Studienarbeit

Inject/J
Ein Werkzeug zur skriptgesteuerten Quellcodetransformation

Forschungszentrum Informatik Karlsruhe (FZI)
Fachbereich Programmstrukturen

Volker Kuttruff

Betreuer: Thomas Genßler

19. Januar 2000

Erklärung

Hiermit erkläre ich, daß ich diese Arbeit eigenhändig, ohne unzulässige Hilfsmittel und unter Angabe aller Quellen angefertigt habe.

Karlsruhe, den 19. Januar 2000

Volker Kuttruff

Inhaltsverzeichnis

1 Einführung **1**
1.1 Problemstellung und Ziel . . . 1
1.2 Lösungsskizze . . . 2
1.3 Aufbau der Arbeit . . . 2

2 Theoretische Grundlagen **3**
2.1 Einordnung in AOP . . . 3
2.2 Das Metaobjektprotokoll (MOP) . . . 5
2.3 Zusammenhang AOP und MOP . . . 6

3 Die Aspekt-Beschreibungssprache **7**
3.1 Einführung . . . 7
3.2 Ziele der Beschreibungssprache . . . 7
3.3 Unterstützte Webepunkte . . . 7
3.4 Syntax . . . 8
3.5 Semantik . . . 9
3.5.1 Identifizierer . . . 9
3.5.2 Benutzung von Aliasen . . . 10
3.5.3 Aspektdeklarationen . . . 10
3.5.4 Navigationsbefehle . . . 11
3.5.5 Befehle zur Ablaufsteuerung . . . 12
3.5.6 Webebefehle . . . 14
3.5.7 Weitere Befehle . . . 19
3.6 Präprozessor-Anweisungen . . . 19
3.6.1 Makros . . . 19
3.6.2 Konstantendefinitionen . . . 21
3.6.3 Ablaufsteuerung . . . 21

4 Der Weber **23**
4.1 Einführung . . . 23
4.2 Struktur des Webers . . . 23
4.2.1 Kapselung des benutzten Metaobjektprotokolls . . . 23
4.2.2 Die Kontrolldatei . . . 24
4.2.3 Präprozessor . . . 26
4.2.4 Zerteiler . . . 26
4.2.5 Ausführungseinheit . . . 26
4.2.6 Steuereinheit . . . 27
4.3 Implementierung für Together SCI . . . 27
4.3.1 Together SCI . . . 28
4.3.2 Implementierungsdetails . . . 28
4.3.3 Explizite Webepunkte . . . 29
4.4 Ablauf eines Webevorgangs . . . 30

4.4.1 Prinzipieller Ablaufplan 30
4.4.2 Ablauf anhand eines Beispiels 32

5 Zusammenfassung und Ausblick 37
5.1 Zusammenfassung 37
5.2 Implementierung für andere MOPs 37
5.3 Erweiterungen 38

A Installation und Eigenschaftsliste 39
A.1 Installation 39
A.2 Eigenschaftsliste der Webepunkte 39

Kapitel 1

Einführung

1.1 Problemstellung und Ziel

An vielen Stellen in einem Softwaresystem trifft man folgendes Problem an: Statt der eigentlichen Funktionalität werden noch viele weitere nichtfunktionale Anforderungen definiert, die zwar zum korrekten Ausführen des Codes unabdingbar sind, aber nicht zur Lösung des Problems beitragen. Beispiele für solche nichtfunktionalen Anforderungen sind Synchronisation, Vor- und Nachbedingungsprüfung oder Fehlerbehandlung. Das Problem an diesem zusätzlichen Code liegt darin, daß er im ganzen System verteilt ist. Teilstücke des Quellcodes, welche die genannten zusätzlichen Eigenschaften beschreiben, vermischen sich einerseits untereinander und andererseits mit den funktionalen Codestücken. Der resultierende Code ist dadurch schwerer lesbar, die eigentliche Funktionalität läßt sich unter Umständen nur mit Mühe erfassen. Gelingt es nun aber, den zusätzlichen Code von der eigentlichen Funktionalität abzutrennen, so läßt sich einerseits letztere besser identifizieren und verstehen, und andererseits kann man andere Dinge wie Synchronisation, Vor- und Nachbedingungsprüfung und Fehlerbehandlung getrennt spezifizieren (und so gegebenenfalls wiederverwenden). Ein Mechanismus, um dies zu ermöglichen, stellt somit ein wünschenswertes Ziel dar. Einen Schritt zur Trennung dieser Eigenschaften stellt zum Beispiel das Ausnahmen-Konzept in einigen objektorientierten Sprachen wie C++ und Java dar, welches den eigentlichen Code und die Fehlerbehandlung voneinander trennt.

Ein damit zusammenhängendes Problem, vor allem in komplexeren Systemen, ist die Anpassung einzelner Komponenten aneinander. Die Anpassung der Komponenten besteht oft darin, die nichtfunktionalen Teile den neuen Anforderungen anzupassen. Die Funktionalität einer Komponente muß im allgemeinen nicht angepaßt werden, da sie ja sonst nicht zur Lösung des Problems geeignet wäre. Diese Anpassung, auch Adaption genannt, kann in einigen Fällen durch das Enwurfsmuster “Adapter” (siehe [Gamma et.al.]) erfolgen. Oftmals reicht eine Adaption der Schnittstellen allerdings nicht aus. In einigen Fällen muß die Anpassung bis auf Anweisungsebene erfolgen. Dies ist mit dem Enwurfsmuster “Adapter” allerdings nicht möglich, da hierzu der Quellcode transformiert werden muß. Das eine manuelle, konsistente Anpassung über mehrere Module, Klassen und Methoden hinweg sehr fehleranfällig ist, braucht an dieser Stelle nicht weiter belegt werden, man denke nur allein an das Auffinden des zu ändernden Quellcodes.

Ein weiteres, oft anzutreffendes Problem stellt die konsistente Änderung der Struktur eines Systems dar. Wird zum Beispiel die Schnittstelle einer Klasse geändert, so muß oftmals deren Benutzung im ganzen System angepaßt werden. Manuell ist dies in einem komplexen System kaum mit annehmbarem Aufwand zu bewerkstelligen, ein Werkzeug ist dazu notwendig.

Ziel dieser Arbeit ist ein Mechanismus, welcher auf abstrakter Ebene die notwendigen Codeänderungen an einem Softwaresystem beschreiben und durchführen kann. Damit

sind einerseits konsistente Änderungen möglich, andererseits kann Adaption auf Quellcodeebene durchgeführt sowie einzelne Eigenschaften des Quellcodes getrennt voneinander spezifiziert werden.

1.2 Lösungsskizze

Die Lösung der angesprochenen Probleme besteht darin, ein Werkzeug zu entwickeln, welches den Quellcode eines Systems analysieren und nach den Wünschen des Anwenders transformieren kann. Die Trennung der Funktionalität und der anderen Eigenschaften läßt sich mit einem solchen Werkzeug dadurch bewerkstelligen, daß man diese Dinge getrennt spezifiziert und erst zum Schluß miteinander "verwebt". Dem Werkzeug müßte dann nur noch mitgeteilt werden, wie es die einzelnen Teile miteinander verweben soll. Durch einen solchen Mechanismus kann man auch das Problem des Adaption auf Quellcodeebene in den Griff bekommen, da hierbei transformierend auf den Quellcode zugegriffen werden muß.

Natürlich darf die Benutzung eines solchen Werkzeugs nicht komplizierter sein als die manuelle Durchführung der erforderlichen Transformationen. Eine Beschreibung der Form *"In Zeile x füge den Quelltext y ein"* wäre sicher kein Fortschritt gegenüber dem manuellen Vorgehen. Es ist also eine entsprechende Beschreibungssprache zu entwickeln, die auf einem abstrakten Niveau die erforderlichen Transformationen beschreibt.

Das in dieser Studienarbeit beschriebene Werkzeug *Inject/J* stellt einen solchen "Verweber" dar (im folgenden nur noch *Weber* genannt). Der Weber und die dazugehörige Beschreibungssprache beschränkt sich dabei auf die objektorientierte Programmiersprache Java.

1.3 Aufbau der Arbeit

Der Rest der Arbeit gliedert sich in 4 Kapitel.

1. *Theoretische Grundlagen*
   Dieses Kapitel gibt einen Überblick über die zugrundeliegenden theoretischen Grundlagen. Diese sind vor allem in der aspektorientierten Programmierung und der Metaprogrammierung zu finden.

2. *Die Beschreibungssprache*
   Hier wird die Beschreibungssprache vorgestellt. Sie stellt die Grundlage für die Arbeit des Webers dar. Die Sprache beschreibt, wo und was am Quellcode transformiert werden soll.

3. *Der Weber*
   Dieser Teil der Arbeit stellt die Struktur und die Arbeitsweise des Webers vor. Es werden dabei die einzelnen Module dieses Werkzeugs durchleuchtet. In diesem Kapitel wird auch die Implementierung des Webers und dessen Anbindung an die Entwicklungsumgebung TogetherJ von TogetherSoft (vormals Object International) beschrieben. Es wird hierbei nur auf die wichtigsten Implementierungsdetails eingegangen. Die Dokumentation der einzelnen Klassen ist dem Quelltext bzw. der daraus generierten Dokumentation zu entnehmen. Zuletzt wird noch die Anwendung des Webers anhand eines Beispiels demonstriert.

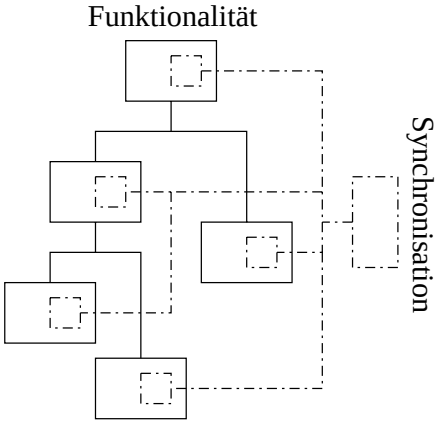
4. *Zusammenfassung und Ausblick*
   Dieser letzte Teil gibt, neben einer Zusammenfassung, abschließend einen Überblick über mögliche Verbesserungen und Erweiterungen von Inject/J.

Kapitel 2

Theoretische Grundlagen

2.1 Einordnung in AOP

AOP steht für aspektorientiertes Programmieren. Das Ziel von AOP ist die bereits in der Einleitung angedeutete Trennung der einzelnen Teilaspekte eines Programms. Der Algorithmus zur Lösung einer gegebenen Problemstellung kann hierbei als der funktionale Aspekt angesehen werden, während andere Aspekte nichtfunktionale Anforderungen wie Synchronisation, Kommunikation und Fehlerbehandlung sind. Von großer Bedeutung ist in diesem Zusammenhang der Begriff der Orthogonalität von Aspekten. Orthogonalität von Aspekten beschreibt deren Eigenschaft, weitgehend getrennt voneinander spezifizierbar zu sein. Sie nehmen nur an bestimmten Stellen Bezug aufeinander, den sogenannten Anknüpfungspunkten (engl. “join points”). Stellt man sich die Funktionalität hierarchisch angeordnet vor, so durchschneidet beispielsweise die Synchronisation diese orthogonal.



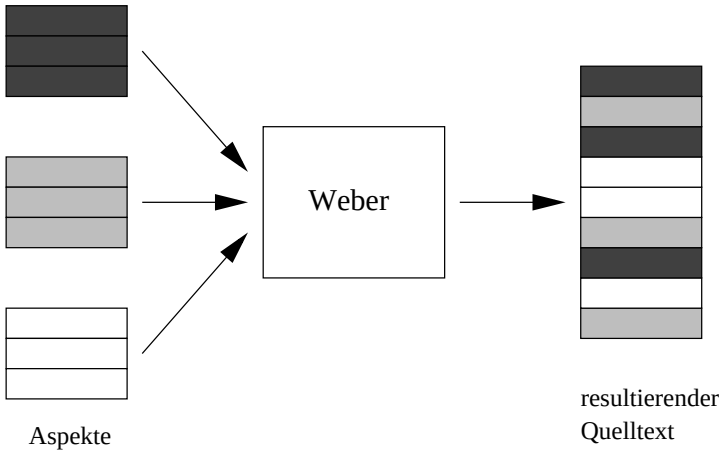
Synchronisation und Funktionalität sind im Prinzip unabhängig voneinander spezifizierbar, da der Aspekt Synchronisation die Funktionalität nicht verändert. Anhand des folgenden Beispiels soll verdeutlicht werden, wie sich die Aspekte Funktionalität, Synchronisation und Fehlerbehandlung durchschneiden:

```
...
Stack myStack = new Stack();            [Funktionalität]
if (myStack==null) System.exit(1);      [Fehlerbehandlung]
...
synchronized(myStack) {                 [Synchronisation]
```

```
        if (myStack.empty()) System.exit(2); [Fehlerbehandlung]
        Object top = myStack.pop();          [Funktionalität]
    }
    ...
```

Bereits an diesem kurzen Beispiel ist zu sehen, daß die Funktionalität nur einen Teil des gesamten Quelltextes ausmacht. Die übrigen Aspekte wie Synchronisation und Fehlerbehandlung vergrößern den Quelltext in nicht unerheblichen Maße. Dies führt im schlimmsten Fall dazu, daß die Funktionalität nur noch schwer zu identifizieren ist. Gelänge es nun, diese Aspekte unabhängig voneinander zu spezifizieren und erst vor der Kompilierung miteinander zu "verweben", so würde dies der Verständlichkeit, Wartbarkeit und der Wiederverwendung des Quelltextes dienen. Die Verständlichkeit würde erhöht, da die Funktionalität nicht erst im Quelltext identifiziert werden muß. Die Wartbarkeit und Wiederverwendung würden ebenfalls verbessert, da die Aspekte gegebenenfalls durch andere, den neuen Anforderungen gerechten, ersetzt und schließlich wieder mit einem Aspektweber zusammengeführt werden können.

Diese "separation of concerns", also die Trennung der Aspekte eines Programms, ist die Grundidee des aspektorientierten Programmierens. Leider bieten die verbreiteten Programmiersprachen keine Möglichkeit, diese Trennung auszudrücken. Stattdessen müssen alle Aspekte direkt als Codestücke, welche im ganzen System verteilt sein können, angegeben werden. Falls nun zu einem Zeitpunkt ein Aspekt verändert werden soll (zum Beispiel asynchrone statt synchrone Kommunikation), so müssen alle diese Codestücke im System gesucht und geändert werden. Dies manuell durchzuführen ist natürlich sehr aufwendig und fehleranfällig. Aus diesem Grund bedient man sich des schon angesprochenen Aspektwebers. Er webt die einzelnen Aspekte zu einem kompilier- und lauffähigen System zusammen.



Ausgehend von der Definition der einzelnen Aspekte ist der Aspektweber in der Lage, diese in der richtigen Reihenfolge zu einem resultierenden, den heutigen Programmiersprachen genügenden System zusammenzuführen. Dieses Verweben kann sowohl statisch als auch dynamisch erfolgen. Beim statischen Verfahren wird Quelltext erzeugt, welcher anschließend übersetzt werden kann. Die verschiedenen Aspekte sind darin fest verdrahtet. Das Ändern eines Aspektes erfordert somit ein erneutes Verweben mit anschließendem Übersetzen. Im dynamischen Fall ist es möglich, Aspekte zur Laufzeit eines Systems zu ändern (zum Beispiel verschiedene Kommunikationsarten). Dies erhöht natürlich die Flexibilität des Systems, wird andererseits aber meist mit einer geringeren Effizienz erkauft, da viele Indirektionen (zum Beispiel durch Delegation) auftreten. Dynamisch änderbare Aspekte besitzen oftmals auch nicht die gleiche Mächtigkeit wie statisch verwobene

Aspekte, da viele Informationen des Quelltextes zur Laufzeit nicht mehr zur Verfügung stehen.

Damit der Weber übehaupt weiß, an welchen Stellen er die verschiedenen Aspekte zusammenführen soll, gibt es die schon erwähnten "join points". An diesen Stellen werden die Aspekte zusammengeführt. Diese Webepunkte können sowohl implizit (durch Muster im Quelltext) als auch explizit angegeben sein. Im obigen Codebeispiel wären zum Beispiel das Erzeugen des Kellerobjekts *mystack* ein Webepunkt, an welchem die Aspekte Funktionalität und Fehlerbehandlung zusammengeführt werden. Ein weiterer solcher Webepunkt wäre der Aufruf der Operation *pop()* auf dem Kellerobjekt *mystack*, an welchem die Aspekte Funktionalität, Synchronisation und Fehlerbehandlung aufeinandertreffen.

Aspektorientiertes Programmieren stellt somit eine Möglichkeit zur Lösung der in der Einführung genannten Probleme dar. Die funktionalen und nichtfunktionalen Anforderungen können getrennt voneinander spezifiziert werden, Adaption kann durch Einweben neuer bzw. Ersetzen vorhandener Aspekte bewerkstelligt werden.

Weitere Informationen zum Thema AOP können beispielsweise den grundlegenden Arbeiten von Kiczales et. al. [KLM97] entnommen werden.

2.2 Das Metaobjektprotokoll (MOP)

Um werkzeuggesteuert auf einen gegebenen Quelltext Transformationen durchführen zu können, muß dieser erst in einer geeigneten Weise repräsentiert werden. Ein Übersetzer bedient sich in diesem Fall seines Zerteilers und der semantischen Analyse. Nach deren Ausführung steht dem Übersetzer der sogennante abstrakte Syntaxbaum (AST, abstract syntax tree) zur Verfügung.

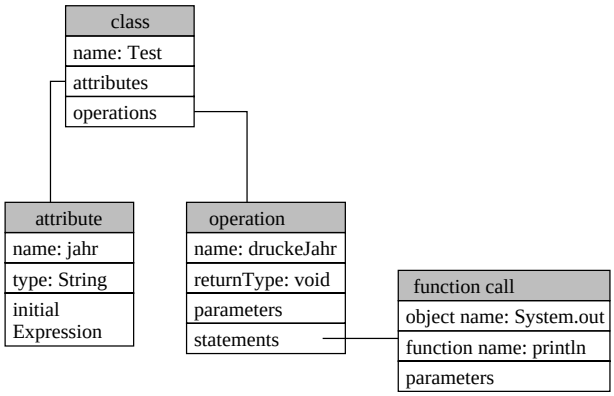
Dies gilt natürlich auch für objektorientierte Sprachen. Die Knoten des Baumes entsprechen dabei den einzelnen Elementen der Sprache wie Klassen, Methoden, Attribute, Ausdrücke etc. Die Kanten zwischen diesen Knoten bilden dabei die Beziehungen der verschiedenen Elemente untereinander, wie zum Beispiel die Vererbungsbeziehungen zwischen Klassen.

In objektorientierten Systemen können die Knoten durch entsprechende Objekte repräsentiert werden, die Kanten werden dann durch Referenzen auf andere Objekte dargestellt. Auf diesen Knotenobjekten kann man nun Operationen definieren. Diese Operationen können sowohl dem Auslesen und dem Verändern der in den Objekten gekapselten Informationen dienen, als auch dem Erzeugen neuer Knotenobjekte und dem Setzen der entsprechenden Referenzen. Durch das Ändern der Knotenobjekte und der Referenzen untereinander ist es nun möglich, indirekt auch den Quellcode zu ändern, da dieser ja nur durch die Baumelemente repräsentiert wird. Einen solchen Mechanismus nennt man *Metaobjektprotokoll (MOP)*.

Angenommen, man hätte folgende Klassendeklaration:

```
class Test {
   String jahr = new String("1999");
   void druckeJahr() {
      System.out.println("Aktuelles Jahr: "+jahr);
   }
}
```

Diese Klasse könnte nun (ausschnittsweise) durch folgenden Baum repräsentiert werden:



Je nachdem, wie leistungsfähig ein solches MOP ist, stellt es die vielfältigsten Informationen zur Verfügung, wie zum Beispiel statische Typinformationen von Variablen mit Verweisen auf die entsprechenden Klassen oder Referenzen auf alle Benutzungsstellen einer deklarierten Variable.

Ein MOP stellt immer auch Operationen zu Navigation innerhalb des Baumes zur Verfügung. Dies kann einerseits durch explizite Navigation mit Hilfe der Objektreferenzen geschehen. Andererseits kann die Navigation auch mit Hilfe des Entwurfsmusters "Besucher" (siehe [Gamma et.al.]) erfolgen. Dabei werden die gewünschten Knoten von einem Besucher nacheinander aufgesucht und die gewünschten Operationen auf ihnen ausgeführt.

Wie aus einem veränderten Baum wieder Quelltext generiert wird, bleibt dem Metaobjektprotokoll überlassen. Es kann sowohl bei jeder Änderung am Modell den entsprechenden Quelltext ändern, es kann aber auch erst zum Schluß den resultierenden Quelltext generieren.

Eine detailierte Beschreibung eines Metaobjektprotokolls kann zum Beispiel in [KRB] nachgelesen werden.

2.3 Zusammenhang AOP und MOP

Da durch ein Metaobjektprotokoll auf einen gegebenen Quelltext transformierend zugegriffen werden kann, eignet sich dieser Mechanismus zur Implementierung eines Aspektwebers. Er kann unter Zurhilfenahme eines Metaobjektprotokolls den bisherigen Quelltext analysieren, die entsprechenden Webepunkte herausfinden und den resultierenden Code erzeugen. Das MOP stellt somit eine mögliche Basistechnologie für die Implementierung eines Aspektwebers dar. Im Gegensatz zu Aspektwebern gibt es bereits eine ganze Reihe kommerziell verwendeter Metaobjektprotokolle. Sie sind oftmals, für den Anwender unsichtbar, in Entwicklungsumgebungen integriert. Diese benutzen die Metaobjektprotokolle einerseits dazu, Informationen aus dem eingegebenen Quelltext zu gewinnen, wie zum Beispiel die oftmals angezeigten Methoden- und Klassenlisten. Andererseits wird der durch Assistenten generierte Quellcode in einigen Fällen durch eben solch ein Metaobjektprotokoll erstellt (beispielsweise in den Entwicklungsumgebungen JBuilder von Borland/Inprise und TogetherJ von TogetherSoft).

In dieser Arbeit wird ein solches Metaobjektprotokoll als Basistechnologie benutzt. Eine zu entwickelnde Skriptsprache stellt dabei einen abstrakten Zugriff auf das MOP bereit.

Kapitel 3

Die Aspekt-Beschreibungssprache

3.1 Einführung

Die hier vorgestellte Sprache stellt die Grundlage für den Weber dar. Mit ihr werden die zu besuchenden Webepunkte durch Navigationsanweisungen eindeutig bestimmt. Neben diesen Navigationsanweisungen werden auch Operationen auf diesen Webepunkten sowie Befehle zur Ablaufsteuerung definiert.

3.2 Ziele der Beschreibungssprache

Ein wesentliches Entwurfsziel der hier vorgestellte Beschreibungssprache war eine einfache Syntax und eine einprägsame Semantik, so daß keine langen Einarbeitungszeiten notwendig sind. Da die einzuwebenden Aspekte im allgemeinen klein sind, waren kurze Turnaround-Zeiten wichtig. Aus diesem Grunde wurde eine Skriptsprache gewählt, welche interpretiert wird. Eine Skriptsprache gewährt einerseits ein entsprechendes Abstraktionsniveau, andererseits sind kurze Turnaround-Zeiten möglich.

Ein weiteres Ziel war die Einsetzbarkeit der Sprache für verschiedene Teilaspekte eines Programms. Im Gegensatz zu spezialisierten Aspektbeschreibungssprachen braucht der Anwender nicht für jeden Aspekt eine neue Sprache erlernen. Er programmiert die Funktionalität der Aspekte weiterhin in Java, nur muß er nun darüber hinaus Webepunkte angeben, an welchen dieser Java-Quelltext eingefügt werden soll. Die Beschreibungssprache setzt bewußt auf diesem Abstraktionsniveau auf, um einerseits die gewünschte Flexibilität zu erreichen und andererseits den Weber auch für nicht aspektorientierte Programmtransformationen zu nutzen. Dieses Entwurfsziel ist vor dem Hintergrund zu sehen, daß der Weber dem Umfeld des TROOP-Projekts (TRansforming Object-Oriented Design using Design Patterns) am Forschungszentrum Informatik entsprungen ist. Aus diesem Grund finden sich auch Operationen in der Sprache wieder, welche für das reine Einweben von Aspekten nicht notwendig wären. Zu diesen Befehlen zählen zum Beispiel das Ändern der Vererbungsbeziehungen wie auch das der Zugriffsmodifizierer einzelner Klassen/Methoden/Attributen.

3.3 Unterstützte Webepunkte

Durch die Beschreibungssprache werden sowohl implizite als auch explizite Webepunkte unterstützt. Letztgenannte werden durch ihren Namen identifiziert, während sich implizite Webepunkte durch Muster im abstrakten Syntaxbaum qualifizieren. Es ist möglich, mehreren expliziten Webepunkte den gleichen Namen zu geben. In diesem Fall handelt es sich

um einen sogenannten verteilten Webepunkt. Alle diese Webepunkte werden ihm Rahmen des Sichtbarkeitsbereiches gleichwertig behandelt.

An impliziten Webepunkten werden unterstützt (d.h. sind durch entsprechende Navigationsanweisungen erreichbar):

- Klassen
- Methoden
- Deklarationen von Klassenattributen und lokalen Variablen
- Methodenaufrufe
- Zuweisungen an eine Variable (Klassenattribut oder lokale Variable)

Deklarationen und Zuweisungen können auf Klassen- und auf Methodenebene betrachtet werden. Deklarationen auf Klassenebene sind nur Deklarationen von Attributen, da die Deklaration von Methoden als eigener Webepunkt vorhanden ist. Deklarationen auf Methodenebene sind Deklarationen lokaler Variablen. Zuweisungen auf Klassenebene sind all jene Attributdeklarationen, die auf Klassenebene initialisiert werden. Zuweisungen auf Methodenebene sind alle Zuweisungsausdrücke.

3.4 Syntax

Die Syntax der Aspektbeschreibungssprache ist relativ einfach gehalten. Die unten angegebene Syntaxbeschreibung ist in erweiterter Backus-Naur-Form gegeben. Fettgedruckte Wörter bzw. Zeichen stellen Terminale, kursivgedruckte stellen Nichtterminale dar. Die Notation entspricht dabei [GOOS].

Innerhalb der Aspektdatei sind Kommentare erlaubt. Einzeilige Kommentare beginnen dabei mit "//", Kommentarblöcke beginnen mit "/*" und enden mit "*/". Geschachtelte Kommentarblöcke werden nicht unterstützt.

Aspectdeclaration	**aspect** *Identifier* [**needs** *Identifierlist*] [**implements** *Identifierlist*] **{** *Aspectbody* **}** | **aspect interface** *Identifier* [**needs** *Identifierlist*] [**extends** *Identifierlist*]
Aspectbody	(*Statement*)*
Statementblock	*Statement* | **{** (*Statement*)* **}**
Statement	*Searchstatement* | *Branchstatement* | *Foreachstatement* | *Weavestatement* | *Message* | **exit**
Message	**message ("** *String* **")**
Searchstatement	**in class** *Classidentifier* [*Declarealias*] **do** *Statementblock* | **in method** *Methodidentifier* [*Declarealias*] **do** *Statementblock* |
Branchstatement	(**if** *Boolvalue* [(**&** *Boolvalue* | | *Boolvalue*)] **do** | **if in** (**class** *Classidentifier* [*Declarealias*] | **method** *Methodidentifier* [*Declarealias*] **from class** *Classidentifier*) **do** *Statementblock* [**else** *Statementblock*] **if exists** (**class** *Classidentifier* [*Declarealias*] | **method** *Methodidentifier* [*Declarealias*]

	| **attribute** *Attributeidentifier* [*Declarealias*]) *Statementblock* [**else** *Statementblock*]
Foreachstatement	**foreach** *Joinpoint* [*Declarealias*] **do** *Statementblock*
Weavestatement	**before** [*Joinpoint*] *SourceCodeBlock* | **after** [*Joinpoint*] *SourceCodeBlock* | **replace** [*Joinpoint*] *SourceCodeBlock* | **delete** [*Joinpoint*] **change modifier** [*Joinpoint*] *SourceCodeBlock* | **change package** *Packageidentifier* | **add to imports** Importidentifier | **add to extends** Extendsidentifier | **add to implements** Implementsidentifier | **add to throws** Throwsidentifier | **add to file** *Classidentifier SourceCodeBlock*
Joinpoint	**class** *Classidentifier* **method** *Methodidentifier* | **call** *Methodidentifier* [**to (class|subclass)** *Classidentifier*] | **assignment to** *Variableidentifier* | **declaration of** *Declarationidentifier* | **joinpoint** *ExplicitJoinpointidentifier*
Identifier	(*Letter* | *Wildcard*) (*Letter* | *Digit* | *Wildcard* | **.**)*
IdentifierList	*Identifier (* **,** *Identifier*)*
Letter	**a-z** | **A-Z** | **_**
Digit	**0-9**
Wildcard	***** | **?**
SourceCodeBlock	**${** Java Quellcode **}$**

Die nicht angegebenen Identifizierer bzw. *Boolvalue* und *Declarealias* werden in den folgenden Abschnitten näher erläutert.

3.5 Semantik

3.5.1 Identifizierer

Identifizierer sind die Ausdrücke welchen einen Knoten des abstrakten Syntaxbaums als Webepunkt qualifizieren oder nicht. Dies kann im einfachsten Fall, wie zum Beispiel bei Klassen, ein einfacher Name sein. Kompliziertere Beispiele stellen die Identifizierer für Methoden dar, welche die komplette Signatur berücksichtigen. Identifizierer müssen im allgemeinen eindeutig bezüglich des aktuellen Sichtbarkeitsbereiches sein. Die Ausnahme bildet die foreach-Schleife, in denen Identifizierer Platzhalter (Wildcards) enthalten dürfen. Die Platzhalter in einem Identifizierer sind '*' für beliebig viele Zeichen bzw. '?' für genau ein Zeichen. Bei Methodensignaturen steht '*' für beliebig viele Parametertypen, '?' für genau einen Parametertyp.

Beispiele für Identifizierer:

Test.Testklasse	Klasse “Testklasse” im Paket “Test”
Test.Test*	alle Klassen im Paket “Test”, welche mit “Test” beginnen
m()	parameterlose Methode m
m(*)	alle Methoden mit Namen m
m(x,*)	alle Methoden mit Namen m und erstem Parametertyp x und beliebigen weiteren Parametertypen
m(*,y)	alle Methoden mit Namen m und letztem Parametertyp y
m(x,*,y,*)	alle Methoden mir Namen m, dessen erster Parametertyp x ist und noch einen weiteren Parameter vom Typ y hat
m(x,?,y)	alle Methoden mit Namen m und 3 Parametern, dessen zweiter Parametertyp egal ist
get*(*)	alle Methoden, deren Namen mit ’get’ beginnt

3.5.2 Benutzung von Aliasen

Die meisten Navigationsbefehle erlauben das Setzen von Verweisen auf den aktuell besuchten Knoten, einen sogenannten Alias. Mit Hilfe dieser Aliase ist es möglich, innerhalb ihres Gültigkeitsbereiches auf verschiedenene Eigenschaften dieser Webepunkte zuzugreifen. Der Gültigkeitbereich eines Alias ist der Anweisungsblock der Navigationsanweisung. Beim Iterieren mit Hilfe der *foreach*-Schleife ist der Alias immer ein Verweis auf den aktuell besuchten Webepunkt.

Ein Alias wird durch die Zeichenkette <= *Aliasname* > deklariert. Der Aliasname ist dabei eine Zeichenkette, welche mit einem Buchstaben oder dem Zeichen “_” beginnt, gefolgt von beliebig vielen Buchstaben, Zahlen oder dem Zeichen “_”.

Aliase werden an zwei Stellen benutzt. Der erste Fall ist die *If*-Anweisung, der zweite ist die Benutzung innerhalb des Java-Quellcodes der Webeanweisungen. Im Fall der If-Anweisung läßt sich der Ablauf unter Zuhilfenahme der Eigenschaften der durch die Aliase referenzierten Webepunkte steuern. Im zweiten Fall kann der eingewebte Quelltext abhängig vom Webepunkt und der dazugehörigen Navigation verändert werden. Jedes Vorkommen eines Aliasausdruckes innerhalb des einzuwebenden Java-Quelltextes wird, falls möglich, durch den Wert dieses Ausdruckes ersetzt. Ist ein solcher Ausdruck innerhalb des Java-Quelltextes nicht auswertbar, so bleibt er unverändert.

Aliasausdrücke stehen immer in spitzen Klammern: < *Aliasausdruck* >. Der Typ des Aliasausdruckes muß immer vom Typ *Bool* oder *String* sein. Der Typ *Bool* ist eigentlich auch ein *String*, da die Werte “true” und “false” durch Zeichenketten ausgedrückt werden, so daß die Ergebnisse gegebenenfalls in den Java-Quelltext eingefügt werden können. Der einfachste Aliasausdruck ist von der Form *Aliasname.Eigenschaft*, wobei Eigenschaft den Typ *Bool* oder *String* hat. Diese beiden Typen lassen sich dann direkt in den Java-Quelltext einfügen. Der Typ *Bool* kann, wie bereits erwähnt, als Bedingung für eine *if*-Anweisung benutzt werden. Manche Eigenschaften geben allerdings nur eine Referenz auf einen weiteren Webepunkt zurück. In diesem Fall kann mittels “.” auf weitere Eigenschaften dieses Webepunktes zugegriffen werden. So liefert zum Beispiel der Aliasausdruck <*a.inMethod.name*> den Namen der Methode zurück, in welcher die Zuweisung mit dem Alias *a* steht.

Eine genaue Auflistung aller Eigenschaften der einzelnen Webepunkte ist dem Anhang zu entnehmen.

3.5.3 Aspektdeklarationen

Eine Aspektdeklaration wird mit den Schlüsselwort *aspect* eingeleitet. Falls weiterhin das Schlüsselwort *interface* folgt, so handelt es sich um ein Aspektinterface, welches keinen Rumpf besitzt. Nach dem Namen des Aspektes werden nun die Aspekte angegeben, welche von diesem Aspekt benötigt werden. Bei Aspektinterfaces kann weiterhin angege-

ben werden, welche Aspekte durch diesen Aspekt erweitert werden. Dadurch lassen sich Vererbungshierarchien aufbauen. Bei gewöhnlichen Aspektdeklarationen kann angegeben werden, welche Aspektinterfaces durch diesen Aspekt implementiert werden. Diese zusätzlichen Angaben (needs, extends und implements) werden vom Weber genutzt, um die Abhängigkeiten zwischen den Aspekten zu prüfen. Wird zum Beispiel von einem Aspekt ein anderer Aspekt benötigt, so muß eine Implementierung für letztgenannten vorliegen. Der Aspektrumpf schließlich enthält die eigentlichen Anweisungen und wird vom Weber sequentiell abgearbeitet.

3.5.4 Navigationsbefehle

Unter Navigation versteht man das Auffinden der gewüschten Webepunkte innerhalb des abstrakten Syntaxbaums. Die Navigation geschieht unter Zuhilfenahme des Entwurfsmusters "Besucher" [Gamma et.al.]. Ausgehend von der aktuellen Position des Besuchers im abstrakten Syntaxbaum wird zu einem darunterliegenden Knoten navigiert.

Zu Beginn befindet sich der Besucher in der sog. Wurzel. Von ihr aus sind alle (nicht inneren) Klassen auffindbar welche in der Menge "namespace" definiert sind. Diese Menge wird durch eine externe Konfigurationsdatei oder der graphischen Bedienoberfläche belegt.

in class *Klassenidentifizierer*

Falls sich der Besucher in der Wurzel befindet, dann wird die angegebene Klasse im Namensraum gesucht und als aktueller Knoten des Besuchers gesetzt. Die angegebene Klasse darf keine innere Klasse sein. Der Name muß vollständig qualifiziert sein.

Befindet sich der Besucher in einem Klassenknoten, so wird nach einer inneren Klasse innerhalb dieser Klasse gesucht.

Ist die aktuelle Position des Besuchers ein anderer Knotentyp, so führt diese Anweisung zu einem Fehler (insbesondere werden keine inneren Klassen innerhalb von Methoden berücksichtigt).

in method *Methodenidentifizierer*

Diese Anweisung ist nur erlaubt, falls sich der Besucher in einem Klassenknoten befindet. Die angegebene Methode wird in dieser Klasse durch den Besucher aufgesucht.

foreach class *Klassenidentifizierer*

Diese Schleife ist nur in Klassenknoten und in der Wurzel erlaubt. Es werden nacheinander alle Klassen besucht, dessen Namen dem angegebenen Klassennamen genügen. Innerhalb der Wurzel werden alle Klassen berücksichtigt, die innerhalb von "namespace" deklariert sind. Innerhalb eines Klassenknotens werden nur dessen innere Klassen herangezogen.

foreach method *Methodenidentifizierer*

"foreach method" ist nur innerhalb von Klassenknoten erlaubt. Es werden nacheinander alle Methodenknoten besucht, die der angegebenen Signatur genügen.

foreach declaration of *Attribut-/Variablenidentifizierer*

Diese Anweisung ist nur innerhalb von Klassen- und Methodenknoten zulässig. Befindet sich der Besucher in einem Klassenknoten, so werden nacheinander alle Attributdeklarationen aufgesucht, dessen Attributname dem angegebenen Attributidentifizierer genügt. Befindet sich der Besucher dagegen in einem Methodenknoten, so werden alle Deklarationen von lokalen Variablen innerhalb des Methodenrumpfes besucht, dessen Variablenname dem angegebenen Variablenidentifizierer genügt.

foreach assignment to *Attribut-/Variablenidentifizierer*

Diese Schleife ist für Klassenknoten und für Methodenknoten erlaubt. Innerhalb eines Klassenknotens werden nur die Attributdeklarationen besucht, welche auf Klassenebene initialisiert werden (dies ist die Zuweisung auf Klassenebene) und dessen Attributname dem angegebenen Attributidentifizierer genügt. In Methodenknoten werden alle Zuweisungen an lokale Variablen oder Klassenattribute innerhalb des Methodenrumpfes besucht, dessen Namen dem angegebenen Name genügen.

foreach call *Methodenidentifizierer* [to (class|subclass) *Klassenidentifizierer*]

Methodenaufrufe können nur aufgesucht werden, falls sich der Besucher in einem Methodenknoten befindet. Es werden alle Aufrufe zu Methoden besucht, deren Signatur dem angegebenen Methodenidentifizierer genügt. Ist kein "to class" oder "to subclass" angegeben, dann ist es egal welchem Objekttyp dieser Aufruf gilt. Ist ein "to class" vorhanden, so werden nur Methodenaufrufe auf Objekten berücksichtigt, die genau vom Typ der angegebenen Klasse sind. Bei diesen Objekten ist der im Quelltext deklarierte Typ ausschlaggebend, nicht der Typ zur Laufzeit. Ist ein "to subclass" angegeben, so muß der Objekttyp, dem der Aufruf gilt, entweder die angegebene Klasse oder eine ihrer Unterklassen sein. Auch in diesem Fall ist der deklarierte Objekttyp im Quelltext ausschlaggebend.

foreach joinpoint *Webepunktname*

Diese Anweisung ist für alle Knoten erlaubt, welche noch Unterknoten besitzen. Dies sind im einzelnen: Wurzel, Klassenknoten, Methodenknoten. Es werden alle expliziten Webepunkte mit dem angegebenen Namen besucht, welche sich im Sichtbarkeitsbereich des aktuellen Knotens befinden. Ist die Position des Besuchers die Wurzel, so werden alle expliziten Webepunkte mit dem angegebenen Namen besucht. Ist der aktuelle Knoten ein Klassenknoten, so werden alle expliziten Webepunkte innerhalb dieser Klasse besucht. Befindet sich der Besucher in einem Methodenknoten, so werden nur die expliziten Webepunkte innerhalb des Methodenrumpfes berücksichtigt.

3.5.5 Befehle zur Ablaufsteuerung

Mit Hilfe der Befehle zur Ablaufsteuerung ist eine bedingte Ausführung von Anweisungen möglich. Ist die Bedingung der If-Anweisung erfüllt, so werden die Anweisungen des If-Anweisungsblocks abgearbeitet. Ist die Bedingung nicht erfüllt, so werden die Anweisungen des eventuell angegebenen Else-Anweisungsblocks ausgeführt.

if in class *Klassenidentifizierer*

Die Bedingung ist erfüllt, falls sich der Besucher in einem Klassenknoten befindet, dessen Klassenname dem angegebenen Klassenidentifizierer genügt. Im Klassenidentifizierer darf kein Platzhalter vorkommen. Diese Anweisung ist vor allem in Verbindung mit der Foreach-Schleife interessant, um für einzelne Klassen spezielle Anweisungen auszuführen.

if in method *Methodenidentifizierer*

Befindet sich der Besucher in einem Methodenknoten, dessen Signatur dem angegebenen Methodenidentifizierer genügt, so werden die angegebenen Anweisungen des If-Blocks ausgeführt. Der Methodenidentifizierer darf keine Platzhalter besitzen.

if exists class *Klassenidentifizierer*

Die Anweisungen des If-Blocks werden nur ausgeführt, falls die angegebene Klasse existiert. Ansonsten wird ein eventuell angegebener Else-Block ausgeführt. Der Klassenidentifizierer muß wie üblich vollständig qualifiziert sein. Platzhalter sind erlaubt, doch der If-Block wird nur einmal ausgeführt, auch falls mehrere Klassen diesem Klassenidentifizierer genügen. Falls Platzhalter benutzt werden, darf aus diesem Grund auch kein Alias vergeben werden, da die spezifizierte Klasse nicht unbedingt eindeutig ist. Innere Klassen können ebenfalls angesprochen werden. In diesem Fall muß das Format *Basisklasse.Innereklasse* lauten, Platzhalter sind auch in diesem Fall erlaubt.

if exists method *Methodenidentifizierer*

Existiert die angegebene Methode, so wird der If-Block ausgeführt. Der Methodenidentifizierer muß auch die Klasse spezifizieren. Dies muß im Format *Klassenname.Methodensignatur* erfolgen. Der Klassenname muß wie immer vollständig qualifiziert sein, es sei denn Platzhalter werden benutzt. Für die Methodensignatur gilt das in der Einleitung gesagte. Werden Platzhalter benutzt, so darf kein Alias vergeben werden, da der If-Block nur einmal ausgeführt wird und die identifizierte Methode nicht unbedingt eindeutig ist. Befindet sich der Besucher in einem Klassenknoten, so kann der Methodenidentifizierer auch ohne Klassenangabe erfolgen. In diesem Fall werden nur die Methoden der aktuellen Klasse berücksichtigt.

if exists attribute *Attributidentifizierer*

Der If-Block wird ausgeführt, falls das angegebene Attribut existiert. Analog zu *if exists method* muß auch die Klasse spezifiziert werden, es sei denn der Besucher befindet sich in einem Klassenknoten, dann werden nur die Attribute dieser Klasse berücksichtigt. Der Attributidentifizierer muß im Format *Klassenname.Attributname* angegeben werden. Werden Platzhalter benutzt, so darf kein Alias vergeben werden, da der If-Block nur einmal ausgeführt wird, auch falls mehrere Attribute dem Identifizierer genügen (Attribut ist dann nicht eindeutig).

if *BoolscherWert*

Ist der boolsche Wert gleich der Zeichenkette “true”, so wird der If-Anweisungsblock abgearbeitet. Ist der boolsche Wert gleich der Zeichenkette “false”, so wird ein eventuell angegebener Else-Anweisungsblock ausgeführt. Die boolschen Werte bekommt man aus der Eigenschaftsliste der besuchten Knoten, welche über Aliase ansprechbar sind. Typische Eigenschaften sind beispielsweise der Zugriff (öffentlich, geschützt oder privat). Eine ausführliche Liste der Eigenschaften der einzelnen Knotentypen ist dem Anhang zu entnehmen.

Beispiel:

```
foreach class * <=c> do {
   if <c.name.equals(“Class1”)> do {
      ...
   }
   else if <c.name.equals(“Class2”)> do {
      ...
   }
   else {
```

```
        ...
    }
}
```

Durch diese Anweisungen werden alle bekannten Klassen besucht, doch je nachdem ob sich der Besucher in der Klasse “Class1”, “Class2” oder einer anderen Klasse befindet, werden unterschiedliche Anweisungen ausgeführt.

Die aus der Eigenschaftsliste erworbenen boolschen Werte lassen sich in gewissen Grenzen logisch verknüpfen. An logischen Operatoren gibt es das logische NICHT (“!”), das logische UND (“&”) und das logische ODER (“|”). Ein solcher boolscher Ausdruck hat die Syntax:

> [“!”] ”<” *Boolsche Eigenschaft* “>” ([“&” | ”|”] [“!”] ”<” *Boolsche Eigenschaft* “>”)*

Es werden also keine Klammern unterstützt, insbesondere muß eine Negation direkt beim boolschen Wert erfolgen.

Ein gültiger boolscher Ausdrücke wären zum Beispiel:

```
!<c.name.equals(“Class1”) & <a.inMethod.name.equals(“Method1”)
```

Der Alias “c” verweise dabei auf eine Klasse, “a” sei eine Zuweisung auf Methodenebene. Der Teilausdruck “c.name” liefert den Namen der Klasse als Zeichenkette zurück. Auf Zeichenketten ist die Funktion “equals” definiert, welche einen entsprechenden boolschen Wert zurückliefert. Dieser zurückgelieferte Wert wird durch das vorangestellte “!” negiert. Der zweite Teilausdruck, welcher mit dem ersten logisch UND-verknüpft ist, liefert “true” zurück falls die Zuweisung sich in einer Methode namens “Method1” befindet (“a.inMethod” liefert einen Verweis auf die Methode zurück, in welcher sich die Zuweisung befindet). Dieser Ausdruck würde also die Zeichenkette “true” zurückliefern falls sich der Besucher in einem Zuweisungsknoten innerhalb einer Methode mit dem Namen “Method1” befindet und falls der Alias “c” verweist nicht auf eine Klasse mit Namen “Class1” verweist.

3.5.6 Webebefehle

Alle bisher beschriebenen Befehle dienten ausschließlich dem Auffinden der gewünschten Webepunkte, gegebenenfalls unter Nebenbedingungen. Mit den folgenden Befehlen ist es nun möglich, neuen Quelltext einzufügen. Bei einigen Webebefehlen ist es möglich noch einen Webepunkt anzugeben. Dies ist nur eine Kurzschreibweise für einen Navigationsbefehl, dessen Befehlsblock eben dieser Webebefehl ohne Webepunktangabe ist. Zum Beispiel sind die folgenden Befehle äquivalent:

```
before class DummyClass ${ ... }$
```

und

```
in class DummyClass do { before ${...}$ }
```

Die Identifizierer für diese Webepunkte müssen eindeutig sein, d.h. es dürfen keine Platzhalter vorkommen. Dies ist keine Einschränkung, da ansonsten die *foreach*-Navigationsanweisung benutzt werden kann.

Innerhalb des Quelltextes wird das Auftreten der Zeichenkette %SOURCETEXT% durch den bisherigen Quelltext des Webepunktes ersetzt. Außerdem werden Aliase wie in Abschnitt 3.5.2 beschrieben ersetzt.

Beim Einweben von Quelltext an expliziten Webepunkten ist der Benutzer dafür verantwortlich, daß der Quelltext bezüglich des Webepunktes korrekt ist (Memberdeklarationen oder Anweisungen).[1]

before *[Webepunkt] Quelltext*

Dieser Befehl fügt den angegebenen Quelltext vor der aktuellen Position des Besuchers ein. Für die einzelnen Webepunkte sind dies folgende Positionen:

Webepunkt	Ort des Einwebens
Klasse	Anfang des Klassenrumpfes. Quelltext muß daher gültige Memberdeklaration(en) sein.
Methode	Anfang des Methodenrumpfes.
Deklaration	Klassenattribut: Vor der Deklaration des Attributs. Quelltext muß daher gültige Memberdeklaration(en) sein. lokale Variable: Vor der Anweisung, welches Deklaration enthält.
Methodenaufruf	Vor der Anweisung, welches Aufruf enthält.
Zuweisung	Auf Klassenebene: Vor der Deklaration, welche initialisiert wird. Quelltext muß daher gültige Memberdeklaration(en) sein. Auf Methodenebene: Vor der Anweisung, welches Zuweisungsausdruck enthält.
Expliziter Webepunkt	Vor dem expliziten Webepunkt

after *[Webepunkt] Quelltext*

Durch Anwendung dieses Befehls ist es möglich, neuen Quelltext nach der aktuellen Position des Besuchers einzufügen.

Webepunkt	Ort des Einwebens
Klasse	Ende des Klassenrumpfes. Quelltext muß daher gültige Memberdeklaration(en) sein.
Methode	Vor jedem regulären Methodenaustritt, d.h. vor jeder *return*-Anweisung oder am Methodenende. Exceptions werden nicht betrachtet.
Deklaration	Klassenattribut: Nach der Deklaration des Attributs. Quelltext muß daher gültige Memberdeklaration(en) sein. lokale Variable: Nach der Anweisung, welches Deklaration enthält.
Methodenaufruf	Nach der Anweisung, welches Aufruf enthält.
Zuweisung	Auf Klassenebene: Nach der Deklaration, welche initialisiert wird. Quelltext muß daher gültige Memberdeklaration(en) sein. Auf Methodenebene: Nach der Anweisung, welches den Zuweisungsausdruck enthält.
Expliziter Webepunkt	Nach dem expliziten Webepunkt

[1] Bei der vorliegenden Implementation des Webers werden explizite Webepunkte auf Klassenebene nur sehr unzureichend unterstützt, da das zugrundeliegende Metaobjektprotokol SCI kaum Unterstützung von Kommentaren bietet. Diese Einschränkung muß aber nicht für andere MOPs gelten und soll deshalb nicht die Semantik der Webesprache negativ beinflussen.

In der vorliegenden Version des Webers gibt es noch Einschränkungen der Befehle *before* und *after* in Bezug auf Java-Schleifenanweisungen. Deklarationen, Methodenaufrufe und Zuweisungen in den Vergleich- und Inkrementanweisungen der *for*-Schleife werden nicht berücksichtigt, d.h. die Befehle *before* und *after* sind wirkungslos. Analoges gilt für die Abbruchbedingung der *while*-Schleife, auch hier sind die genannten Webebefehle wirkungslos. Schleifensteuernde Anweisungen, welche bei jedem Schleifendurchlauf ausgeführt werden, bleiben also z.Z. unberücksichtigt. Diese Einschränkung gilt nicht für die Initialisierungsanweisung der *for*-Schleife. Hier bewirkt der Befehl *before* ein Weben vor der *for*-Anweisung und ein *after* ein Weben nach dem Schleifenrumpf.

replace *[Webepunkt] Quelltext*

Mit dieser Anweisung ist es möglich den Quelltext des bisherigen Webepunktes durch den angegebenen Quelltext zu ersetzen. Im einzelnen gilt folgende Semantik.

Webepunkt	Ort des Einwebens
Klasse	Der bisherige Klassenrumpf wird ersetzt. Quelltext muß daher ein gültiger Klassenrumpf sein.
Methode	Der Methodenrumpf wird ersetzt. Quelltext muß daher ein gültiger Methodenrumpf sein
Deklaration	Klassenattribut: Die Deklaration des Attributs wird ersetzt. Quelltext muß daher gültige Memberdeklaration(en) sein. lokale Variable: Deklarationsausdruck wird ersetzt. Dieser Ausdruck kann auch innerhalb einer Anweisung sein. Quelltext muß daher gültiger Ausdruck sein.
Methodenaufruf	Der Aufrufausdruck wird ersetzt. Quelltext muß daher gültiger Ausdruck sein.
Zuweisung	Auf Klassenebene: Der linke Teil der Zuweisung wird ersetzt, d.h. der initiale Wert des Attributs. Quelltext muß daher gültiger Ausdruck sein. Auf Methodenebene: Der Zuweisungsausdruck wird ersetzt. Quelltext muß daher gültiger Ausdruck sein.
Expliziter Webepunkt	Nur für mehrzeilige explizite Webepunkte sinnvoll. In diesem Fall wird alles zwischen Webepunktbeginn und Webepunktende durch den angegebenen Quelltext ersetzt. Bei einzeiligen Webepunkten geschieht nichts.

delete *[Webepunkt] Quelltext*

Der Knoten, in dem sich der Besucher befindet, wird aus dem abstrakten Syntaxbaum gelöscht.

Webepunkt	Entfernter Knoten
Klasse	Die gesamte Klassendeklaration wird gelöscht
Methode	Die gesamte Methode wird gelöscht
Deklaration	Klassenattribut: Das Attribut wird aus der Klasse gelöscht lokale Variable: Nur der Deklarationsausdruck wird gelöscht

Methodenaufruf	Nur der Aufrufausdruck wird gelöscht
Zuweisung	Auf Klassenebene: Die Initialisierung des Klassenattributs wird gelöscht, nicht aber die Attributdeklaration Auf Methodenebene: Der Zuweisungsausdruck wird entfernt
Expliziter Webepunkt	Nur für mehrzeilige explizite Webepunkte sinnvoll. In diesem Fall wird alles zwischen Webepunktbeginn und Webepunktende gelöscht. Bei einzeiligen expliziten Webepunkten geschieht nichts.

Nachdem ein Knoten gelöscht wurde dürfen keine weiteren Aktionen auf ihm ausgeführt werden.

change modifier *[Webepunkt] Quelltext*

Ändert den Zugriffsmodifizierer für ein(e) Klassen/Methode/Attribut. Die gewünschten neuen Zugriffsmodifizierer müssen im Quelltext angegeben werden. Soll ein eventuell vorhandener Zugriffsmodifizierer explizit gelöscht werden, so ist ihm ein “!” voranzustellen.

Beispiele:

`change modifier ${ public abstract }$`
setzt die Zugriffsmodifizierer “public” und “abstract”.

`change modifier ${ public !abstract }$`
setzt den Zugriffsmodifizierer “public” und löscht ein eventuell angegebenes “abstract”.

Da nicht alle möglichen Kombinationen von Zugriffsmodifizierern erlaubt sind kommt es unter Umständen zu Konflikten mit bereits gesetzten Zugriffsmodifizierern. In diesem Fall hat immer der hier angegebene Modifizierer Vorrang. Konflikte werden also automatisch aufgelöst.

Für die unterstützten Webepunkte gelten folgende Konfliktauflösungen:

- Klassenknoten
  Erlaubte Zugriffsmodifizierer sind: “public”, “abstract” und “final”
  Die Konfliktauflösung folgt dem Schema:

gegeben	gewünscht	Ergebnis
final	abstract	abstract
abstract	final	final

- Methodenknoten
  Erlaubte Zugriffsmodifizierer sind: “public”, “protected”, “private”, “final”, “static”, “transient” und “volatile”
  Wird als neuer Zugriffsmodifizierer “abstract” gewünscht, so werden automatisch gegebene “private”, “static”, “final”, “native”, oder “synchronized” gelöscht. Ist zusätzlich zu "abstract" eines dieser Modifizierer gewünscht, so führt dies zu einem Fehler.
  Die Konfliktauflösung erfolgt ansonstem dem Schema:

gegeben	gewünscht	Ergebnis
public	protected	protected
public	private	private
protected	public	public
protected	private	private
private	public	public
private	protected	protected

- Attributknoten
  Erlaubte Zugriffsmodifizierer sind: “public”, “protected”, “private”, “final”, “static”, “transient”, “volatile”
  Die Konfliktauflösung erfolgt nach folgendem Schema:

gegeben	gewünscht	Ergebnis
public	protected	protected
public	private	private
protected	public	public
protected	private	private
private	public	public
private	protected	protected

add to imports *Importsidentifizierer*

Diese Anweisung ist nur für Klassenknoten erlaubt. Sie fügt in die Datei, in welcher die aktuelle Klasse sich befindet, eine neue *imports*-Anweisung ein. Dabei wird der angegebene *Importsidentifizierer* übernommen.

Beispiel:

```
add to imports java.util.*
```

fügt die Anweisung “import java.util.import;” ein.

add to extends *Klassenidentifizierer*

Dieser Befehl ist nur für Klassenknoten erlaubt. Er fügt den angegebenen Klassenidentifizierer in die Liste der Klassen ein, welche diese Klasse erweitert. Im Klassenidentifizierer darf deshalb kein Platzhalter vorkommen.

add to implements *Klassenidentifizierer*

Analog zu *add to extends* fügt diese Anweisung den angegebenen Klassenidentifizierer in die Liste der Klassen ein, welche diese Klasse implementiert. Aus diesem Grund ist diese Anweisung nur für Klassenknoten zulässig. Der Klassenidentifizierer darf keine Platzhalter besitzen.

add to throws *Klassenidentifizierer*

Diese Anweisung ist nur für Methodenknoten erlaubt. Sie fügt den angegebenen Klassenidentifizierer in die Liste der Ausnahmen ein, welche diese Methode werfen kann. Platzhalter sind deshalb natürlich im Klassenidentifizierer nicht erlaubt.

add to file *Klassenidentifizierer Quelltext*

Add to file ist nur erlaubt, falls sich der Besucher in der Wurzel befindet. Es wird nach der Datei gesucht, welche die angegebene Klasse enthält. Anschließend wird der angegebene Quelltext, bei welchem es sich um eine gültige Klassendeklaration handeln muß, in die Datei eingefügt.

Existiert die angegebene Klasse nicht, so wird eine neue Datei angelegt, dessen Pfad und Name sich aus dem angegebenen Klassenidentifizierer ableiten läßt. Anschließend wird der angegebene Quelltext, bei welchem es sich wiederum um eine gültige Klassendeklaration handeln muß, in die neu erstellte Datei eingefügt. Da es sich beim angegebenen Quelltext um eine Klassendeklaration handelt, müssen *imports*-Anweisungen gegebenenfalls zusätzlich mit Hilfe von *add to imports* eingefügt werden. Das Paket braucht nicht angegeben werden, da es sich aus dem Klassenidentifizierer ableiten läßt.

change package *Paketidentifizierer*

Dieser Webebefehl wird zur Zeit noch nicht unterstützt, ist aber schon in die Grammatik aufgenommen worden. Mit ihm soll es möglich sein, eine Klasse in ein anderes Paket zu verschieben.

3.5.7 Weitere Befehle

exit

Falls dieses Schlüsselwort auftritt, wird der Webevorgang umgehend abgebrochen.

message

Durch diese Anweisung ist es möglich, eine Nachricht für den Benutzer auszugeben. Aliasausdrücke innerhalb der Nachricht werden vor dem Anzeigen ausgewertet. So ist zum Beispiel eine Positionsangabe des Besuchers möglich.

3.6 Präprozessor-Anweisungen

Der Weber enthält einen einfachen Präprozessor. Die Präprozessor-Direktiven sind im einzelnen:

#macro *Makrosignatur*	Leitet eine Makrodefinition ein
#endmacro	Beendet Makrodefinition
#call *Makroaufruf*	Ruft ein Makro auf
#define *Konstante* [*Wert*]	Definiert eine Konstante mit dem angegebenen Text
#undef *Konstante*	Löscht die angegebene Konstante wieder
#ifdef *Konstante*	Der folgende Quelltext wird nur übernommen, falls die angegebene Konstante definiert wurde
#else	Falls #ifdef nicht zum Erfolg führte, dann wird der folgende Quelltext übernommen
#endif	Beendet #ifdef-#else Block

3.6.1 Makros

Mit Hilfe von Makros lassen sich immer wiederkehrende Anweisungsfolgen automatisieren. Da Makros parametrisierbar sind, stellen sie eine Art Funktionsaufruf dar.

Eine Makrodefinition hat immer die Form:

```
#macro Makroname ( [Parameterliste] )
Makrorumpf
#endmacro
```

Der *Makroname* muß mit einem Buchstaben beginnen, gefolgt von beliebig vielen weiteren Buchstaben oder Zahlen. Falls Parameter angegeben werden, so müssen diese mit "%" beginnen, gefolgt von einem Namen, welcher aus einer beliebigen Kombination aus Buchstaben und Zahlen bestehen kann. Dieser Name muß nicht notwendigerweise mit einem Buchstaben beginnen. Mehrere Parameter werden durch ein Komma getrennt.

Nach dieser Makrosignatur folgt der Makrorumpf. Dieser Makrorumpf kann ein beliebiges Stück Quelltext der Webesprache sein. Makroparameter können innerhalb des Makrorumpfes an beliebigen Stellen auftauchen. Sie werden bei einem Aufruf dieses Makros durch den beim Aufruf übergebenen Text ersetzt. Diese Makroparameter müssen das gleiche Aussehen wie in der Makrosignatur besitzen, also insbesondere mit einem "%" beginnen[2].

An Präprozessorbefehlen sind innerhalb des Makrorumpfes nur Makroaufrufe erlaubt. Es ist dabei allerdings zu beachten, daß die Makros sich nicht rekursiv wieder selbst aufrufen. In diesem Fall wird mit einem Fehler abgebrochen.

Durch den Präprozessorbefehl *#endmacro* wird schließlich die Makrodefinition beendet.

Um den durch ein Makro angegebenen Quelltext in die Ausgabe zu übernehmen muß dieses aufgerufen werden. Dies geschieht durch die Präprozessoranweisung *#call*.

Ein Makroaufruf hat immer die Form:

```
#call Makroname ( [Parameterliste] )
```

Der Makroname ist dabei der Name, welcher bei der Makrodefinition vergeben wurde. Falls mehrere Parameter vorhanden sind, so müssen diese mit Kommas getrennt werden. Bei Parametern unterscheidet man zwischen einfachen und komplexen Parametern. Einfache Parameter sind einzelne Wörter, komplexe enthalten darüberhinaus noch Sonderzeichen (wie zum Beispiel das Komma). Komplexe Parameter müssen in eckige Klammern eingeschlossen werden, einfache Parameter nicht. Diese eckigen Klammern zu Beginn und zum Ende des Parameters werden bei der Textersetzung innerhalb des Makrorumpfes nicht übernommen.

Ein Beispiel für eine Makrodefinition und einen Makroaufruf wäre also:

```
#macro inAllMethods(%classAlias,%methodAlias,%source)
   foreach class * <= %classAlias > do {
      foreach method * <= %methodAlias > do {
         %source
      }
   }
#endmacro
```

(Der Rumpf dieses Makros veranlaßt den Weber alle Methoden in allen ihm bekannten Klassen zu besuchen)

Ein dazugehöriger Makroaufruf könnte zum Beispiel so aussehen:

[2] Die Makroparameter sollten zur Zeit im Makrorumpf als eigene Zeichenkette erkennbar sein, d.h. davor und danach sollte ein Trennzeichen wie das Leerzeichen, ein Tabulator, ein Zeilenumbruch oder ein Wagenrücklauf stehen.

```
#call inAllMethods( c, m, [
        before ${ System.out.println("Entering method <m.name>
from class <c.name>"); }$
        ])
```

Der resultierende Weber-Quelltext wäre dann:

```
foreach class * <= c > do {
   foreach method * <= m > do {
      before ${ System.out.println("Entering method <m.name>
from class <c.name>"); }$
   }
}
```

Makros besitzen keinen Gültigkeitsbereich. Sie sind sichtbar, sobald sie definiert wurden. Sie werden nicht gelöscht, falls eine andere Quelltextdatei eingelesen wird. Dies bietet die Möglichkeit, Makrodefinitionen bzw. Makrobibliotheken am Anfang einzulesen.

3.6.2 Konstantendefinitionen

Mit Hilfe der Präprozessoranweisung *#define* können Konstanten definiert werden. Konstantennamen müssen wie Makronamen mit einem Buchstaben beginnen, gefolgt von einer beliebigen Kombination aus Buchstaben und Zahlen. Konstanten muß dabei nicht unbedingt ein Wert zugewiesen werden. In diesem Fall ist dem Präprozessor nur bekannt, daß diese Konstante definiert ist. Dadurch ist es zum Beispiel mit Hilfe der *#ifdef*-Anweisung möglich, verschiedene Quelltextalternativen auszuwählen. Ist einer Konstante allerdings ein Wert zugewiesen, so wird im Quelltext jedes Vorkommen des Konstantennamens durch dessen Wert ersetzt. Dieser Wert ist beliebig, muß allerdings in der gleichen Zeile enden. Die Konstante muß als solche zu erkennen sein, d.h. davor und danach muß ein Trennzeichen (Leerzeichen, Tabulator, Zeilenumbruch) stehen. Die Textersetzung erfolgt auch in Makrorümpfen.

Mit Hilfe der Anweisung *#undef* läßt sich eine Konstante wieder löschen. Der Gültigkeitsbereich von Konstanten ist analog zu dem der Makros. Konstanten sind sichtbar, sobald sie definiert wurden, auch über Dateigrenzen hinweg.

3.6.3 Ablaufsteuerung

Durch die Präprozessoranweisungen *#ifdef* - *#else* - *#endif* ist es möglich zwischen verschiedenen Alternativen des Weber-Quelltextes auszuwählen. Ist die angegebene Konstante definiert, so wird der Quelltext zwischen *#ifdef* und *#endif* bzw. zwischen *#ifdef* und *#else* betrachtet. Ist die angegebene Konstante nicht definiert, so wird ein eventueller *#else*-Block betrachtet, d.h. alles zwischen *#else* und *#endif*. Innerhalb dieser Blöcke sind weiterhin alle Präprozessoranweisungen erlaubt, geschachtelte *#ifdef* - *#else* - *#endif* Anweisungen sind also ebenso möglich wie Konstantendefinitionen, Makrodefinitionen und Makroaufrufe.

Kapitel 4

Der Weber

4.1 Einführung

Der Weber ist kein eigenständiges Werkzeug, welches selbstständig Quelldateien transformieren kann. Er ist als Erweiterung zu geeigneten Entwicklungswerkzeugen gedacht, dessen MOP er nutzen kann. Der Weber selbst ist nur eine Schaltzentrale, welche die weitgehend unabhängigen Module Präprozessor, Zerteiler und Ausführungseinheit steuert. Aus diesem Grund ist der Weber eine relativ kompakte Java-Klasse, deren Methoden die anfallenden Arbeiten nur an diese Module delegiert.

4.2 Struktur des Webers

Der Weber und seine Module lassen sich grob in 5 Teile gliedern:

- Kapselung des Metaobjektprotokolls
- Präprozessor
- Zerteiler (Parser)
- Ausführungseinheit
- Steuereinheit (der “Weber”)

Die für einen Webevorgang benötigten Grundfunktionalitäten sind in der Kapselung des Metaobjektprotokolls enthalten. Sie werden von der Ausführungseinheit benutzt um Navigation, Ablaufsteuerung und Programmtransformation zu bewerkstelligen. Da aus Sicht der Ausführungseinheit jedes Metaobjektprotokoll durch die Kapselung gleich ist, braucht diese nicht weiter an ein neues MOP angepaßt werden.

4.2.1 Kapselung des benutzten Metaobjektprotokolls

Wie bereits erwähnt wird eine eigene Sicht auf das verwendete MOP benutzt. Die Kapselung besteht aus einer Reihe von Interfaces, welche die von der Ausführungseinheit benötigten Grundfunktionalitäten bereitstellen. Für jeden Webepunkt gibt es daher ein Interface:

- *JPRoot* stellt die Wurzel dar, über welche sämtliche weiteren Webepunkte erreichbar sind.
- *JPClass* kapselt sowohl öffentliche als auch innere Klassen der Quelle.
- *JPMethod* ermöglicht den Zugriff auf Methoden innerhalb einer Klasse.

- *JPDeclaration* kapselt alle Variablendeklarationen. Dies können sowohl Klassenattribute als auch lokale Variablen innerhalb von Methoden sein.
- *JPAssignment* bietet Zugriff auf alle Zuweisungen an eine Variable, sowohl auf Klassenebene wie auch auf Methodenebene.
- *JPCall* stellt Zugriff auf Methodenaufrufe sicher.
- *JPExplicit* kapselt die expliziten Webepunkte.

Der gesamte Zugriff auf das verwendete Metaobjektprotokoll erfolgt über diese Interfaces. Daher besteht die Anpassung des Webers an ein anderes Metaobjektprotokoll im wesentlichen in der Implementierung dieser Interfaces. Die restlichen Module wie Präprozessor, Zerteiler, Ausführungs- und Steuereinheit brauchen nicht angepaßt werden.

4.2.2 Die Kontrolldatei

Als erste Aktion eines Webevorgangs liest die Steuereinheit eine Kontrolldatei ein, welche alle weiteren Informationen enthält. Eine Kontrolldatei hat folgende Struktur:

Optionen

Aspekt-Quelldateien

Namensraum

Reihenfolge der Aspektanwendung

Optionen

Es stehen folgende Optionen zur Verfügung: “PARSEONLY”, “NONAVIGATIONCHECK” und “CHECKONLY”. Die Optionen müssen jeweils direkt (d.h. ohne Leerzeichen etc.) in eckige Klammern eingeschlossen werden. Die Optionen sollten aus Übersichtlichkeitsgründen im Kopf der Kontrolldatei stehen, dürfen aber auch an jeder anderen Position untergebracht werden.

Die einzelnen Optionen bewirken folgendes Verhalten:

PARSEONLY	Der Webevorgang wird nach dem Einlesen und Zerteilen der Aspektdateien abgebrochen. Diese Option darf nicht zusammen mit den Optionen NONAVIGATIONCHECK und CHECKONLY benutzt werden. Sie ist dazu gedacht, die Syntax der Aspektdeklarationen durch den Zerteiler prüfen zu lassen.
NONAVIGATIONCHECK	Der Weber prüft vor der eigentlichen Programmtransformation die Navigation der einzelnen Aspekte und meldet Fehler, falls bestimmte Webepunkte nicht gefunden werden können. Falls nun allerdings durch einen Aspekt ein neuer Webepunkt angelegt wird, welcher später wieder referenziert wird, führt dies zu diesem Zeitpunkt zu einem Fehler. Deshalb läßt sich durch diese Option die oben gennante Überprüfung unterdrücken. Darf nicht zusammen mit PARSEONLY benutzt werden.
CHECKONLY	Der Weber prüft vor der Programmtransformation

sowohl die Navigation durch die Aspekte als auch die Aspektabhängigkeiten untereinander. Erst nach erfolgreicher Überprüfung beginnt die Programmtransformation. Durch setzten der Option CHECKONLY kann vor diesem letzten Schritt abgebrochen werden. Es werden dann nur noch die Navigation (falls NONAVIGATIONCHECK nicht gesetzt ist) und die Aspektabhängigkeiten geprüft, die Programmtransformation entfällt. Diese Option darf nicht zusammen mit der Option PARSEONLY benutzt werden

Aspekt-Quelldateien

Diese Sektion wird durch das Schlüsselwort "[FILES]" eingeleitet. Alle nun folgenden Einträge in der Kontrolldatei werden als Dateinamen für Aspekt-Quelldateien interpretiert. Die Dateien werden später in der angegebenen Reihenfolge eingelesen.

Namensraum

Dieser Abschnitt wird durch das Schlüsselwort "[CLASSES]" eingeleitet. Alle folgenden Namen werden als Klassennamen interpretiert. Platzhalter sind nicht erlaubt. Die Klassennamen müssen vollständig qualifiziert sein. Nur die hier aufgeführten Klassen sind dem Weber später bekannt. Es ist so also möglich den gesamten sichtbaren Namensraum auf bestimmte Klassen einzuschränken.

Es können auch gesamte Pakete in den Namensraum eingefügt werden. Dazu muß das Wort "package", gefolgt von einem Leerzeichen und dem Paketnamen anstatt eines Klassennamens in diese Sektion eingetragen werden. Alle Klassen im Wurzel-Paket können durch "package *" eingefügt werden. Sind hier angegebene Klassen nicht vorhanden, führt dies zu einem Fehler.

Reihenfolge der Aspektanwendung

Nach Angabe des Schlüsselwortes "[ORDER]" kann die Reihenfolge der Aspektanwendungen angegeben werden. Dem Schlüsselwort müssen die Namen der Aspekte folgen, und zwar genau in der Reihenfolge, in der sie angewendet werden sollen. Es dürfen keine Aspektinterfaces angegeben werden.

Kommentare

Alle Zeilen, deren erstes Zeichen ein ";" oder ein "#" ist werden als Kommentare behandelt und somit ignoriert.

Beispiel für eine Kontrolldatei

```
# Keine Prüfung der Navigation
[NONAVIGATIONCHECK]

# Ab hier stehen die Aspektquelldateien
[FILES]
z:\AspectSources\AspectInterfaces.aspect
z:\AspectSources\concreteAspect1.aspect
z:\AspectSources\concreteAspect2.aspect
```

```
# Hier wird der Namensraum definiert
[CLASSES]
package *
package MyPackage
AnotherPackage.AClass

# Reihenfolge der Aspektanwendung: concreteAspect2 zuerst,
# dann concreteAspect1
[ORDER]
concreteAspect2
concreteAspect1
```

4.2.3 Präprozessor

Der Präprozessor ist dafür verantwortlich, alle Makros und Konstanten innerhalb der Aspektquelldateien aufzulösen. Dazu muß er in den Quelldateien nach den in Kapitel 3.6 definierten Präprozessordirektiven suchen. Makro- und Konstantendefinitionen werden von ihm selbst verwaltet und bei einem Aufruf bzw. Konstantenbenutzung direkt in den Ausgabestrom aufgelöst. Deshalb sind auch keine Vorwärtsreferenzen erlaubt, d.h. alle Konstanten und Makros müssen bei ihrer Anwendung dem Präprozessor bekannt sein.

Der Zerteiler des Präprozessors wurde von dem Zerteiler-Generator JavaCC [JavaCC] generiert. JavaCC generiert, ausgehend von einer Syntaxbeschreibung, einen reinen Java-Zerteiler. Ein Großteil der Makro- und Konstantenverwaltung konnte direkt in dieser Syntaxbeschreibungsdatei als Java-Quelltext mit angegeben werden, da JavaCC diese Codefragmente in die entsprechenden Positionen übernimmt.

Der Ausgabestrom des Präprozessors dient dem Aspektzerteiler schließlich als Eingabe.

4.2.4 Zerteiler

Der Zerteiler hat die Aufgabe, den Ausgabestrom des Präprozessors zu zerteilen und letztendlich die abstrakten Syntaxbäume der Aspektdeklarationen aufzubauen. Hierzu werden die einzelnen Befehle durch entsprechende Java-Klassen repräsentiert, welche von einer der Klassen *Weave*, *Navigation* und *Branch* erben. Webebefehle sind dabei alles Unterklassen von *Weave*, Navigationsbefehle sind Unterklassen von *Navigation* und Verzweigungsbefehle sind Unterklassen von *Branch*. Dies hat für die Ausführungseinheit später den Vorteil, nur diese 3 Typen unterscheiden zu müssen (von den Befehlen *message* und *exit* abgesehen, welche extra behandelt werden).

Auch der Zerteiler wurde mit dem Zerteiler-Generator JavaCC erstellt. Die Java-Codefragmente zur Erstellung des abstrakten Syntaxbaums wurden dabei ebenfalls in die Syntaxbeschreibungsdatei integriert, so daß JavaCC sie gleich an die entsprechenden Stellen einfügen konnte. Eine nachträgliche Änderung der generierten Zerteiler-Klasse ist somit nicht mehr nötig, es kann sofort die generierte Klasse benutzt werden.

4.2.5 Ausführungseinheit

Die Ausführungseinheit hat die Aufgabe, die Anweisungsliste eines Aspektes abzuarbeiten. Dabei werden die Anweisungen dieser Liste sequentiell interpretiert. Für diese Aufgabe ist die Klasse *StatementListExecutor* zuständig. Für jede Anweisungsebene wird eine eigene Instanz dieser Klasse erzeugt. Eine neue Anweisungsebene ist zum Beispiel die Anweisungsliste einer *foreach*-Anweisung oder die Anweisungsblöcke einer Verzweigung.

Um eine Anweisungsliste auszuführen muß zunächst der Typ der aktuellen Anweisung ermittelt werden. Hier sind nur die Obertypen *Navigation*, *Branch* und *Weave* interessant. Ist der Typ der Anweisung ermittelt, so werden für die folgende Aktionen gestartet:

Navigation

Es wird ein neuer Besucher erzeugt, der selbstständig, ausgehend von der aktuellen Position im abstrakten Syntaxbaum des Java-Quellprogramms, die weiteren Knoten selbstständig aufsucht. Für jeden dieser Knoten wird eine neue Instanz von *StatementListExecutor* erzeugt und diesem den Anweisungsblock der *foreach*-Anweisung zum Ausführen übergeben.

Branch

Zuerst wird überprüft, ob die Bedingung erfüllt ist oder nicht. Dann wird eine neue Instanz der Klasse *StatementListExecutor* erzeugt, der entweder die Anweisungen des *if*-Blocks oder des *else*-Blocks zur Ausführung übergeben wird.

Weave

Der Webeanweisung wird die aktuelle Position innerhalb des Java-Syntaxbaums übergeben. Nun ist es der Webeanweisung möglich, ihre Aktion auf diesem Knoten des abstrakten Syntaxbaums (des Java-Quellprogramms) auszuführen.

Sonstige

Die beiden Anweisungen *exit* und *message* werden von *StatementListExecutor* selbstständig erkannt und ausgeführt.

4.2.6 Steuereinheit

Die Steuereinheit ist der Teil, welcher von außen als Weber wahrgenommen wird. Sie koordiniert die Zusammenarbeit der oben aufgeführten Module. Sie veranlaßt das Einlesen der Kontrolldatei, das Einlesen der Quelldateien durch den Präprozessor und die Weiterreichung des Ausgabestroms an den Zerteiler. Danach werden die Abhängigkeiten der Aspekte durch die Steuereinheit überprüft. Zuletzt werden die Aspekt-Syntaxbäume in der gewünschten Reihenfolge an die Ausführungseinheit übergeben.

4.3 Implementierung für Together SCI

TogetherJ 3.0 ist ein Werkzeug zum visuellen Modellieren von objektorientierten Systemen. TogetherJ arbeitet dabei direkt auf Quellcodeebene, d.h. es gibt keine Zwischenrepräsentation, aus welcher der endgültige Quelltext generiert wird. TogetherJ bietet darüberhinaus die Möglichkeit, eigene Skripte bzw. Java-Programme in seine Entwicklungsumgebung einzubinden. Dazu wird die sogenannte Open API benutzt. Es bietet Zugriff auf die wichtigsten internen Funktionen von TogetherJ. Ein Subsystem dieser Open API ist SCI (Abkürzung für *S*ource *C*ode *I*nterface). SCI ist ein Metaobjektprotokoll, welches Zugriff auf fast alle Elemente des Quelltextes bietet. Aus diesem Grunde kann der Weber gut auf diesem Metaobjektprotokoll aufsetzen.

4.3.1 Together SCI

Together SCI ist ein sehr leistungsfähiges Metaobjektprotokoll. Ausgehend von einer Wurzel, welches eine Instanz der Klasse *SciModel* ist, kann man Pakete und Klassen erreichen (repräsentiert durch *SciPackage* und *SciClass*). Von einer Klasse aus ist es möglich, zu ihren Attributen und Methoden zu navigieren. Da diese Navigation der Navigation des Webers entspricht, läßt sich diese auch einfach implementieren. Der Zugriff auf die Anweisungen der Methodenrümpfe ist ebenfalls möglich. Die Navigation innerhalb dieser Anweisungsblöcke, wie zum Beispiel das Auffinden von Ausdrücken in Anweisungen, geschieht in SCI hauptsächlich mit Hilfe des Entwurfsmusters Besucher (vgl. [Gamma et.al.]). SCI bietet außerdem die Möglichkeit, den statischen Typ von Ausdrücken und Objekten zu ermitteln.

SCI beschränkt sich aber nicht nur auf das Auffinden von Informationen über den repräsentierten Quelltext. Es bietet darüberhinaus die Möglichkeit, neue Knoten zu erstellen, diese in das bestehende Modell einzufügen bzw. bestehende Knoten zu ersetzen. Die neuen Knoten werden mit Hilfe des Entwurfsmusters *abstrakte Fabrik* erstellt. Für jeden relevanten Knotentyp existiert dort eine *create*-Methode, welche einen entsprechend neu erzeugten Knoten zurückgibt. Werden Knoten verändert, so spiegelt sich dies sofort im Quelltext wider.

Es wurde versucht, SCI sprachunabhängig zu gestalten, so daß es den Quellcode von verschiedenen objektorientierten Sprachen repräsentieren kann. In der vorliegenden Version wird allerdings nur Java vollständig unterstützt. Da der Weber allerdings nur Java unterstüzt, stellt dies keine Einschränkung dar.

4.3.2 Implementierungsdetails

Da der Weber kein eigenes Metaobjektprotokoll besitzt, sondern nur ein darunterliegendes MOP kapselt um eine einheitliche Schnittstelle zu erhalten, mußte zunächst eine entsprechende Abbildung der Webepunkt-Klassen auf die SCI-Klassen gefunden werden:

Webepunkt	SCI-Klasse(n)
JPRoot	SciModel
JPClass	SciClass
JPMethod	SciOperation
JPDeclaration	SciAttribute oder SciVariable
JPAssignment	SciAttribute oder SciAssignmentExpression
JPCall	SciFunctionCallExpression
JPExplicit	SciMember oder SciStatement

Die Webepunkte *JPDeclaration*, *JPAssignment* und *JPExplicit* werden dabei jeweils durch zwei SCI-Klassen repräsentiert, je nachdem ob die Webepunkte im Klassen- oder im Methodenkontext auftreten.

Inject/J ist in mehrere Pakete unterteilt, welche die verschiedenen Module enthalten. Die verschiedenen Pakete und ihre Funktion ist wie folgt:

Paket	Funktion
Joinpoints	Webepunkte, die das darunterliegende MOP kapseln.
Statements	Klassen, dessen Instanzen die Knoten im abstrakten Syntaxbaum der Aspektdeklarationen bilden. Teil der Ausführungseinheit.
Parser	Der generierte Zerteiler mit Hilfsklassen.
PreProcessor	Der Präprozessor mit Hilfsklassen und Makroverwaltung.
Visitors	Schnittstellen und Standardimplementierungen der verschiedenen *foreach*-Besucher. Teil der Ausführungseinheit.
SciImplementation	Die Implementierung der Klassen aus dem Paket *Joinpoints* für Together SCI.
Utils	Diverse Hilfsklassen, die vom System benötigt werden.
Exceptions	Alle Ausnahmen, die geworfen werden können.
Enumeration	Schnittstellen für die benötigten Aufzählungen.
Configuration	Graphische Bedienoberfläche zur Konfiguration von Inject/J

Neben den Klassen in den genannten Paketen existiert noch die Klasse *Weaver*, welche die Steuereinheit darstellt.

Im Paket *Utils* befinden sich unter anderem die wichtigen Klassen *Environment* und *IdentifierMatcher*. Die Klasse *Environment* stellt die Umgebung dar, in welcher der Weber läuft. In ihr werden zum Beispiel die Aspekt-Syntaxbäume, der Namensraum der bekannten Klassen und die Ausführungsreihenfolge verwaltet. Die Klasse *IdentifierMatcher* stellt statische Methoden zum Umgang mit Identifizierern und ggf. enthaltenen Platzhaltern bereit. Diese Methoden werden bei jedem Identifiziervergleich benutzt.

4.3.3 Explizite Webepunkte

Leider bietet SCI kaum Unterstützung von Kommentaren. Die einzige unterstützte Art sind sogennante *Tags*. Dies sind die Kommentare vor den Elementen Klasse, Methode und Attribut. Ein Tag beginnt immer mit dem "@"-Zeichen, gefolgt von dem Tag-Namen und einem optionalen Wert. TogetherJ benutzt diese Tags, um darin Dinge seines Modells zu speichern, die keine Entsprechung im Quelltext besitzen. Tags müssen immer direkt vor dem dazugehörigen Element stehen.

Da SCI keine weiteren Kommentare unterstützt, können diese auch nicht zur Identifikation expliziter Webepunkte benutzt werden (was von Vorteil gewesen wäre, da sie keinen Bytecode produzieren). Explizite Webepunkte werden in dieser Implementation durch folgende Methodenaufrufe identifiziert:

- JOINPOINT.BEGIN("*Webepunktname*");
  Für den Beginn eines mehrzeiligen Webepunktes innerhalb eines Methodenrumpfes.
- JOINPOINT.END("*Webepunktname*");
  Für das Ende eines mehrzeiligen Webepunktes innerhalb eines Methodenrumpfes.
- JOINPOINT.ID("*Webepunktname*");
  Für einen einzeiligen Webepunkt innerhalb eines Methodenrumpfes.

Damit solch ein Quelltext ohne Entfernen dieser Methodenaufrufe übersetzt werden kann, muß die abstrakte Klasse JOINPOINT im Standardpaket existieren. Diese Klasse muß die statischen, öffentlichen Methoden BEGIN, END und ID enthalten. Der Quelltext dieser Klasse wäre also:

```
public abstract class JOINPOINT {
   static public void BEGIN(String s) {}
   static public void END(String s) {}
```

```
    static public void ID(String s) {}

}
```

Ein Beispiel für einen mehrzeiligen Webepunkt in einem Methodenrumpf wäre:

```
...

JOINPOINT.BEGIN("expliziterWebepunkt1");

int i = 0;

int j = i * 5;

JOINPOINT.END("expliziterWebepunkt1");

...
```

Zu jedem JOINPOINT.BEGIN muß ein JOINPOINT.END folgen.

Explizite Webepunkte auf Klassenebene werden durch Tags identifiziert. Ein solcher hat die Form `@JOINPOINT Webepunktname`. Auf Klassenebene werden in dieser Implementation nur einzeilige Webepunkte unterstützt. Ein möglicher Webepunkt auf Klassenebene wäre zum Beispiel:

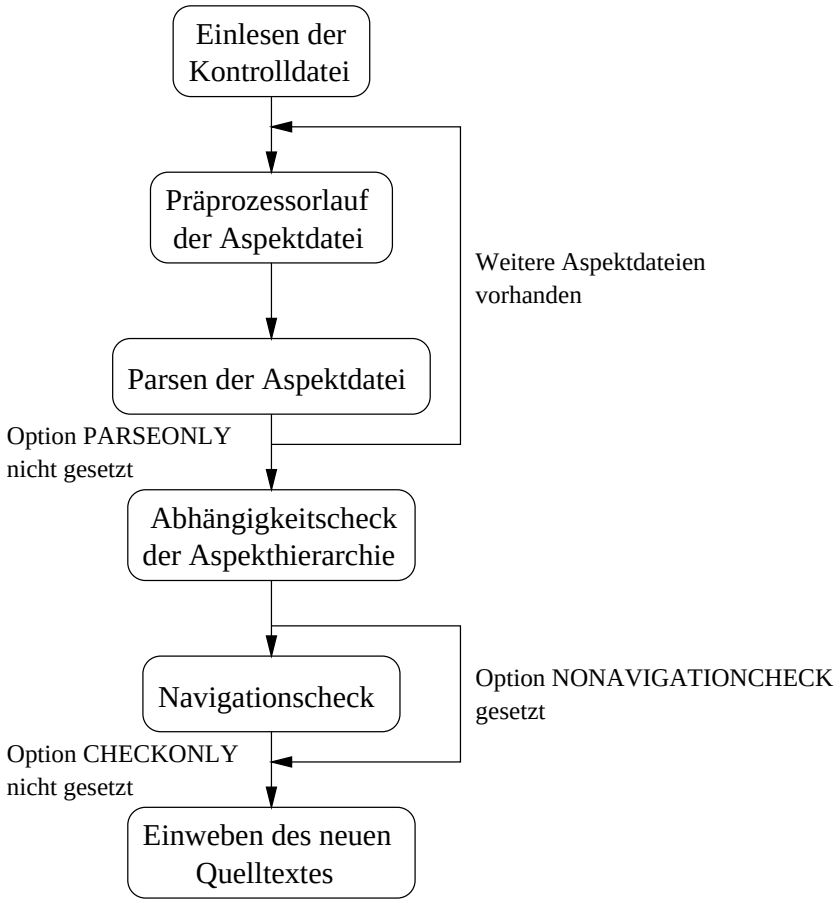
```
/** Ein Attribut

  * @JOINPOINT expliziterJoinpoint

  */

  int einAttribut;
```

In diesem Fall wäre die Deklaration von "einAttribut" ein expliziter Joinpoint.

4.4 Ablauf eines Webevorgangs

4.4.1 Prinzipieller Ablaufplan

Wie bereits in Abschnitt 4.2.6 angedeutet gliedert sich ein Webevorgang in mehrere Phasen. Die einzelnen Phasen werden von der Steuereinheit nacheinander angestoßen. Insgesamt werden sechs Phasen unterschieden:

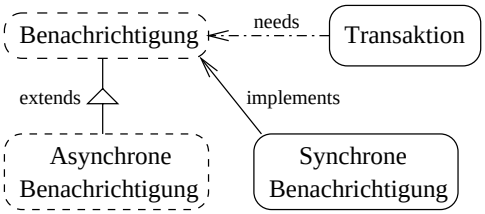


Die möglichen Optionen in der Konfigurationsdatei weisen die Steuerungseinheit an, nach einer bestimmten Phase abzubrechen bzw. Phasen zu überspringen.

Zuerst wird die Kontrolldatei eingelesen, welche alle Informationen zur weiteren Ablaufsteuerung enthält. So steht nach dem Einlesen der Namensraum der bekannten Java-Klassen fest, ebenso wie die einzulesenden Aspektdateien und die Anwendungsreihenfolge der Aspekte.

Anschließend wird jede angegebene Aspektdatei eingelesen und durch den Präprozessor vorverarbeitet. Die Reihenfolge ist dabei die gleiche, wie sie auch in der Kontrolldatei zu finden ist. Dies ist vor allem dann zu beachten, falls Makros und Definitionen über Dateigrenzen hinweg benutzt werden. Danach wird durch den Zerteiler ein abstrakter Syntaxbaum für die Aspektdeklarationen aufgebaut. Nach dem Einlesen und Zerteilen aller Quelldateien wird geprüft, ob die Option *PARSEONLY* in der Kontrolldatei angegeben wurde. Ist dies der Fall, so wird zu diesem Zeitpunkt abgebrochen. Dies ist dann sinnvoll, falls nur die Syntax der angegebenen Aspekte geprüft werden soll.

Die nächste Phase besteht im Prüfen der Aspektabhängigkeiten. Es muß sichergestellt sein, daß alle Aspekte, die in der *needs*-Liste einer Aspektdeklaration vorkommen, bekannt sind und das für sie auch eine Implementierung vorhanden ist. Ein Beispiel für einen solchen Abhängigkeitsgraph wäre:



Hierbei stellen Aspekte mit gestrichelten Rahmen Aspektinterfaces dar, wohingegen Aspekte mit durchgezogenen Rahmen Aspektimplementierungen repräsentieren. Angenommen, der Aspekt Transaktion soll bearbeitet werden. Dann müssen dem Weber mindestens die Aspekte *Transaktion*, *Benachrichtigung* und *SynchroneBenachrichtigung* bekannt sein. *Transaktion* benötigt den Aspekt *Benachrichtigung*. Da Benachrichtigung aber ein Interface ist muß zumindest eine Implementierung dafür vorhanden sein. Dies kann aber nur der Aspekt *SynchroneBenachrichtigung* leisten, da der Aspekt *AsynchroneBenachrichtigung* wiederum nur ein Interface ist, für welches noch keine Implementierung vorhanden ist. Der Aspekt *Transaktion* benötigt zwar eine Implementierung des Aspekts *Benachrichtigung*, doch diese Implementierung wird nicht automatisch angewendet. Der Implementierung muß in der Aspekt-Anwendungsreihenfolge in der Kontrolldatei explizit aufgeführt werden. Zuletzt wird in dieser Phase überprüft, ob auch alle Klassen existieren, die in der *CLASSES*-Sektion der Kontrolldatei angegeben wurden.

Als nächste Phase wird ein Navigationscheck durchgeführt, vorausgesetzt die Option *NONAVIGATIONCHECK* ist nicht gesetzt. Es wird versucht, alle in der Aspektdeklaration angegebenen Webepunkte zu finden. Schlägt dies fehl, so wird mit einem Fehler abgebrochen. Dies soll sicherstellen, das in der nächsten Phase, dem Einweben des neuen Quelltextes, kein Navigationsfehler mehr auftritt und somit nur ein Bruchteil transformiert wurde. Durch die Option *NONAVIGATIONCHECK* kann diese Überprüfung allerdings übersprungen werden. Dies ist zum Beispiel dann notwendig, falls durch den Aspekt ein neues Element erzeugt wird, auf welches später Bezug genommen wird. Zum Zeitpunkt des Navigationschecks ist dieses neue Element noch nicht sichtbar und würde beim anschließenden Referenzieren zu einem Navigationsfehler führen.

Die letzte Phase ist schließlich das Einweben des neuen Quelltextes. Dies geschieht nur, falls die Option *CHECKONLY* nicht gesetzt wurde.

4.4.2 Ablauf anhand eines Beispiels

Die Arbeitsweise des Webers soll nun anhand eines Beispiels demonstriert werden.

Stackchecker

Bei diesem Aspekt soll sichergestellt werden, daß die Vorbedingungen bei einer Operation auf einem Keller eingehalten werden. Sind die Vorbedingungen nicht erfüllt, so soll das Programm mit einem Fehler abbrechen. Kellerobjekte müssen alle den Obertyp *java.util.Stack* besitzen. In diesen Klassen müssen die Operationen *push()*, *pop()*, *peek()* und *empty()* definiert sein. Die interessanten Vorbedingungen sind dabei für die beiden Operationen *peek()* und *pop()* einzuhalten. Es muß sichergestellt sein, daß der Keller bei Anwendung diese Methoden nicht leer ist, d.h. die Operation *empty()* darf nicht *true* zurückliefern. Weitere Vorbedingungen wären zum Beispiel, daß die Anwendung der Operation *pop()* nicht zu einem Kellerüberlauf führt. Dieses Beispiel betrachtet aber nur die Vorbedingungen für die Methoden *peek()* und *pop()*.

Die Aspektdeklaration könnte zum Beispiel so aussehen:

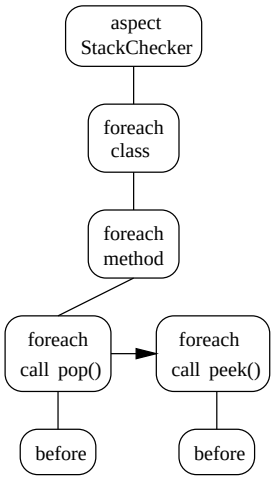
```
aspect StackChecker {
   foreach class * do {
```

```
        foreach method * do {
           foreach call pop() to subclass java.util.Stack <=c> do {
              before ${
                  if (<c.objectName>.empty()) {
                     System.err.println("Can't pop from stack.");
                     System.exit(1);
                  }
              }$
           }
           foreach call peek() to subclass java.util.Stack <=c> do {
              before ${
                 if (<c.objectName>.empty()) {
                    System.err.println("Can't peek stack.");
                    System.exit(1);
                 }
              }$
           }
        }
     }
  }
```

Eine mögliche Kongigurationsdatei hat folgendes Aussehen:

```
# Die Aspektquelldatei, in welcher der Aspekt StackChecker deklariert
ist.
[FILES]
z:\Stackchecker.ij

# Die Klassen/Pakete, die den Namensraum des Webers bilden
TestClass1
MyPackage.TestClass2
# Die Reihenfolge, in der die Aspekte angewendet werden
StackChecker
```

Zuerst wird diese Konfigurationsdatei eingelesen. Dann wird die Aspekt-Quelldatei *Stack-checker.ij* vom Präprozessor eingelesen. Da in diesem Fall keine Präprozessor-Direktiven benutzt worden sind erreicht diese Quelldatei den Zerteiler unverändert. Dieser liefert dann einen Syntaxbaum mit folgender Struktur:



Der nun folgende Abhängigkeitscheck ist in diesem Fall überflüssig, da von *StackChecker* keine Abhängigkeiten ausgehen. Die beiden Klassen *TestClass1* sowie *MyPackage.TestClass2* müssen existieren. In diesem Beispiel sollen die Java-Quelldateien folgendes Aussehen besitzen:

Datei TestClass1.java:

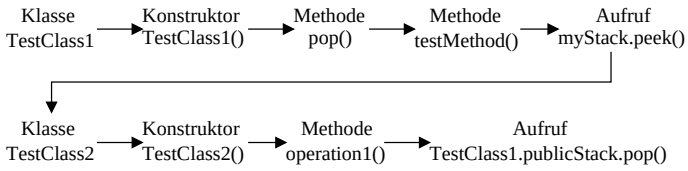
```
import java.util.Stack;
class TestClass1 {
   private Stack myStack = new Stack();
   static public Stack publicStack = new Stack();
   public TestClass1() {}
   public void pop() {}
   public void testMethod() {
      publicStack.push(new Object());
      pop();
      Object top = myStack.peek();
   }
}
```

Datei TestClass2.java:

```
package MyPackage;
import java.util.Stack;
class TestClass2 {
   public TestClass2() {}
   public operation1() {
      TestClass1.publicStack.pop();
   }
}
```

Die beiden Klassen verfolgen keinen tieferen Sinn, sie dienen nur der Demonstration.

Nun wird der Navigationscheck durchgeführt. Dabei besuchen die einzelnen Besucher die folgende Knotensequenz:



Da alle Knoten gefunden werden, liefert der Navigationscheck ein positives Ergebnis. Als letztes werden die neuen Quelltextzeilen eingefügt. Die transformierten Klassen haben dann folgendes Aussehen:

Datei TestClass1.java:

```
import java.util.Stack;
class TestClass1 {
   private Stack myStack = new Stack();
   static public Stack publicStack = new Stack();
   public TestClass1() {}
   public void pop() {}
   public void testMethod() {
      publicStack.push(new Object());
      pop();
      if (myStack.empty()) {
         System.err.println("Can't peek from stack.");
         System.exit(1);
      }
      Object top = myStack.peek();
   }
}
```

Datei TestClass2.java:

```
package MyPackage;
import java.util.Stack;
class TestClass2 {
   public TestClass2() {}
   public operation1() {
      if (TestClass1.publicStack.empty()) {
         System.err.println("Can't pop from stack.");
         System.exit(1);
      }
      TestClass1.publicStack.pop();
   }
}
```

Zu beachten ist hierbei, daß sich der Aufruf von *pop()* in der Methode *testMethod* nicht als Webepunkt qualifiziert. Der Aufruf erfolgt auf dem Objekt *this*, und dies hat nicht den Obertyp *java.util.Stack*. Wie in Methode *operation1* zu sehen ist, werden auch statische Objekte berücksichtigt.

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

Mit Inject/J wurde ein Werkzeug vorgestellt, mit dem auf einfache Weise Transformationen auf einem gegebenen Java-Quelltext durchgeführt werden können. Dazu wurde eine Sprache entwickelt, welche die durchzuführenden Transformationen beschreibt. Diese Beschreibung besteht einerseits aus Navigationsanweisungen, welche die gewünschten Webepunkte eindeutig identifizieren, andererseits aus Webebefehlen, welche an den gefundenen Webepunkten die Transformationen ausführen. Darüberhinaus existieren noch Anweisungen zur Ablaufsteuerung. Der Weber interpretiert diese Skriptsprache und führt die Anweisungen mit Hilfe eines Metaobjektprotokolls aus. Das Metaobjektprotokoll stellt dabei die Schnittstelle zum Quelltext dar.

Inject/J läßt sich zum Einweben von einfachen Aspekten nutzen, allerdings sind auch andere skriptgesteuerte Programmtransformationen möglich. Im Rahmen dieser Studienarbeit wurde das Werkzeug für die Entwicklungsumgebung TogetherJ implementiert. Inject/J ist nicht selbstständig lauffähig, da es das Metaobjektprotokoll SCI von TogetherJ nutzt.

5.2 Implementierung für andere MOPs

Together SCI stellt natürlich nicht das einzige mögliche Metaobjektprotokoll dar. Neben SCI ist auch eine Implementierung für andere Metaobjektprotokolle bzw. eine Integration in andere Entwicklungsumgebungen möglich.

Ein möglicher Kandidat ist das MOP der Java-Entwicklungsumgebung Borland JBuilder zu nennen. Dieses MOP mit Namen JOT (*J*ava *O*bject *T*ool) hat ähnliche Eigenschaften wie SCI, nur geschieht die Navigation dort nicht mit Hilfe von Besuchern. Die entsprechenden Webepunkte müssen also “von Hand” zusammengesucht werden. Eine Implementierung für JOT hätte den Vorteil, daß eine ausgereifte und weitverbreitete Entwicklungsumgebung zur Verfügung stände.

Ein weiteres Metaobjektprotokoll ist COMPOST, welches am Institut für Programmtechnik und Datenstrukturen (IPD) an der Universität Karlsruhe entwickelt wird. COMPOST steht dabei für COMPOsition SysTem. Das System kann Quellcode einlesen, daraus einen Syntaxbaum aufbauen und auf diesem Operationen ausführen. Die Navigation geschieht innerhalb von COMPOST, ebenso wie in Together SCI, mit dem Besucher-Konzept und könnte somit gut in den aktuellen Weber integriert werden.

5.3 Erweiterungen

Der im Rahmen dieser Studienarbeit entwickelte Weber stellt naturgemäß nur einen Prototyp dar. Es ist noch viel Raum für Erweiterungen und Weiterentwicklungen offen. Einige dieser Erweiterungen und Weiterentwicklungen wären:

- Anpassung der Aspekte
  Einmal fertiggestellte Aspektdeklarationen lassen sich zur Zeit nur sehr unzureichend anpassen, ohne den Quelltext verändern zu müssen. Die einzige Möglichkeit stellt im Moment der Präprozessor dar, doch sind viele denkbare Anpassungen damit nicht durchführbar, wie zum Beispiel benutzergesteuerte Verzweigungen zur Laufzeit. Eine Erweiterung wäre, Befehle zur interaktiven Anpassung der Aspekte bereitzustellen. Damit könnten Entscheidungen vom Benutzer zur Laufzeit entgegengenommen und entsprechend verarbeitet werden.

- Aspekte als kompilierte Klassen
  Um Aspekte einzuweben ist momentan eine Installation des Webers nötig. Um diesen zu bedienen und zu konfigurieren ist ein Mindestmaß an Verständnis über dessen Arbeitsweise nötig. Für einige Anwendungsfälle ist dies nicht erwünscht. Um diese direkte Konfrontation des Benutzers mit dem Weber zu umgehen wäre es möglich, Aspekte in Operatoren zu fassen. Statt der Interpretation der Aspekte generiert der Weberr Klassen, welche die gleiche Aufgabe erledigen. Diese Klassen könnten fertig konfiguriert und kompiliert an den Anwender übergeben werden. Er bekommt dann fertige Operatoren, welche als “Wizards” in die entsprechenden Entwicklungsumgebungen eingebunden werden können.

- Listen und globale Aliase
  Der aktuelle Stand des Webers bietet kaum eine Möglichkeit, in Abhängigkeit von bisherigen Operationen neue Aktionen zu starten. Dies liegt daran, daß der Weber zustandslos ist, d.h. er kann sich nicht an frühere Aktionen/Ergebnisse “erinnern”. Es hat sich herausgestellt, daß für manche denkbaren Anwendungsgebiete des Webers allerdings die Speicherung von Zuständen erforderlich ist. Aus diesem Grund wäre eine denkbare Erweiterung die Einführung von globalen Aliasen. Deren Gültigkeit wäre dann nicht auf den Anweisungsblock einer Navigationsanweisung beschränkt. Die Referenz auf einen Knoten stände auch noch zu einem späteren Zeitpunkt zur Verfügung. In diesem Zusammenhang könnten auch Listen eingeführt werden. Mit ihnen wäre es möglich, z.B. durch die *foreach*-Schleife Knoten zu sammeln und diese in einer späteren Operation wieder zu referenzieren.

- Exceptions als eigene Webepunkte
  Exceptions können als eigens Konzept in die Webesprache aufgenommen werden. So stellen dann das Werfen bzw. Auffangen von Exceptions eigene Webepunkte dar.

Einige dieser Erweiterungen, wie das interaktive Anpassen der Aspekte und das Listen-Konzept, werden in die nächste Version von Inject/J einfließen, welche im Anschluß an diese Studienarbeit entwickelt wird. Inject/J wird in diesem Zuge auch auf das Metaobjektprotokoll COMPOST portiert.

Anhang A

Installation und Eigenschaftsliste

A.1 Installation

Inject/J liegt als gepackte ZIP-Datei vor. Die Installation ist sehr einfach. Der Inhalt des Archivs muß nur, unter Beibehaltung der Pfadnamen, in das TogetherJ Installationsverzeichnis entpackt werden. Beim nächsten Start von TogetherJ befindet sich dann unter dem Menüpunkt *Tools* der Eintrag *Inject/J Configuration*, welches die graphische Bedienoberfläche zur Konfiguration des Webers aktiviert.

Das Archiv mit den kompilierten Java-Klassen bzw. deren Quellen können bei Bedarf per E-Mail angefragt werden unter: *kuttruff@fzi.de* oder *genssler@fzi.de.*

A.2 Eigenschaftsliste der Webepunkte

Die einzelnen Webepunkte haben bestimmte Eigenschaften. Diese sind über Aliase ansprechbar. Mehr dazu ist in Abschnitt 3.5.2 nachzulesen.

Auf dem Typ String ist eine Operation *equals* definiert. Damit kann auf Gleichheit der Zeichenkette geprüft werden. Beispiel:

```
<c.name.equals("TestClass")>
```

Dieser Ausdrück würde den boolschen Wert "true" zurückliefern, falls *c* ein Alias auf eine Klasse mit dem Namen "TestClass" ist.

Webepunkt	Eigenschaft	Typ	Beschreibung
Class	lineBegin	String	Startzeile der Klassendeklaration
	lineEnd	String	Endezeile der Klassendeklaration
	name	String	Qualifizierter Klassenname
	package	String	Name des Pakets, in welchem sich Klasse befindet
	interface	Bool	Gibt an, ob Klasse Interface ist
	public	Bool	Gibt an, ob Klasse öffentlich ist
	abstract	Bool	Gibt an, ob Klasse abstrakt ist
	final	Bool	Gibt an, ob Klasse final ist
	innerClass	Bool	Gibt an, ob Klasse eine innere Klasse ist
	file	String	Name der Datei, in welcher Klasse steht
	extends(pos)	String	Eintrag an Stelle *pos* der extends-Liste
	extends(name)	Bool	Gibt an, ob Klasse die angegebene Klasse erweitert. *name* muß qualifizierter Klassenname oder Alias sein
	implements(pos)	String	Eintrag an Stelle *pos* in implements-Liste

	implements(name)	Bool	Gibt an, ob Klasse die angegebene Klasse implementiert. *name* muß qualifizierter Klassenname oder Alias sein
	text	String	Quelltext der Klasse
Method	lineBegin	String	Startzeile der Methodendeklaration
	lineEnd	String	Endezeile der Methodendeklaration
	name	String	Name der Methode
	inClass	Class	Gibt Klasse zurück, die diese Methode enthält
	returnType	String	Typ des Rückgabewerts, “void” falls keiner
	public	Bool	Gibt an, ob Methode öffentlich ist
	protected	Bool	Gibt an, ob Methode geschützt ist
	private	Bool	Gibt an, ob Methode privat ist
	abstract	Bool	Gibt an, ob Methode abstrakt ist
	static	Bool	Gibt an, ob Methode statisch ist
	synchronized	Bool	Gibt an, ob Methode synchronized ist
	final	Bool	Gibt an, ob Methode final ist
	native	Bool	Gibt an, ob Methode native ist
	parameter(pos)	Param	Gibt Parameter an Position *pos* zurück
	text	String	Quelltext der Methode
Declaration	lineBegin	String	Startzeile der Attribut-/Variablendeklaration
	lineEnd	String	Endezeile der Attribut-/Variablendeklaration
	attribute	Bool	Gibt an, ob es sich um Attribut handelt
	variable	Bool	Gibt an, ob es sich um lokale Variable handelt
	inClass	Class	Falls Attribut: Klasse, die Attribut enthält, sonst leere Zeichenkette
	inMethod	Method	Falls lokale Variable: Methode, in der Variable deklariert wurde, sonst leere Zeichenkette
	name	String	Name Attribut/Variable
	type	String	Typ des Attributs / der Variable
	public	Bool	Falls Attribut: Gibt an, ob Attribut öffentlich Falls Variable: leere Zeichenkette
	protected	Bool	Falls Attribut: Gibt an, ob Attribut geschützt Falls Variable: leere Zeichenkette
	private	Bool	Falls Attribut: Gibt an, ob Attribut privat Falls Variable: leere Zeichenkette
	static	Bool	Falls Attribut: Gibt an, ob Attribut statisch Falls Variable: leere Zeichenkette
	final	Bool	Falls Attribut: Gibt an, ob Attribut final Falls Variable: leere Zeichenkette
	transient	Bool	Falls Attribut: Gibt an, ob Attribut transient Falls Variable: leere Zeichenkette
	volatile	Bool	Falls Attribut: Gibt an, ob Attribut volatile Falls Variable: leere Zeichenkette
	text	String	Deklarationstext Attribut/Variable
Assignment	lineBegin	String	Startzeile der Zuweisung
	lineEnd	String	Endezeile der Zuweisung
	leftExpression	String	linker Teil der Zuweisung
	rightExpression	String	rechter Teil der Zuweisung
	text	String	Quelltext der Zuweisung
Call	lineBegin	String	Startzeile des Methodenaufrufs
	lineEnd	String	Endezeile des Methodenaufrufs
	inMethod	Method	Gibt Methode zurück, in der dieser Aufruf steht
	name	String	Name der aufgerufenen Methode

	signature	String	Signatur der aufgerufenen Methode
	toClass	String	Klasse, zu der dieser Aufruf geht. Dabei wird der deklarierte Typ betrachtet.
	objectName	String	Name des Objekts, auf der Methode aufgerufen wird
	inAssignment	Bool	“true”, falls Aufruf innerhalb von Zuweisung
	inExpression	Bool	“true”, falls Aufruf innerhalb eines Ausdrucks
	parameter(pos)	Param	Aufrufparameter an Stelle *pos*
	text	String	Quelltext des Methodenaufrufs
Param	type	String	Typ des Parameters
	name	String	Name des Parameters
	hasType(type)	String	Gibt an, ob Parameter den angegebenen Typ hat.

Literaturverzeichnis

[KLM97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J.Irwin. Aspect-oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP*), LNCS 1241. Springer-Verlag, 1997.

[GOOS] Prof. Dr. Gerhard Goos: Vorlesungen über Informatik, Band 1: Grundlagen und funktionales Programmieren, Springer-Lehrbuch, 1995, ISBN 3-540-57281-3

[Gamma et.al.] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, 1996. ISBN 3-89319-950-0

[KRB] Gregor Kiczales, Jim des Rivières, Daniel G.Bobrow: The art of the meta-object protocol, MIT Press, ISBN 0-262-61074-4

[JavaCC] Homepage von Metamata: www.metamata.com (JavaCC steht dort zum kostenlosen Herunterladen bereit)

[TogetherJ] Homepage von TogetherJ 3.0: www.togetherj.com (Herunterladen der kostenlosen Whiteboard-Edition möglich, Dokumentation zu SCI ist darin enthalten)