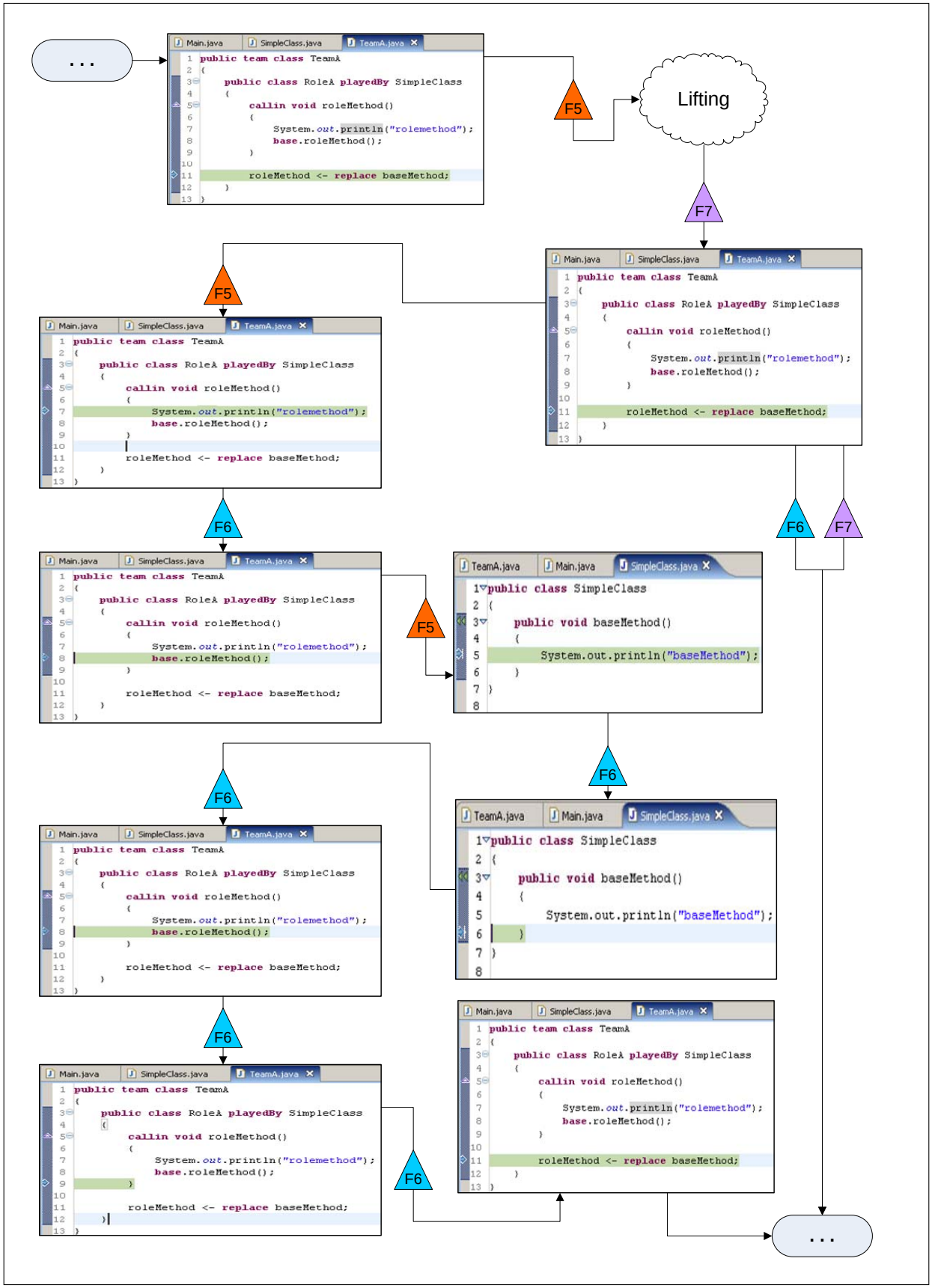


Abbildung A.6: Beispiel für Debug-/Sprungverhalten bei Callin-Methodenbindungen (replace mit basecall)