Situationsbasierte Aktivierung von Aspekten in ObjectTeams/Java

Diplomarbeit

von

Sven Kampfhenkel

Matrikelnr. 193323

Berlin, 24.07.2008

Technische Universität Berlin

Fakultät IV - Elektrotechnik und Informatik

Institut für Softwaretechnik und Theoretische Informatik

Fachgebiet Softwaretechnik

Inhaltsverzeichnis

Inhalt

Abbildungsverzeichnis

Listingverzeichnis

1 Einleitung

Wiederverwendung in der objektorientierten Softwareentwicklung beschränkt sich häufig nur auf die wiederholte Verwendung von Klassen. Doch auch diese wiederholte Verwendung macht die Wartung einer komplexen Applikation nicht leichter. Des Weiteren werden dadurch viele Klassen an den verschiedensten Stellen im Code einer Applikation genutzt. Dies führt jedoch dazu, dass die Kapselung der funktionalen Anforderungen durchbrochen wird.

Um die Trennung der funktionalen Anforderung zu erreichen, wurden der objektorientierten Programmierung Konzepte der Aspektorientierung aufgesetzt. Einzelne Aspekte einer Applikation finden sich jetzt an einer Stelle und nicht im ganzen Code verteilt. Dies führt auch zu einer höheren Modularisierung der Applikation. Es können jetzt nicht nur Klassen als Modul eingesetzt werden, sondern ganze Aspekte.

Durch das Aufsetzen der neuen Konzepte bleibt die Kompatibilität mit der ursprünglichen Programmiersprache vorhanden. Bestehende Applikationen, Bibliotheken und Klassen können weiterhin verwendet werden. Die Aspekte werden an Stellen des Basiscodes eingewoben, die durch den Aspektcode bestimmt werden. Man nennt diese Stellen Joinpoints.

In ObjectTeams/Java werden Aspekte in Form von Klassen gekapselt. Diese Klassen werden Teams genannt. Objekte aus der Basisapplikation werden in ein Team als Rolle eingebunden. Rollenmethoden werden dabei an Methoden oder Attribute des Basisobjektes gebunden. Man unterscheidet zwischen Callin-Bindung und Callout-Bindung. Bei Callin-Bindungen wird der Aufruf der Basismethode abgefangen und die Rollenmethode aufgerufen. Callout-Bindungen delegieren den Aufruf der Rollenmethode weiter an die Basismethode.

1.1 Motivation und Ziele

Abhängig vom Kontext der Applikation kann ein Team aktiviert oder deaktiviert werden. Erst wenn ein Team aktiviert ist, werden die Callin- und Callout-Bindungen wirksam. Bei deaktivierten Teams verhält sich das Basisobjekt unverändert und die Steuerung wird nicht an das Team weiter gegeben.

Bei einem aktivierten Team werden Aufrufe einer Basismethode, für die eine Callin-Bindung existiert, an die Rollenmethode weitergeleitet. Durch so genannte Guards kann entschieden werden, ob der zusätzliche Aspekt ausgeführt werden soll. Diese Ausführungsprüfung ist in der Regel jedoch unabhängig vom aktuellen Ausführungszustand der Applikation. Das bedeutet, dass bei den Guards nicht darauf geachtet wird welche Aktionen die Applikation bereits ausgeführt hat. Die Ausführung der Rollenmethode ist jedoch nicht in allen Situationen, also nicht in allen Ausführungszuständen, sinnvoll oder sogar erst erwünscht, wenn die Applikation einen definierten Punkt in ihrer Abarbeitung erreicht hat.

Beispielhaft sei hier eine Basismethode genannt, die sich per Rekursion selbst wieder aufruft. Bei der ersten Ausführung der Methode mag es noch sinnvoll sein die Rollenmethode zusätzlich auszuführen. Aber schon beim zweiten Aufruf der Basismethode macht die Ausführung der Rollenmethode meist wenig Sinn. Abhängig vom Kontrollfluss könnte man hier jedoch entscheiden, ob die Rollenmethode ausgeführt werden soll oder nicht.

In dem genannten Beispiel ist die Ausführung der Rollenmethode also abhängig davon, ob die Basismethode als oberstes Element auf dem Callstack der virtuellen Maschine liegt. Dies zeigt auch sehr gut eine einfache Art den Zustand einer Applikation zu bestimmen, nämlich zu prüfen welche Methoden sich gerade auf dem Callstack befinden. Hierbei wird aber noch nicht in Betracht gezogen, dass die zu beobachtende Methode zwar ausgeführt wurde, sich jedoch nicht mehr auf dem Callstack befindet. Soll diese Situation zusätzlich betrachtet werden, muss ein Zustandsautomat eingeführt werden, der den Zustand der Applikation überwacht. Dabei könnte es auch interessant sein zu beobachten in welchem Thread die überwachte Methode ausgeführt wurde.

Um zu entscheiden ob die Rollenmethode zusätzlich zur Basismethode ausgeführt werden soll, muss ein Mechanismus implementiert werden der prüft ob die Ausführung im aktuellen Zustand der Basisapplikation sinnvoll ist. Dazu werden zusätzliche Informationen über die Rollenmethode benötigt. Es muss mitgeteilt werden unter welchen Voraussetzungen die Ausführung der Rollenmethode sinnvoll ist. Diese

Metainformation kann in Form von Annotationen an die Rollenmethode oder besser an die Callin-Bindung notiert werden. In dem konkreten Beispiel muss durch die Annotation also mitgeteilt werden welche Methoden der Basisapplikation sich auf dem Callstack befinden sollen oder welche eben nicht enthalten sein sollen.

Bei Verwendung eines Zustandsautomaten muss dieser beim Betreten oder Verlassen der beobachteten Methode aktualisiert werden. Dazu können spezielle Rollenmethoden an die beobachteten Methoden gebunden werden, die dem Zustandsautomaten mitteilen, dass die Basismethode ausgeführt wurde.

Da das zu entwickelnde System aus Standardelementen der Sprache Java bestehen soll, kann es einfach um weitere Auswertungsmöglichkeiten ergänzt werden. Des Weiteren bietet die Verwendung von Standardkomponenten die Möglichkeit der einfachen Integration in die bestehende Entwicklungsumgebung.

1.2 Gliederung

In Kapitel 2 folgt eine kurze Einführung in die Sprache ObjectTeams/Java. Dort werden die für diese Arbeit wichtigen Sprachkonzepte erläutert.

Kapitel 3 liefert eine Beschreibung der Annotationen aus dem Java5 Standard. Diese werden im praktischen Teil der Arbeit benötigt und fließen somit in die nachfolgenden Kapitel ein.

Die benötigten Ergänzungen an der ObjectTeams/Java Entwicklungsumgebung Object Teams Development Tooling werden in Kapitel 4 beschrieben. Diese Ergänzungen sind notwendig, damit die Annotationsverarbeitung, die in den folgenden Kapiteln verwendet wird, mit den Konzepten der Sprache ObjectTeams/Java benutzt werden kann.

In Kapitel 5 wird gezeigt, wie die Ausführung eines Aspektes beeinflusst werden kann. Diese Technik wird in Kapitel 6 benutzt um die Ausführung einer Callin-Bindung vom Kontrollfluss einer Applikation abhängig zu machen. Ergänzend hierzu wird in Kapitel 7 ein zusätzlicher Datenkanal für diese Steuerung beschrieben.

Das Kapitel 8 befasst sich mit der Erläuterung von Tracematches. Es wird beschrieben was Tracematches sind und wie diese in ObjectTeams/Java umgesetzt werden können.

Kapitel 9 beendet die Arbeit mit einem Fazit und einem Ausblick.

2 Aspektorientierte Programmierung mit ObjectTeams/Java

2.1 Was ist ein Aspekt?

Wie bereits in Kapitel 1 erwähnt, ist die aspektorientierte Sprache ObjectTeams/Java als Erweiterung zur bestehenden objektorientierten Sprache Java realisiert. Es liegt also nahe die erste, zentrale Frage der Objektorientierung, in leicht veränderter Form, für diese Betrachtung heran zu ziehen. So wird die Frage „Was ist ein Objekt?“ abgewandelt in „Was ist ein Aspekt?“

Als Aspekt wird eine Aufgabe eines Softwaresystems bezeichnet, die sich nicht als einzelnes Modul, also als Klasse in Java, implementieren lässt. Typischerweise sind dies nicht funktionale Anforderung an die Software, die sich häufig in weiten Teilen des Softwaresystems wiederfinden lassen. Die Aspekte schneiden also viele Teile des Softwaresystems und unterwandern damit das gewünschte Konzept der Kapselung von Modulen. Man nennt dieses Problem auch „Cross-Cutting-Concerns“. Oft genannte Beispiele von Aspekten sind Fehlerbehandlung und Logging, also die Protokollierung des Programmablaufs eines Softwaresystems.

Da die Cross-Cutting-Concerns andere Module eines Softwaresystems schneiden, lassen sich beide Seiten, also die grundlegenden Module und die zusätzlichen Anforderungen, nicht unabhängig von einander behandeln. Dies erschwert einerseits die Entwicklung beider Seiten aber auch die Wartbarkeit der einzelnen Module leidet darunter.

Die Aspektorientierte Programmierung ist ein Programmierparadigma das diesen Cross-Cutting-Concerns entgegenwirken soll. Dazu werden in der Aspektorientierten Programmierung die zusätzlichen Anforderung, die Aspekte, als Modul des Softwaresystems angesehen. Die Einbindung dieser Aspekt-Module in Module des objektorientierten Basissystems geschieht innerhalb des Aspekt-Moduls. Das Basis-Modul kann somit unabhängig vom Aspekt entwickelt und gewartet werden.

Um diese nachträgliche Bindung in der Aspektorientierten Programmierung zu realisieren, müssen die Punkte identifiziert werden, an die die Aspekte gebunden werden sollen. Die Punkte, die diese Stellen innerhalb des Basis-Moduls markieren, werden Joinpoints genannt. Da jedoch ein Aspekt nicht entwickelt wird um ein einzelnes Basis-Modul

zu ergänzen, werden die Joinpoints zu einer Menge von Punkten zusammengefasst die Pointcut heißt.

Wenn die Abarbeitung des Basisprogramms einen Joinpoint erreicht hat, wird der zusätzliche Aspektcode, Advice genannt, ausgeführt. Die Ausführung der Advices kann an drei definierten Stellen geschehen:

1. Vor der Ausführung des Basiscodes
2. Nach der Ausführung des Basiscodes
3. Um dies Ausführung des Basiscodes herum (Tatsächlich wird hier die Ausführung des Basiscodes durch den Advice ersetzt. Der Basiscode kann als Aufruf aus dem Advice heraus ausgeführt werden.)

2.2 Konzepte der Sprache ObjectTeams/Java

In den nächsten Abschnitten wird beschrieben, wie die oben genannten Konzepte der Aspektorientierten Programmierung in der Sprache ObjectTeams/Java umgesetzt werden.

2.2.1 Teams und Rollen

ObjectTeams/Java führt Teams als neue Art von Klassen ein. Ein Team wird wie eine normale Klasse in Java deklariert, erhält jedoch zusätzlich noch das Schlüsselwort **team**.

```
public team class DemoTeam {}
```

Listing 2.1: Teamdeklaration

Jede innere Klasse eines Teams ist definiert als eine Rolle. Ein Team dient somit als Container für Rollen. Damit die Rollenklassen immer einen Bezug zu ihrem Team haben, dürfen sie nicht statisch sein. Innerhalb eines Teams bildet eine Rolle die Schnittstelle zum zugrundeliegenden Modul der Basisanwendung und stellt so das Verhalten des Basisobjektes dar. Rollenklassen können gebunden oder ungebunden sein. Eine gebundene Rollenklasse ist mittels des Schlüsselwortes **playedBy** an eine Basisklasse gebunden.

```
public team class DemoTeam {
        public class BoundRole playedBy BaseClass {}
        public class UnboundRole {}
}
```

Listing 2.2: Rollendeklaration

Wie in Java üblich können auch Teamklassen von anderen Klassen erben. Hier gilt jedoch die Einschränkung, dass die Superklasse auch eine Teamklasse sein muss. Wie auch für die aus Java bekannte Vererbung wird für die Teamvererbung das Schlüsselwort **extends** benutzt. Die Implementierung von Interfaces ist für Teamklassen unverändert und kann wie aus Java bekannt benutzt werden.

```
public team class DemoTeam extends SuperTeam {}
```

Listing 2.3: Teamvererbung

Für Rollenklassen gelten bei der Teamvererbung spezielle Regeln. Einerseits sind sie, wie innere Klassen, in der Subklasse ebenfalls verfügbar. Andererseits können sie, ähnlich wie Methoden, in der Subklasse überschrieben werden. Das Überschreiben einer Rollenklasse resultiert in einer sogenannten impliziten Vererbung. Hierbei wird die Rollenklasse zwar überschrieben, erbt aber alle Funktionen der überschriebenen Rollenklasse. Werden innerhalb von den überschreibenden Rollenklassen auch Methoden überschrieben, so kann die überschriebene Methode mittels des Schlüsselwortes **tsuper** aufgerufen werden. Des Weiteren können ungebundene Rollenklassen durch diese Art der Vererbung nachträglich gebunden werden.

```
public team class DemoTeam {
        public class UnboundRole {
                private String aString = "a String";
                public void print() {
                        System.out.println(aString);
                }
        }
```

```
}
public team class SubDemoTeam extends DemoTeam {
        public class UnboundRole playedBy BaseClass {
                public void print() {
                        System.out.print("String is: ");
                        tsuper.print();
                }
        }
}
```

Listing 2.4: Implizite Vererbung

Normalerweise sind Rollen durch ihre umschließenden Teams vor Zugriffen von außen geschützt. Manchmal ist es jedoch nötig Rollen auch außerhalb ihrer Teams zu benutzen. Hierbei spricht man von externen Rollen. Nur Rollen die mit dem Schlüsselwort **public** markiert sind, die also überall sichtbar sind, können zu externen Rollen werden. Um eine externe Rolle zu benutzen, wird eine Referenz zu einer Instanz des umschließenden Teams benötigt. Die Variable die diese Referenz liefert muss mit dem Schlüsselwort **final** als unveränderbar markiert sein.

```
public team class DemoTeam {
        public class ExternalRole playedBy BaseClass {}
}

public team class AnotherTeam {
        private final DemoTeam demoTeam;
        public AnotherTeam(DemoTeam demoTeam) {
                this.demoTeam = demoTeam;
        }
        public class DemoRole playedBy ExternalRole<@demoTeam> {}
}
```

Listing 2.5: Externe Rolle

2.2.2 Methodenbindungen

Ist eine Rollenklasse an eine Basisklasse gebunden, so kann das Verhalten der Basisklasse mittels Methodenbindungen verändert werden. Dabei wird eine Methode der Rollenklasse, die Rollenmethode, an eine Methode der Basisklasse gebunden. Man unterscheidet hierbei zwischen Callin-Bindungen und Callout-Bindungen.

Durch eine Callin-Bindung wird der Aufruf der gebundenen Basismethode abgefangen und die Steuerung an die Rollenmethode übergeben. Nach der Ausführung der Rollenmethode wird die Steuerung an die Basismethode zurückgegeben.

Es existieren drei Punkte an denen Rollenmethoden ausgeführt werden können:

- Vor der Ausführung der Basismethode. Dies wird durch das Schlüsselwort **before** an der Callin-Bindung realisiert. Hierbei wird zuerst die Rollenmethode und danach die Basismethode ausgeführt.
- Nach der Ausführung der Basismethode. Hierfür wird das Schlüsselwort **after** benötigt. Es wird zuerst die Basismethode und danach die Rollenmethode ausgeführt.
- Anstatt der Ausführung der Basismethode. Wird das Schlüsselwort **replace** benutzt, wird nur die Rollenmethode ausgeführt. Die implizite Ausführung der Basismethode entfällt.

Um eine Callin-Bindung zu notieren, wird der Name der Rollenmethode gefolgt von ‚<-' aufgeschrieben. Des Weiteren wird noch die Art der Callin-Bindung anhand des Schlüsselwortes (**before**, **after**, **replace**) und der Name der Basismethode notiert. Sollte der Name der Basismethode nicht eindeutig sein, da sie überladen wurde, muss die vollständige Deklaration der Basismethode notiert werden. Hierbei muss aber auch die vollständige Deklaration der Rollenmethode benutzt werden.

Wird die Rollenmethode vor oder nach der Basismethode ausgeführt, kann die Signatur der Rollenmethode von der der Basismethode abweichen. Die Rollenmethode wird hierbei wie eine normale Javamethode notiert. Wird jedoch die Basismethode durch die Rollenmethode ersetzt (durch Benutzung des Schlüsselwortes **replace**), muss die Rollenmethode mit dem Schlüsselwort **callin** beginnen. Die Rollenmethode sollte dabei die gleiche Signatur wie die Basismethode haben. (Inwiefern die Signatur der Rollenmethode abweichen darf, ist beschrieben in [HHM07] §4.5(d)) Die ersetzte Basismethode kann durch

einen Basecall aufgerufen werden. Dazu wird das Schlüsselwort **base** gefolgt vom Aufruf der Basismethode benutzt. Der Name der Basismethode wird jedoch durch den Name der Rollenmethode ersetzt. Der Aufruf ist somit vergleichbar mit dem Aufruf einer Methode einer Superklasse durch das Schlüsselwort **super**.

```
public team class DemoTeam {

        public class DemoRole playedBy BaseClass {
                roleMethod <- after myBaseMethod;
                void roleMethod() <- before int baseMethod(String string);
                callinMethod <- replace someBaseMethod;

                private void roleMethod() {}
                callin String callinMethod(int value) {
                        return base.callinMethod(value);
                }
        }
}
```

Listing 2.6: Callin-Bindungen

Durch Callout-Bindungen werden Zugriffe aus der Rollenklasse auf Methoden der gebundenen Basisklasse realisiert. Dabei wird nicht nur der Aufruf der Rollenmethode zur gebundenen Basismethode weitergeleitet, sondern auch die Steuerung an die Basismethode übergeben. Somit würde die Auflösung von Selfcalls nicht in der Rollenklasse sondern in der Basisklasse beginnen. Da die Rollenmethode hier als Stellvertreter benötigt wird, muss sie abstrakt implementiert sein.

Eine Callout-Bindung wird notiert durch den Name der Rollenmethode gefolgt von einem ‚->' und dem Namen der Basismethode. Wie bei einer Callin-Bindung können die Parameter der Basismethode vernachlässigt werden wenn der Name der Basismethode eindeutig ist. Sollte der Name der Basismethode nicht eindeutig sein, muss die vollständige Signatur der Basismethode und der Rollenmethode notiert werden.

Bei einer Callout-Bindung können die Parameter der Rollenmethode auf die Parameter der Basismethode übersetzt werden. Es wird zwischen expliziter und impliziter Übersetzung unterschieden. Bei der expliziten Parameterübersetzung kann die Signatur der Rollenmethode von der Signatur der Basismethode abweichen. Bei der impliziten Parameterübersetzung müssen die Signaturen gleich sein. (Die explizite

Parameterübersetzung und die möglichen Signaturabweichungen der impliziten Parameterübersetzung sind beschrieben in [HHM07] §3.2)

Des Weiteren können Callout-Bindung dazu benutz werden den Wert von Variablen der Basisklasse auszulesen oder zu überschreiben. Die Rollenmethoden werden also als Getter- bzw. Settermethode in der Rollenklasse implementiert. Bei der Notation der Callout-Bindung wird der Name der Basismethode ersetzt durch den Namen der Variablen aus der Basisklasse. Vor dem Namen der Variablen muss der Typ der Callout-Bindung notiert werden. Hierfür werden das Schlüsselwort **get**, zu Lesen des Wertes, bzw. **set**, zum Setzen des Wertes, benutzt.

```
public team class DemoTeam {
        public class DemoRole playedBy BaseClass {
                calloutMethod -> baseMethod;
                getString -> get string;
                setString -> set string;

                abstract void calloutMethod();
                abstract String getString();
                abstract void setString(String string);
        }
}
```

Listing 2.7: Callout-Bindungen

2.2.3 Guards

Die Ausführung von Callin-Bindungen kann durch sogenannte Guards beeinflusst werden. Guards bestehen aus dem Schlüsselwort **when** gefolgt von einem boolschen Ausdruck. Wird dieser Ausdruck zu **false** ausgewertet, so wird die durch die Callin-Bindung gebundene Rollenmethode nicht ausgeführt. Guards können an vier verschiedenen Stellen auftreten:

- An einer Callin-Bindung: Der Guard ist nur für diese eine Callin-Bindung gültig.
- An einer Rollenmethode: Der Guard ist gültig für alle Callin-Bindungen die diese Rollenmethode an eine Basismethode binden.
- An einer Rollenklasse: Der Guard ist für alle Callin-Bindungen innerhalb dieser Rollenklasse gültig.

- An einem Team: Der Guard ist für alle Callin-Bindungen gültig die innerhalb der Rollenklassen dieses Teams existieren.

```
public team class DemoTeam
when (invokeCallin()) {
        private boolean invokeCallin() {…}

        public class RoleClass playedBy BaseClass
        when (invokeCallin()) {
                roleMethod <- after baseMethod
                when (invokeCallin());

                private void roleMethod()
                when (invokeCallin()) {…}
        }
}
```

Listing 2.8: Guards

Werden wie in Listing 2.1.3.1 mehrere Guards angegeben, so müssen alle zu **true** ausgewertet werden damit die Callin-Bindung aktiv wird.

2.2.4 Teamaktivierung

Damit die Methodenbindungen einer Rolle wirksam werden können, ist es nötig eine Instanz des umschließenden Teams der Rolle zu aktivieren. Diese Teamaktivierung kann für einen bestimmten Thread durchgeführt werden oder global für alle Threads. Des Weiteren wird zwischen expliziter und impliziter Teamaktivierung unterschieden.

Bei der expliziten Teamaktivierung wird eine Referenz auf das zu aktivierende Team benötigt. Diese Referenz kann entweder innerhalb eines Ausführungsblockes mittels des Schlüsselwortes **within** aktiviert werden oder durch die Methoden `activate()` und `deactivate()` direkt aktiviert bzw. deaktiviert werden. Die explizite Teamaktivierung kann innerhalb einer normalen Javaklasse, insbesondere also auch innerhalb eines Teams, erfolgen.

Wird das Team mittels **within** innerhalb eines Blockes aktiviert, so wird es nach der Ausführung des Blockes wieder in den vorherigen Zustand

überführt. War das Team bereits vor der Ausführung des Blockes aktiv, so bleibt es auch nach der Ausführung aktiv. War es vorher deaktiviert, so wird es nach der Ausführung wieder deaktiviert. Die Teamaktivierung innerhalb eines Blockes ist nur für den Thread gültig der den Block ausführt.

Die Methoden activate() und deactivate() aktivieren bzw. deaktivieren das Team nur für den Thread der die Methoden ausführt. Soll das Team für einen anderen Thread aktiviert oder deaktiviert werden, so muss dieser Thread als Parameter an die Methoden activate(Thread thread) bzw. deactivate(Thread thread) übergeben werden. Soll das Team für alle Threads aktiviert bzw. deaktiviert werden, so muss die vordefinierte Konstante Team.ALL_THREADS als Parameter an die Methoden übergeben werden.

```
public class DemoClass {
        public void startDemo() {
                DemoTeam demoTeam = new DemoTeam();
                demoTeam.activate();

                within (demoTeam) {…}

                demoTeam.deactivate();
        }
}
```

Listing 2.9: Explizite Teamaktivierung

Wird im Zuge der Programmausführung die Steuerung durch einen Methodenaufruf an ein Team oder an eine externe Rolle übergeben, so wird das Team bzw. das Team das die Rolle umschließt automatisch aktiviert. Man spricht hierbei von impliziter Teamaktivierung. Wird nach der Ausführung der Methode die Steuerung wieder zurückgegeben, so wird, wie bei der expliziten Teamaktivierung mittels **within**, das Team in den vorherigen Aktivierungszustand versetzt.

2.2.5 Entkapselung

Ein wichtiges Prinzip der Objektorientierten Programmierung ist die Kapselung von Daten. Dabei werden innere Klassen, Methoden und

Attribute, die in ihrer Sichtbarkeit eingeschränkt sind, vor dem Zugriff von außerhalb der Klasse geschützt. Die möglichen Sichtbarkeitsstufen dieser Elemente sind:

- Public: Das Element ist in der Sichtbarkeit nicht eingeschränkt. (Schlüsselwort **public**)
- Protected: Das Element ist für die eigene Klasse und Spezialisierungen der eigenen Klasse sichtbar. (Schlüsselwort **protected**)
- Private: Das Element ist nur innerhalb der eigenen Klasse sichtbar. (Schlüsselwort **private**)
- Package local: Das Element ist innerhalb des eigenen Paketes sichtbar. (kein Schlüsselwort angegeben)

Der Zugriff auf solche geschützten Elemente erfolgt über wohldefinierte Schnittstellen. Dadurch werden sowohl die Daten als auch die Implementierung vor der Außenwelt verborgen.

Bei der Aspektorientierten Programmierung reichen die durch die Schnittstelle zur Verfügung gestellten Möglichkeiten zur Veränderung von Daten einer Instanz einer Klasse meist nicht aus. Darum führt ObjectTeams/Java die Entkapselung oder auch Decapsulation ein. Durch die Entkapselung sind Methoden die in ihrer Sichtbarkeit eingeschränkt sind trotzdem nach außen sichtbar. Somit können Callin- und Callout-Bindungen auch an private Methoden einer Basisklasse gebunden werden.

2.2.6 Lowering und Lifting

Jede Instanz einer gebundenen Rollenklasse speichert eine Referenz auf eine Instanz ihrer Basisklasse. Diese Referenz kann während der gesamten Lebenszeit des Rollenobjektes nicht verändert werden. Das Auslesen des referenzierten Basisobjektes aus dem Rollenobjekt wird als Lowering bezeichnet. In den meisten Fällen geschieht das Lowering implizit um die Typsicherheit der Anwendung zu gewährleisten. Dazu fügt der Compiler an allen Stellen wo ein Objekt der Basisklasse erwartet wurde, jedoch ein Objekt der Rollenklasse übergeben wurde, automatisch eine Übersetzung mittels Lowering ein.

```
public team class DemoTeam {
        public class DemoRole playedBy BaseClass {

                private void print(BaseClass baseClass) {…}
```

```
            private DemoRole getRole() {…}

            public void start() {
                DemoRole demoRole = getRole();
                print(demoRole);
            }
        }
}
```

Listing 2.10: Implizites Lowering

Wenn der Compiler nicht in der Lage ist die Typisierung automatisch aufzulösen, ist es nötig das Lowering explizit anzugeben. Dieser Fall tritt ein, wenn der angegeben Typ Supertyp von Basisklasse und Rollenklasse ist. Damit das Lowering explizit angegeben werden kann, muss die Rollenklasse das Interface ILowerable implementieren. Der Compiler generiert dann automatisch die Methode **public** Object lower(), mit der das Lowering explizit durchgeführt werden kann.

```
public team class DemoTeam {
        public class DemoRole implements ILowerable playedBy Base {
            private DemoRole getRole() {…}

            public void run() {
                DemoRole demoRole = getRole();
                Object roleObject = demoRole;
                Object baseObject = demoRole.lower();
            }
        }
}
```

Listing 2.11: Explizites Lowering

Lifting ist der umgekehrte Prozess zum Lowering. Beim Lifting wird eine Instanz der Basisklasse in eine Instanz der Rollenklasse übersetzt. Im Gegensatz zum Lowering ist das Lifting nur bei Methodenbindungen implizit anwendbar. Explizites Lifting kann in Methodensignaturen innerhalb von Teams benutzt werden. Dazu wird das Schlüsselwort **as** gefolgt von der Rollenklasse als zweiter Parametertyp benutzt.

```
public team class DemoTeam {
        public class DemoRole playedBy BaseClass {…}

        public void doSomething(BaseClass as DemoRole param) {…}
}
```

Listing 2.12: Explizites Lifting

3 Automatische Codeerzeugung mit Java Annotationen

Seit Version 5 bietet die Programmiersprache Java die Möglichkeit einige Elemente der Sprache mit zusätzlichen beschreibenden Informationen zu versehen. Diese zusätzlichen Informationen, oft auch Metainformationen genannt, werden nicht nur in Java als Annotationen bezeichnet. Man unterscheidet zwischen Annotationstyp, der der Deklaration entspricht, und einer konkreten Annotation, die als Metainformation an einem Element der Sprache notiert ist. Annotationen werden wie Java-Doc Tags mit dem Zeichen @ eingeleitet und als Modifizierer für Elemente der Sprache benutzt. Java5 bietet bereits einige vorgefertigte Annotationstypen die in Java6 um weitere Typen ergänzt wurden. Im Gegensatz zu den Java-Doc Tags können jedoch eigene Annotationstypen implementiert werden. Solche selbst implementierten Annotationstypen können selbst wieder mit Annotationen markiert werden. Diese Annotationen werden dann Meta-Annotationen genannt.

3.1 Implementierung von Annotationen

Neue Annotationstypen werden wie Schnittstellen mit dem Schlüsselwort **interface** deklariert. Im Unterschied zu Schnittstellen wird dem Schlüsselwort jedoch ein @ vorangestellt. Der so neu angelegt Annotationstyp kann mit der Meta-Annotation `@Retention` in seiner Sichtbarkeit eingeschränkt werden. Dabei gibt `RetentionPolicy.SOURCE` an, dass die Annotation nur im Quellcode sichtbar ist und vom Compiler und der Laufzeitumgebung verworfen wird. Bei der `RetentionPolicy.CLASS` ist die Information der Annotation auch beim kompilieren verfügbar, nicht jedoch in der Laufzeitumgebung. Wohingegen bei der `RetentionPolicy.RUNTIME` die Information sowohl dem Compiler als auch der Laufzeitumgebung zur Verfügung stehen.

Mit der Meta-Annotation `@Target` kann beeinflusst werden welche Element der Sprache mit dieser Annotation markiert werden können. Mögliche Parameter dieser Annotation sind in der Klasse `java.lang.annotation.ElementType` enthalten. Die möglichen Parameter sind:

- ANNOTATION_TYPE
- CONSTRUCTOR
- FIELD
- LOCAL_VARIABLE
- METHOD
- PACKAGE
- PARAMETER
- TYPE

Ist die Annotation `@Target` nicht vorhanden, kann der neue Annotationstyp für alle Elemente verwendet werden.

Annotationstypen können unterschiedlich viele Parameter haben. Dabei werden Parameter von Annotationen deklariert wie Methoden in Schnittstellen. Im Unterschied dazu kann der Typ eines Parameters jedoch nicht beliebig sein. Es stehen folgende Typen zu Verfügung:

- alle primitiven Datentypen (byte, short, int, long, float, double, boolean)
- String
- Class
- Aufzählungstypen (enum-Typen)
- Andere Annotationen
- Eindimensionale Felder der genannten Typen

Hat ein Annotationstyp keine Parameter, spricht man von einer Marker-Annotation.

Parameter von Annotation werden als Schlüssel-Wert Paare angegeben. Bei Annotationstypen mit einem Parameter sollte dieser den Namen `value` haben. Dadurch kann bei der Verwendung der Annotation der Wert direkt in runde Klammern geschrieben werden und muss nicht als Schlüssel-Wert Paar angegeben werden.

```
@Retention(RetentionPolicy.CLASS)          // value only
@Target(value = ElementType.METHOD)        // key value pair
public @interface DemoAnnotation {
        String value();
}
```

Listing 3.1: Annotationstyp mit einem Parameter

Zur Laufzeit können Annotationen mittels der Reflection API ausgewertet werden. Wie dies funktioniert zeigt [Ull07] ist jedoch für die weiteren Betrachtungen hier nicht relevant.

3.2 AnnotationProcessor und AnnotationProcessorFactory

Seit Java6 können Annotationen direkt vom Compiler verarbeitet werden. Java5 benötigt dazu noch Unterstützung vom Annotation Processing Tool (APT). Da die ObjectTeams Entwicklungsumgebung ebenfalls vom APT Gebrauch macht, wird hier nur der Weg mit APT erläutert. Das APT wird direkt nach dem Compiler gestartet. Wird bei einem Durchlauf neuer Quellcode erzeugt, so wird dieser gleich kompiliert und das APT erneut gestartet. So können mehrere Runden durchlaufen werden bis im neu erzeugten Quellcode keine Annotationen mehr enthalten sind.

Damit das APT eine Annotation verarbeiten kann, benötigt es eine Instanz eines `AnnotationProcessor`. Diese Instanz erhält es indem alle verfügbaren `AnnotationProcessorFactories` geprüft werden ob sie einen Processor für die gegebene Annotation anbieten können. Der Factory wird dabei eine Instanz des `AnnotationProcessorEnvironment` übergeben. Mittels dieses Environments kann der `AnnotationProcessor` lesend auf den abstrakten Syntaxbaum des Projektes zugreifen.

3.3 Codeerzeugung

Um mittels einer Annotation Code zu erzeugen, wird als ersten ein selbst implementierter Annotationstyp benötigt. Da dieser Annotationstyp nur für den Compiler bzw. das APT sichtbar sein muss, reicht als Sichtbarkeitsart des Annotationstyps `Retention.CLASS` aus. Abhängig vom Anwendungsfall werden die Zielelemente der Annotation mit der Meta-Annotation `@Target` notiert.

Damit das APT den `AnnotationProcessor` findet, wird eine `AnnotationProcessorFactory` benötigt. Diese liefert als unterstützten Annotationstyp die selbst implementierte Annotation. Bei der Anfrage nach dem `AnnotationProcessor` wird eine neue Instanz des Processors erzeugt und das `AnnotationProcessorEnvironment` an diese übergeben.

Der `AnnotationProcessor` kann nun mittels des `AnnotationProcessorEnvironment` alle annotierten Elemente bestimmen lassen. Aus diesen Elementen kann dann der zu schreibende Code

erzeugt werden. Dies kann als normaler Text oder auch in Form eines abstrakten Syntaxbaumes geschehen. Der erzeugte Code wird als Text durch einen FileWriter den das AnnotationProcessorEnvironment bereitstellt in eine Datei geschrieben.

3.4 Annotationen in Eclipse

Damit die Entwicklungsumgebung mittels der Annotation neuen Code erzeugen kann, ist es nötig die erstellte AnnotationProcessorFactory der Entwicklungsumgebung bekannt zu machen. Dazu werden alle erstellten Klassen in ein Java Archiv verpackt. Des Weiteren muss das Archiv eine Datei mit dem Namen *com.sun.mirror.apt.AnnotationProcessorFactory* im Ordner *META-INF/services* enthalten. In dieser Datei steht der voll qualifizierte Klassenname der erstellten AnnotationProcessorFactory.

Soll ein Projekt die neue Annotation benutzen, so muss das Java Archiv in den Projekteinstellungen dem *FactoryPath* für das *AnnotationProcessing* hinzugefügt werden. Dort kann auch der Ausgabeort der zu erzeugenden Dateien festgelegt werden. Wird während des Kompilierens die Annotation erkannt, so wird automatisch nach dem Kompilieren das *AnnotationProcessing* ausgeführt.

4 Erweiterung der ObjectTeams/Java Entwicklungsumgebung

Die ObjectTeams/Java Entwicklungsumgebung Object Teams Develpoment Tooling basiert auf Eclipse und dessen Java Development Tooling. Damit das Object Teams Development Tooling die neuen Konzepte von ObjectTeams/Java verarbeiten kann, wurde das Java Development Tooling von Eclipse an mehreren Stellen um zusätzliche Klassen ergänzt. Da dies bis jetzt noch nicht nötig war, wurden die neuen Sprachkonzepte jedoch nicht an allen Stellen des Java Development Tooling eingefügt.

Das Verarbeiten von Annotationen in Eclipse basiert auf der *Mirror* Programmierschnittstelle von SUN und ist im Java Development Tooling integriert. Da das Annotation Processing für das Object Teams Development Tooling nicht angepasst wurde, sind Annotationen bisher nur an Elementen von Standard Java verfügbar. Da es im Zuge dieser Diplomarbeit nötig ist auch Elemente der Sprache ObjectTeams/Java mit Annotationen zu versehen, muss die Annotationsverarbeitung des Java Development Tooling angepasst werden.

Die nötigen Anpassungen können durch zwei verschiedene Ansätze realisiert werden. Zum einen können die nötigen Änderungen direkt in den Quellcode des Java Development Tooling eingefügt werden, andererseits ist es aber auch möglich die nötigen Ergänzungen mittels eines aspektorientierten Ansatzes an das Java Development Tooling zu hängen. Da für diese Diplomarbeit die vollständige Erweiterung des Annotation Processings um die ObjectTeams Konzepte nicht nötig ist, wird hier der zweite Weg gewählt. Die Umsetzung dieses aspektorientierten Ansatzes wird in Form eines Plugins für die Eclipse Plattform realisiert.

4.1 Annotationen an Elementen der Sprache ObjectTeams/Java

Mit dem aktuellen Java Development Tooling ist es nicht möglich ein neues Element der Sprache ObjectTeams/Java als annotiertes Element zu erkennen. Dies liegt an der Funktionsweise des Annotation Processings. Um annotierte Elemente zu erkennen, wird ein Visitor an die Wurzel des abstrakten Syntaxbaum der zu untersuchenden Klasse geschickt. Hierbei handelt es sich um den `AnnotatedNodeVisitor` aus dem Paket

org.eclipsejdt.apt.core.internal.util. Die Elemente des abstrakten Syntaxbaumes schicken den Visitor an alle ihre untergeordneten Elemente. Somit wird der komplette abstrakte Syntaxbaum traversiert. Des Weiteren werden alle Elemente dem Visitor gezeigt, so dass dieser entscheiden kann ob sie für die Verarbeitung innerhalb des Visitors relevant sind.

Da das Annotation Processing noch nicht an die ObjectTeams/Java Elemente angepasst wurde, entscheidet der AnnotatedNodeVisitor, dass die neuen Elemente der Sprache nicht für die Weiterverarbeitung relevant sind, da er sie nicht verarbeiten kann. Somit werden diese Elemente nicht als annotierte Elemente erkannt.

Der AnnotatedNodeVisitor spezialisiert den ASTVisitor aus dem Paket org.eclipse.jdt.core.dom. Der ASTVisitor wurde bereits an ObjectTeams/Java angepasst und kann somit die neuen Elemente der Sprache verarbeiten. Die angestrebte Lösung besteht also darin, mittels Callin-Bindung im AnnotatedNodeVisitor auch Elemente der Sprache ObjectTeams/Java zu verarbeiten und diese dann per Callout-Bindung in die Ergebnisliste des Visitors aufzunehmen.

Die Objekte in der Ergebnisliste des AnnotatedNodeVisitor sind vom Typ BodyDeclaration aus dem Paket org.eclipse.jdt.core.dom. Diese können jedoch nicht von der *Mirror* Programmierschnittstelle, die für die Weiterverarbeitung der Annotationen verantwortlich ist, erkannt werden. Darum müssen diese Objekte vorher übersetzt werden in Objekte des Typs Declaration aus dem Paket com.sun.mirror.declaration. Normalerweise geschieht diese Übersetzung durch das Java Development Tooling, das dafür Objekte aus dem Paket org.eclipse.jdt.apt.core.internal.declaration bereitstellt. Objekte aus diesem Paket können sowohl von der *Mirror* Programmierschnittstelle als auch vom Java Development Tooling weiterverarbeitet werden. Innerhalb dieses Pakets sind jedoch noch keine Typen für die Elemente der Sprache ObjectTeams/Java implementiert. Es müssen also neue Klassen für die ObjectTeams/Java Elemente angelegt werden.

Damit das Java Development Tooling diese neuen Klassen verarbeiten kann, müssen sie die Klasse EclipseDeclarationImpl aus dem Paket org.eclipse.jdt.apt.core.internal.declaration spezialisieren. Die genannte Klasse ist zwar öffentlich sichtbar, ihr Konstruktor jedoch nur innerhalb des genannten Pakets. Somit ist es nicht möglich die Spezialisierung zu implementieren.

Für die Weiterverarbeitung der Annotationen ist es aber nicht nötig, dass das Java Development Tooling die annotierten Objekte erkennt, denn es reicht aus, wenn sie von der *Mirror* Programmierschnittstelle verarbeitet werden können. Es reicht hier also aus, wenn die neuen Klassen die Schnittstelle Declaration aus dem Paket com.sun.mirror.declaration implementieren.

4.2 Zusätzlich Klassen

In diesem Abschnitt werden die zusätzlich benötigten Klassen, deren Aufgaben und Funktionsweise kurz vorgestellt. Die vollständige Implementierung der Klassen bleibt dem praktischen Teil der Arbeit überlassen.

4.2.1 CFlowPluginTeam

Dieses Team ist das Kernstück des Plugins. Innerhalb dieses Teams sind zwei Rollen implementiert. Die Rolle ASTVisitorRole adaptiert den AnnotatedNodeVisitor und lässt diesen die neuen Elemente der Sprache ObjectTeams/Java verarbeiten. Da das Java DevelopmentTooling diese Elemente nicht übersetzen kann, werden sie innerhalb der Rolle BaseProcessorEnvRole übersetzt.

```
public team class CFlowPluginTeam {
        public class BaseProcessorEnvRole playedBy BaseProcessorEnv {
                // translate dom objects to mirror declarations
                doGetDeclarations <- replace getDeclarations;

                callin void doGetDeclarations(ASTNode node,
                                List<Declaration> declarations) {…}
        }

        public class ASTVisitorRole playedBy AnnotatedNodeVisitor {
                // process callin bindings
                boolean doVisit(CallinMappingDeclaration decl) <- replace
                        boolean visit(CallinMappingDeclaration decl);
                callin boolean doVisit(CallinMappingDeclaration decl) {…}

                // callout to add elemente to result list
                doVisitBodyDeclaration -> visitBodyDeclaration;
                abstract void doVisitBodyDeclaration(BodyDeclaration decl);
        }
}
```

Listing 4.1: CFlowPluginTeam

4.2.2 CallinMappingDeclarationImpl

Die Objekte der Ergebnisliste, die der AnnotatedNodeVisitor liefert, müssen vom Typ Declaration aus dem Paket com.sun.mirror.declaration sein. Darum bildet die Klasse CallinMappingDeclarationImpl die Repräsentation einer Callin-Bindung für die *Mirror* Programmierschnittstelle. Des Weiteren werden Objekte dieser Klasse dem AnnotationProcessor als annotierte Elemente geliefert. Darum werden in dieser Klasse zusätzliche Methoden zur Auswertung der Rolle und des Teams, in denen die annotierte Callin-Bindung enthalten ist, benötigt.

```
public class CallinMappingDeclarationImpl implements Declaration {
        public Collection getAnnotationMirrors() {…}

        public String getRoleClassName() {…}
        public String getBaseMethodName() {…}
        public String getRoleMethodName() {…}
        public String getTeamClassName() {…}
}
```

Listing 4.2: CallinMappingDeclarationImpl

4.2.3 CFlowAnnotationMirrorImpl und CFlowAnnotationValue

Da für die Implementierung der CallinMappingDeclarationImpl das Java Development Tooling der Eclipse Plattform umgangen wurde, ist es nötig den AnnotationMirror und den AnnotationValue für Annotationen an Callin-Bindungen selber zu implementieren. Der AnnotationMirror bildet dabei die Repräsentation der Annotation innerhalb der *Mirror* Programmierschnittstelle. Der AnnotationValue repräsentiert die Parameter eines Annotationstyps.

```
public class CFlowAnnotationMirrorImpl implements AnnotationMirror {
        public Map getElementValues() {…}
}
```

Listing 4.3: CFlowAnnotationMirrorImpl

```
public class CFlowAnnotationValue implements AnnotationValue {
        public Object getValue() {…}
}
```

Listing 4.4: CFlowAnnotationValue

5 Steuerung von Callin-Bindungen

Damit ein Aspekt in ObjectTeams/Java als deaktiviert angesehen werden kann, reicht es aus zu verhindern, dass der Advice ausgeführt wird. Da der Advice von einer Callin-Bindung aufgerufen wird, muss diese Callin-Bindung deaktiviert werden damit der Advice deaktiviert ist. Für die Deaktivierung der Callin-Bindung sollen jedoch noch zusätzliche Randbedingungen gelten. Zum einen soll die Deaktivierung durchgeführt werden ohne den Aktivierungszustand des Teams zu verändern das die Callin-Bindung umschließt. Außerdem sollen möglichst wenige Änderungen an bestehendem Quellcode vorgenommen werden.

In den folgenden Abschnitten werden zwei Lösungen für das genannte Problem aufgezeigt und verglichen, um dann zu entscheiden welcher der beiden Ansätze in dieser Diplomarbeit umgesetzt werden soll.

5.1 Ersetzung des Advice

Die erste Methode beruht darauf, den zu deaktivierenden Advice zu ersetzen und somit die Ausführung zu steuern. Die Ersetzung kann durch einen objektorientierten Ansatz oder durch einen aspektorientierten Ansatz erfolgen. Auch hier haben beide Wege Vor- und Nachteile. Der allgemeiner verwendbare soll für diese Methode genutzt werden.

5.1.1 Objektorientierter Ansatz

Der einfachste denkbare Ansatz ist es den Advice, der ja nur eine Java-Methode ist, zu überschreiben. In der überschriebenen Form wird geprüft ob der Advice ausgeführt werden soll und bei positiver Prüfung der originale Advice aufgerufen. Damit die Überschreibung des Advice möglich ist, muss die Rollenklasse, die den Advice enthält, durch eine neue Rollenklasse spezialisiert werden. Da Rollenklassen innere Klassen von Teamklassen sind, muss auch ein neues spezialisiertes Team erzeugt werden. Dieser Ansatz beruht darauf, dass explizite Vererbung genutzt wird. Die Ergebnisse der Betrachtung sind jedoch relativ genau auf die implizite Vererbung übertragbar. Abbildung 5.1 zeigt die Grundliegende Idee dieses Ansatzes.

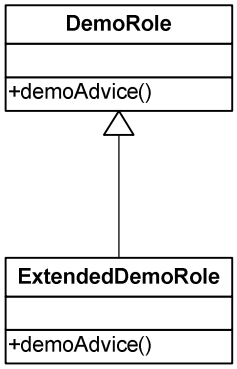

Abbildung 5.1: Spezialisierung der Rollenklasse

Dieser Ansatz ist schlank und einfach, bringt aber einige Nachteile mit sich. Würde es eine weitere Spezialisierung der originalen Rollenklasse geben, so wäre der Advice hier nicht überschrieben und würde ungeprüft ausgeführt. Außerdem ist es denkbar, dass die originale Rollenklasse abstrakt ist und somit noch nicht alle Methoden vollständig implementiert sind. In diesem Fall müssten die Implementierungen innerhalb der spezialisierten Rollenklasse automatisch generiert werden, da die spezialisierte Rollenklasse nur generierten Quellcode enthalten soll. Abbildung 5.2 verdeutlicht die genannten Probleme.

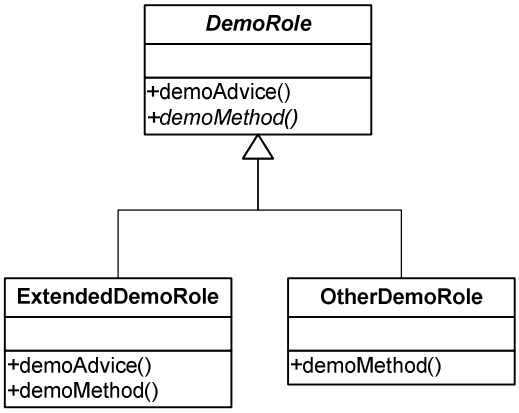

Abbildung 5.2: Probleme bei Spezialisierung

Eine mögliche Lösung für dieses Problem wäre die neue generierte Rollenklasse zwischen die originale und alle anderen spezialisierten Rollenklassen zu bringen. Die bestehenden spezialisierten Rollenklassen müssen dann die generierte Rollenklasse spezialisieren. Wird jedoch der Advice nochmals überschrieben, wird dieser Ansatz dadurch unterwandert. Abbildung 5.3 zeigt die Idee und das Problem dieses Ansatzes.

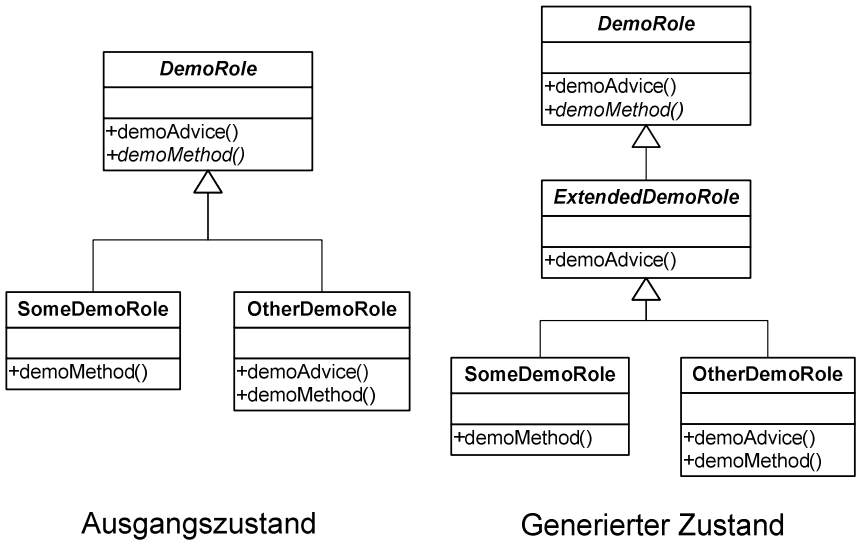

Abbildung 5.3: Generierte Zwischenklasse

Damit der Advice der originalen Rollenklasse überhaupt überschrieben werden kann, ist es zwingend notwendig, dass die Java-Methode, die den Advice darstellt, keine private Methode der originalen Rollenklasse ist. Die hierbei entstehende Kapselung könnte mit keiner objektorientierten Methode durchbrochen werden. Somit ergibt sich für den objektorientierten Ansatz die Bedingung, dass der Advice für eine spezialisierende Rolle sichtbar sein muss.

Ein weiteres Problem bereitet die Aktivierung des generierten Teams. Geht man von Abbildung 5.3 aus, ist es nicht nötig das zusätzliche Team zu aktivieren, da es durch die Aktivierung der spezialisierenden Teams aktiviert wird. Sind jedoch, wie in Abbildung 5.1, keine weiteren Spezialisierungen vorhanden, so muss das generierte Team explizit aktiviert werden. Des Weiteren darf die Aktivierung des originalen Teams nicht durchgeführt werden. Am einfachsten wäre es alle Stellen an denen das originale Team instanziiert wird durch eine Instanziierung des generierten Teams zu ersetzen. Dies würde jedoch einen zu tiefen Eingriff in bestehenden Code erfordern.

Eine andere Möglichkeit bedient sich eines aspektorientierten Ansatzes. Hierfür ist es nötig die Aktivierung des originalen Teams durch die Aktivierung des generierten Teams zu ersetzen. Dafür ist ein weiteres Team notwendig, dass die Aktivierung des originalen Teams überwacht. Abbildung 5.4 zeigt die statische Struktur dieses Ansatzes.

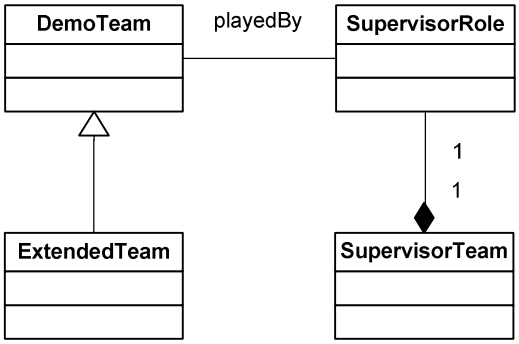

Abbildung 5.4: SupervisorTeam

Das überwachende Team (SupervisorTeam) muss jetzt noch das generierte Team (ExtendedTeam) anstatt des originalen Teams (DemoTeam) aktivieren. Dazu wird das `activate()` des originalen Teams ersetzt und im Advice das generierte Team aktiviert, das dazu innerhalb des `SupervisorTeams` instanziiert werden muss.

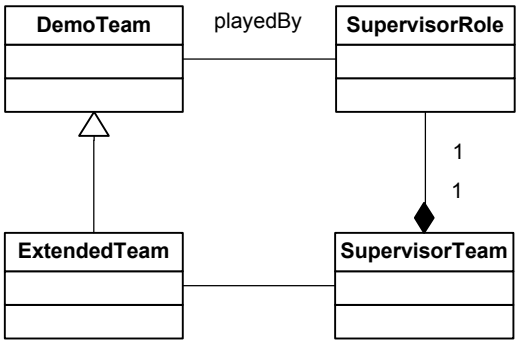

Abbildung 5.5: Aktivierung durch SupervisorTeam

Wie in Abbildung 5.5 auch grafisch zu sehen ist, ergibt sich aus diesem Ansatz eine Kreisreferenz. Durch das Abfangen der Aktivierung am `DemoTeam` werden auch die Aktivierungen des `ExtendedTeam` abgefangen, da diese eine Spezialisierung des `DemoTeam` ist. Innerhalb des Advice wird jetzt wieder das `ExtendedTeam` aktiviert, woraufhin die Aktivierung erneut abgefangen wird. Dieser Prozess wiederholt sich bis die Ausführung der Anwendung mit einem `StackOverflowError` abbricht. Allein durch diesen Umstand ist der objektorientierte Ansatz zum Ersetzen des Advice gescheitert.

5.1.2 Aspektorientierter Ansatz

Eine weitere Möglichkeit zum Ersetzen einer Methode bietet die Aspektorientierte Programmierung. Die Idee dieses Ansatzes ist es die Ausführung des Aspektes zu verhindern, indem der Advice durch eine Callin-Bindung ersetzt wird. Der Advice der durch diese neue Callin-Bindung ausgeführt wird, prüft zuerst ob die Ausführung des ursprünglichen Advice stattfinden soll, und führt diesen bei positiver Prüfung mittels Basecall aus. In Abbildung 5.6 ist dies Ablaufsequenz grafisch dargestellt.

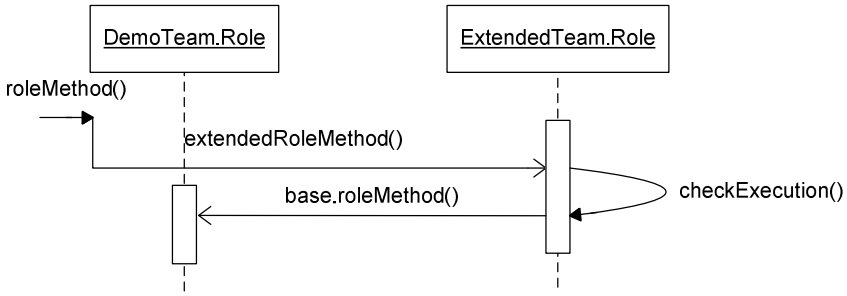

Abbildung 5.6: Ausführungsprüfung von Rollenmethoden

Da dieser Ansatz ohne Spezialisierung auskommt, treten hier auch nicht die Probleme auf, die beim objektorientierten Ansatz existieren. Es ist hier also nicht nötig auf abstrakte Methoden zu reagieren und auch auf wiederholtes Überschreiben des Advice muss keine Rücksicht genommen werden. Die statische Struktur, die in Abbildung 5.7 gezeigt wird, verdeutlicht den Kern dieses Ansatzes.

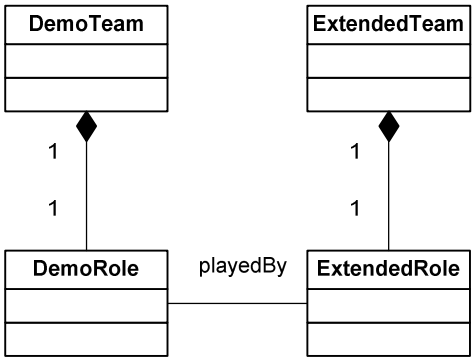

Abbildung 5.7: Statische Struktur

Damit die gezeigte statische Struktur umgesetzt werden kann, ist es nötig die originale Rollenklasse als externe Rolle bereitzustellen, da sie außerhalb ihres umschließenden Teams referenziert werden muss. Somit ergibt sich, dass bei der Instanziierung der generierten Teamklasse eine Instanz der original Teamklasse zur Verfügung stehen muss. Die generierte Teamklasse soll jedoch nicht bei der Instanziierung der originalen Teamklasse instanziiert werden, da dies in der Regel ein zu tiefer Eingriff in bestehenden Code darstellen würde.

Da auch das originale Team aktiviert werden muss damit die Callin-Bindungen aktiv sind, bietet sich dessen activate() Methode an um die Instanziierung des erzeugten Teams durchzuführen. Dazu überwacht eine weitere Rolle eben diese activate() Methode des originalen Teams und ersetzt sie durch die Instanziierung des generierten Teams und aktiviert dieses. Mittels Basecall wird dann auch das originale Team aktiviert. Da das generierte Team nicht benötigt wird wenn das originale Team deaktiviert ist, wird auch die deactivate() Methode des originalen Teams ersetzt. Die neue deactivate() Methode deaktiviert erst das generierte Team und danach, mittels Basecall das originale Team.

Da die überwachende Rolle in einer **playedBy** Beziehung mit dem originalen Team steht, kann die Instanziierung des generierten Teams hier problemlos erfolgen. Dazu wird mittels Lowering die überwachende Rolle in ein originales Team überführt, mit dem dann das generierte Team instanziiert wird. Listing 5.1 zeigt die wichtigsten Methoden des überwachenden Teams.

```
public team class SupervisorTeam {
        public class SupervisorRole playedBy DemoTeam {
                private ExtendedTeam innerTeam;

                void doActivate(Thread thread)
                        <- replace void activate(Thread thread);
                void doDeactivate(Thread thread)
                        <- replace void deactivate(Thread thread);

                callin void doActivate(Thread thread) {
                        if (innerTeam == null)
                                innerTeam = new ExtendedTeam(this);
                        innerTeam.activate(thread);
                        base.doActivate(thread);
                }
```

```
                callin void doDeactivate(Thread thread) {
                        if (innerTeam != null)
                                innerTeam.deactivate(thread);
                        base.doDeactivate(thread);
                }
        }
}
```

Listing 5.1: Überwachendes Team

Die statische Struktur in Abbildung 5.8 verdeutlicht nochmals die Funktionsweise dieses Ansatzes. Das originale Team hat in diesem Beispiel den Namen DemoTeam. Das SupervisorTeam ist das überwachende Team und das ExtendedTeam ist das generierte Team. Die SupervisorRole überwacht die activate() und deactivate() Methoden des DemoTeam und aktiviert bzw. deaktiviert zusätzlich das ExtendedTeam. Die ExtendedRole überwacht den Advice der DemoRole und entscheidet ob dieser mittels Basecall ausgeführt werden soll.

Auch wenn das Klassendiagramm in Abbildung 5.8 wieder eine zyklische Abhängigkeit vermuten lässt, tritt diese hier nicht auf. Die Kreisreferenz im objektorientierten Ansatz entstand durch den Einsatz einer Spezialisierungsbeziehung. Da der aspektorientierte Ansatz keine Spezialisierung benötigt, tritt hier auch keine Kreisreferenz auf.

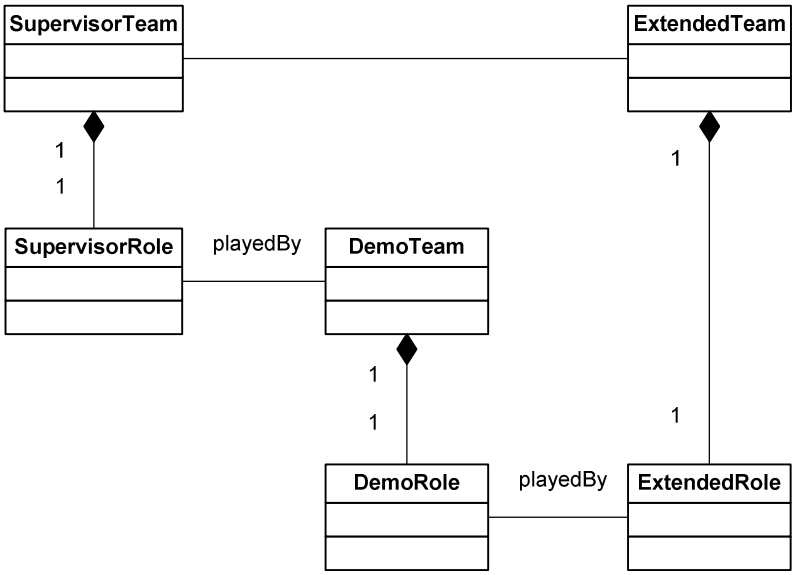

Abbildung 5.8: Statische Struktur des aspektorientierten Ansatzes

Gegenüber dem objektorientierten Ansatz hat der aspektorientierte Ansatz den Vorteil, dass er überhaupt umsetzbar ist. Allerding bringt dieser Ansatz auch kleine Randbedingungen mit sich. Da das ursprüngliche Team als Variable innerhalb des generierten Teams benötigt wird, muss entweder die Sichtbarkeit dieses Teams **public** sein oder die beiden Klassen müssen im gleichen Paket liegen. Da aber der generierte Code nicht im gleichen Paket liegen sollte, muss also die Sichtbarkeit des ursprünglichen Teams zwingend **public** sein.

Da das überwachende Team alle activate() Aufrufe des originalen Teams überwachen soll, ist es notwendig, dass das überwachende Team vor dem originalen Team aktiviert wird.

Eine weitere Schwäche dieses Ansatzes liegt darin, dass die Ausführung des Advice nicht verhindert wird sondern der Advice durch eine andere Java-Methode ersetzt wird. Diese neue Methode prüft nur ob der Advice ausgeführt werden soll oder nicht. Bei positiver Prüfung wird der Advice per Basecall ausgeführt. Fällt die Prüfung negativ aus, wird der Basecall nicht aufgerufen und somit der Advice nicht ausgeführt. Das Ergebnis hieraus ist, dass kein zusätzlicher Code ausgeführt wird. Für Callin-Bindungen die zusätzlich zu bestehendem Code ausgeführt werden funktioniert dieses Verfahren auch zufriedenstellend. Der Code der Basisapplikation wird ausgeführt und nach Ausführungsprüfung wird der zusätzliche Advice ausgeführt oder kein weiterer Code.

Für ersetzende Callin-Bindungen funktioniert dieses Verfahren jedoch nicht. In diesem Fall wird die ursprüngliche Methode durch den Advice ersetzt. Das generierte Team ersetzt wiederum diesen Advice. Schlägt nun die Ausführungsprüfung fehl, wird weder der ursprüngliche Code noch der eigentliche Advice ausgeführt. Das gewünschte Verhalten sieht jedoch vor, dass wenn der Advice unterdrückt wird, der ursprüngliche Code ausgeführt wird. Dazu wäre im generierten Advice jedoch ein Base-Basecall auf den ursprünglichen Code nötig, da per einfachen Basecall ja nur der ursprüngliche Advice ausgeführt werden würde. Das Sequenzdiagramm in Abbildung 5.9 zeigt wie ein Base-Basecall aussehen müsste. Da solch ein Base-Basecall zurzeit nicht in ObjectTeams/Java vorgesehen ist, müsste dieser erst implementiert werden damit der aspektorientierte Ansatz zur Ersetzung des Advice funktionieren kann.

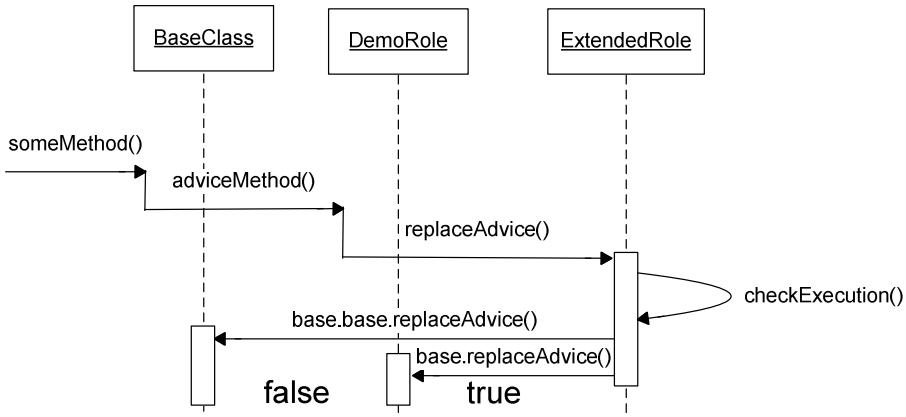

Abbildung 5.9: Sequenzdiagramm Base-Basecall

5.2 Generierte Guards

Die Sprache ObjectTeams/Java bringt von Hause aus eine Möglichkeit mit die Ausführung von Aspektcode zu steuern. Dazu werden sogenannte Guards benutzt. Guards bieten die Ausführungsprüfung die für die gewünschte Steuerung von Callin-Bindungen benötigt wird. Allerdings müssen Guards explizit in dem Team bzw. an der Callin-Bindung implementiert werden an der sie benötigt werden. Aus diesem Umstand ergibt sich auch schon die erste Randbedingung für diesen Ansatz. Zusätzlich zu den eventuell bestehenden Guards muss ein weiterer Guard an die Callin-Bindung generiert werden. Dieser Ansatz kommt also nicht ohne die Veränderung bestehenden Codes aus.

Um die Veränderungen am bestehenden Code möglichst gering zu halten, wird nur der benötigt Guard erzeugt bzw. ein bestehender Guard modifiziert. Zusätzlich wird eine Methode erzeugt, die von diesem Guard aufgerufen wird. Damit sich das Laufzeitverhalten des originalen Codes nicht ändert, liefert diese Methode nur den Wert **true** zurück. Ein aspektorientierter Ansatz kann diese Methode ersetzen und somit die Callin-Bindung steuern. Würde diese Methode innerhalb der Rollenklasse generiert, wäre eine externe Rolle nötig um die Methode mittels Callin-Bindung zu ersetzen. Darum wird die Methode innerhalb des umschließenden Teams erzeugt. Da eine Rolle aus Java-Sicht eine nicht statische innere Klasse ist, kann diese problemlos auf die Methode zugreifen.

Da die Annotationsverarbeitung, die nach dem Kompilieren ausgeführt wird, bestehenden Code nicht verändern kann, taugt diese Methode nicht um den Guard und die Methode zu generieren. Da Annotationen jedoch

eine schlanke und gut lesbare Metainformation bilden, sollen diese trotzdem eingesetzt werden um den Guard und die Methode zu generieren. Dazu muss der ObjectTeams/Java Compiler angepasst werden, da die dafür angelegte Annotation schon beim Kompilieren verarbeitet werden muss. Listing 5.2 zeigt beispielhaft welcher Code in eine Rollenklasse und dessen umschließende Teamklasse generiert werden muss.

```
// Originaler Code
public team class DemoTeam {
	protected class DemoRole playedBy BaseClass {
		@Guard()
		callinMethod <- replace baseMethod;

		private void callinMethod() {…}
	}
}

// Code inklusive generiertem Guard und generierter Methode
public team class DemoTeam {
	protected class DemoRole playedBy BaseClass {
		@Guard()
		callinMethod <- replace baseMethod
			when (createdGuardMethod());

		private void callinMethod() {…}
	}

	private boolean createdGuardMethod() {
		return true;
	}
}
```

Listing 5.2: Beispielcode für generierte Guards

Damit die Ausführung der Callin-Bindung gesteuert werden kann, ist eine weitere Rolle nötig, die die generierte Methode ersetzt. Innerhalb dieser neuen Rolle kann dann bewertet werden, ob in der aktuellen Situation die Ausführung der originalen Callin-Bindung stattfinden soll oder nicht. Listing 5.3 zeigt wie die neue Rolle aussehen muss.

```
public team class ExtendedTeam {
        protected class ExtendedRole playedBy DemoTeam {
                replacedGuardMethod <- replace createdGuardMethod;

                private boolean replacedGuardMethod() {…}
        }
}
```

Listing 5.3: Generierte Team- und Rollenklasse

Bei den vorherigen Ansätzen ist es nötig die Sichtbarkeit der Advicemethode bzw. des ursprünglichen Teams so einzustellen, dass die Kapselung aufgeweicht wird und die genannten Elemente nach außen sichtbar sind. Bei diesem Ansatz ist diese Aufweichung nicht nötig, da ObjectTeams/Java das Konzept der Entkapselung beherrscht und somit auch private Methode ersetzen kann. Des Weiteren können dadurch auch nicht öffentliche Klassen als Basisklassen importiert werden.

Beim aspektorientierten Ansatz war es nötig das erzeugte Team vor dem ursprünglichen Team zu aktivieren. Diese Restriktion kann für diesen Ansatz etwas entschärft werden. Im Normalfall kann diese Einschränkung ganz vernachlässigt werden. Nur wenn die Ausführungsreihenfolge von Callin-Bindungen relevant wird, ist auch in diesem Ansatz die Reihenfolge der Aktivierung wichtig. Da dies jedoch als Spezialfall angesehen werden darf, beeinträchtigt dies nicht die Umsetzung dieses Ansatzes. Das generierte Team muss jedoch aktiviert sein wenn die Guardmethode ausgeführt wird, damit die Methodenersetzung aktiv ist.

5.3 Vergleich

Vorgestellt wurden drei Methoden um Callin-Bindungen nachträglich zu beeinflussen. Da alle drei Stärken und Schwächen zeigen, werden hier nochmal alle gegenübergestellt um herauszufinden welche Methode im Verlauf dieser Arbeit benutzt werden soll.

Der objektorientierte Ansatz und der aspektorientierte Ansatz benötigen überwachende Teams, die die generierten Teams aktivieren müssen. Damit dies zuverlässig funktioniert, müssen alle Aufrufe der überwachten Methoden erkannt werden. Darum müssen die

überwachenden Teams als erstes aktiviert werden. Beim Ansatz des generierten Guards muss das generierte Team zwar auch aktiviert werden, hier ist die Aktivierungsreihenfolge aber nur in Ausnahmefällen relevant. Somit hat der Ansatz des generierten Guards hier einen Vorteil.

Für alle drei Ansätze muss zusätzlicher Code erstellt werden. Für den Ansatz des generierten Guards muss bestehender Code verändert werden um den Guard an eine Callin-Bindung zu generieren. Der objektorientierte Ansatz funktioniert nur wenn die Instanziierung des ursprünglichen Teams durch die Instanziierung des generierten Teams ersetzt wird. Nur beim aspektorientierten Ansatz muss kein Code verändert werden, was hier einen Vorteil darstellt.

Beim aspektorientierten Ansatz muss der Compiler angepasst werden, um einen Base-Basecall zu ermöglichen. Und auch beim Ansatz des generierten Guards muss der Compiler angepasst werden um den Guard während des Kompilierens zu erzeugen. Der objektorientierte Ansatz hat hier den Vorteil, dass der Compiler unangetastet bleiben kann.

Beim objektorientierten Ansatz kann die Funktionsweise nicht sichergestellt werden, da durch wiederholte Spezialisierung der überschrieben Advice erneut überschrieben werden könnte. Dies stellt einen entscheidenden Nachteil für diesen Ansatz dar.

Somit stehen nur der aspektorientierte Ansatz und der Ansatz des generierten Guards zur Auswahl. Durch seine größere Übersichtlichkeit und die einfachere Teamaktivierung stellt sich der Ansatz des generierten Guards als der Ansatz heraus, der am einfachsten und elegantesten umzusetzen ist und somit in dieser Diplomarbeit umgesetzt werden soll.

6 Situationsbewertung auf Basis des aktuellen Kontrollflusses

In den vorangegangenen Kapiteln wurde immer wieder von einer Situation gesprochen in der ein Aspekt aktiviert oder deaktiviert sein soll. Diese Situation wurde bisher jedoch nicht weiter definiert. Eine mögliche Definition soll in diesem Kapitel gezeigt werden. Des Weiteren sollen alle zur Auswertung dieser Situation benötigten Komponenten skizziert werden.

Die hier vorgestellte Situation bewertet den aktuellen Kontrollfluss einer Applikation. Es werden hier also nur die Methoden zur Bewertung herangezogen die sich zum Bewertungszeitpunkt auf dem Aufrufstapel des aktuellen Threads befinden. Zuerst wird also eine Möglichkeit benötigt um den Aufrufstapel auszuwerten.

6.1 Auswertung des Aufrufstapels

Für die Auswertung des Aufrufstapels kommen erneut verschiedene Techniken in Frage. Die nächsten Abschnitte erläutern zwei der möglichen Techniken und überprüfen die Umsetzbarkeit dieser Techniken.

6.1.1 Auswertung mittels Standard Java Methoden

Die Programmiersprache Java bietet bereits eine Möglichkeit den Aufrufstapel auszulesen. Mittels der Methode `Thread.getStackTrace()` erhält man eine Liste von `StackTraceElements`, die den aktuellen Aufrufstapel repräsentieren. Um anhand von diesen `StackTraceElements` herauszufinden ob sich eine bestimmte Methode auf dem Aufrufstapel befindet, muss für alle Elemente der Liste geprüft werden, ob der Name der Methode und der Klassenname mit dem Namen und dem Klassennamen der gesuchten Methode übereinstimmen.

Da in Java Methoden überladen werden können, können in derselben Klasse zwei Methoden mit dem gleichen Namen, jedoch verschiedenen Parametern, existieren. Um also exakt zu bestimmen ob sich eine Methode auf dem Aufrufstapel befindet, müssen auch die Parameter der Methode überprüft werden. Die Klasse `StackTraceElement` bietet jedoch

nicht die Möglichkeit die Parameter der ausgeführten Methode anzuzeigen.

Ein StackTraceElement bietet jedoch die Möglichkeit die Zeile, in der die Deklaration der Methode steht, anzuzeigen. Theoretisch ist es möglich anhand dieser Zeilennummer auf die Methode und deren Parameter zu schließen. Dazu müssen jedoch einige Randbedingungen gelten. Zum einen ist diese Zeilennummer nur vorhanden, wenn beim Kompilieren die Debuginformationen in den Bytecode kompiliert werden. Bei den Klassen des Java Runtime Environments ist dies zum Beispiel nicht der Fall. Des Weiteren muss der Quellcode vorliegen, damit dort analysiert werden kann, um welche Methode es sich tatsächlich handelt. Diese Debuginformationen sind aber nicht zwangsläufig zuverlässig denn so gut wie jeder Obfuscator vertauscht diese Zeileninformationen, damit eben nicht auf die ausgeführte Methode geschlossen werden kann.

6.1.2 Organisation des Aufrufstapels mittels Aspektorientierer Programmierung

Der Java Standard bietet also keine ausreichenden Informationen über den aktuellen Aufrufstapel. Daraus ergibt sich jedoch die Notwendigkeit den Aufrufstapel selber aufzubauen. Da in diesem Beispiel nur interessant ist, ob eine Methode auf dem Aufrufstapel ist oder nicht, ist es auch nur für die angegebenen Methoden notwendig den Aufrufstapel zu organisieren. Er muss nicht vollständig, also für alle Methoden, sondern nur selektiv aufgebaut werden.

Mittels einer Callin-Bindung soll signalisiert werden, dass sich eine Methode auf dem Aufrufstapel befindet. Die zu überwachende Methode muss also ersetzt werden, durch einen Advice der als erstes ein Event generiert, das den Eintritt in die Methode signalisiert. Als nächstes wird per Basecall die eigentliche Methode ausgeführt. Und zum Schluss wird erneut ein Event erzeugt welches diesmal jedoch das Verlassen der Methode signalisiert. Damit der Event beim Verlassen der Methode auch erzeugt wird wenn der Basecall mit einer Exception beendet wird, muss die Erzeugung des Events in einem Codeblock geschehen, der auch im Fehlerfall ausgeführt wird. Dies wird mittels des Schlüsselwortes **finally** gewährleistet. Lisitng 6.1 zeigt beispielhaft die Implementierung einer solchen Überwachung. Zur Erzeugung der Events werden die Methoden enterMethod(String methodName) und leaveMethod(String methodName) aus der Oberklasse benutzt. Da es sich bei diesem überwachenden Team um automatisch erzeugten Code handeln wird, muss durch die Benutzung von Methoden der Oberklasse nur wenig neuer Code erzeugt werden.

```
public team class SupervisorTeam extends AbstractSupervisorTeam {
        protected class SupervisorRole playedBy demo.DemoClass {
                replacedMethod <- replace demoMethod;

                callin void replacedMethod() {
                        enterMethod("demo.DemoClass.demoMethod()");
                        base.replacedMethod();
                        finally {
                                leaveMethod
                                        ("demo.DemoClass.demoMethod()");
                        }
                }
        }
}
```

Listing 6.1: Aufbau des Aufrufstapels

Durch die Benutzung einer Callin-Bindung ergibt sich jedoch auch ein Spezialfall. Wird die Methode, die durch die Callin-Bindung ersetzt wird, durch eine weitere Callin-Bindung ersetzt, so entscheidet die Reihenfolge der Aktivierung der Teams über die Ausführungsreihenfolge der Callin-Bindungen. Die Callin-Bindung deren Team zuletzt aktiviert wird, wird dabei als erstes ausgeführt. Die Abbildungen 6.1 und 6.2 zeigen die beiden möglichen Sequenzdiagramme, die entstehen wenn die vom SupervisorTeam überwachte Methode von einer weiteren Callin-Bindung ersetzt wird.

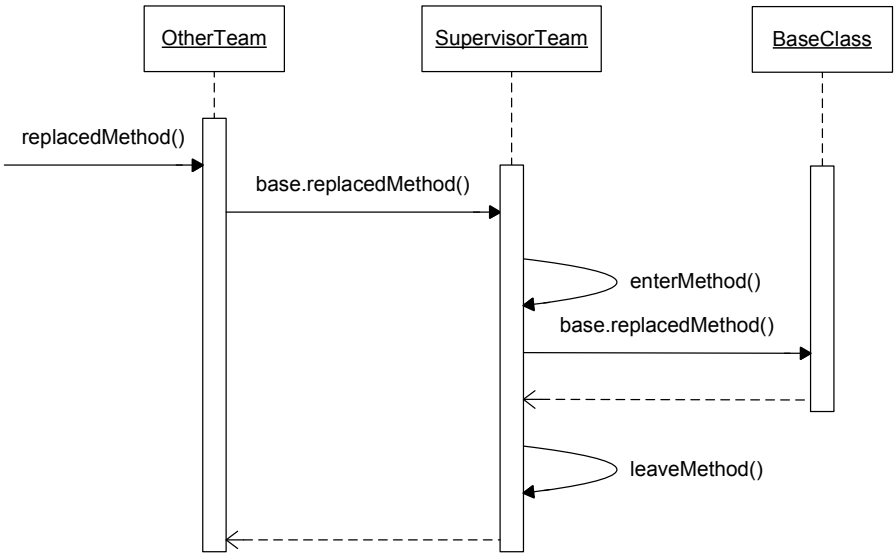

Abbildung 6.1: SupervisorTeam aktiviert vor OtherTeam

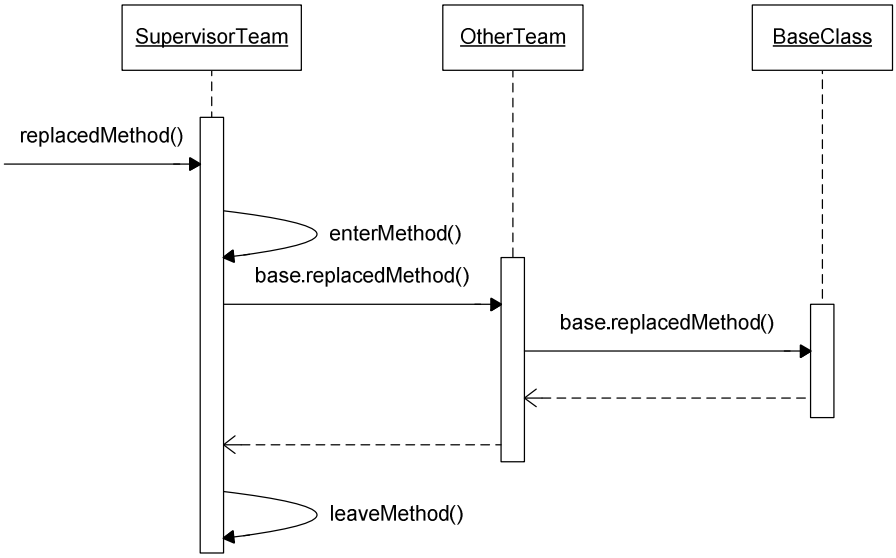

Abbildung 6.2: OtherTeam aktiviert vor SupervisorTeam

Wird bei der Ausführung der Callin-Bindung im `OtherTeam` geprüft ob bereits der Event beim Eintreten der Methode gefeuert wurde, so erhält man bei den beiden Varianten auch zwei verschiedene Ergebnisse. Im Sequenzdiagramm von Abbildung 6.1 wurde die Methode `enterMethod()` noch nicht ausgeführt. Da die ursprüngliche Basismethode tatsächlich

noch nicht betreten wurde, ist dies auch das erwartete Ergebnis. In Abbildung 6.2 erhält man somit das falsche Ergebnis, hier würde das Ergebnis lauten, dass die Methode bereits betreten wurde.

Und auch beim Verlassen der Basismethode liefert Abbildung 6.1 das korrekte Ergebnis. Hier wird zuerst der Event erzeugt, der mitteilt, dass die Methode verlassen wurde und danach mit der Ausführung in der Callin-Bindung von `OtherTeam` fortgefahren. Das Sequenzdiagramm in Abbildung 6.2 zeigt die falsche Reihenfolge beim Verlassen der Methode.

Allgemein kann gesagt werden, dass man das erwartete Ergebnis erhält, wenn die Callin-Bindung des überwachenden Teams zuletzt ausgeführt wird. Dies ist der Fall, wenn das überwachende Team als erstes aktiviert wird. Somit ergibt sich aus diesem Spezialfall die Notwendigkeit das überwachende Team als eines der ersten Teams zu aktivieren.

Man könnte argumentieren, dass es sich hierbei nur um einen Spezialfall handelt, der nur selten auftritt und dadurch ignoriert werden kann. Ein kleines Beispiel verdeutlicht jedoch wie schnell dieser Spezialfall relevant wird. In diesem Beispiel wird die Ausführung einer Methode geloggt, die sich rekursiv selbst aufruft. Als Mechanismus zum Loggen dient ein aspektorientierter Ansatz, der die Methode per Callin-Bindung ersetzt, und bei der Ausführung der Callin-Bindung eine Log-Meldung auf der Konsole ausgibt. Die ursprüngliche Methode wird danach per Basecall ausgeführt. Da sich die Methode selbst wieder aufruft, würde dieser Mechanismus für jeden Aufruf der Methode eine Log-Meldung erzeugen. Mittels der Steuerung von Callin-Bindungen soll jetzt nur der erste Aufruf der Methode geloggt werden alle weiteren jedoch nicht. Hierbei ist es nötig zu prüfen, ob sich die Methode bereits auf dem Aufrufstapel befindet. Daraus ergibt sich, dass die Methode mittels einer weiteren Callin-Bindung, wie oben beschrieben, überwacht wird. Es tritt also genau der Spezialfall ein. Da jedoch eine Rekursion keine Seltenheit in der Softwaretechnik ist, kann der genannte Spezialfall nicht ignoriert werden. Es ergibt sich also, dass es zwingend notwendig ist das überwachende Team vor allen anderen Teams zu aktivieren um das Funktionieren der Steuerung von Callin-Bindungen sicherzustellen.

Die Aktivierung des generierten Teams geschieht im Moment nicht automatisch. Das Team muss, zusätzlich zum ursprünglichen Team, per Hand aktiviert werden. Das generierte Team kann dabei durch alle in ObjectTeams/Java verfügbaren Techniken zur Aktivierung eines Teams aktiviert werden. Da die Aktivierung eines Teams für jeden Thread separat durchgeführt werden kann, muss das generierte Team mindestens für dieselben Threads aktiviert werden, für die auch das originale Team aktiviert werden soll. Der Einfachheit halber, kann das generierte Team auch für alle Threads aktiviert werden. Dies ändert nichts am Aktivierungszustand des originalen Teams, insbesondere nicht an den

Threadbindungen. Abgesehen von der Situationsbewertung ändert sich somit auch das Laufzeitverhalten des originalen Teams nicht.

Wird das generierte Team jedoch nicht für alle Threads aktiviert für die das originale Team aktiviert ist, so findet für die Threads in denen das generierte Team nicht aktiviert ist keine Situationsbewertung statt. Das ursprüngliche Laufzeitverhalten des originalen Teams bleibt bestehen.

6.2 Markierung von relevanten Methoden mittels Annotation

Für die Umsetzung der Steuerung von Callin-Bindungen soll das Konzept der Annotationen verwendet werden. Dabei soll die Callin-Bindung annotiert werden, für die die Steuerung aktiviert werden soll. Es müssen jedoch auch die Methoden markiert werden, die sich zum Zeitpunkt der Ausführungsprüfung der Callin-Bindung auf dem Aufrufstapel befinden dürfen bzw. sich nicht auf dem Aufrufstapel befinden dürfen.

Eine Möglichkeit um diese Markierung durchzuführen, besteht darin diese Methoden mit einer weiteren Annotation zu versehen. Damit dies funktioniert, ist es jedoch erforderlich, dass der Quelltext der Methode, bzw. der Klasse die die Methode enthält, vorliegt und veränderbar ist. Dies ist jedoch nicht in jedem Fall gegeben. Somit scheidet die Möglichkeit die zu überwachenden Methoden mit einer weiteren Annotation zu versehen aus.

Viel übersichtlicher ist es, die zu überwachenden Methoden an derselben Callin-Bindung zu markieren, die auch gesteuert werden soll. Da diese Callin-Bindung sowieso schon annotiert ist, kann dieser Annotationstyp dazu benutzt werden auch die zu überwachenden Methoden zu markieren. Dieser Annotationstyp benötigt also einen Parameter der eine oder mehrere Methoden markieren kann. Damit mehrere Methoden überprüft werden können und das Ergebnis dieser Prüfung logisch zusammengefasst werden kann, ist es nötig, dass dieser Parameter eine logische Formel darstellen kann. Da die möglichen Datentypen von Annotationsparametern sehr begrenzt sind, bietet sich der Datentyp `String` an um die genannten Anforderungen zu erfüllen. Da `Class` ein weiterer möglicher Parametertyp ist, könnte man auch eine Klasse angeben, die die Formel in einer geeigneten Weise repräsentiert. Diese Lösung ist aber mit einem erhöhten Programmieraufwand für den Entwickler verbunden, da diese zusätzliche Klasse nicht generiert werden kann. Da der Aufwand für den Entwickler so minimal wie möglich gehalten werden soll, wird für den weiteren Verlauf der Arbeit der Datentyp `String` als Datentyp des Formelparameters benutzt.

Mittels des Datentyps `String` kann einfach eine logische Verknüpfung notiert werden, in der die Namen der zu überwachenden Methoden als Variablen der Formel angesehen werden können. Die Verknüpfung der Variablen geschieht innerhalb der Formel mittels logischer Operatoren. Damit diese Formel ausgewertet werden kann, ist jedoch ein zusätzlicher Formelparser nötig. Dieser wird in Kapitel 6.3 beschrieben.

In Kapitel 5.2 wurde beschrieben wie die Steuerung der Callin-Bindung funktionieren soll. Damit der benötigte Guard erzeugt werden kann, wird die Annotation `@Guard()` benötigt. Da in einem Team mehrere Callin-Bindungen gesteuert werden sollen, müssen die erzeugten Guards mit eindeutigen Kennungen versehen werden. Dazu reicht ein Parameter vom Typ **int** aus. Listing 6.2 zeigt wie die Implementierung des Annotationstyps `Guard` aussehen muss.

```
public @interface Guard {
        int value();
}
```

Listing 6.2: Der Annotationstyp Guard

Damit die Annotation zur Steuerung der Callin-Bindung auf diese Kennung zugreifen kann, benötigt sie auch einen Parameter vom Typ **int**. Dieser markiert für welchen Guard die Steuerung aktiviert werden soll. Listing 6.3 zeigt den vollständigen Annotationstyp zur Steuerung einer Callin-Bindung.

```
public @interface CFlowGuard {
        int guardID();
        String cflowFormula();
}
```

Listing 6.3: Annotationstyp zur Steuerung von Callin-Bindungen

Wie bereits erwähnt, besteht die Formel aus logischen Operationen und Variablen die von Methodennamen repräsentiert werden. Die unterstützten logischen Operationen sind `!` (Negation), `&&` (logisches Und) und `||` (logisches Oder). Aus den Methodennamen müssen die Rollen erzeugt werden, die die angegebenen Methoden überwachen. Somit muss zusätzlich zum Namen der Methode auch der voll

qualifizierte Klassenname angegeben werden. Der Klassenname wird dabei für die **playedBy** Beziehung genutzt und bildet somit die Basisklasse einer überwachenden Rolle. Der Methodenname wird genutzt um eine Callin-Bindung zu generieren, die die angegebene Methode ersetzt und die in Kapitel 6.1 beschriebenen Events beim Betreten und Verlassen der Methode erzeugt. Wie bei einer Callin-Bindung reicht es wenn nur der Methodenname angegeben wird, sofern dieser eindeutig ist. Sollte der Methodenname nicht eindeutig sein, müssen zusätzlich der Rückgabetyp und die Parameter der Methode angegeben werden. Listing 6.4 zeigt ein Beispiel für den Einsatz der Annotationen.

```
public class DemoTeam {
        protected class FirstRole playedBy SomeClass {
                @Guard(1)
                @CFlowGuard(guardID = 1, cflowFormula =
                        "test.SomeClass.demoMethod &&" +
                        "! test.SomeClass.otherMethod")
                callinMethod <- replace demoMethod;

                callin void callinMethod() {…}
        }

        protected class SecondRole playedBy OtherClass {
                @Guard(2)
                @CFlowGuard(guardID = 2, cflowFormula =
                        "void test.SomeClass.otherMethod(String)")
                callinMethod <- replace aMethod;

                callin void callinMethod() {…}
        }
}
```

Listing 6.4: Anwendungsbeispiel der Annotationen

Stehen die Annotationen `@Guard(..)` und `@CFlowGuard(..)` direkt nebeneinander und somit an der selben Callin-Bindung, so ist der Parameter `guardID` der Annotation `@CFlowGuard` redundant und könnte weggelassen werden. Dann ist es notwendig, dass während der Annotationsverarbeitung die Kennung aus der `@Guard(..)` Annotation

gelesen wird und für die `@CFlowAnnotation(..)` übernommen wird. Theoretisch ist es sogar möglich die Kennung in der `@Guard(..)` Annotation weg zu lassen. Dann muss der Compiler eine Kennung generieren. Während der Annotationsverarbeitung muss dann die gleiche Kennung generiert werden. Dadurch würde eine indirekte Kommunikation zwischen Compiler und Annotationsverarbeitung entstehen. Da die `@CFlowGuard(..)` Annotation sich aber nicht auf eine Callin-Bindung bezieht sondern auf einen Guard-Kennung, ist es nicht notwendig diese Annotation an eine Callin-Bindung zu schreiben. Sie kann auch an jedes andere annotierbare Element im selben Team geschrieben werden. Dann ist es jedoch zwingend notwendig die Guard-Kennung in der `@CFlowAnnotation(..)` anzugeben. Aus diesem Grund wird die `guardID` als Parameter der `@CFlowGuard(..)` Annotation benötigt.

Anhand des Beispiels in Listing 6.4 lassen sich weitere Eigenschaften der Funktionsweise der Steuerung von Callin-Bindungen zeigen. Durch die Annotation `@Guard` muss durch den Compiler eine Team-Level-Methode erzeugt werden, die später durch eine Callin-Bindung einer erzeugten Rolle ersetzt werden kann. Damit dies zuverlässig funktioniert, muss der Name der Methode für die verschiedenen Kennungen der Guards unterschiedlich sein. Der einfachste Weg um dies zu gewährleisten ist, dass die Kennung des Guards in den Namen der Methode aufgenommen wird. Somit müsste die Annotation `@Guard(1)` eine Methode mit der Signatur **`protected boolean`** `guard1()` erzeugen.

Da diese Methoden auf Ebene des Teams erzeugt werden, muss die Kennung nicht nur eindeutig innerhalb der Rolle sondern auch innerhalb des umschließenden Teams sein.

Ein weiterer Punkt der in dem Beispiel von Listing 6.4 zu sehen ist, ist dass die Namen der Parameter der zu überwachenden Methoden nicht angegeben werden müssen. Im Beispiel wurde nur "`void test.SomeClass.oterMethod(String)`" angegeben. Es ist jedoch auch möglich den Namen des Parameters anzugeben. Dadurch kann die Signatur der Methode aus dem originalen Quelltext durch Kopieren und Einfügen als Parameter in die Annotation übernommen werden.

Da zur Abgrenzung von Parametern von Methoden runde Klammern benutzt werden, werden zur Klammerung einzelner Teile einer Formel eckige Klammern verwendet.

6.3 Der Formelparser

Da die Formeln die hier verwendet werden einfach und überschaubar sind, wird hier nur ein einfacher, dafür aber intuitiver und leicht erweiterbarer Formelparser benutzt.

Der erste Schritt beim Parsen von Formeln ist die vollständige Übersetzung der Formel in Objekte vom Typ `Token`. Der Einfachheit halber werden dabei auch Klammern in Objekte übersetzt. Da Klammern aber nicht Teil des Ergebnisses sein dürfen, implementieren sie nur die Schnittstelle `Token`. Alle Elemente die Teil des Ergebnisses sein können, implementieren dagegen die Schnittstelle `FunctionPart`. Das Ergebnis dieser Übersetzung ist eine Liste von `Token`, die in den nächsten Schritten weiterverarbeitet wird.

Die bisher implementierten Elemente, die beim Parsen von Formeln verwendet werden können, sind im Klassendiagramm von Abbildung 6.3 gezeigt. Damit das Ergebnis des Parsers nur ein einzelnes Objekt vom Typ `FunctionPart` ist, müssen die übersetzten Objekte in eine Baumstruktur gebracht werden. Dazu wird das Entwurfsmuster Composite genutzt, das eben diese Baumstruktur möglich macht. Anhand dieses Klassendiagramms erkennt man auch, wie zusätzliche Operationen implementiert werden können, sofern sie benötigt werden. Eine Operation hat dabei ein oder zwei Objekte vom Typ `FunctionPart`, die die Argumente der abgebildeten Funktion bilden.

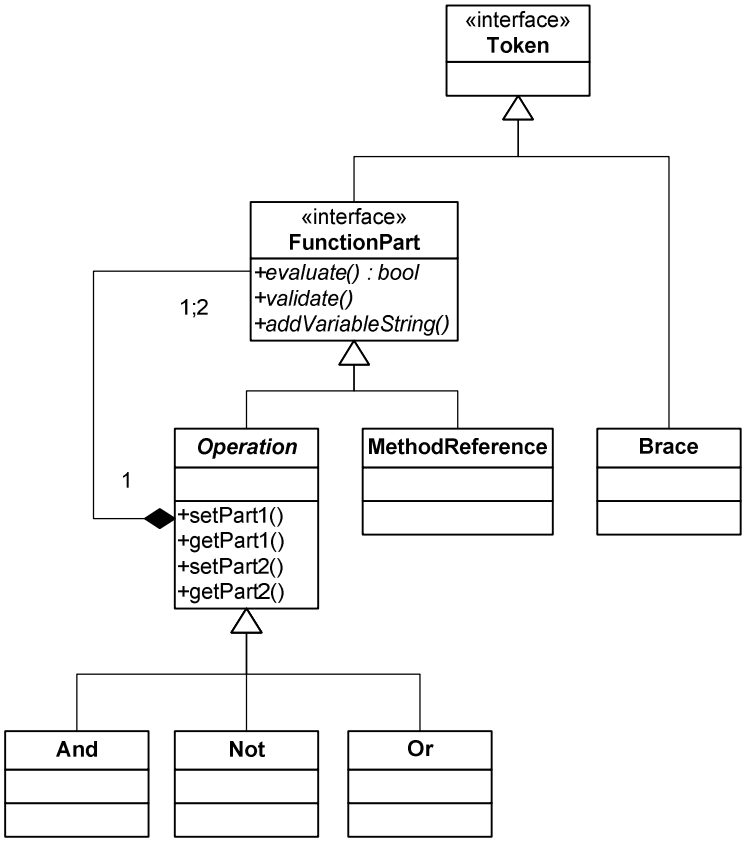

Abbildung 6.3: Elemente des Formelparsers

Im zweiten Schritt wird die Auflistung von Token reduziert. Dabei wird zuerst nach Klammerpaaren gesucht, und dann der Term innerhalb der Klammer zu einem einzelnen Objekt reduziert. Die dabei aus der Auflistung entfernten FunctionParts werden bei der Reduzierung zu Argumenten der Funktionen. Da die Klammern nach der Reduzierung nicht mehr benötigt werden, werden sie während des Prozesses entfernt. Das Ergebnis des zweiten Schrittes ist ein Baum aus Operationen und Referenzen auf Methoden, dessen Blätter immer Referenzen auf Methoden sind. Dabei gilt, dass das Ergebnis des Reduzierens immer ein einzelnes Objekt sein muss. Ist dies nicht der Fall, ist die eingegebene Formel ungültig. Abbildung 6.4 zeigt beispielhaft solch einen Baum für die Formel a && ! [b || c].

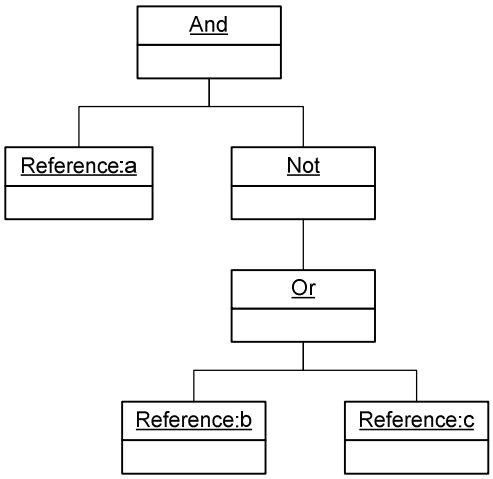

Abbildung 6.4: Objektbaum nach dem Parsen

Als letzter Schritt des Parsens wird die Formel validiert. Dabei wird geprüft, ob für jede Operation die benötigte Menge an Argumenten vorhanden ist. Dazu wird die Validierung beim Wurzelelement gestartet und an die jeweiligen Kinder des Knoten weitergereicht. Wird dabei eine Referenz auf eine Methode validiert, so ist diese per Definition valide. Ein Seiteneffekt dieser Prüfung besteht darin, dass sichergestellt wird, dass jedes Blatt zwingend eine Referenz ist.

Die Auswertung einer Formel geschieht zur Laufzeit der Applikation, nämlich genau dann, wenn geprüft wird ob eine Callin-Bindung ausgeführt werden soll. Die Auswertung geht einen ähnlichen Weg wie die Validierung einer Formel. Gestartet wird die Auswertung beim Wurzelelement. Wird eine Referenz auf eine Methode getroffen, wird der Wert für die repräsentiert Variable aus einem VariableHolder ausgelesen. Wird bei der Auswertung der Formel eine Operation ausgewertet, werden zuerst alle Kinder der Operation ausgewertet und danach die Operation ausgeführt. Listing 6.5 zeigt die Schnittstelle VariableHolder, die benötigt wird um den Wert einer Variablen zu ermitteln. Der Wert einer Variablen

kann hierbei **true** oder **false** sein und gibt an, ob sich die entsprechende Methode auf dem Aufrufstapel befindet.

```
public interface VariableHolder {
        boolean evaluate(String methodName);
}
```

Listing 6.5: Schnittstelle VariableHolder

6.4 Ausführungsprüfung für überwachte Methoden

In Kapitel 6.1 wurde erläutert wie relevante Methoden überwacht werden und auf dem selbst implementierten Aufrufstapel organisiert werden. Dabei blieb jedoch die Funktionsweise des Aufrufstapels außen vor. In den folgenden Abschnitten wird nun diese Funktionsweise erklärt.

Da nicht alle Methoden jeder in der Applikation geladenen Klasse überwacht werden können, wird der Aufrufstapel nicht vollständig aufgebaut. Vielmehr enthält er nur einen Ausschnitt aus dem vollständigen Aufrufstapel, der nur die relevanten Methoden beinhaltet. Für die Situationsanalyse anhand des Beispiels der Betrachtung des Kontrollflusses ist es auch gar nicht nötig den vollständigen Aufrufstapel aufzubauen, da nur interessant ist ob eine relevante Methode auf dem Aufrufstapel ist, nicht an welcher Stelle des Aufrufstapels sie sich befindet.

Überwachte Methoden feuern einen Event beim Betreten und Verlassen der Methode. Der Aufrufstapel muss nun diese Events verarbeiten. Dazu wird in einer `Map` die Zuordnung von Methodenname zu Aufrufzustand gespeichert. Der Aufrufzustand in diesem Beispiel besteht eigentlich nur aus einem logischen Wert, nämlich die Methode wird ausgeführt oder die Methode wird nicht ausgeführt. Der Event beim Betreten einer Methode würde diesen Zustand auf wahr setzen, der Event beim Verlassen auf falsch.

Leider reicht diese einfache Art der Zustandsspeicherung nicht aus. Wenn eine Methode per direkter oder indirekter Rekursion erneut aufgerufen wird, so wird der Zustand beim wiederholten Betreten der Methode nicht verändert, da der gespeicherte Zustand schon auf wahr, die Methode wird gerade ausgeführt, steht. Beim ersten Verlassen der Methode wird dieser Zustand auf falsch gesetzt. Da die Methode durch den rekursiven Aufruf aber noch auf dem Aufrufstapel ist, stimmt die

gespeicherte Aussage jetzt nicht mehr. Es reicht also nicht aus, den Ausführungszustand einer Methode mit einem logischen Wert zu speichern.

Die Lösung dieses Problem besteht darin, den Ausführungszustand als natürliche Zahl zu speichern. Jedes Betreten einer Methode erhöht den Zähler für diese Methode um eins. Das Verlassen einer Methode verringert ihn wieder um eins. Da eine Methode zuerst betreten werden muss bevor sie verlassen werden kann, ist der Zähler für diese Methode mindestens null. Wenn er jedoch null ist, so wird diese Methode gerade nicht ausgeführt.

Werden die überwachten Methoden von mehreren Threads ausgeführt, kann es dazu kommen, dass der Zähler einer Methode nicht null ist, obwohl die Methode gar nicht auf dem Aufrufstapel des prüfenden Threads ist. Die Methode wird dann gerade in einem anderen Thread ausgeführt. Dies würde zu einer fehlerhaften Überprüfung des Ausführungszustandes einer Methode führen. Damit sichergestellt ist, dass die Ausführungsprüfung auch in einer Umgebung mit mehreren Threads zuverlässig funktioniert, muss die `Map`, die die Zähler speichert, eine Thread lokale Variable sein, damit jeder Thread eine eigene Instanz dieser `Map` erhält. Dadurch wird sichergestellt, dass ein Thread der eine Ausführungsprüfung einer Methode macht auch nur die Methoden auf dem Ausführungsstapel sieht, die im Kontrollfluss dieses Threads ausgeführt werden.

Damit für eine Callin-Bindung geprüft werden kann ob sie ausgeführt werden soll, muss die Formel, die in der Annotation angegeben wurde, ausgewertet werden. Damit diese Formel nicht bei jeder Prüfung erneut geparst werden muss, wird das Parsen einmalig im Konstruktor des Teams gemacht, das die Callin-Bindung umschließt, und danach in einer weiteren `Map` gespeichert. Da in dieser `Map` nur die Zuordnung von der Kennung des Guards zu einer Formel gespeichert wird, und dies nur innerhalb des Teams geschieht in dem die Annotation steht, muss diese `Map` nicht threadlokal sein. Bei der Ausführungsprüfung wird das Objekt, das die Formel repräsentiert aus dieser `Map` geholt und ausgewertet. Trifft diese Auswertung auf eine Variable, die ja die Methoden innerhalb einer Formel repräsentieren, so wird geprüft ob die entsprechende Methode gerade ausgeführt wird. Dazu muss nur überprüft werden, ob der Zähler für diese Methode, für den aktuellen Thread, größer als null ist.

Listing 6.6 zeigt die Implementierung der Klasse, die den Aufrufstapel organisiert. Da die Methoden später innerhalb von Rollenklassen aufgerufen werden sollen, ist diese Klasse als abstraktes Team implementiert. Damit der Wert einer Variablen einfach ausgelesen werden kann, implementiert das abstrakte Team die Schnittstelle `VariableHolder`.

```
public abstract team class AbstractCFlowGuardTeam
        implements VariableHolder {

        private final Map<Integer, FunctionPart> restrictionMap =
                new HashMap<Integer, FunctionPart>();
        private final ThreadLocal<Map<String, Integer>> methodMap =
                new ThreadLocal<Map<String, Integer>>();

        protected void addRestriction(int guardID, String restriction) {
                FunctionPart part = RestrictionParser.parse(restriction);
                restrictionMap.put(guardID, part);
        }

        protected void enterMethod(String method) {
                methodMap.get().put(method, getRunCount(method) + 1);
        }

        protected void leaveMethod(String method) {
                int runCount = getRunCount(method);
                if (runCount > 1)
                        methodMap.get().put(method, runCount – 1);
                else if (runCount == 1)
                        methodMap.get().remove(method);
        }

        protected boolean checkRestriction(int guardID) {
                FunctionPart part = restrictionMap.get(guardID);
                if (part == null)
                        return true;
                else
                        return part.evaluate(this);
        }

        private int getRunCount(String method) {
                Integer result = methodMap.get().get(method);
                if (result == null)
                        return 0;
                else
```

```
                return result;
        }

        public boolean evaluate(String variable) {
                return getRunCount(variable) > 0;
        }
}
```

Listing 6.6: AbstractCFlowGuardTeam

6.5 AnnotationProcessorFactory und AnnotationProcessor

Damit aus den Annotationen, die die zu steuernden Callin-Bindungen markieren, zusätzlicher Code erzeugt werden kann, werden noch weiteren Komponenten benötigt. Zum Erzeugen des Codes wird ein AnnotationProcessor benutzt. Die AnnotationProcessorFactory liefert eine Instanz des AnnotationProcessors. Und damit das Annotation Processing Tool, das die Verarbeitung der Annotationen übernimmt, eine AnnotationProcessorFactory instanziieren kann, muss diese zusammen mit allen benötigten Klassen in einer Jar Dateien verpackt werden, die außerdem eine Service Datei enthalten muss, die den Klassenname der AnnotationProcessorFactory enthält.

6.5.1 CFlowAnnotationProcessorFactory

Da die Annotationsverarbeitung für jede Klasse separat gemacht werden muss, liefert die CFlowAnnotationProcessorFactory für jede Anfrage eine neue Instanz eines CFlowAnnotationProcessors. Dadurch ist die Klasse CFlowAnnotationProcessorFactory sehr einfach aufgebaut. Die Factory kann nur AnnotationProcessors für den oben beschriebenen CFlowGuard Annotationstyp liefern. Benutzerspezifische Optionen werden hierbei nicht berücksichtigt. Listing 6.7 zeigt die Klasse CFlowAnnotationProcessorFactory.

```
public class CFlowAnnotationProcessorFactory
        extends AnnotationProcessorFactory {

        public AnnotationProcessor getProcessorFor(
                        Set<AnnotationTypeDeclaration> annotationTypes,
                        AnnotationProcessorEnvironment environment) {
                return new CFlowAnnotationProcessor(environment);
        }

        public Collection<String> supportedAnnotationTypes() {
                return Collections.singleton(CFlowGuard.class.getName());
        }

        public Collection<String> supportedOptions() {
                return Collections.emptySet();
        }
}
```

Listing 6.7: CFlowAnnotationProcessorFactory

6.5.2 CFlowAnnotationProcessor

Der CFlowAnnotationProcessor bearbeitet alle Annotationen vom Typ CFlowGuard. Eine Instanz dieses Processors bearbeitet genau eine Klasse. Dadurch wird es nötig, dass für jede Klasse die die Annotation verwendet auch ein überwachendes Team erzeugt wird. Man könnte zwar die zu überwachenden Methoden in einer statischen Liste sammeln und erst nach Abschluss der Annotationsverarbeitung ein Team erstellen. Es ist jedoch nicht klar, wann die Annotationsverarbeitung für alle Klassen vollständig abgeschlossen ist. Dadurch ist einerseits unklar wann das überwachende Team erstellt werden soll. Andererseits ist auch nicht geklärt, wann die Liste wieder geleert werden muss. Somit kann es passieren, dass beim nächsten Durchlauf der Annotationsverarbeitung noch Methodennamen in der Liste enthalten sind, für die die Annotation beim Bearbeiten der Klasse entfernt wurde. Aus diesem Grund kann keine statische Liste benutzt werden.

Die die Verarbeitung des CFlowGuard Annotationstyps erfolgt innerhalb des CFlowAnnotationProcessors in vier Schritten. Zuerst werden Kennungen aller @Guard Annotationen ermittelt. Dies ist nötig um

sicherzustellen, dass in den @CFlowGuard Annotationen auch nur Kennungen benutzt werden, die tatsächlich existieren. Verwendet eine Annotation eine Kennung die nicht existiert, wird dies per Fehlermeldung mitgeteilt. Die entsprechende Annotation wird nicht in die Verarbeitung einbezogen.

Der zweite Schritt ist nicht zwingend für die Verarbeitung der Annotationen notwendig. Für die Benutzbarkeit des Programms ist er jedoch sehr hilfreich. In diesem Schritt werden alle @Guard Annotationen markiert, deren Kennung von keiner @CFlowGuard Annotation verwendet wird. Dadurch werden dem Benutzer unbenutzte Kennungen sichtbar gemacht, die auf potentielle Fehler hinweisen.

Im dritten Schritt wird eine Rollenklasse generiert, die die Methoden ersetzt, die vom Compiler bei der Verarbeitung der @Guard Annotation erzeugt werden. Durch die Ersetzung dieser Methoden kann die Ausführung einer Callin-Bindung gesteuert werden indem der Rückgabewert der Methode verändert wird.

Zum Abschluss werden im letzten Schritt alle Rollen erstellt, deren Callin-Bindungen die Methoden überwachen, die in der @CFlowGuard Annotation angegeben sind. Dabei werden zuerst alle Methoden gesammelt und nach der angegebenen Klasse sortiert. Dadurch wird für jede benutzte Basisklasse genau eine Rollenklasse erzeugt. Verweisen dabei zwei Annotationen auf dieselbe Methode wird nur eine Callin-Bindung erzeugt die diese Methode überwacht.

Treten während der Annotationsverarbeitung Fehler auf, werden diese dem Benutzer innerhalb der Entwicklungsumgebung angezeigt. Dazu kann einer Instanz der Klasse com.sun.mirror.apt.Messager, die vom AnnotationProcessorEnvironment bereitgestellt wird, die Fehlermeldung übergeben werden. Das AnnotationProcessorEnvironment wird dem CFlowAnnotationProcessor im Konstruktor übergeben. Listing 6.8 zeigt den rudimentären Aufbau der Klasse. Die vollständige Implementierung ist im praktischen Teil der Arbeit zu finden.

```
public class CFlowAnnotationProcessor implements AnnotationProcessor {
        private final AnnotationProcessorEnvironment environment;

        public CFlowAnnotationProcessor(
                        AnnotationProcessorEnvironment environment) {
                this.environment = environment;
        }
```

```
        public void process() {
                Map createdIDs = processGuards();
                Collection annotatedTypes = obtainAnnotatedTypes(
                                CFlowGuard.class.getName());
                Collection usedIDs  = processCFlows(createdIDs,
                                annotatedTypes);
                checkIDs(createdIDs, usedIDs);
                Collection restrictions = processRestrictions(annotatedTypes);

                createTeam(usedIDs, restrictions);
        }

        private Map processGuards() {…}

        private Collection obtainAnnotatedTypes(String className) {…}

        private Collection processCFlows(Map createdIDs,
                        Collection annotatedTypes) {…}

        private void checkIDs(Map createdIDs, Collection usedIDs) {…}

        private Collection processRestrictions(
                        Collection annotatedTypes) {…}

        private void createTeam(
                        Collection usedIDs, Collection restrictions) {…}
}
```

Listing 6.8: CFlowAnnotationProcessor

6.6 Zusammenspiel der Komponenten

In den vorangegangenen Abschnitten wurde erläutert welche zusätzlichen Komponenten benötigt werden, um Code erzeugen zu können. In diesem Abschnitt wird das Zusammenspiel der gesamten Komponenten und somit der vollständige Ablauf vom annotierten Quellcode zum zusätzlich erzeugten und kompilierten Code gezeigt.

6.6.1 Vorbereitungen

Damit überhaupt Code erzeugt werden kann, muss die Annotations-Verarbeitung aktiviert werden. Die geschieht in Eclipse in den Projekteigenschaften. In der Kategorie Java Compiler findet sich dort der Eintrag Annotation Processing. Dies muss für das aktuelle Projekt aktiviert werden. Zusätzlich muss im Unterpunkt Factory Path die Jar Datei eingetragen werden, die die CFlowAnnotationProcessorFactory enthält. Durch diese Einstellungen findet Das Annotation Processing Tool die benötigte Factory für den CFlowGuard Annotationstyp.

Damit der Annotationstyp im Quellcode benutzt werden kann, muss die eben genannte Jar Datei auch in den Buildpath aufgenommen werden. Dies geschieht ebenfalls in den Projekteigenschaften in der Kategorie Java Build Path. Auf dem Reiter Libraries kann die Jar Datei zum Buildpath hinzugefügt werden. Da der CFlowGuard Annotationstyp und der Guard Annotationstyp in dieser Jar Datei enthalten sind, kann der Compiler diese nun verarbeiten.

6.6.2 Quellcode

Da die Annotationstypen Guard und CFlowGuard nur an Callin-Bindungen erlaubt sein sollen, muss die Klasse in der diese Annotationstypen benutzt werden als Team implementiert sein. An einer Callin-Bindung können nun beide Annotationstypen als Modifizierer benutzt werden. Da der Annotationstyp Guard nur die Kennung als Parameter besitzt, muss dieser nicht als Schlüssel-Wert-Paar angegeben werden sondern der Wert kann ohne Schlüssel aufgeschrieben werden. Der Annotationstyp CFlowGuard besitzt 2 Parameter. Hier müssen die Kennung des Guards (guardID) und die Einschränkung für diese Callin-Bindung (cflowFormula) als Schlüssel-Wert-Paar angegeben werden. Listing 6.9 zeigt hierfür eine beispielhafte Implementierung.

```
public team class DemoTeam {
    protected class DemoRole playedBy demo.BaseClass {
        @Guard(1)
        @CFlowGuard(guardID = 1,
                cflowFormula = "! demo.BaseClass.aMethod")
        logExecution <- after demoMethod;
```

```
            private void logExecution() {
                System.out.println("Method invoked");
            }
        }
}
```

Listing 6.9: Implementierungsbeispiel DemoTeam

6.6.3 Kompilieren

Beim Kompilieren dieses Teams muss der Compiler die Annotation `@Guard(1)` erkennen und daraus den benötigten Guard für die Callin-Bindung und auf Team-Level-Ebene eine Methode erzeugen, die im Guard der Callin-Bindung überprüft werden kann. Da die Kennung der `@Guard(1)` Annotation in die zu erzeugende Methode einfließen muss, hat die Methode die Signatur **private boolean** `gurad1()`. An der Callin-Bindung muss ein Guard erzeugt werden, der eben diese erzeugte Methode aufruft. Sollte die Callin-Bindung bereits einen Guard besitzen, so muss dieser so angepasst werden, dass das Ergebnis des ursprünglichen Guards mit dem Ergebnis der erzeugten Methode durch ein logisches Und verknüpft wird.

Damit allein diese Veränderung keine Änderung am Laufzeitverhalten des ursprünglichen Codes hervorruft, liefert die erzeugte Methode nur den Wert **true** als Ergebnis zurück. Zum Zeitpunkt der Erstellung dieser Arbeit wurde der Compiler noch nicht erweitert, so dass die genannten Änderungen noch per Hand gemacht werden müssen. Die Erweiterung des Compilers ist nicht Bestandteil dieser Arbeit. Listing 6.10 zeigt den veränderten Quellcode aus Listing 6.9.

```
public team class DemoTeam {
    private boolean guard1() {
        return true;
    }

    protected class DemoRole playedBy demo.BaseClass {
        @Guard(1)
        @CFlowGuard(guardID = 1,
            cflowFormula = "! demo.BaseClass.aMethod")
        logExecution <- after demoMethod when (guard1());
```

```
            private void logExecution() {
                System.out.println("Method invoked");
            }
        }
}
```

Listing 6.10: DemoTeam nach Kompilieren

6.6.4 AnnotationProcessing

Da in der Klasse DemoTeam eine Annotation gefunden wurde, wird nach dem Kompilieren das Annotation Processing Tool gestartet. Dieses sucht mittels aller bekannten AnnotationProcessorFactories nach einem AnnotationProcessor der die enthaltene Annotation verarbeiten kann. Für die Klasse DemoTeam wir hier der CFlowAnnotationProcessor gestartet. Dieser erzeugt ein Team, das zwei Rollen enthält. Die erste Rolle ersetzt mittels einer Callin-Bindung die in der Klasse DemoTeam erzeugte Methode **private boolean** guard1() und steuert somit die Ausführung der annotierten Callin-Bindung. Die zweite Rolle überwacht ebenfalls mittels Callin-Bindung die Ausführung der in der CFlowGuard Annotation angegebenen Methode und organisiert somit den relevanten Teil des Aufrufstapels. Dadurch bildet dieses generierte Team ebenfalls den in Kapitel 6.1.2 eingeführten Supervisor, der die Organisation des Aufrufstapels übernimmt. Listing 6.11 zeigt den Code der vom CFlowAnnotationProcessor erstellt wird.

```
public team class CFlow_Generated_DemoTeam
            extends AbstractCFlowGuardTeam {

    public CFlow_Generated_DemoTeam() {
        addRestriction(1, "! demo.BaseClass.aMethod");
    }
    protected class CreatedRole playedBy demo.DemoTeam {
        replaceGuard_gurad1 <- replace guard1;
        callin boolean replaceGuard_guard1() {
            return checkExecution(1);
        }
    }
    protected class Generated_Role_demoBaseClass
            playedBy demo.BaseClass {
```

```
                replaceaMethod <- replace aMethod;

                callin void replaceaMethod() {
                        enterMethod("demo.BaseClass.aMethod");
                        base.replaceaMethod();
                        finally {
                                leaveMethod("demo.BaseClass.aMethod");
                        }
                }
        }
}
```

Listing 6.11: Aus DemoTeam erzeugter Code

6.6.5 Laufzeit

Die Steuerung der Callin-Bindung muss nun noch aktiviert werden. Dies geschieht über die Aktivierung des generierten Teams. Damit auch rekursive Methoden zuverlässig überwacht werden können, muss das erzeugt Team als eines der ersten aktiviert werden. Die Aktivierung des Teams kann im Code über den Aufruf **new** `CFlow_Generated_DemoTeam().activate()` oder aber durch hinzufügen des Teams, zu den beim Start der Applikation zu startenden Teams, realisiert werden.

Beim Erzeugen einer Instanz des generierten Teams werden alle Einschränkungen von Callin-Bindungen durch den Formelparser geparst und die erzeugten Formelobjekte gespeichert. Die geschieht mittels des Aufrufs `addRestriction()` im Konstruktor des Teams.

Wird eine, durch das Team überwachte, Methode ausgeführt, setzt die Callin-Bindung mittels des Aufrufs `enterMethod()` den Ausführungszähler für diese Methode um eins hoch. Sofern dieser Zähler vorher nicht gesetzt oder null war, landet die Methode damit auf dem Aufrufstapel. Danach wird mittels Basecall die ursprüngliche Methode aufgerufen. Der Aufruf von `leaveMethod()` setzt den Zähler wieder um eins runter. Sollte dieser dabei null werden, wird die Methode wieder vom Aufrufstapel entfernt.

Vor der Ausführung der ursprünglich annotierten Callin-Bindung wird der Guard ausgewertet. Da dieser mittels Callin-Bindung ersetzt ist, wird die Prüfung innerhalb des generierten Teams fortgesetzt. Liegt dort eine Einschränkung für diese Callin-Bindung vor, so wird die Formel für diese Einschränkung ausgewertet und das Ergebnis als Ergebnis der

Ausführungsprüfung zurückgegeben. Wird dabei der logische Wert wahr zurückgegeben, wird die Callin-Bindung ohne Einschränkung ausgeführt. Wird der Wert falsch zurückgeliefert, verhindert der Guard die Ausführung der Callin-Bindung.

7 Zusätzlicher Datenfluss bei der Bewertung des Kontrollflusses

Bei der bisherigen Betrachtung des Kontrollflusses wurde nur darauf geachtet, ob sich bestimmte Methoden auf dem Aufrufstapel befinden oder nicht. Des Weiteren könnte jedoch interessant sein, mit welchen Parametern eine Methode ausgeführt wurde die sich auf dem Aufrufstapel befindet. Da hierbei die Parameter an einer Stelle verfügbar gemacht werden, an der sie eigentlich nicht mehr zur Verfügung stehen, entsteht ein zusätzlicher Datenfluss. Dieser zusätzliche Datenfluss wird auch Wurmlocheffekt genannt.

Beim Ausführen einer Methode können die Parameter der Methode zwar an weitere Methoden übergeben werden, in der Regel sind die Parameter jedoch nicht in allen Methoden auf dem Aufrufstapel verfügbar. Hier setzt der Wurmlocheffekt an. Mittels dieses Effektes können Daten von der zuerst ausgeführten Methode an Methoden geleitet werden, die sich weiter oben im Aufrufstapel befinden, die also später ausgeführt wurden.

Um diesen zusätzlichen Datenkanal sichtbar zu machen, zeigen die Abbildungen 7.1 und 7.2 in einfaches Beispiel. Abbildung 7.1 zeigt ein Sequenzdiagramm in dem kein zusätzlicher Datenfluss entsteht. Die Parameter der ursprünglich ausgeführten Methode `demoMethod(String)` sind in der Methode `handleData()` nicht mehr verfügbar.

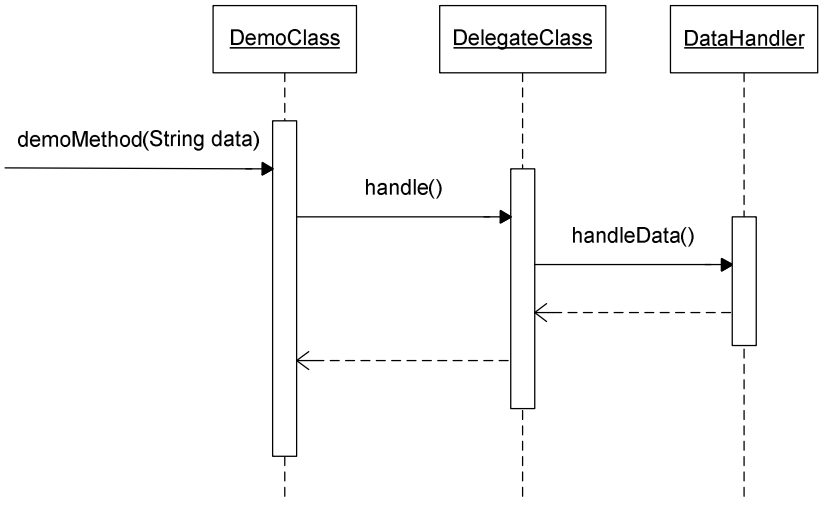

Abbildung 7.1: Sequenzdiagramm ohne Wurmlocheffekt

Wenn der Quelltext der Klassen vorhanden und veränderbar ist, würde man die Daten den Methoden handle() und handleData() als Parameter mitgeben. Steht der Quellcode jedoch nicht zu Verfügung, wird ein anderer Weg benötigt um die Daten in der Methode handleData() verfügbar zu machen. Abbildung 7.2 zeigt wo dieser zusätzliche Datenfluss benötigt wird. Das Diagramm hat dabei keinen Anspruch auf Korrektheit gemäß der UML-Spezifikation sondern soll nur verdeutlichen welche Daten transferiert werden sollen.

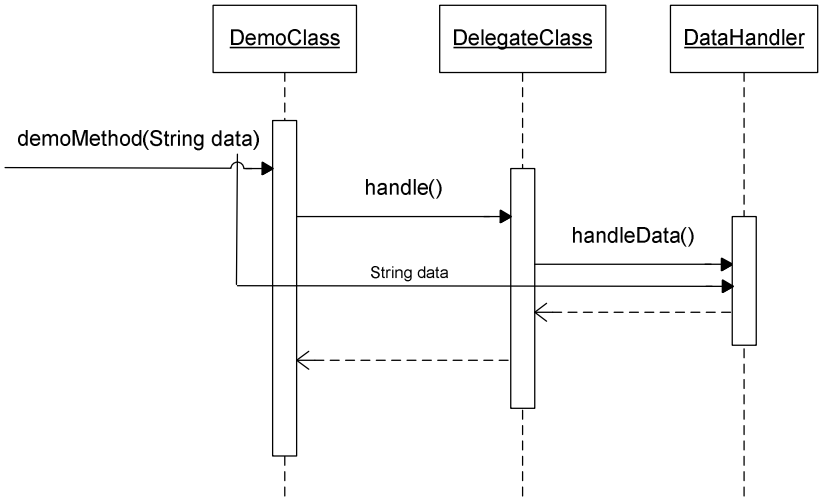

Abbildung 7.2: Sequenzdiagramm mit Wurmlocheffekt

Damit die Parameter der zuerst ausgeführten Methode später verfügbar sind, müssen sie irgendwo gespeichert werden. Um sie zu speichern kann die Methode mittels Callin-Bindung ersetzt werden. Bei der Ausführung der Callin-Bindung werden zuerst die Parameter der Methode gespeichert, dann wird per Basecall die ursprüngliche Methode ausgeführt und danach werden die Parameter wieder aus dem Speicher entfernt. Würden die Parameter nicht aus dem Speicher entfernt werden, wären sie zwar auch später noch verfügbar, für den Speicherverbrauch der Anwendung wäre dieses Verfahren jedoch nicht sehr sinnvoll. Da durch das Speichern der Parameter noch zusätzliche Referenzen auf sie gehalten werden, kann der Garbagecollector den Speicher, den die Parameter verbrauchen, nicht wieder freigeben. Lisitng 7.1 zeigt wie das Team aussehen muss, das für die Ersetzung der Methode benötigt wird.

```
public team class DataHandlerTeam extends AbstractDataHandlerTeam {
        protected class DataHandlerRole playedBy DemoClass {
                void saveParams(String data) <-
                        replace void demoMethod(String data);

                callin void saveParams(String data) {
                        saveParams(data);
                        base.saveParams(data);
                        finally {
                                removeParams();
                        }
                }
        }
}
```

Listing 7.1: DataHandlerTeam zum Speichern von Parametern

Die Methoden saveParams(..) und removeParams() sind in der abstrakten Oberklasse implementiert und sorgen für das Speichern und Löschen der Parameter. Die Methode saveParams(..) macht dabei Gebrauch von zwei in Java 5 eingeführten Funktionen. Zum einen besitzt die Methode eine variable Argumentliste. Dadurch müssen die zu speichernden Parameter nicht erst in ein Feld von Objekten übertragen werden, da dies bereits automatisch erledigt wird. Außerdem findet auch Autoboxing statt. Dadurch werden primitive Datentypen in ihre Objektrepräsentation verpackt und können somit an die Methode saveParams(..) übergeben und gespeichert werden. Aus den genannten Punkten ergibt sich **protected void** saveParams(Object… args) als Signatur der Methode.

In Listing 7.1 sieht man recht deutlich die Ähnlichkeit zum Ansatz der kontrollflussbasierten Aktivierung von Aspekten. Wie in Listing 6.1 wird auch in Listing 7.1 bei der Ausführung der Callin-Bindung um den Basecall herum noch weiterer Code aufgerufen. In Listing 6.1 wurden diese Aufrufe benötigt um die ersetzte Methode auf den selbst organisierten Aufrufstapel zu packen und wieder zu entfernen. Hier werden die Methoden benutzt um die Parameter zu speichern und zu löschen. Da sich die Ansätze sehr ähnlich sind, liegt es nahe sie zu vereinen und somit die Parameter der Methoden auf dem Aufrufstapel bei der Auswertung des Kontrollflusses verfügbar zu machen.

Damit die Parameter gespeichert werden können, muss die Signatur der Methode enterMethod(..) angepasst werden. Zusätzlich zum Parameter

der den Name der aufgerufenen Methode repräsentiert, wird die Signatur der Methode um die variable Argumentliste erweitert. Für Methoden die keine Parameter haben, bleibt die alte Methode enterMethod(String methodName) bestehen. Werden beim Aufbauen des Aufrufstapels die Parameter einer Methode nicht in der @CFlowGuard Annotation angegeben, können die Parameter nicht ermittelt werden. In diesem Fall wird der Aufruf der Methode im Aufrufstapel gespeichert ohne die Parameter zu speichern. Dafür wird eine neue Methode namens enterMethodIgnoreParams(String methodName) implementiert.

Da eine Methode per Rekursion mehrfach aufgerufen werden kann, müssen die Parameter der Methode für jeden Aufruf separat gespeichert werden. Dazu wird für jede Methode eine Liste angelegt, die die Parameter der einzelnen Aufrufe speichert. Jeder Eintrag in der Liste markiert einen Aufruf der Methode. Die Reihenfolge der Elemente der Liste ist die Aufrufreihenfolge in der sich die Methode aufgerufen hat. Der erste Eintrag in der Liste symbolisiert dabei den ersten Aufruf der Methode.

Da jede Methode eine unterschiedliche Anzahl an Parameter hat, besteht jeder Eintrag der Liste aus einem Feld von Objekten. Die Struktur die dabei entsteht entspricht einem zweidimensionalen Feld. Wird eine Methode aufgerufen, die keine Parameter besitzt, wird ein leeres Feld in die Liste aufgenommen. Dies ist nötig um zu signalisiert, dass die Methode sich gerade auf dem Aufrufstapel befindet. Wird die Methode per Rekursion erneut aufgerufen, wird ein weiteres leeres Feld in die Parameterliste der Methode geschrieben. Werden die Parameter der Methode nicht überwacht, da sie nicht in der @CFlowGuard Annotation angegeben wurden, wird der Wert **null** in die Liste aufgenommen. Auch dies ist nötig um zu markieren ob und wie oft sich die Methode auf dem Aufrufstapel befindet. Listing 7.2 zeigt die Signaturen der drei genannten Methoden. Diese müssen in der Klasse AbstractCFlowGuardTeam implementiert werden, die in Listing 6.6 gezeigt wurde.

```
public void enterMethod(String methodName, Object… args) {…}

public void enterMethod(String methodName) {…}

public void enterMethodIgnoreParams(String methodName) {…}
```

Listing 7.2: Erweiterte Signatur der Methode enterMethod()

Die Signatur der Methode leaveMethod(String methodName) ändert sich dagegen nicht. Dies ist auch nicht nötig, da beim Verlassen der Methode

nur der letzte Eintrag in der Parameterliste für die entsprechende Methode entfernt werden muss.

Durch diese Änderungen können die Parameter der Methoden, die auf den Aufrufstapel gepackt werden, ebenfalls gespeichert werden. Jetzt gilt es noch diese Parameter für die spätere Verarbeitung bereitzustellen. Eine Möglichkeit besteht darin, die Parameter der `@CFlowGuard` Annotation so zu modifizieren, dass der Code, der die Parameter verarbeitet, ebenfalls mittels Annotationsverarbeitung automatisch erzeugt werden kann. Da es jedoch nicht möglich ist selbst implementierte Typen als Parameter an einen Annotation zu übergeben, wäre solch eine Lösung an die wenigen verfügbaren Parametertypen von Annotationen gebunden. Mit dieser Einschränkung ist es sehr schwierig die Verarbeitung der Parameter umzusetzen. Darum wird diese Möglichkeit verworfen.

Einfacher ist hier ein Ansatz der den Code, der die Parameter auswertet, nicht automatisch erzeugt. Es bleibt hierbei dem Anwender dieses Verfahrens selbst überlassen wie die Parameter ausgewertet werden. Dazu muss er den auswertenden Code aber auch selbst implementieren. Damit dies möglich wird, müssen die Parameter in der Klasse verfügbar gemacht werden, in der auch die zugehörige `@CFlowGuard` Annotation benutzt wurde. Dies funktioniert aber nur, wenn diese Klasse als Callback am selbst organisierten Aufrufstapel angemeldet wird. Dafür muss die Klasse die Schnittstelle `CFlowParameterHandler` implementieren, die in Listing 7.3 zu sehen ist.

```
public interface CFlowParameterHandler {
        boolean handleGuardCall(int guardID, CFlowParameterHolder
                parameterHolder, boolean restrictionPassed);
}
```

Listing 7.3: Schnittstelle CFlowParameterHandler

Die einzige Methode dieser Schnittstelle wird aufgerufen während der Überprüfung ob die annotierte Callin-Bindung ausgeführt werden soll. Der Parameter `guardID` gibt die Kennung des Guards an der gerade überprüft wird. Vom `CFlowParameterHolder` können die gespeicherten Parameter abgerufen werden. Die Schnittstelle `CFlowParameterHolder` wird in Listing 7.4 gezeigt. Der Parameter `restrictionPassed` zeigt, ob die Einschränkung, die in der `@CFlowGuard` Annotation angegeben ist, erfüllt wurde oder nicht. Der Rückgabewert der Methode `handleGuardCall(..)` bildet die Grundlage für die Ausführungsprüfung der Callin-Bindung Somit kann der Rückgabewert der Methode das Ergebnis der

Ausführungsprüfung noch verändern. Dadurch kann die Ausführungsprüfung nicht nur auf Basis des Kontrollflusses entschieden werden sondern auch durch zusätzliche Datenanalysen. Soll die ursprüngliche Einschränkung nicht verändert werden, muss der Parameter restrictionPassed als Ergebnis der Methode zurückgegeben werden.

```
public interface CFlowParameterHolder {
        List<Object[]> getParameters(String method);
}
```

Listing 7.4: Schnittstelle CFlowParameterHolder

Über die Methode getParameters(String methodName) können die Parameter der angegebenen Methode abgefragt werden. Dies funktioniert nur wenn die Methode in der @CFlowGuard Annotation aufgeführt war und wenn sie sich gerade auf dem Aufrufstapel befindet. Der an die Methode zu übergebende Methodenname muss identisch mit dem in der @CFlowGuard Annotation angegebenen Methodennamen sein. Dies bezieht auch die Parameter und den Rückgabetyp mit ein. Die Länge der erhaltenen Parameterliste spiegelt die Anzahl der Aufrufe der angegebenen Methode wieder. Ist die Liste leer, so befindet sich die angegebene Methode aktuell nicht auf dem Aufrufstapel.

Implementiert das Team in der die @CFlowGuard Annotation benutzt wurde die Schnittstelle CFlowParameterHandler, so wird die Methode handleGuardCall(..) bei der Ausführungsprüfung einer Callin-Bindung aufgerufen. Implementiert das Team dieses Interface nicht, so bleibt die in Kapitel 6 beschrieben Funktionalität der Ausführungsprüfung unverändert erhalten.

Bleibt nur noch die Frage zu klären, wie die Methode handleGuardCall(..) aufgerufen werden kann, ohne dass der Klasse die den Aufrufstapel organisiert eine Referenz auf das Team bekannt ist. Auch hierfür gibt es eine einfache Lösung. Die Rolle, von der aus die Ausführungsprüfung einer Callin-Bindung ausgeführt wird, spielt mittels der **playedBy** Beziehung bereits die Rolle des benötigten Teams. Mittels Lowering kann also eine Referenz auf dieses Team ermittelt werden. Damit hier der Klassenname des Teams nicht angegeben werden muss, wird an dieser Stelle explizites Lowering verwendet. Bei der Abfrage des Aufrufstapels wird das per Lowering erhaltene Objekt mit übergeben. Nun muss nur noch geprüft werden, ob dieses Objekt die Schnittstelle CFlowParameterHandler implementiert um den Aufruf der Methode zu machen. Diese Änderungen müssen ebenfalls in der Klasse

AbstractCFlowGuardTeam vorgenommen werden, das in Listing 6.6 gezeigt wurde. Listing 7.5 zeigt hier nur die überarbeitete Methode zur Ausführungsprüfung. Die Implementierung der restlichen Methoden ist im praktischen Teil der Arbeit zu sehen.

```
public abstract team class AbstractCFlowGuardTeam
        implements VariableHolder, CFlowParameterHolder {

        protected boolean checkRestriction(Object paramHandler,
                        int guardID) {
                CFlowParameterHandler handler;
                if (paramHandler instanceof CFLowParameterHandler)
                        handler = (CFlowParameterHandler) paramHandler;
                else
                        handler = null;

                FunctionPart part = restrictionMap.get(guardID);

                boolean restrictionPassed;
                if (part == null)
                        restrictionPassed = true;
                else
                        restrictionPassed = part.evaluate(this);

                if (handler == null)
                        return restrictionPassed;
                else
                        return handler.handleGuardCall(guardID, this,
                                restrictionPassed);
        }
}
```

Listing 7.5: Überarbeitete Methode zur Ausführungsprüfung

In der Implementierung der Methode checkRestriction(…) ist gut zu erkennen, dass das ursprüngliche Verhalten der Methode nicht geändert wird, wenn das übergebene Objekt nicht vom Typ CFlowParameterHandler ist. In diesem Fall wird weiterhin nur das Ergebnis der geparsten Formel als Ergebnis der Ausführungsprüfung benutzt. Wenn jedoch ein Objekt

vom Typ CFlowParameterHandler übergeben wird, kann dieses das Ergebnis der Formel überschreiben.

Listing 7.6 zeigt beispielhaft eine Implementierung der Methode handleGuardCall(..). Hier werden nur die gespeicherten Parameter auf der Konsole ausgegeben. Die Methode liefert den Parameter restrictionPassed als Ergebnis zurück. Dadurch bleibt das ursprüngliche Ergebnis der Ausführungsprüfung erhalten. Im Beispiel wird eine Methode überwacht, die sich per Rekursion wieder selbst aufruft. Die Callin-Bindung soll dabei nur ausgeführt werden, wenn sich die überwachte Methode auf dem Aufrufstapel befindet. In diesem Beispiel bedeutet dies, dass für alle Aufrufe außer dem ersten die Callin-Bindung ausgeführt werden soll.

```
public class DemoTeam implements CFlowParameterHandler {
	private static final String METHOD = "int test.Main.recurse(int o)";

	private boolean guard1() {
		return true;
	}

	protected class DemoRole playedBy Main {
		@Guard(1)
		@CFlowGuard(guardID = 1, cflow = METHOD)
		logRecurse <- replace recurse when(guard1());

		callin void logRecurse() {
			System.out.println("Method invoked");
			base.logRecurse();
		}
	}

	public boolean handleGuardCall(int guardID,
		CFlowParameterHolder holder, boolean passed) {

		List<Object[]> list = holder.getParameter(METHOD);
		for (Object[] args : list) {
			if (args == null) {
				System.out.println("Args not collected");
			}
			else if (args.length == 0) {
```

```
                        System.out.println("Method has no args");
                    }
                    else {
                        for (Object object : args)
                            System.out.print(object + ", ");
                    }
                    System.out.println();
                }
                return passed;
            }
        }
```

Listing 7.6: Beispielimplementierung

8 Tracematching

Bei der bisherigen Situationsbewertung wurden nur Ereignisse betrachtet, die sich in der aktuellen Aufrufsequenz befinden. Als Ereignis wurde dabei der Aufruf einer Methode angesehen. Somit wurden bisher nur Methoden zur Entscheidung herangezogen die sich aktuell auf dem Aufrufstapel einer Applikation befinden.

Eine weitere Möglichkeit zur Situationsbewertung bezieht auch Ereignisse ein, die in der Vergangenheit liegen. Als Ereignis werden weiterhin nur Methodenaufrufe betrachtet. Als Bewertungszeitraum wird jedoch nicht der aktuelle Aufrufstapel angenommen, sondern die vergangene Laufzeit der Applikation. Ein Ansatz für diese Form der Situationsbewertung wird Tracematching genannt. In diesem Kapitel wird die Idee des Tracematchings erläutert, Schwierigkeiten bei der Umsetzung diskutiert und Lösungsideen dafür gezeigt.

8.1 Tracematching in der Theorie

Wie bereits erwähnt, werden beim Tracematching auch Ereignisse zur Situationsbewertung herangezogen, die in der Vergangenheit der Anwendung liegen. Dabei wird in der Regel jedoch nicht nur ein einzelnes Ereignis betrachtet, sondern eine Sequenz von Ereignissen. Wird während der Laufzeit einer Applikation die angegebene Sequenz ausgeführt, wird danach noch der Advice des Tracematches ausgeführt. Daraus ergeben sich die drei Punkte aus denen jeder Tracematch besteht. Als erstes müssen dabei die Ereignisse definiert werden, die für einen konkreten Tracematch relevant sind. Danach müssen diese Ereignisse in eine Reihenfolge gebracht werden. Diese Reihenfolge ist die Sequenz der Ereignisse, die für einen Tracematch benötigt wird. Zum Schluss muss noch der Code definiert werden, der nach dem Auftreten der Sequenz von Ereignissen ausgeführt werden soll.

Normalerweise kann bei der Aspektorientierten Programmierung entschieden werden, wann ein Advice ausgeführt werden soll. Es besteht die Möglichkeit den Advice vor, nach oder anstatt einer Methode der Basisapplikation auszuführen. Beim Tracematching muss diese Unterscheidung etwas abgeschwächt werden. Hier kann der zusätzliche Code nur ausgeführt werden, wenn das Auftreten einer Ereignissequenz erkannt wurde.

Die Ausführung vor der Ereignissequenz ist nicht möglich, da nicht bekannt ist, ob und wann die Ereignissequenz auftritt. Tritt also ein Ereignis auf, das dem ersten Ereignis der Sequenz gleicht, kann noch nicht bestimmt werden, ob dieses Ereignis auch wirklich das erste Ereignis einer folgenden Sequenz ist. Anhand eines Beispiels soll dieser Umstand verdeutlicht werden. Die möglichen Ereignisse in diesem Beispiele seinen F und G. Die gesuchte Sequenz sei hierbei FG. Das heißt, dass das Ereignis G nach dem Ereignis F auftritt. Der zusätzliche Code soll also immer dann ausgeführt werden, nachdem die Ereignissequenz FG ausgeführt wurde. Wenn jetzt zum ersten Mal das Ereignis F auftritt, kann daraus noch nicht geschlossen werden, dass dieses Ereignis zwingend der Beginn einer Ereignissequenz ist denn es könnte nach dem ersten Aufruf von F ein weiteres Ereignis F auftreten. Dadurch ist das erste Auftreten des Ereignisses F nicht mehr der Beginn der Ereignissequenz FG.

Für die Ersetzung einer Methode der Basisapplikation gilt Ähnliches. Auch hierbei müsste man wissen, welches Ereignis das erste Ereignis einer Sequenz ist noch bevor die Sequenz abgeschlossen ist. Des Weiteren ist nicht klar definiert, wie die Ersetzung einer Ereigniskette aussehen soll. Es könnte zum Beispiel jeder Methodenaufruf durch einen eigenen Advice ersetzt werden. Es ist aber auch denkbar, dass jeder Methodenaufruf durch denselben Advice ersetzt wird. Außerdem sind natürlich Mischformen dieser beiden Extreme denkbar. Die Ersetzung einer Ereignissequenz ist somit nicht möglich.

Es ist jedoch wichtig zu unterscheiden wann die Ereignisse, also die Methodenaufrufe, in der Ereignissequenz gespeichert werden. Dies kann vor dem Eintreten in die Methode oder nach dem Verlassen der Methode geschehen. Somit muss ein Ereignis im Sinne eines Tracematches definiert werden als Eintritt in eine Methode bzw. das Verlassen einer Methode.

8.2 Tracematching mit ObjectTeams/Java

Da das Konzept des Tracematching nicht in ObjectTeams/Java implementiert ist, soll zuerst geklärt werden, wie Tracematching mittels Elementen der Sprache ObjectTeams/Java umgesetzt werden kann. Dazu muss geklärt werden, wie die drei relevanten Punkte eines Tracematches in ObjectTeams/Java umgesetzt werden können.

Der erste Punkt ist die Definition der Ereignisse. Ein Ereignis eines Tracematches ist das Eintreten in eine Methode bzw. das Verlassen einer Methode. Damit eine Ereignissequenz erkannt werden kann, müssen also das Eintreten bzw. das Verlassen einer Methode protokolliert werden.

Mittels einer Callin-Bindung braucht dabei nicht in den bestehenden Code eingegriffen zu werden. Es muss nur vor oder nach der Ausführung der Methode noch eine Callin-Bindung ausgeführt werden, die das entsprechende Ereignis, also das Betreten oder Verlassen der Basismethode, protokolliert.

Nach dem Protokollieren eines Ereignisses muss geprüft werden, ob das Ereignis den Abschluss einer Sequenz bildet. Dazu muss die komplette bisher protokollierte Sequenz der Ereignisse überprüft werden, ob sie die gesuchte Sequenz als Teilsequenz enthält. Die Umsetzung hierfür könnte beispielhaft mittels Pattern-Matching realisiert werden. Wurde eine passende Sequenz gefunden, muss noch der Advice ausgeführt werden. Dieser kann beim Tracematching mittels ObjectTeams/Java aus einer Rollenmethode bestehen. Da alle für das Tracematching relevanten Rollen innerhalb desselben Teams liegen, kann hier auch eine Java-Methode auf Team-Level-Ebene als Advice genutzt werden. Listing 8.1 zeigt die Umsetzung des Tracematchings mittel ObjectTeams/Java

---

```
public team class TraceMatchTeam {
    private StringBuffer sequence = new StringBuffer();

    protected class TraceRole1 playedBy BaseClass1 {
        logA <- after demoMethod;

        private void logA() {
            log("A");
        }
    }

    protected class TraceRole2 playedBy BaseClass2 {
        logB <- after someMethod;

        private void logB() {
            log("B");
        }
    }

    private void log(String identifier) {
        sequence.append(identifier);
        if (matches())
            runAdvice();
    }
```

```
        private boolean matches() {…}

        private void runAdvice() {..}
}
```

Listing 8.1: Tracematching mit ObjectTeams/Java

Die Schwierigkeit des Ansatzes in Listing 8.1 bildet die Methode matches(). Diese Methode muss überprüfen ob die aktuell gespeicherte Sequenz eine gegebene Teilsequenz enthält. Enthält die gespeicherte Sequenz die Teilsequenz, wird der Advice ausgeführt. Da die gefundene Teilsequenz bei der nächsten Teilsequenzprüfung nicht mehr gefunden werden darf, kann die gespeicherte Sequenz nach der Ausführung des Advice gelöscht werden. Da die Ereignissequenz für jeden Tracematch separat gespeichert werden muss, ist das Löschen der Ereignissequenz möglich ohne andere Tracematches zu beeinflussen.

Des Weiteren könnte es interessant sein Ereignisse zu markieren, die sich nicht zwischen den gesuchten Ereignissen befinden dürfen. Anhand eines Beispiels soll dies verdeutlicht werden. Hierbei sind die überwachten Ereignisse A, B, C und D. Die gesuchte Teilsequenz ist ABC. Das bedeutet, dass das Ereignis D nicht innerhalb der Sequenz auftreten darf. Die bisher gespeicherte Ereignissequenz besteht aus den Ereignissen AB. Wird nun das Ereignis C protokolliert, wird die gesuchte Teilsequenz gefunden und der Advice ausgeführt. Wird an Stelle des Ereignisses C jedoch das Ereignis D protokolliert, kann die gesuchte Teilsequenz mit der bisher gespeicherten Ereignissequenz nicht mehr auftreten. Die gespeicherte Ereignissequenz kann daraufhin gelöscht werden. Verallgemeinert kann man sagen, dass die Ereignissequenz gelöscht werden kann, wenn ein Ereignis auftritt das nicht dem nächsten Ereignis innerhalb der gesuchten Teilsequenz entspricht.

Bei der Notation eines Tracematches kann dieses Verhalten realisiert werden, indem das Ereignis D in der Definition der Ereignisse auftaucht nicht jedoch in der Definition der Sequenz. Das Ereignis D würde somit zum relevanten Ereignis erklärt, darf sich jedoch nicht innerhalb der gesuchten Teilsequenz befinden.

In Listing 8.1 wurde ein StringBuffer benutzt um die Ereignissequenz zu speichern. Um nun eine Teilsequenzprüfung durchzuführen, muss die gespeicherte Sequenz nur mit der gesuchten Teilsequenz verglichen werden. Dies reicht bei der Durchführung der Teilsequenzprüfung aus, da bei Auftreten eines nicht gewünschten Ereignisses die gespeicherte Sequenz gelöscht wird.

8.3 Objektbindungen

Bisher wurden nur die Methodenaufrufe für sich alleine betrachtet. Dadurch lässt sich eine Form der Situationsbewertung schaffen, die nicht nur den aktuellen Kontrollfluss sondern auch den vergangenen Kontrollfluss bewerten kann. Es wurde also betrachtet, ob eine Methode einer bestimmten Klasse aufgerufen wurde. Sofern es sich nicht um statische Methoden handelt, wird eine Methode aber an einem bestimmten Objekt, also einer Instanz einer Klasse, aufgerufen. Diese Bindung der Methoden an bestimmte Objekte wurde bisher vernachlässigt.

Anhand eines Beispiels soll verdeutlicht werden warum dies zu Schwierigkeiten führen kann. Kern des Beispiels ist die Klasse `Connection` mit ihren Methoden `open()`, `close()` und `read()`. Mit den Methoden `open()` und `close()` wird die Verbindung geöffnet und wieder geschlossen, mittels der Methode `read()` können Daten von der Verbindung gelesen werden. Hierbei gilt, dass Daten nur von einer geöffneten Verbindung gelesen werden können. Denkbar wäre also ein Tracematch der auf die Sequenz `close()`, `read()` reagiert, also auf einen lesenden Zugriff nach dem Schließen der Verbindung. In diesem Fehlerfall soll die Verbindung erneut geöffnet werden. Betrachtet man nur eine Instanz von `Connection`, so funktioniert dieses Verhalten wie gewünscht. Betrachtet man jedoch zwei gleichzeitige Verbindungen, so kann der Advice des Tracematch ausgeführt werden ohne dass dies nötig wäre. Dazu werden zuerst beide Verbindungen geöffnet. Danach wird von der ersten Verbindung gelesen und diese dann geschlossen. Wird jetzt von der zweiten Verbindung gelesen, so sieht der Tracematch nur ein `read()` nach einem `close()` und würde den Advice ausführen obwohl die beiden Ereignisse von unterschiedlichen Objekten erzeugt wurden und somit ein `open()`, das im Advice ausgeführt werden sollte, gar nicht nötig ist.

Anhand des Beispiels könnte man argumentieren, dass es ausreicht die Sequenz für jedes Objekt separat zu erzeugen. Im Beispiel würde das auch den gewünschten Erfolg bringen. Dies liegt jedoch nur daran, dass das Beispiel zu wenig Komplexität enthält. Soll der Tracematch nämlich nicht nur Objekte von einem Typ überwachen sondern mehrere Objekte mit unterschiedlichen Typen, so macht es keinen Sinn mehr pro Objekt eine Sequenz aufzubauen, da die Sequenz ja Methodenaufrufe von anderen Objekten enthalten soll.

Andererseits kann es durchaus auch Fälle geben in denen es gewünscht ist, dass Methodenaufrufe von unterschiedlichen Instanzen derselben Klasse in einen Tracematch einbezogen werden. Es kann also nicht

programmatisch entschieden werden wie mit unterschiedlichen Objekten derselben Klasse verfahren wird. Es muss dem Anwender überlassen werden, ob die Objekte, die im Tracematch betrachtet werden, identisch sein sollen oder nicht. Damit dies möglich ist, muss eine Möglichkeit geschaffen werden die Objekte zu speichern und die gespeicherten Objekte zu vergleichen.

Dies zieht jedoch weitreichende Konsequenzen nach sich. Wenn die Objekte, die an der gespeicherten Ereignissequenz beteiligt sind, gespeichert werden sollen, reicht die gezeigte Speicherung der Ereignisse als einfache Zeichenkette nicht mehr aus. Es muss nun eine Möglichkeit gefunden werden damit die Objekte zusätzlich gespeichert werden können. Eine Lösungsidee hierfür könnte darin bestehen, für jedes Objekt eine eigene Ereignissequenz zu speichern. Dieses Verfahren stößt allerdings an seine Grenzen sobald mehrere verschiedene Datentypen für einen Tracematch herangezogen werden. Hierbei müsste eine Ereignissequenz pro Objektkombination gespeichert werden.

Ein weiterer Punkt ist die Speicherung der Objekte selber. Diese werden nur als Referenzen gespeichert. Wenn jedoch eine Referenz auf ein Objekt gespeichert bleibt, so kann das Objekt nicht vom Garbagecollector aus dem Speicher entfernt werden. Für dieses Problem sieht die Lösung recht einfach aus. Die Objektreferenzen können als `WeakReference` gespeichert werden. Dadurch kann das referenzierte Objekt zwar aus dem Speicher entfernt werden, in der Sequenz des Tracematches sind aber immer noch die Ereignisse gespeichert, die von diesem Objekt ausgegangen sind. Dies kann positive Auswirkungen auf den Tracematch haben, da eine vollständige Abdeckung der vergangenen Ereignisse gewährleistet ist. Es kann jedoch auch negative Auswirkungen haben, da Ereignisse von Objekten in den Tracematch einfließen die nicht mehr existieren und über die somit auch keine Informationen mehr verfügbar sind. Dies ist für die hier angestellten Betrachtungen noch zweitrangig. Allerdings wird die Speicherung von Objekten in Kapitel 8.6 nochmals aufgegriffen, dort ist es notwendig, dass die gespeicherten Objekte auch verfügbar bleiben.

Aufgrund der beschriebenen Komplexität wird im weiteren Verlauf der Betrachtung von Tracematches auf die Objektbindungen verzichtet. Eine mögliche Lösung für die beschriebenen Probleme bietet [All05].

8.4 Threadabhängigkeiten

In den vorangegangenen Betrachtungen von Tracematches wurde davon ausgegangen, dass eine Anwendung nur aus einem einzelnen Thread besteht. Dabei wird Nebenläufigkeit auch im Hinblick auf das

Voranschreiten der Technik immer wichtiger, denn der Trend der Prozessorentwicklung geht nicht nur in Richtung von Dual-Core- und Quad-Core-Prozessoren sondern vielmehr in Richtung Many-Core-Prozessoren. In den folgenden Absätzen soll darum betrachtet werden, wie Nebenläufigkeit mit Tracematches vereinbart werden kann.

Wie auch bei der Bewertung des aktuellen Kontrollflusses werden bei den Tracematches Methodenaufrufe auf einer Art Aufrufstapel organisiert. Bei der Betrachtung des aktuellen Kontrollflusses wurde beim Verlassen der Methode diese wieder vom Aufrufstapel entfernt. Damit dort auch wirklich nur die Aufrufsequenz einer Methode betrachtet werden konnte, war es notwendig den Aufrufstapel für jeden Thread separat zu organisieren.

Bei den Tracematches werden die Ereignisse nicht wieder aus der Sequenz entfernt. Dies ist notwendig um ein eine vollständige Abdeckung der vergangenen Ereignisse zu erreichen. Man kann jedoch nicht mit Bestimmtheit sagen, dass eine Ereignissequenz nur für einen einzelnen Thread, und somit für jeden Thread separat, erstellt werden darf. Es lassen sich sicherlich Anwendungsfälle konstruieren bei denen es relevant ist, dass die gespeicherten Ereignisse nur aus einem konkreten Thread stammen. Bei den Tracematches soll der vollständige vergangene und gegenwärtige Kontrollfluss betrachtete werden. Es wird also der Zustand aller überwachten Objekte, und somit ein ausgewählter Teil aus dem Gesamtzustand der Applikation, betrachtet. Theoretisch ist es also möglich den vollständigen Zustand einer Applikation zu überwachen. Zum Gesamtzustand einer Applikation gehören aber auch alle Threads aus denen sich die Applikation zusammensetzt. Ein Zustandsübergang eines Objektes müsste somit in den Sequenzen aller Threads gespeichert werden. Dies spricht dafür, dass auf die Betrachtung von Threads bei den Tracematches verzichtet werden könnte.

Ein Gegenbeispiel, bei dem es also wichtig ist die Ereignissequenzen für jeden Thread separat zu verwalten, könnte eine Anwendung sein, bei der jeder Thread die gleiche Aufgabe mit jeweils verschiedenen Daten berechnet. Hier wäre der Zustand eines Objektes nur innerhalb des Threads wichtig, zu dem das Objekt gehört. Das zeigt also, dass es, wie bei den Objektbindungen, dem Nutzer überlassen sein sollte ob eine Ereignissequenz nur für einen Thread oder für alle Threads erstellt werden soll. Interessant könnte es auch sein eine Ereignissequenz für eine Gruppe von Threads zu erstellen. Die Handhabung von Threads wird ebenfalls in [All05] eingehender diskutiert.

8.5 Generierte Tracematches

Wie bei der Situationsbewertung auf Basis des aktuellen Kontrollflusses ist das erklärte Ziel, dass der Code, der für die Tracematches benötigt wird, automatisch erzeugt wird. Auch hier soll dies mittels Annotationsverarbeitung geschehen. In diesem Abschnitt werden die dazu benötigten Komponenten skizziert.

Wie bei der Annotationsverarbeitung üblich, werden ein `AnnotationProcessor` und eine `AnnotationProcessorFactory` benötigt. Die `AnnotationProcessorFactory` ist dabei trivial, da sie nur eine Instanz des `AnnotationProcessors` liefern muss. Der `AnnotationProcessor` hat bei den Tracematches die gleiche Aufgabe wie bei der Situationsbewertung auf Basis des aktuellen Kontrollflusses. Er erzeugt aus der Annotation ein Team, das die in der Annotation angegebenen Methoden mittels Callin-Bindungen um zusätzlichen Code erweitert. Die Callin-Bindungen erzeugen zur Laufzeit die Ereignisse, die das Betreten bzw. Verlassen einer Methode signalisieren, und speichern es in der Ereignissequenz.

Für die Tracematches ist es nicht notwendig die Methode zu ersetzen, da das gespeicherte Ereignis beim Verlassen der Methode nicht wieder aus der Ereignissequenz gelöscht werden darf. Eine interessante Designentscheidung ist, ob die Callin-Bindung vor oder nach der Methode ausgeführt wird. Für Ereignisse die die Ausführung des Advice nicht auslösen würden, ist diese Entscheidung noch nicht relevant. Wird jedoch ein Ereignis erzeugt, das die Ausführung des Advice veranlasst, so kann die Stelle der Ausführung von großer Bedeutung sein. Um dies zu veranschaulichen kommt wieder das oben eingeführte Beispiel der `Connection` zum Zuge. Der Tracematch in diesem Beispiel bestand aus `close()` und `read()`, also wenn von der Verbindung gelesen werden sollte nachdem sie geschlossen wurde. Durch die Ausführung von `read()` wird der Advice des Tracematches ausgeführt. Für dieses Beispiel bestand der Advice darin, dass die Verbindung erneut geöffnet werden sollte. Wird der Advice nun vor dem `read()` ausgeführt, wird die Verbindung erst erneut geöffnet und dann werden die Daten gelesen. Wird der Advice jedoch nach dem `read()` ausgeführt, so wird erst versucht die Daten zu lesen und danach wird die Verbindung geöffnet. Dies führt natürlich zu einem Fehler beim lesenden Zugriff auf die `Connection`, da ja nur von offenen Verbindungen gelesen werden durfte. In diesem Fehlerfall wäre es aber denkbar, dass der Advice darin besteht eben diesen Fehler zu behandeln. Es ist also auch hier dem Anwender zu überlassen wann der Advice ausgeführt werden soll. Diese Entscheidung muss für jedes Ereignis separat getroffen werden und muss somit bei jedem Event separat notiert werden. Der `AnnotationProcessor` erzeugt dann die Callin-Bindungen so, dass sie vor oder nach den Basismethoden ausgeführt werden.

Der Code für die Speicherung der Ereignissequenz sollte in einer abstrakten Oberklasse implementiert sein. Damit später die Objektbindungen verarbeitet werden können, sollte die Signatur der speichernden Methode das Objekt, das die ursprüngliche Methode ausführt, berücksichtigen. Da der Aufruf der speichernden Methode aus einer Callin-Bindung und somit aus einer Rollenklasse kommt, muss das Objekt, das die ursprüngliche Methode ausführt, mittels Lowering ermittelt werden. Hierbei wird explizites Lowering benötigt, da aus der Methodensignatur nicht hervorgeht welcher Datentyp benötigt wird. Theoretisch reicht es auch aus die Rollenobjekte zu speichern. Soll jedoch mit den Basisobjekten im Advice weitergearbeitet werden, so müssen diese dann durch Lowering der Rollenobjekte innerhalb des Advice ermittelt werden. Der einfachere Weg besteht darum darin die Basisobjekte direkt an die speichernde Methode zu übergeben, da das Lowering hier per Codegenerierung implementiert wird.

Beim Speichern eines Ereignisses in der Ereignissequenz ist es nicht notwendig den aktuellen Thread an die speichernde Methode zu übergeben, da dieser mittels des Aufrufes `Thread.currentThread()` innerhalb der Methode ermittelt werden kann. Listing 8.2 zeigt eine mögliche Signatur der Methode.

```
public void methodExecuted(String methodName, Object executingObject);
```

Listing 8.2: Methodensignatur zur Speicherung der Ereignissequenz

Bei jedem Ereignis das gespeichert wird, muss geprüft werden ob eine passende Aufrufsequenz für einen Tracematch vorliegt. Ein einfacher Vergleichsalgorithmus wurde in Kapitel 8.2 vorgestellt. Dieser muss in der abstrakten Oberklasse des zu generierenden Teams implementiert sein, damit er nicht in jedes zu erzeugenden Team hinein generiert werden muss. Ein weiterer Vorteil ist, dass er dadurch an einer zentralen Stelle implementiert ist und somit leicht austauschbar ist.

Bleibt noch die Frage zu klären, wie der Advice ausgeführt werden soll nachdem eine passende Ereignissequenz gefunden wurde. Bei den Tracematches in ObjectTeams/Java ist der Advice nichts anderes als eine normale Java-Methode. Somit bleibt zu klären wie diese Methode aufgerufen werden kann. Damit eine Methode aufgerufen werden kann, wird ein Objekt benötigt an dem die Methode aufgerufen wird. Wenn die gebundenen Objekte aus Kapitel 8.3 gespeichert werden, so stehen diese im Advice des Tracematches zur Verfügung. Damit könnten im Advice Methoden dieser Objekte aufgerufen werden. Werden diese Objekte nicht gespeichert, so steht wenigstens das Objekt zur Verfügung, bei dem

aktuell ein Methodenaufruf in der Ereignissequenz gespeichert wird. Veranschaulicht am Connection Beispiel mit dem Tracematch close(), read() bedeutet dies, dass beim Speichern des Ereignisses read() eine Instanz der Klasse Connection verfügbar ist. An dieser Instanz kann dann die Methode open() aufgerufen werden.

Damit der Advice die gewünschte Methode aufrufen kann, muss vorher beschrieben werden welche Methode ausgeführt werden soll. Dies kann über verschiedene Wege realisiert werden. Ein Weg wäre, dass ein ähnlicher Ablauf wie beim Wurmlocheffekt in Kapitel 7 genutzt wird. Die Klasse in der die Annotation steht, muss dabei eine bestimmte Schnittstelle implementieren, damit eine Instanz dieser Klasse als Callback für den Advice benutzt werden kann. Der Advice delegiert die Ausführung dann nur an die in der Schnittstelle beschriebene Methode.

Eine weitere Möglichkeit besteht darin, dass die Methode als Parameter der Annotation implementiert wird. Durch die beschränkte Anzahl an möglichen Parametertypen müsste die Methode als String codiert werden. Der AnnotationProcessor kann diesen String dann verarbeiten und einen Weg in dem zu erzeugenden Team generieren, wie die Methode aufgerufen werden kann. Damit die Sichtbarkeit der Methode dabei nebensächlich bleibt, sollte der Aufruf der Methode über eine Callout-Bindung erfolgen. Die Implementierung der Methode runAdvice() aus Listing 8.1 würde den Aufruf nur an die Callout-Bindung delegieren.

Daraus ergibt sich, dass die @Tracematch Annotation nicht nur innerhalb eines Teams, sondern in jeder beliebigen Klasse benutzt werden kann. Dieser Gegensatz zur Situationsbewertung auf Basis des aktuellen Kontrollflusses entsteht, da ein Tracematch nicht ein eine einzelne Callin-Bindung gebunden ist, sondern nach einer Sequenz von Ereignissen ausgeführt wird.

Da nun der generelle Ablauf geklärt ist, kann das Herzstück der Annotationsverarbeitung designt werden. Damit nämlich die Annotationsverarbeitung überhaupt stattfinden kann, wird eine Annotation und somit ein Annotationstyp benötigt. In den vorangegangenen Absätzen wurden schon die wichtigsten Parameter des Annotationstyps erwähnt. Listing 8.3 zeigt eine mögliche Implementierung des Annotationstyps.

```
public @interface Tracematch {

    String[] events();
    String trace();
```

```
        String adviceMethod();
}
```

Listing 8.3: Annotationstyp Tracematch

Der Parameter events gibt alle benötigten Ereignisse an. Als Ereignis muss hier nicht nur die Methode angegeben werden, sondern auch ob das Eintreten oder das Verlassen der Methode gemeint ist. Hierbei sind nicht nur die Ereignisse wichtig die innerhalb des Tracematches vorkommen sollen, sondern auch die, die nicht vorkommen dürfen. Der Methodenname eines Ereignisses muss dabei inklusive des Klassennamen notiert werden. Der Klassenname wird benötigt um die Basisklasse der Rolle zu beziehen, die die Methode überwachen soll. Warum die Events separat definiert werden müssen, wurde bereits in Kapitel 8.2 erläutert.

Der Parameter trace gibt den regulären Ausdruck an, der in der gespeicherten Ereignissequenz gefunden werden soll. Innerhalb dieses Parameters dürfen nur Ereignisse benutzt werden, die durch den Parameter events definiert wurde. Sinnvoll wäre hier eine alphanumerische Zuordnung auf die in events angegebenen Ereignisse. So könnte mit trace="AC" die Ereignissequenz aus dem ersten und dritten Ereignis innerhalb des Feldes events gemeint sein.

Letztlich gibt der Parameter adviceMethode den Methodennamen der Methode an, die bei der Ausführung des Advice aufgerufen werden soll. Bei diesem Parameter muss die Methode nicht mit Klassenname angegeben werden, da die Klasse bekannt ist, nämlich die Klasse in der die Annotation steht. Der AnnotationProcessor muss die Callout-Bindung, die aus diesem Parameter erzeugt werden soll, nur in die richtige generierte Rolle einsortieren. Das Objekt an dem das letzte Ereignis aufgerufen wurde, muss als Parameter an die Methode übergeben werden. Listing 8.4 zeigt die Verwendung der @Tracematch Annotation anhand des Connection Beispiels.

```
public class ConnectionTracematch {
        @Tracematch(
                events = new String[] {
                        "after: Connection.open()",
                        "after: Connection.close()",
                        "before: Connection.read()"
                },
                trace = "BC",
```

```
                adviceMethod = "reopen")
        private void reopen(Object executingObject) {
                Connection connection = (Connection) executingObject;
                connection.open();
        }
}
```

Listing 8.4: Annotation Tracematch am Beispiel

8.6 Wurmlocheffekt

Da die Funktionsweise der Tracematches sehr ähnlich zur Funktionsweise der Situationsbewertung auf Basis des aktuellen Kontrollflusses ist, könnte man sich die Frage stellen ob auch bei den Tracematches der Wurmlocheffekt realisiert werden kann. Für die Situationsbewertung auf Basis des aktuellen Kontrollflusses wurde der Wurmlocheffekt bereits ausführlich in Kapitel 7 beschrieben. Hier soll nun der Wurmlocheffekt bei Tracematches betrachtet werden und gezeigt werden welche Probleme dabei beachtet werden müssen.

Bei jedem Aufruf einer überwachten Methode wird ein Ereignis erzeugt und dieses in der Ereignissequenz gespeichert. Mit diesem Ereignis könnten auch die Parameter der überwachten Methode übergeben werden. Damit dies geschehen kann, muss die Signatur der Methode, die die Ereignisse speichert, um eine variable Argumentliste ergänzt werden. Um auch Ereignisse von Methoden zu speichern, die keine Parameter haben oder bei denen die Parameter nicht überwacht werden, sind zwei weitere Methoden nötig die eben diese Gegebenheiten berücksichtigen. Dieses Vorgehen ist also völlig analog zur Speicherung von Parametern bei der Situationsbewertung auf Basis des aktuellen Kontrollflusses in Kapitel 7. Listing 8.5 zeigt die Signaturen der benötigten Methoden.

```
public void enterMethod(String methodName, Object executingObject,
                Object… parameters);
public void enterMethod(String methodName, Object executingObject);
public void enterMethodIgnoreParams(String methodName,
                Object executingObject);
```

Listing 8.5: Methodensignaturen zur Speicherung von Parametern

Bei dem vorgestellten Tracematch-Ansatz wird kein Ereignis erzeugt, dass ein Event nach der Ausführung einer Methode wieder aus der Ereignissequenz entfernt. Darum werden auch die Parameter an dieser Stelle nicht wieder aus der Ereignissequenz entfernt. Dies wäre auch die falsche Stelle dazu, da sonst die Parameter der überwachten Methoden nicht im Advice zur Verfügung stehen würden.

Da bei den Tracematches der vollständige Ablauf der Applikation überwacht werden soll, müssten die Parameter auch über die vollständige Laufzeit der Applikation in der Ereignissequenz gespeichert werden. Dies ist jedoch für den Speicherverbrauch der Applikation nicht akzeptabel, da somit alle Objekte die als Parameter einer überwachten Methode benutzt werden, nie wieder freigegeben werden. Des Weiteren würde der Speicher, den die Ereignissequenz benötigt, sehr viel Platz belegen. Aus diesen Gründen ist es nötig, dass die Referenzen auf Parameter wieder gelöscht werden und der Garbagecollector den Speicher, den die Parameter belegen, wieder freigeben kann.

Der einzige Punkt der als sinnvoll erachtet werden kann um die Objekte wieder aus der Ereignissequenz zu entfernen ist, wenn das Objekt an dem die Methode aufgerufen wurde aus dem Speicher entfernt wird. Bereits in Kapitel 8.3 wurde gezeigt, dass diese gebundenen Objekte als `WeakReference` gespeichert werden können, damit sie aus dem Speicher entfernt werden können. Wenn das Objekt an dem eine Methode aufgerufen wurde aber nicht mehr zur Verfügung steht, dann ist es auch nicht sinnvoll, dass die Parameter der aufgerufenen Methode noch im Speicher liegen. Darum sollten die Parameter ebenfalls entfernt werden. Der einfachste Weg um diese Bindung an die schwachen Referenzen zu realisieren ist mittels einer Instanz der Klasse `WeakHashMap`. Bei dieser Implementierung der Schnittstelle `Map` werden die Schlüssel nur als schwache Referenz gespeichert. Dadurch kann der Garbagecollector sie wieder freigeben. Wird ein Schlüssel einer `WeakHashMap` freigegeben, so wird auch das in der `Map` zugeordnete Objekt aus der `Map` entfernt. Somit existieren innerhalb der Ereignissequenz dann keine weiteren Referenzen auf die Parameter mehr, woraufhin diese ebenfalls vom Garbagecollector freigegeben werden können.

Beim Abrufen der Parameter innerhalb des Advice sind jedoch die Objekte an denen die Methoden aufgerufen wurden nicht bekannt. Darum können sie nicht aus der `Map` abgerufen werden. Ein Ansatz um dieses Problem zu lösen wäre, die Schlüssel der `Map`, und somit alle gespeicherten Objekte, dem Advice bekannt zu machen. Eine andere Möglichkeit besteht in einer zusätzlichen Zuordnung der, in der Annotation verwendeten, Events zu den konkreten Objekten. Da die Events bekannt sind, kann somit der gesuchte Schlüssel selektiv bezogen werden.

Bei der Ausführung des Advice müssen nun die Parameter der überwachten Methoden innerhalb des Advice verfügbar gemacht werden. Dazu kann der in Kapitel 8.5 beschrieben Weg zur Ausführung des Advice genutzt werden. Dabei muss die aufgerufene Methode aber einen Parameter von Typ TraceParameterHolder besitzen. Die Schnittstelle TraceParameterHolder stellt die Parameter der überwachten Methoden bereit und funktioniert somit analog zum beschriebenen CFlowParameterHolder in Kapitel 7. Da es aber sein kann, dass die Methode nicht verändert werden darf, da sie mit der vorhandenen Signatur in anderen Klassen benutzt wird, ist der bessere Weg die Ausführung des Advice mittels einer anderen Methode zu realisieren.

Bei der Situationsbewertung auf Basis des aktuellen Kontrollflusses wurde ein Callback-Mechanismus benutzt um die Parameter der überwachten Methoden für den Wurmlocheffekt bereitzustellen. Bei den Tracematches kann dieses Verfahren adaptiert werden und außerdem dazu genutzt werden den Advice auszuführen. Dazu muss die Klasse in der die @Tracematch Annotation benutzt wird die Schnittstelle TracematchHandler implementieren. Wird in der Ereignissequenz ein passender Tracematch gefunden, so wird die in der Schnittelle beschriebene Methode aufgerufen welche damit den Advice bildet. Als Parameter der Methode wird eine Instanz vom Typ TraceParameterHolder geliefert, welche die Parameter der überwachten Methoden bereitstellt. Listing 8.6 zeigt die Schnittstellen TraceParameterHolder und TracematchHandler. Da eine Methode mehrfach im Tracematch auftauchen kann, liefert der TraceParameterHolder eine Liste von Objektfeldern zurück. Ein Objektfeld ist dabei gleichzusetzen mit der variablen Parameterliste die für jeden Methodenaufruf zusätzlich in der Ereignissequenz gespeichert wurde. Als Parameter der Methode wird der Event angegeben der auch in der @Tracematch Annotation verwendet wurde.

```
public interface TraceParameterHolder {
        List<Object[]> getParameters(String event);
}
public interface TracematchHandler {
        void runAdvice(TraceParameterHolder parameterHolder);
}
```

Listing 8.6: Schnittstellen TraceParameterHolder und TracematchHandler

9 Zusammenfassung und Ausblick

Ziel der Arbeit war es eine Möglichkeit zu finden mit der Callin-Bindungen gesteuert werden können. Diese Steuerung sollte zusätzlich zu den in ObjectTeams/Java gegebenen Möglichkeiten der Aktivierung von Aspekten implementiert werden. Dabei sollte berücksichtigt werden, dass die neue Art der Steuerung einfach zu benutzen ist und möglichst wenig bestehender Code verändert werden muss. In diesem Kapitel wird zurückblickend beleuchtet ob und wie die gewünschten Ziele erreicht wurden. Außerdem soll ein Ausblick zeigen wie dieses Thema weiterentwickelt werden kann.

9.1 Zusammenfassung

Es wurden mehrere Möglichkeiten gezeigt um die Ausführung einer Callin-Bindung und somit die Ausführung eines Aspektes zu steuern. Diese Steuerung wurde dazu benutzt einen Aspekt zu aktivieren bzw. zu deaktivieren ohne den Aktivierungszustand des Teams zu verändern, das die Callin-Bindung umschließt. Somit wurde eine Möglichkeit geschaffen die Ausführung eines Aspektes nicht nur an die Aktivierung eines Teams zu binden sondern von weiteren Faktoren abhängig zu machen.

Es ist somit möglich die Ausführung eines Aspektes von weiteren Analysen abhängig zu machen. In dieser Arbeit wurden drei Analysetechniken erklärt und die für ihre Implementierung benötigten Komponenten beschrieben. Dabei befassen sich zwei der Analysen mit der Bewertung des Zustandes einer Applikation, also der Situation in der sich die Applikation gerade befindet. Zuerst wurde nur der gegenwärtige Zustand einer Applikation betrachtet. Hierfür wurde die Situationsbewertung auf Basis des aktuellen Kotrollflusses durch Analyse des Aufrufstapels durchgeführt. Danach wurde auch der vergangene Kontrollfluss in die Bewertung des Zustandes einbezogen.

Des Weiteren wurde eine mögliche Analyse gezeigt, sie sich nicht nur auf den Kontrollfluss einer Applikation bezieht, sondern die Kontrollflussanalysen um zusätzliche Datenanalysen erweitert. Hierzu wurden Parameter von ausgeführten Methoden solange gespeichert wie die Methode auf dem Aufrufstapel lag. Somit konnten die Parameter aller Methoden des Aufrufstapels für diese Form der Analyse herangezogen werden auch wenn sie zum Zeitpunkt der Situationsbewertung normalerweise nicht vorlagen.

Um dem Benutzer Programmierarbeit abzunehmen, wurde die Annotationsverarbeitung vorgestellt und auch zur Implementierung der Situationsbewertung genutzt. Dadurch ist es nicht mehr notwendig alle Komponenten per Hand zu implementieren, da mittels Annotationsverarbeitung vieles automatisch erzeugt werden kann. Es wurden alle Komponenten gezeigt, die für die automatische Codegenerierung bei der Situationsbewertung auf Basis des aktuellen Kontrollflusses nötig sind. Einige dieser Komponenten sind direkt auf andere Verfahren übertragbar, so dass diese Diplomarbeit den Weg aufzeigt, der gegangen werden muss um weitere Analyseverfahren in die ObjectTeams/Java Entwicklungsumgebung zu integrieren.

9.2 Ausblick

9.2.1 Umstellung auf Java6

Da die ObjectTeams/Java Entwicklungsumgebung auf Eclipse basiert, wurde die Annotationsverarbeitung mittels der Eclipse Plattform vorgestellt. Diese Annotationsverarbeitung basiert auf dem Java5 Standard und wird somit mittels des Annotation Processing Tools durchgeführt. Das Annotation Processing Tool wird hierbei nach dem Kompilieren der Klassen gestartet. Seit Java6 wird die Annotationsverarbeitung vom Compiler erledigt. Auch hier wird sie nach dem Kompilieren gestartet aber dies sind nun nicht mehr zwei separate Arbeitsschritte. Für die Anpassung an Java6 muss untersucht werden wie die Schnittstellen der Annotationsverarbeitung aussehen. Außerdem müssen die hier gezeigten Implementierungen an die neuen Schnittstellen angepasst werden. Des Weiteren bleibt zu klären wie die Annotationsverarbeitung in Java6 in die Entwicklungsumgebung Eclipse integriert werden kann.

9.2.2 Codeerzeugung durch abstrakte Syntaxbäume

Der `AnnotationProcessor`, der für die Erzeugung des Codes verantwortlich ist, ist so implementiert, dass der zu erzeugende Quelltext als Zeichenkette in eine Datei geschrieben wird. Die Erstellung dieser Zeichenkette wird bisher nur mittels des Datentyps `String` gemacht. Es werden also nur mehrere `Strings` aneinandergereiht und in eine Datei geschrieben. Es ist jedoch auch möglich den Quelltext einer Klasse über einen neu angelegten abstrakten Syntaxbaum zu realisieren. Dabei wird im `AnnotationProcessor` der abstrakte Syntaxbaum kreiert und dieser kann dann den resultierenden Quelltext in eine Datei schreiben. Der Quellcode

der dabei entsteht, ist nach den in der Entwicklungsumgebung eingestellten Regeln formatiert und, durch die Verwendung des abstrakten Syntaxbaumes, zwingend syntaktisch korrekt. Bei der Codeerzeugung mittels `Strings` muss der Quellcode natürlich auch syntaktisch korrekt sein, allerdings muss dies aufwendiger geprüft sein. Für die Umsetzung dieses Verfahrens muss untersucht werden, wie der abstrakte Syntaxbaum erzeugt wird und dieser dann in eine Datei geschrieben wird. Dabei besteht jedoch kein direkter Zugriff auf die Datei mittels eines Objektes vom Typ `File`, da die Datei vom Annotation Processing Tool angelegt wird und somit nur der `OutputStream` zur Verfügung steht.

9.2.3 Teamaktivierung

Sowohl bei der Situationsbewertung auf Basis des aktuellen Kontrollflusses als auch bei den Tracematches werden zusätzliche Teams erzeugt. Damit die zusätzliche Funktionalität, die diese Teams bieten, genutzt werden kann, müssen die Teams aktiviert werden. Diese Teamaktivierung muss bisher per Hand, nach der Annotationsverarbeitung, ausgeführt werden. Damit der Benutzer so wenig wie möglich selber erledigen muss, sollte die Teamaktivierung für die erzeugten Teams automatisiert werden. Hierbei muss untersucht werden, wie die Teams automatisch aktiviert werden können. Die Teamaktivierung könnte dabei über eine zusätzliche Konfigurationsdatei ausgeführt werden, die bei der Erzeugung der Teams ebenfalls erzeugt wird. Dies ist jedoch nur ein Vorschlag um die Teamaktivierung zu automatisieren. So wäre es auch interessant zu untersuchen ob die Teams auch explizit im Quellcode oder über die Laufzeiteigenschaften der Applikation automatisch aktiviert werden können.

9.2.4 Compileranpassungen

Für die Steuerung der Callin-Bindungen bei der Situtationsbewertung auf Basis des aktuellen Kontrollflusses ist es notwendig einen Guard für die annotierte Callin-Bindung zu erzeugen bzw. einen bestehenden Guard zu modifizieren. Des Weiteren muss eine Team-Level-Methode erzeugt werden, die von dem erzeugten Guard aufgerufen werden kann. Zurzeit muss dies noch per Hand erledigt werden. Da die Annotation, die an der Callin-Bindung steht, auch im Compiler zur Verfügung steht, können die genannten Aktionen auch automatisch vom Compiler erledigt werden. Da sowohl der Guard als auch die genannte Team-Level-Methode keine zusätzliche Logik enthalten, kann beides ohne manuelle Eingriffe automatisch erstellt werden.

9.2.5 Matchingalgorithmus des Tracematching

Bei dem in der Arbeit vorgestellten Tracematchingansatz wird ein Matchingalgorithmus benötigt um eine Teilsequenz in der gespeicherten Ereignissequenz zu finden. Im Zuge dieser Arbeit wurde ein einfacher Matchingalgorithmus eingeführt. Dieser einfache Algorithmus ist nicht in der Lage die Ereignisse von unterschiedlichen Objekten und unterschiedlichen Threads zu handhaben. Hier muss ein Weg gefunden werden wie die Objekte sinnvoll gespeichert werden können und in den Matchingalgorithmus einfließen können. Für die Speicherung der Objekte muss eine Strategie ausgearbeitet werden, wie die gespeicherten Referenzen gelöscht werden, damit der Garbagecollector den Speicher, den die Objekte benötigen, wieder freigeben kann, wenn die Objekte außerhalb des Tracematchings nicht mehr benötigt werden.

Mit [All05] wurde bereits ein Tracematchingansatz für die aspektorientierte Sprache AspectJ implementiert. In einer weiterführenden Arbeit könnte untersucht werden, ob dieser Ansatz oder Teile davon auch für ObjectTeams/Java übernommen werden können. Dazu ist es nötig Schnittstellen innerhalb dieses Ansatzes zu identifizieren und diese in ObjectTeams/Java umzusetzen. Da dort Lösungen für den Umgang mit gespeicherten Objekte und Ereginissen von unterschiedlichen Threads implementiert sind, wäre es sinnvoll den dort implementierten Matchingalgorithmus genauer zu untersuchen und für ObjectTeams/Java zu adaptieren.

Literaturverzeichnis

[All05] Allen, Chris u.a.: Adding Trace Matching with Free Variables to AspectJ. 2005

[Gam96] Gamma, Erich u.a.: Entwurfsmuster, Elemente wiederverwendbarer objektorientierte Software, Addison-Wesley. 1996

[HHM07] Herrmann, Stephan; Hundt, Christine; Mosconi, Marco: ObjectTeams/Java Language Definition – version 1.0. 2007

[Ull07] Ullenboom, Christian: Java ist auch eine Insel, 6. Auflage, Galileo Computing. 2007

[Wik08] Wikipedia: Aspektorientierte Programmierung, URL: http://de.wikipedia.org/wiki/Aspektorientierte_Programmierung [Stand 07.2008]

Eidesstattliche Erklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, 24.07.2008

Sven Kampfhenkel