Jürgen Widiker

Policy-basierte Zugriffskontrolle für Joinpoints in der aspektorientierten Sprache ObjectTeams/Java

Diplomarbeit

Berlin, 10. April 2007

Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation und Ziele 2

1.2 Gliederung der Arbeit 4

2 Einführung in ObjectTeams/ Java **5**

2.1 Konzepte der Sprache 5

2.1.1 Teams und Rollen 6

2.1.2 Callin- und Callout-Bindungen 7

2.1.3 Decapsulation 9

2.2 Die Webestrategie 10

2.2.1 Das Classfile-Format 11

2.2.2 Die ObjectTeams/Java-Laufzeitumgebung 15

3 Sicherheitskonzepte in Java **17**

3.1 Policies und Access Controller 18

3.2 Schlüssel und Zertifikate 22

3.3 Digitale Signaturen und Code-Signierung 25

3.4 Tools zur Signierung 27

4 Szenarien und technische Anforderungen **31**

4.1 Beschreibung 32

4.2 Angreifermodell und Angriffsszenarien 33

4.3 Ansatzverfeinerung 35
4.3.1 Benutzerkategorien 36
4.3.2 Zugriffskategorien von Joinpoints 37
4.3.3 Entwicklungsversion und Release 38
4.3.4 Schlüsselarten 39
4.3.5 Policies 40
4.3.6 Ausführungsmodi 40
4.3.7 Behandlung von Zugriffsrechteverletzungen 42
4.4 Anwendungsszenarien 43
4.4.1 Aspektentwicklung 43
4.4.2 Ausführung der erweiterten Basisanwendung 44

5 Deklaration von Joinpoints 47
5.1 Joinpoints in ObjectTeams/Java 49
5.2 Joinpoint Policy 50
5.3 Request Policy 56

6 Erweiterung der ObjectTeams/Java-Laufzeitumgebung 59
6.1 Policy-Parser und Zugriffsrechteverwaltung 59
6.1.1 Parser-Klassen 60
6.1.2 Infrastruktur zur Verwaltung von Zugriffsrechten . . . 60
6.2 Join Point Access Controller 64
6.2.1 Aktivierung 65
6.2.2 Verifizierung des Basispaketes 66
6.2.3 Verifizierung der Aspekte und Zugriffsrechtekontrolle . 67

7 Das Packaging Tool 73
7.1 Anforderungen 73
7.2 Entwurf 74
7.2.1 Model-View-Controller 75

7.2.2 Die Benutzeroberfläche 75
7.2.3 Die `Controller`-Klasse 76
7.2.4 Generierung der Policy-Dateien 77
7.3 Verwaltung von Basispaketen 78
7.4 Signieren von Java-Archiven 81

8 Zusammenfassung und Ausblick 83
8.1 Zusammenfassung 83
8.2 Ausblick 85

A Dateiformate für Policies 91
A.1 Joinpoint Policy (EBNF) 91
A.2 Request Policy (EBNF) 92

Literaturverzeichnis 93

Erklärung 97

Abbildungsverzeichnis

3.1 Realisierung des Sandkastens 19
3.2 Überprüfung der Zugriffsrechte 22
3.3 Signierung und Verifizierung einer Nachricht nach DSA . . . 26

4.1 Architektur 33
4.2 Anwendungsszenario 45

6.1 Die Parser-Klassen 60
6.2 Infrastruktur zur Verwaltung von Zugriffsrechten 61
6.3 Methodendeklaration in der Oberklasse 70
6.4 Einordnung des JPAC in die Sicherheitsarchitektur 71

7.1 Das Paket `generator` 77

Kapitel 1

Einleitung

Das Konzept der Objektorientierung, das seit Jahrzehnten die Softwareentwicklung prägt, fördert Flexibilität und Wiederverwendbarkeit von Softwareprodukten. Die Grundidee der objektorientierten Programmierung ist Kapselung von funktionalen Anforderungen, genannt *Core-Level-Concerns*, in separaten Modulen. Die Modularisierung geschieht meistens in Form von Klassen oder einzelnen Funktionen. Doch diese Sicht vernachlässigt eine weitere Gruppe von Kriterien. Diese können das gesamte System betreffen und sind somit quer über die Anwendung zerstreut, was sich auch in deren Bezeichnung widerspiegelt: *Crosscutting Concerns*. Durch Modularisierung wird der Umgang mit diesen modulübergreifenden *Aspekten* erheblich erschwert. Das klassische Beispiel eines solchen *Aspektes* ist Logging.

Zur Behandlung von Crosscutting Concerns wurde die objektorientierte Programmierung um aspektorientierte Konzepte erweitert. Die aspektorientierte Programmierung definiert eine neue Dimension der Modulbildung. Anders als im klassischen objektorientierten Paradigma steht in der Aspektorientierung die Kapselung von Aspekten im Vordergrund. Der Aspektcode bildet ein eigenständiges Modul und wird mit dem Programmcode der Basisanwendung[1] verknüpft. Diese Technik ermöglicht Erweiterung von Anwendungen um Aspekte. Das Zusammenfügen des Basissystems und Aspekten wird auf der technischen Ebene realisiert, indem der Aspektcode mit dem Code der Basisanwendung *verwoben* wird. Beim Weben werden einzelne Abschnitte des Aspektcodes in den Basiscode an wohldefinierten Stellen im Programmfluss, genannt *Joinpoints*, eingefügt.

Die Entstehungsgeschichte des aspektorientierten Paradigmas spiegelt sich auch in der Entwicklung der aspektorientierten Sprachen wider. Aspektorientierte Sprachen basieren auf bereits existierenden objektorientierten

[1] Als Basisanwendung wird hier die zu erweiternde Anwendung bezeichnet

Sprachen und erweitern diese auf syntaktischer und konzeptueller Ebene.

Die aspektorientierte Sprache *ObjectTeams/Java* ist, wie der Name schon sagt, eine Erweiterung von *Java*, einer der bekanntesten objektorientierten Sprachen. ObjectTeams/Java führt Kapselung von Aspekten in Form von Klassen, genannt *Teams*, ein. Ein Basisobjekt kann innerhalb eines Aspektes eine bestimmte *Rolle* spielen. Eine Rolle ist somit ein Bestandteil eines Teams und *adaptiert* die entsprechende Basisklasse. Als Bezugspunkte zum Weben von Aspekten fungieren in der Basisanwendung Klassen, Methoden und Attribute. Dabei werden Rollenmethoden an die Methoden oder Attribute der adaptierten Basisklasse gebunden. Man unterscheidet zwei Arten von Bindungen der Rollenmethoden: *Callin*-Bindungen und *Callout*-Bindungen. Eine detaillierte Beschreibung des Sprachkonzeptes wird im Kapitel 2 gegeben.

1.1 Motivation und Ziele

Eine aspektorientierte Sprache wird nicht nur bei der Entwicklung neuer Anwendungen verwendet. Sie kommt auch dann zum Einsatz, wenn ein fertiges objektorientiertes System um zusätzliche Anforderungen erweitert werden soll. Sehr oft entsteht eine solche Notwendigkeit, wenn die Software bereits auf dem Markt ist. Die Entscheidung über die Erweiterung wird von Kunden oder einem Dritten getroffen, der diese Anwendung unmittelbar benutzt bzw. als Komponente in eine weitere Anwendung einsetzt. Abgesehen von den rechtlichen und wirtschaftlichen Kriterien steht einer Erweiterung um Aspekte kaum etwas im Wege. Der Quellcode der Basisanwendung muss nicht geändert werden. Zur Bestimmung von Joinpoints sind lediglich Kenntnisse über die Struktur der Basisanwendung vorausgesetzt. Diese Information kann der Dokumentation entnommen oder (sofern die Dokumentation nicht vorhanden ist) mit Hilfe zahlreicher Tools wie z. B. Decompiler für Java-Anwendungen ermittelt werden. Man kann also zwei Entwicklergruppen hervorheben. Die erste Gruppe bilden Entwickler der Basisanwendung, zu der zweiten gehören Entwickler von Aspekten, die die Anwendung erweitern.

Vertreter der beiden Gruppen verfolgen bei der Entwicklung jeweils eigene Interessen, die nicht unbedingt an Interessen der anderen Seite abgestimmt sind. Der Interessenkonflikt zwischen den beiden Parteien wird in [Oss06] dargestellt. Wie oben gesehen, wird der Konflikt dadurch verursacht, dass die traditionellen Vorstellungen über die Programmcode-Besitzerschaft in der aspektorientierten Softwareentwicklung nicht mehr zutreffen.

Der Eigentümer[2] der Basisanwendung verliert beim Aspektweben die Kontrolle über seinen Code, denn der laufende Programmcode wird vom separat implementierten Aspekt beeinflusst und entspricht nicht dem ursprünglichen Source-Code der Basisanwendung. Der Entwickler ist verantwortlich für die Korrektheit seines Codes und ist dieser Verantwortung bewusst. Aus diesem Grund möchte der Eigentümer der Basisanwendung sein Produkt gegen ungewollte Änderungen und Erweiterungen schützen. In der objektorientierten Programmierung steht ihm das Konzept der Datenkapselung zur Hilfe. In der aspektorientierten Welt erhält jedoch die Kapselung eine neue Dimension, indem objektübergreifende Aspekte in neuen Objekten gekapselt werden. Die traditionelle Datenkapselung kann durchbrochen werden und reicht somit nicht aus, um wichtige Teile der Anwendung wie z. B. Klasseninvarianten vor Änderungen zu schützen. Es müssen Verbindungspunkte, an denen der Aspektcode verknüpft werden kann, gekapselt werden. Es handelt sich also um die Kapselung von Joinpoints.

Auf der anderen Seite möchte der Aspekteigentümer über einen gewissen Grad an Flexibilität verfügen, um die Anwendung an neue Anforderungen anzupassen. Die Anwendung muss eine Erweiterung zulassen. Das heißt, der Aspektentwickler muss die Erlaubnis haben, den Programmcode der Basisanwendung zu inspizieren, damit relevante Joinpoints ermittelt werden können. Nicht weniger wichtig ist die Erlaubnis zur Adaptierung der Basisanwendung an den ermittelten Stellen.

Es muss also eine Lösung gefunden werden, die für die beiden Seiten zufrieden stellend ist. Ossher spricht dabei von einem sozialen Problem zwischen den beiden Entwicklergruppen, die für unterschiedliche Teile der Anwendung verantwortlich sind. Als Kompromiss stellt er den Ansatz von *confirmed join points* vor. Dieser basiert auf der Kapselung von Joinpoints und anschließenden Deklaration von zugelassenen Pointcuts[3], die bestimmte zur Freigabe ausgewählte Joinpoints matchen. In [HHP06] wird der Bezug von confirmed join points zu ObjectTeams/Java hergestellt. Dabei werden eventuelle Vorteile einer solchen Technik am Beispiel der Adaptierung von Eclipse-Plugins mit Hilfe von ObjectTeams/Java diskutiert.

Im Rahmen dieser Diplomarbeit soll eine Technik entwickelt werden, die Kapselung und gezielte Freigabe von Joinpoints in der aspektorientierten Sprache ObjectTeams/Java realisiert. Hier wird der Begriff *Joinpoint* als Bezugspunkt für Callins und Callouts definiert. Dies ist eine Stelle im Programmcode der Basisanwendung, an der ein Aspekt durch eine Callin- oder Callout-Bindung verknüpft werden kann. Die neue Technik soll dem

[2] Wie auch in der Arbeit von Harold Ossher [Oss06] ist hier unter dem Begiff *Eigentümer* der Entwickler gemeint, also diejenige Person, die für die Entwicklung des Programcodes verantwortlich ist.

[3] Ein *Pointcut* definiert eine Menge von Joinpoints, die für einen Aspekt relevant ist.

Eigentümer der Basisanwendung eine Möglichkeit bieten, sein Produkt vor ungewollten Erweiterungen durch ObjectTeams/Java-Aspekte zu schützen. Gleichzeitig soll dem Aspekt-Eigentümer exklusiver Zugriff auf bestimmte Joinpoints ermöglicht werden.

1.2 Gliederung der Arbeit

Die zwei nachfolgenden Kapitel befassen sich mit dem theoretischen Hintergrund dieser Arbeit. Kapitel 2 gibt eine Einführung in ObjectTeams-Konzepte sowie einen Einblick in die ObjectTeams-Architektur. Kapitel 3 beschreibt Sicherheitskonzepte in der Programmiersprache Java, welche bei der Realisierung des in Kapitel 4 definierten Konzeptes zum Einsatz kommen.

Kapitel 4 leitet die Entwurfsphase des Konzeptes ein, indem es Anwendungsszenarien beschreibt und technische Anforderungen an die zu entwickelnde Technik aufstellt.

In Kapitel 5 wird das Beschreibungsformat für Joinpoints definiert. Das Beschreibungsformat bildet eine Schnittstelle zwischen dem Entwurf und der Implementierung der in dieser Arbeit zu entwickelnden Komponenten.

Die beiden nächsten Kapitel beschreiben die Implementierungsphase und bilden somit den praktischen Teil der Arbeit. Kapitel 6 befasst sich mit der Erweiterung der ObjectTeams-Laufzeitumgebung um Sicherheitsmechanismen, die die Zugriffskontrolle auf Joinpoints realisieren. Kapitel 7 erläutert die Entwicklung des Packaging Tools, das dem Entwickler Unterstützung beim Erstellen von Beschreibungsdateien für Joinpoints bietet und die entsprechenden Klassen der Basisanwendung signiert.

Kapitel 8 fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf eventuelle Erweiterungen des vorgestellten Konzeptes.

Kapitel 2

Einführung in ObjectTeams/ Java

Als aspektorientierte Sprache führt ObjectTeams eine neue Art der Aspektmodularisierung ein. Diese basiert auf der Kapselung von Kollaborationen in Modulen, die das Verhalten einzelner Objekte zusammefassen. Im Kontext von ObjectTeams wird ein solches Objekt als *Basisobjekt* bezeichnet. Kollaborationen bilden neben der strukturellen Organisation in Form von Klassen die zweite Dimension für den Aufbau von objektorientierten Anwendungen [Her02]. Die Tatsache, dass ObjectTeams eine Erweiterung der objektorientierten Programmiersprache Java ist, erfordert die Kompatibilität der von ObjectTeams eingeführten Konzepte zu den bereits existierenden objektorientierten Java-Konzepten spätestens dann, wenn die implementierte Anwendung ausgeführt wird.

In diesem Kapitel werden die wichtigsten Konzepte von ObjectTeams und deren technische Realisierung erläutert. Dies liefert das Basis- und Hintergrundwissen, das für die weitere Arbeit erforderlich ist.

2.1 Konzepte der Sprache

Zu den grundlegenden Konzepten von ObjectTeams gehören Modularisierung von Kollaborationen in Form von *Teams* und Kapselung des Verhaltens eines bestimmten Objektes (Basisobjektes) innerhalb der Kollaboration in einer *Rolle*. Die funktionale Beziehung zwischen Basisobjekten und deren Rollen wird durch die Definition von *Callin-* und *Callout-Bindungen* hergestellt. Diese stehen mit der *Decapsulation*, einem weiteren Konzept, im Zusammenhang. In nachfolgenden Abschnitten werden die Konzepte

einzeln vorgestellt.

2.1.1 Teams und Rollen

Ein *Team* ist ein Modul, das die Eigenschaften von Klassen und Paketen kombiniert, und wird als eine Java-Klasse mit dem Schlüsselwort `team` deklariert:

```
public team class MyTeam {}
```

Als Klasse kann ein Team eine Reihe von Attributen deklarieren, die den Zustand einer Team-Instanz definieren, sowie Methoden implementieren, die diesen Zustand ändern. Als Paket kapselt eine Team-Instanz eine Aggregation von Objekten (Rollen) ein, die das Verhalten der korrespondierenden Basisobjekte innerhalb der Kollaboration (Team) darstellen. Auf der Implementierungsebene wird eine Rollenklasse als innere Klasse der Team-Klasse realisiert. Somit ist jede innere Klasse eines Team auch eine Rollenklasse:

```
public team class MyTeam {
    public class MyRole1 {}
    public class MyRole2 {}
}
```

Je nachdem, ob einer Rolle eine Basisklasse zugeordnet wurde oder nicht, unterscheidet man zwischen *gebundenen* und *ungebundenen* Rollen. Eine Rolle ist gebunden, wenn sie eine `playedBy`-Beziehung deklariert:

```
public team class MyTeam {
    public class MyRole playedBy MyBase{}
}
```

Durch die Deklaration der `playedBy`-Beziehung *adaptiert* die Rollenklasse die angegebene Basisklasse, somit wird jedes Rollenobjekt mit einer Instanz der Basisklasse assoziiert. Ein Basisobjekt kann in einer Kollaboration unterschiedliche Rollen spielen, also kann ein Team mehrere Rollenklassen enthalten, die an dieselbe Basisklasse gebunden sind. Das Gleiche gilt auch für Rollenklassen, die aus unterschiedlichen Teams stammen.

Die Funktionalität eines Teams bezieht sich auf die Funktionalität seiner Rollen. Zur schrittweisen Verfeinerung dieser Funktionalität wird in ObjectTeams *implizite Rollenvererbung* unterstützt. Erbt ein Team von einem anderen Team (durch die in Java übliche `extends`-Klausel), so werden dessen Rollen automatisch vererbt. Es besteht die Möglichkeit, die implizit vererbte

Rolle zu verfeinern, indem man die Rolle in der abgeleiteten Klasse überschreibt. Zugriffe auf die Rollen-Oberklasse erfolgen mit Hilfe eines von ObjectTeams neu eingeführten Konstruktes `tsuper()`. Die Verfeinerung erstreckt sich auch auf die `playedBy`-Beziehung: ungebundene Rollen können im abgeleiteten Team an eine Basisklasse gebunden werden:

```
public team class MyTeam1 {
    public class MyRole {
        void m1() {}
    }
}

public team class MyTeam2 extends MyTeam1 {
    public class MyRole playedBy MyBase{
        void m2() {}
    }
}
```

Die Bindung einer Rollenklasse an eine Basisklasse in einem Sub-Team gibt dem Entwickler die Möglichkeit, die Funktionalität der Rollenklasse von einer konkreten Basisklasse zu trennen. In diesem Fall findet im abgeleiteten Team keine Spezialisierung der Rollenklassen statt, diese werden lediglich an konkrete Basisklassen gebunden, sodass die Team-Klasse als *Konnektor* betrachtet werden kann.

Adaptierung von Basisklassen durch korrespondierende Rollen hat nur dann Effekt, wenn die entsprechende Team-Instanz zur Laufzeit der Anwendung aktiv ist. Aktivierung bzw. Deaktivierung von Team-Instanzen legt fest, ob das Verhalten der Basisobjekte durch die entsprechenden Rollenobjekte verändert werden soll. Zu diesem Zweck stellt die Klasse `Team` (Oberklasse jeder Team-Klasse, analog zur Klasse `Object` in Java) Methoden `activate` und `deactivate` bereit.

2.1.2 Callin- und Callout-Bindungen

Die Änderung des Verhaltens von Basisobjekten wird in Methoden der Rollenklasse realisiert, die mit der jeweiligen Basisklasse in der `playedBy`-Relation steht. Diese Methoden werden an Methoden oder Attribute der Basisklasse *gebunden*. Somit wird die Verknüpfung der Rolle und der entsprechenden Basisklasse durch Methodenbindungen, die in der Rollenklasse deklariert werden, erreicht. Man unterscheidet zwei Arten von Methodenbindungen:

- *Callin*-Bindungen realisieren Änderungen am Kontrollfluss der Basis-

anwendung, indem sie das Abfangen von Aufrufen der Basismethoden ermöglichen. Dabei wird die Steuerung an die gebundene Rollenmethode übergeben und wieder an die Basisklasse zurückgegeben, sobald die Ausführung der Rollenmethode abgeschlossen ist.

- *Callout*-Bindungen setzen Delegation von Aufrufen der Rollenmethoden an die korrespondierenden Methoden des Basisobjektes um.

Callin-Bindungen existieren in drei Ausprägungen, je nachdem, wann die Rollenmethode ausgeführt werden soll:

- Bei einem *before*-Callin wird die Rollenmethode vor der Basismethode ausgeführt.
- Bei einem *after*-Callin erfolgt die Ausführung der Rollenmethode nach der Ausführung der Basismethode.
- Bei einem *replace*-Callin wird die Ausführung der Basismethode durch die der Rollenmethode ersetzt.

Eine Rollenmethode, die durch einen *replace*-Callin gebunden ist, kann die korrespondierende Basismethode durch einen *Base Call* aufrufen. Nach dem Base Call übergeht die Kontrolle wieder an die Rollenmethode. Zur Deklaration von Callins wird das Symbol `<-` verwendet. Im folgenden Beispiel wird die gebundene Rollenmethode `roleMethod` nach der Basismethode `baseMethod` ausgeführt.

```
public class MyBase {
    public void baseMethod() {}
}

public team class MyTeam {
    public class MyRole playedBy MyBase{
        void roleMethod() {}

        roleMethod <- after baseMethod;
    }
}
```

Eine Callout-Bindung ermöglicht Zugriffe aus der Rollenklasse heraus auf Methoden und Attribute der korrespondierenden Basisklasse. Somit gibt es zwei Arten von Callout-Bindungen:

- *Callout auf eine Methode* der Basisklasse: Der Aufruf der Rollenmethode wird an die Basismethode delegiert. Da die Rollenmethode lediglich die Delegation des Aufrufes realisiert, fungiert sie als Stell-

vertreter der Basismethode innerhalb der Rolle, und wird daher als abstrakt deklariert.

- *Callout auf ein Attribut* der Basisklasse: Die Rollenmethode dient als eine *Getter*- bzw. *Setter*-Methode für das gebundene Attribut der Basisklasse. Somit unterscheidet man zwischen einem *get*-Callout und einem *set*-Callout auf ein Attribut. Die Rollenmethode wird auch wie bei einem Callout auf eine Methode als abstrakt deklariert und hat die gleiche Signatur wie eine Getter- bzw. Setter-Methode für das Attribut der Basisklasse.

Die Deklaration eines Callouts erfolgt mit Hilfe des Symbols `->`. Nachfolgendes Beispiel zeigt Deklaration eines Callouts auf eine Methode:

```
public class MyBase {
    public void baseMethod() {}
}

public team class MyTeam {
    public class MyRole playedBy MyBase{
        abstract void roleMethod();
        ...
        roleMethod -> baseMethod;
    }
}
```

Die Callin- und Callout-Bindungen sind wirksam, wenn die entsprechende Team-Instanz aktiv ist (s. Kap. 2.1.1).

2.1.3 Decapsulation

Datenkapselung *(Encapsulation)* gehört zu den wichtigsten Konzepten der objektorientierten Programmierung und realisiert den Schutz von Klasseninvarianten und Objektzuständen. Nach dem Prinzip der Datenkapselung sind Implementierungsdetails einer Klasse der Außenwelt verborgen. Zugriffe auf das Innere der Klasse erfolgen über eine fest definierte Schnittstelle, die auf der Sichtbarkeit der einzelnen Klassenelementen basiert. Sichtbarkeit eines Elements[1] wird in dessen *Access Modifier* deklariert. Java bietet vier Sichtbarkeitsstufen zur Realisierung der Datenkapselung: *private*, *protected*, *public* und *default* (falls kein Modifier angegeben wurde) [JLS]. Nur als *public* deklarierte Elemente sind für jede beliebige Klasse sichtbar und somit von überall aus zugreifbar.

[1] Als *Elemente* werden hier innere Klassen, Methoden oder Attribute einer Klasse bezeichnet.

Die durch Klassen bereitgestellte Schnittstelle reicht in den meisten Fällen nicht aus, um das Verhalten deren Instanzen ändern zu können. Dafür sind Zugriffe auf das Innere der Klasse notwendig. Als Lösung wurde in ObjectTeams das Konzept der *Decapsulation (Entkapselung)* realisiert. Decapsulation ist Durchbrechen der Kapselung, wonach Bindungen auf nicht-sichtbare Methoden ermöglicht werden. Das typische Beispiel dafür ist eine Callin- oder Callout-Bindung auf eine als *private* deklarierte Basismethode. Generell spricht man von Decapsulation, wenn ein Zugriff auf ein nach dem OO-Konzept nicht erreichbares Element stattfindet. Somit betrifft das Decapsulation-Konzept in ObjectTeams nicht nur Methodenbindungen, sondern auch Rollenbindungen *(Role-Base-Bindings)*, die in Abschnitt 2.1.1 als `playedBy`-Relationen vorgestellt wurden.

2.2 Die Webestrategie

In Kapitel 2.1 wurden die wichtigsten Konzepte von ObjectTeams vorgestellt. Dieses Kapitel befasst sich dagegen mit der Umsetzung dieser Konzepte. Die Verknüpfung von Aspekten mit der Basisanwendung wird auf der technischen Ebene realisiert und mit dem Begriff *Weben* bezeichnet. Beim Weben wird die Implementierung des Aspektes in die der Basisanwendung eingefügt, wobei der Webeprozess auf der Bytecode-Ebene stattfindet. Im Vordergrund steht die Umwandlung der aspektorientierten Konzepte, die in ObjectTeams auf der Sprachebene definiert sind, in den Java-konformen Bytecode. Für das Weben von Aspekten sind in ObjectTeams zwei Komponenten verantwortlich, wodurch im Webeprozess zwei Phasen unterschieden werden können.

Die erste Phase bezieht sich auf das Übersetzen des Aspektcodes durch den *ObjectTeams-Compiler*. Dieser führt den ersten Teil des Webeprozesses durch und bereitet die zweite Phase des Webens vor. Dazu gehört beispielsweise Folgendes:

- Bearbeitung der *Callout-Bindungen*: In der Rollenklasse werden nicht-abstrakte Callout-Methoden erzeugt, die einen Aufruf der korrespondierenden Basismethode enthalten.
- Realisierung der *impliziten Vererbung*: In die erbenden Team-Klassen wird der Bytecode der vererbten Rollenklassen eingefügt.
- Generierung von *Bytecode-Attributen* in den Team- und Rollenklassen: Die Bytecode-Attribute beinhalten Informationen, die in der zweiten Phase des Webeprozesses verwendet werden (s. Kap. 2.2.1).

Die zweite Phase des Webeprozesses findet in der *ObjectTeams-Laufzeitumgebung* statt und befasst sich mit der Transformation der in Bytecode-Form vorliegenden Aspekt- und Basisklassen. Als Ergebnis dieser Transformation entsteht der Bytecode der Anwendung, den die *Java Virtual Machine* [JVM] ausführen kann. Die OT-Laufzeitumgebung beschäftigt sich im Rahmen des Webeprozesses unter Anderem mit folgenden Aufgaben:

- Verarbeitung der *Callin-Bindungen*: In den Bytecode der Basisklasse werden Methoden eingefügt, die die Dispatch-Logik zur Übergabe des Kontrollflusses an Rollenklassen realisieren.
- Umsetzung von *Base Calls*
- Generierung der Codes zur *Team-Aktivierung*: In den Bytecode der Basisklasse werden Methoden zum Registrieren von aktiven Team-Instanzen eingefügt.
- Durchführen der *Decapsulation*: Bei Elementen der Basisklasse, die bei Zugriffen aus dem Aspekt heraus nicht erreichbar sind, wird die Sichtbarkeit erweitert. Dies geschieht durch die Änderung des Modifiers.

Während die Laufzeitumgebung sowohl den Basis- als auch den Aspectcode verarbeitet, bleibt die Basisanwendung vom ObectTeams-Compiler unberührt. Dies heißt also, dass diese lediglich in Classfile-Form vorliegen muss, um das Weben zu ermöglichen.

Ursprünglich handelte es sich bei Bytecode-Transformationen in der OT-Laufzeitumgebung um das Weben zur *Ladezeit* der Anwendung *(Load Time Weaving)* [Hu03]. Inzwischen wurde ein Ansatz für das Weben zur Programmlaufzeit entwickelt. [Fl06]. In den nächsten Abschnitten wird das Format für Java-Classfiles sowie die Struktur einiger OT-Bytecode-Attribute vorgestellt. Anschließend wird ein Überblick der OT-Laufzeitumgebung gegeben.

2.2.1 Das Classfile-Format

Der Java-Quellcode wird beim Übersetzen durch den Compiler in Bytecode-Dateien, genannt *Classfiles*, umgewandelt. Jeder Classfile enthält den Bytecode von genau einer Klasse oder eines Interfaces und besteht aus den für *Java Virtual Machine (JVM)* interpretierbaren Anweisungen. Classfiles haben ein fest definiertes Format und weisen folgende Struktur auf:

- Der **Header** besteht aus einer Magic-Zahl `0xCAFEBABE`, die die Datei als ein Java-Classfile identifiziert. Zusätzlich enthält der Header zwei

Versionsnummer (`minor_version` und `major_version`). Diese spezifizieren Versionen der virtuellen Maschine, die das Format des Classfiles unterstützen.

- Der **Constant Pool** enthält konstante Informationen in Form einer Liste von Zeichenketten. Die Zeichenketten repräsentieren beispielsweise Namen von Methoden, Feldern[2], Klassen, Interfaces sowie andere konstante Strings innerhalb der Datei.

- In einem Block mit **Klasseninformationen** sind Name der aktuellen Klasse sowie Namen der Oberklasse und der implementierten Interfaces aufgeführt. Bei den Einträgen dieses Blockes handelt es sich um Referenzen auf den Constant Pool, wo die Namen in Form von konstanten Strings enthalten sind.

- **Liste mit Feldern** enthält Informationen zu allen Feldern, die die aktuelle Klasse (bzw. das aktuelle Interface) deklariert. Zu diesen Informationen gehören jeweils Name und Typ des Feldes als Verweise auf entsprechende Einträge im Constant Pool sowie Modifier und Bytecode-Attribute des Feldes.

- **Liste mit Methoden** beinhaltet Informationen über die von dieser Klasse (bzw. diesem Interface) deklarierten Methoden. Analog zur Liste mit Feldern sind es der Name und die Signatur der Methode sowie Modifier und Bytecode-Attribute. Ebenfalls in einem Attribut (sofern es sich bei der aktuellen Klasse nicht um ein Interface handelt) ist die Implementierung der Methode aufgeführt.

- **Bytecode-Attribute** der Klasse sind optional und enthalten zusätzliche Informationen über den Classfile. Dies sind beispielsweise Attribute *SourceFile* und *Deprecated*.

Der Block mit optionalen Bytecode-Attributen ermöglicht Speicherung von zusätzlichen Informationen, die in der Definition der virtuellen Maschine nicht spezifiziert sind. Bei der Bearbeitung des Classfiles ignoriert die JVM diese nicht-spezifizierten Attribute, sodass diese keine Rolle beim Ausführen der Anwendung spielen. In ObjectTeams wird der Block mit Attributen dazu verwendet, Informationen über die Aspektbindung an die OT-Laufzeitumgebung weiter zu reichen. Anhand des Inhaltes dieser Attribute wird die Transformation von Klasssen in der zweiten Phase des Webeprozesses angesteuert. Im Folgenden wird das Format von einigen für die

[2] Der Begriff *Feld* lässt in der deutschen Terminologie doppelte Interpretation zu. In diesem Abschnitt wird er als direkte Übersetzung aus dem Englischen *(field)* zur Bezeichnung von Attributen der Klasse verwendet, um eine Verwechslung mit dem Begriff *Bytecode-Attribut* zu vermeiden.

weitere Arbeit relevanten OT-Bytecode-Attributen vorgestellt. Bei der Beschreibung des Formates wird die Notation verwendet, die auch bei der Beschreibung des Classfile-Formates in [JVM] zum Einsatz kommt.

Informationen über die Callout-Bindungen werden im Attribut `CalloutMappings` gespeichert. Das Format sieht wie folgt aus:

```
CalloutMappings {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 bindings_count;
    CalloutBinding[bindings_count];
}

CalloutBinding {
    u2 role_method_name_index;
    u2 role_method_signature_index;
    u1 element_flags;
    u2 base_class_name_index;
    u2 base_member_name_index;
    u2 base_member_signature_index;
}
```

`CalloutMappings` ist eine Liste von Einträgen, die jeweils durch die Struktur `CalloutBinding` repräsentiert werden. `CalloutBinding` enthält Constant Pool-Referenzen auf den Namen und die Signatur der Rollenmethode, ein Byte mit Flags, den Namen der Basisklasse sowie den Namen und den Typ (im Fall eines Callouts auf eine Methode ist dies die Methodensignatur) des Elements (Feld oder Methode) der Basisklasse. `element_flags` zeigt an, ob es sich um einen Callout auf ein Feld oder eine Methode handelt, und gibt im Falle eines Callouts auf ein Feld dessen Art (`get` oder `set`) an.

Callin-Bindungen sind im Attribut `CallinMethodMappings` aufgeführt:

```
CallinMethodMappings {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 bindings_count;
    CallinMethodMapping[bindings_count];
}

CallinMethodMapping {
    u2 binding_file_name_index;
    u4 binding_line_number;
    u4 binding_line_offset;
    u2 binding_label_index;
    u2 role_method_name_index;
    u2 role_method_signature_index;
```

```
    u4 flags;
    u2 lift_method_name_index;
    u2 lift_method_signature_index;
    u2 binding_modifier_index;
    u2 base_method_mapping_count;
    BaseMethodMapping[base_method_mapping_count];
}

BaseMethodMapping {
    u2 base_method_name_index;
    u2 base_method_signature_index;
    u2 wrapper_name_index;
    u2 wrapper_signatur_index;
    u1 base_flags;
    u4 translation_flags;
}
```

Jede per Callin gebundene Rollenmethode wird durch die Struktur `CallinMethodMapping` repräsentiert. Da eine Rollenmethode an mehrere Basismethoden gebunden werden kann, enthält `CallinMethodMapping` eine Liste von Elementen, die Informationen über die gebundenen Basismethoden beinhalten (Unterstruktur `BaseMethodMapping`). Auf eine ausführliche Erläuterung der Strukturen wird an der Stelle verzichtet. Anzumerken ist es jedoch, dass das Attribut `CallinMethodMappings` analog zu `CalloutMappings` Namen und Signaturen der gebundenen Rollen- und Basismethoden enthält. Der Callin-Typ (*before*, *after* oder *replace*) ist im Constant Pool unter dem Index `binding_modifier_index` zu finden.

Die beiden oben vorgestellten Attribute werden in der Rollenklasse angegeben. Die von der Decapsulation betroffenen Felder und Methoden sind im Bytecode-Attribut `OTSpecialAccess` der Team-Klasse aufgeführt:

```
OTSpecialAccess {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 elements_count;
    OTSpecialAccessElement[elements_count];
}

OTSpecialAccessElement {
    u2 kind;
    (DacapsulatedMethodAccess | CalloutFieldAccess);
    u2 adapted_base_class_count;
    AdaptedBaseClass[adapted_base_class_count];
}

DacapsulatedMethodAccess {
    u2 class_name_index;
```

```
    u2 method_name_index;
    u2 method_signature_index;
}

CalloutFieldAccess {
    u2 field_access_flags;
    u2 class_name_index;
    u2 field_name_index;
    u2 field_type_index;
}

AdaptedBaseClass {
    u2 base_class_name_index;
    u2 adaption_flag;
}
```

Das Attribut hat eine etwas komplexere Struktur, die von der Art des adaptierten Basisklassenelementes abhängig ist. Informationen über die entkapselten Methoden und Felder sind in den Unterstrukturen `DacapsulatedMethodAccess` bzw. `CalloutFieldAccess` angegeben. Dabei handelt es sich um entkapselte Elemente von allen Basisklassen, die durch Rollen des aktuellen Teams adaptiert werden. Die durch die Decapsulation betroffenen Basisklassen sind jeweils in der Unterstruktur `AdaptedBaseClass` aufgelistet. Die in `OTSpecialAccess` aufgeführten Bindungen bilden eine Teilmenge aus der Vereinigung von in den Attributen `CalloutMappings` und `CallinMethodMappings` angegebenen Elementmengen.

2.2.2 Die ObjectTeams/Java-Laufzeitumgebung

Zum Weben von Aspekten verwendet die OT-Laufzeitumgebung eine Reihe von Transformer-Klassen, die Bytecode-Transformationen an Aspekten und Basisklassen durchführen. Im Laufe des Webeprozesses erfüllt jede der Transformer-Klassen eine bestimmte Aufgabe. Als Input für Transformationen dient der Inhalt von zusätzlichen Bytecode-Attributen, die zum Teil in Kapitel 2.2.1 vorgestellt wurden. Die Transformation jeder einzelnen Klasse wird mit dem Einlesen deren Bytecode-Attribute eingeleitet. Diese Funktion übernimmt die Funktion `scanClassOTAttributes` die in der Transformer-Oberklasse `ObjectTeamsTransformation` implementiert ist. Beim Einlesen der Attribute wird die Objektstruktur zur Verwaltung von Informationen über die Aspektbindungen erzeugt. Dazu zählen beispielsweise Methodenbindungen und Rolle-Basis-Bindungen. Als Repository zur Verwaltung der erzeugten Objektstruktur dient die Klasse `CallinBindingManager`, die dazu eine Reihe von statischen Methoden zur Verfügung stellt.

Transformation von Klassen verläuft iterativ, d. h. für jede Klasse werden alle Transformer nacheinander aufgerufen. Stellt die Transformer-Klasse nach dem Auswerten des Inhalts von Bytecode-Attributen fest, dass die Transformation der Klasse nicht notwendig ist, so wird diese Klasse dem nächsten Transformer übergeben.

Beim Transformieren des Bytecodes wird das Framework *BCEL (Byte Code Engineering Library)* verwendet. BCEL ermöglicht Manipulationen des Java-Bytecodes auf etwas abstrakteren Ebene und übernimmt die Behandlung von internen Details der Classfile-Formates wie beispielsweise Berechnung von Bytecode-Offsets.

Kapitel 3

Sicherheitskonzepte in Java

Die Grundlage zum Schutz eines Systems bietet die *Sicherheitsstrategie*, die beschreibt, welche Aktionen durch verschiedene Einheiten des Systems (dazu gehören auch Akteure) ausgeführt werden dürfen. Bei der Realisierung einer Sicherheitsstrategie kommen *Sicherheitsmechanismen* zum Einsatz. In [TvS] sind folgende wichtige Sicherheitsmechanismen hervorgehoben:

- *Verschlüsselung (Encryption)* ist die Umwandlung von Daten in eine nicht einfach interpretierbare Form. Der umgekehrte Prozess der Transformation von verschlüsselten Daten in die interpretierbare Ausgangsform wird mit dem Begriff *Entschlüsselung (Decryption)* bezeichnet.
- *Authentifizierung (Authentication)* bezeichnet die Identitätsprüfung der agierenden Einheit.
- *Autorisierung (Authorization)* beschäftigt sich mit der Erteilung von Zugriffsrechten auf Daten und Ressourcen.
- *Auditing* ist das detaillierte Protokollieren der getätigten Zugriffe.

Im deutschen Sprachraum bezeichnet man mit dem Begriff *Authentisierung* den Vorgang des Nachweises der eigenen Identität, während *Authentifizierung* auf den Vorgang der Identitätsprüfung deutet. Die Vorgänge werden im Englischen nicht unterschieden und fallen beide unter den Begriff *Authentication*.

Bei einer schnellen Verbreitung von Java spielten die in der Sprache realisierten Sicherheitskonzepte eine bedeutende Rolle. Das Sicherheitsmodell von Java basiert auf der Idee des *Sandkastens (Sandbox)* - einer

beschränkten Umgebung, die für eine Java-Anwendung definiert werden kann, wodurch der Spielraum der Anwendung auf diese Umgebung eingeschränkt wird. Der Sandkasten ist verantwortlich für Systemressourcen, die der Java-Anwendung zugeteilt wurden. Die Entscheidung darüber, welche Ressourcen und wie benutzt werden, kann vom Benutzer der Anwendung oder vom Systemadministrator getroffen werden. Für einige Anwendungen, wie z. B. *Applets* sind die Grenzen durch die Java-Plattform vorgegeben.

Zu den grundlegenden Sicherheitskonzepten in Java gehören unter anderem Sicherheitspolicies und Sicherheitsmanager zur Realisierung der Zugriffskontrolle, digitale Signierung zur Erkennung von Datenmanipulationen sowie Verwaltung von digitalen Schlüsseln zur Datenkodierung. Die Java-Plattform bietet mit dem Sicherheitspaket (`java.security`) eine erweiterbare Infrastruktur zur Unterstützung der oben genannten Aspekte. Die nachfolgenden Abschnitte geben eine detaillierte Beschreibung der einzelnen Konzepte.

3.1 Policies und Access Controller

Eine Sicherheitspolicy wird meistens in Form einer Textdatei deklariert und dient zur Verwaltung des Sandkastens. Durch die Rechtevergabe werden die Grenzen der Umgebung definiert, die der laufenden Anwendung zur Verfügung steht. Eine Policy beinhaltet einen Satz von Regeln *(Permissions)*, die genau spezifizieren, welche Aktivitäten die Klassen ausführen dürfen. Eine Permission besteht aus drei Elementen:

- Der *Typ* der Permission wird repräsentiert durch den Namen der entsprechenden Java-Klasse, die diese Permission implementiert.
- Der *Name* der Permission ist von ihrem Typ abhängig. So wird beispielsweise für die Permission vom Typ `java.io.FilePermission` der Name der Datei oder des Verzeichnisses erwartet für die/das diese Permission gelten soll. Es gibt auch Permissions, die keinen Namen enthalten.
- *Aktion* der Permission spezifiziert, was genau mit dem Zielobjekt gemacht werden darf, und steht wie auch der Name der Permission mit dem Permission-Typ im Zusammenhang.

Um Parameter des Sandkastens festzulegen, übergibt man den Pfad der Policy-Datei als Argument der *Java Virtual Machine (JVM)*:

```
-Djava.security.policy=<Pfad>
```

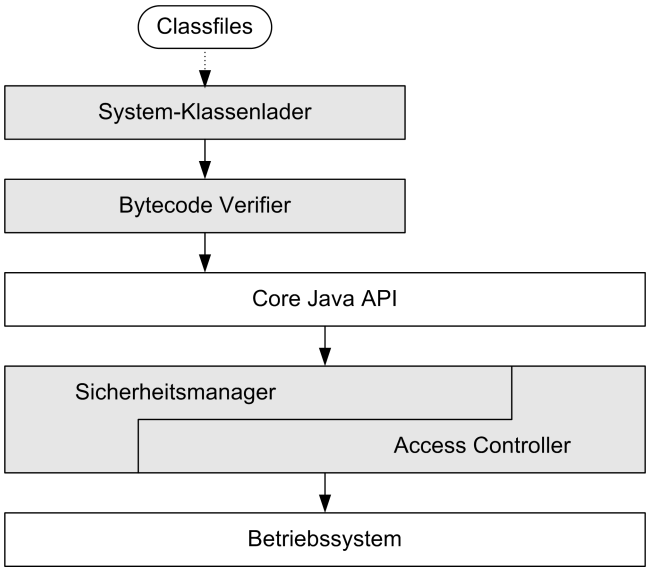

Abbildung 3.1: Realisierung des Sandkastens

[PFS] gibt eine detaillierte Beschreibung der Syntax von Policy-Dateien.

Die Funktionalität des Sandkastens wird durch drei folgende Einheiten definiert:

- *Klassenlader (Class Loader)* kapselt Informationen über Policies und Klassen ein.
- *Sicherheitsmanager (Security Manager)* fungiert als eine Schnittstelle, die von Java API zur Prüfung der Zugriffsrechte verwendet wird.
- *Access Controller* stellt die Standardimplementierung des Sicherheitsmanagers zur Verfügung.

Abbildung 3.1 veranschaulicht die Realisierung des Sandkastens für eine Java-Applikation. JVM verfügt über einen Mechanismus zum Laden der in Bytecode-Form vorliegenden Klassen. Beim Laden wird jede Klasse durch den Klassenlader einer *Sicherheitsdomäne (Protection Domain)* zugeordnet. Sicherheitsdomäne wird durch eine Policy definiert und ist eine Verknüpfung zwischen Permissions und einer bestimmten *Code-Ressource (Code Source)*, also einem Ort, von dem die Klassen geladen werden. Zusätzlich enthält eine Code-Ressource Informationen darüber, wer die zu ladenden Klassen signiert (s. Kap. 3.3) hat. Bevor die Klasse in die virtuelle Maschine gelangt, wird deren Bytecode einer Prüfung unterzogen, die sicher stellt, dass der Bytecode zur Sprachdefinition konform ist. Diese Aufgabe erfüllt der *Bytecode Verifier*.

Sicherheitsmanager ist eine Einheit zwischen dem Aufrufer einer beliebigen Bibliotheksfunktion und dem Betriebssystem, der Zugriffe auf alle Systemressourcen kontrolliert. Der Sicherheitsmanager entscheidet, ob die Aktionen (wie z. B. Netzzugriffe, Zugriffe auf das Dateisystem oder Systemvariablen) ausgeführt werden dürfen oder nicht. In Wirklichkeit leitet der Sicherheitsmanager die Aktionen an den Access Controller weiter, der dann die Aktionen anhand der in der Policy aufgeführten Permissions auswertet und die Entscheidung trifft, ob der Zugriff auf die Ressource gewährt werden darf. Es wird geprüft, ob jede zurzeit aktive Klasse die Erlaubnis hat, die Aktion auszuführen. Die Prüfung der Zugriffsrechte findet also zur Programmlaufzeit statt, nachdem die betroffene Klasse geladen und gelinkt wurde.

Die Duplizierung der Funktionalität des Sicherheitsmanagers durch den Access Controller hat einen historischen Hintergrund. Den Access Controller gibt es in Java erst seit der Version 1.2. In den Versionen davor übernahm die interne Logik des Sicherheitsmanagers die Aufgabe zur Festlegung der Sicherheitspolicy, nach der die Zugriffskontrolle realisiert werden sollte, wobei das Austauschen der Policy den Austausch des Sicherheitsmanagers erforderlich machte. Mit dem Erscheinen des Access Controllers können Sicherheitspolicies mit Hilfe von Policy-Dateien spezifiziert werden, was ein flexibleres Verfahren zur Definition von spezifischen Zugriffsrechten für konkrete Klassen ermöglicht. Der Sicherheitsmanager und der Access Controller implementieren die Sicherheitsmechanismen *Authentisierung* und *Autorisierung*.

Standardmäßig ist beim Starten eines Java-Programms kein Sicherheitsmanager aktiv, somit bekommt das Programm keine Einschränkungen und hat die volle Kontrolle über das System. Es gibt zwei Möglichkeiten, den Sicherheitsmanager für eine Java-Anwendung zu aktivieren. Die erste Möglichkeit besteht darin, aus dem Programmcode der Anwendung heraus die statische Methode `setSecurityManager()` der Klasse `System` mit der Übergabe einer `SecurityManager`-Instanz aufzurufen:

```
SecurityManager s = new SecurityManager();
if(System.getSecurityManager()==null) {
    System.setSecurityManager(s);
}
```

Die zweite Option ist die Aktivierung des Sicherheitsmanagers über die Kommandozeile beim Starten der Anwendung. Dafür wird die System-Property unter der Angabe des entsprechenden Argumentes der JVM gesetzt:

```
java -Djava.security.manager MainClass
```

Der Sicherheitsmanager stellt eine Reihe von Methoden zur Verfügung, die die Klassen der Java-Standard-Bibliothek aufrufen, um festzustellen, ob alle Klassen in der Aufruferkette die Erlaubnis haben, die entsprechende Aktion auszuführen. Klassen der Standard-Bibliothek haben die Erlaubnis, jede Aktion auszuführen. Die Methoden zur Sicherheitsüberprüfung, deren Namen mit dem Teilstring `check` beginnen, werden je nach Art der Ressource, die die Aktion beansprucht, in unterschiedliche Kategorien eingeteilt:

- Dateisystemzugriffe
  - `checkRead(..)`
  - `checkWrite(..)`
  - `checkDelete(..)` usw.
- Netzzugriffe
  - `checkConnect(..)`
  - `checkListen(..)`
  - `checkAccept(..)` usw.
- Schutz der JVM
  - `checkExec(..)`
  - `checkExit(..)` usw.
- Schutz von Anwendungsprozessen
  - `checkAccess(..)`
- Schutz von Systemressourcen
  - `checkPrintJobAccess(..)`
  - `checkPropertiesAccess(..)` usw.
- Schutz von Sicherheitsaspekten
  - `checkSecurityAccess(..)`
  - `checkPackageDefinition(..)` usw.

Methoden der Klassen aus Standardbibliothek, die eine aus der Sicht der Sicherheit sensible Aktion ausführen, enthalten einen Aufruf der entsprechenden Überprüfungsmethode des Sicherheitsmanagers. Diese können allerdings bei Notwendigkeit auch direkt aus einer Benutzerklasse heraus aufgerufen werden. Die Implementierung des Sicherheitsmanagers basiert vollständig auf dem Access Controller. Der Access Controller ist in Java

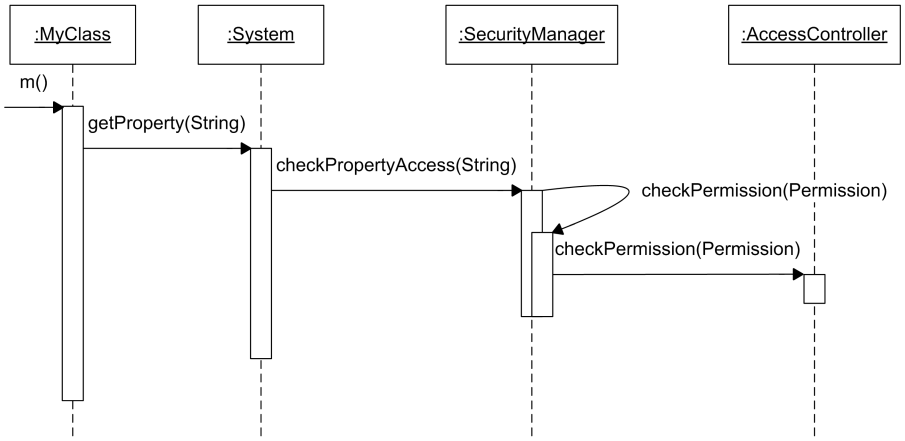

Abbildung 3.2: Überprüfung der Zugriffsrechte

durch die Klasse `AccessController` repräsentiert, die nicht instantiiert werden kann, dementsprechend sind alle Methoden der Klasse statisch. Die zentrale Rolle spielt die Methode `checkPermission`, die vom Sicherheitsmanager aufgerufen wird und ein `Permission`-Objekt entgegen nimmt. Die Permission wird unter Verwendung der in der Policy aufgeführten Berechtigungen überprüft. Ist der Zugriff erlaubt, terminiert die Methode, andernfalls wird eine `AccessControlException` geworfen.

Abbildung 3.2 zeigt ein Sequenzdiagramm einer Zugriffsrechteprüfung beim Lesen einer System-Property. Sollte die Methode `getProperty` der Klasse `System` feststellen, dass ein Sichergeitsmanager aktiv ist, wird dessen Methode `checkPropertyAccess` aufgerufen. Diese ruft über die eigene Methode `checkPermission` die gleichnamige Methode des Access Controllers auf.

3.2 Schlüssel und Zertifikate

Sicherheit basiert auf der Verwendung von Verschlüsselungstechniken (kryptografischer Methoden), die durch Schlüssel parametrisiert werden. Ein Schlüssel wird typischerweise durch eine Reihe von Bits repräsentiert und ist stets der Identität seines Besitzers zugeordnet.

Je nach Art des Verschlüsselungsverfahrens unterscheidet man zwischen *symmetrischen* und *asymmetrischen* Schlüsseln. Unter einem symmetrischen Schlüssel versteht man einen geheimen Schlüssel *(secure key)*, der sowohl bei der Ver- als auch Entschlüsselung verwendet wird. Bei einem asymmetrischen Schlüssel handelt es sich dagegen um ein Schlüsselpaar

aus einem geheimen *(private key)* und einem öffentlichen Teil *(public key)*. Der geheime Schlüssel bleibt immer im Besitz seines Eigentümers und wird von ihm vertraulich behandelt, während der öffentliche Schlüssel jeder beliebigen Person zugänglich ist. Die Asymmetrie bezieht sich auf die Verwendung des Schlüssels. Verschlüsselt man Daten mit dem geheimen Schlüsselteil, so können diese nur mit Hilfe des dazugehörigen öffentlichen Schlüssels entschlüsselt werden. Die beiden Schlüsselteile sind voneinander mathematisch abhängig, es ist jedoch unmöglich, mit vertretbarem Aufwand den geheimen Schlüssel aus dessen öffentlichem Teil herzuleiten. [St05]

Symmetrische Verschlüsselungstechniken kommen in der Regel bei der Datenübertragung zum Einsatz und setzen voraus, dass beide am Datenaustausch teilnehmenden Parteien über den Schlüssel verfügen. Bei asymmetrischen Verschlüsselungstechniken finden der geheime und der öffentliche Teile des Schlüssels unterschiedliche Anwendung, abhängig davon, welcher Sicherheitsmechanismus realisiert wird. Sollte Verschlüsselung (Encryption) umgesetzt werden, so benutzt man den öffentlichen Schlüssel zum Verschlüsseln der Daten und den korrespondierenden geheimen Schlüssel zur Entschlüsselung. Bei der Realisierung der Authentifizierung wird dagegen der geheime Teil zum Unterzeichnen der Daten verwendet, somit können die Daten mit Hilfe des öffentlichen Schlüssels verifiziert werden (s. Kap. 3.3).

Zu den bekanntesten Vertretern von symmetrischen Verschlüsselungsalgorithmen zählt *DES (Data Encryption Standard)*. DES ist für die Arbeit mit 64 Bit langen Datenblöcken ausgelegt, wobei die Länge des Schlüssels 56 Bit beträgt. Jeder Datenblock wird iterativ 16 Verschlüsselungsdurchgängen unterzogen, wodurch jeweils neue Blöcke gleicher Länge entstehen. Der Eingabeblock wird zu Beginn des Verschlüsselungsvorgangs permutiert, auf die verschlüsselte Ausgabe wird die umgehkerte Permutation angewendet. Eine detaillierte Beschreibung des Algorithmus ist in [Pf00] gegeben.

Aus der Gruppe von asymmetrischen Verfahren fand *DSA (Digital Signature Argorithm)*, ein Bestandteil von *Digital Signature Standard (DSS)*, eine große Verbreitung. Dieser Algorithmus, wie der Name bereits erklärt, ist zum Signieren von Daten konzipiert und basiert auf dem diskreten Logarithmus in endlichen Körpern. Die Funktionsweise von DSA wird in Kapitel 3.3 betrachtet.

In Java werden zur elektronischen Übermittlung von öffentlichen Schlüsseln *Zertifikate* verwendet. Ein Zertifikat ist ein Satz von strukturierten Daten mit dem eingebetteten öffentlichen Schlüssel. Darüberhinaus enthält ein Zertifikat Daten über den Eigentümer des öffentlichen Schlüssels, der

typischerweise eine konkrete Person oder eine Organisation ist. Die Daten dienen zur Authentisierung des öffentlichen Schlüssels und sind eine Teilmenge der in X.500-Standard spezifizierten Beschreibung des Aufbaus eines Verzeichnisdienstes. Innerhalb des Zertifikates sind die Daten in Form eines langen Strings zusammengefasst und werden mit dem Begriff *Distinguished Name (DN)* bezeichnet. Im Einzelnen sind das:

- *Common Name (CN):* Vollständiger Name des Eigentümers
- *Organizational Unit (OU):* Organisatorische Einheit, zu der der Eigentümer des Zertifikates gehört
- *Organization (O):* Organisation, zu der der Eigentümer gehört
- *Location (L):* Name der Stadt, in der sich der Eigentümer aufhält
- *State (S):* Name der Provinz, in der sich der Eigentümer aufhält
- *Country (C):* Land, in dem sich der Eigentümer aufhält

Eine Instanz, die das Zertifikat ausstellt, wird als *Zertifizierungsinstanz* bezeichnet. Ein Zertifikat kann jedoch auch vom Eigentümer selbst ausgestellt werden. Dies führt jedoch zu einem Authentisierungsproblem, weil es für den Eingentümer unmöglich ist, mit einem solchen Zertifikat seine Identität nachzuweisen. Aus diesem Grund lässt man das Zertifikat von einer dritten Organisation ausstellen, die für beide Seiten, also für den Eigentümer selbst sowie für das Subjekt, das den Identitätsnachweis benötigt, als vertrauenswürdig gilt.

Zur persistenten Speicherung von Schlüsseln und Zertifikaten stellt Java *Keystores* zur Verfügung. Keystore ist eine passwortgeschützte Datei, die zwischen zwei Arten von *Entries* unterscheidet.

- Eine *Key Entry* dient zur Speicherung von geheimen Schlüsseln im geschütztem Format, dementsprechend ist jede Key Entry durch ein eigenes Passwort geschützt.
- In einer *Trusted Certificate Entry* wird ein Zertifikat gespeichert, das den korrespondierenden öffentlichen Schüssel beinhaltet.

Zugriffe auf Entries beider Arten erfolgen über die Angabe von *Aliases*, eindeutigen Namen, die die Entries in der Keystore identifizieren. Aliases sind nur innerhalb der Keystore bekannt, die Inhalte von Entries (Schlüssel und Zertifikate) haben keine Angaben darüber, unter welchem Alias sie in der Keystore abgelegt wurden. Ein Verwaltungstool für Keystores wird in Kapitel 3.4 vorgestellt.

3.3 Digitale Signaturen und Code-Signierung

Digitale Signierung (Unterzeichnung) ist ein weiteres wichtiges Sicherheitsaspekt, der in Java realisiert wurde. Signierung basiert auf der Erzeugung eines *digitalen Fingerabdruckes (Message Digest)* und dient zur Authentifizierung von Daten. Wie bereits in Kapitel 3.2 erwähnt wurde, wird beim Signieren der geheime Schlüssel des Individuums (genannt *Signer*) verwendet. Erfolgreiche Authentifizierung eines Datenblocks gewährleistet zwei folgende Aspekte:

- Der Datenblock wurde nicht verändert.
- Der Datenblock stammt tatsächlich vom Individuum, dessen Identität angegeben wurde.

Message Digests sind Ergebnisse von Einweg-Hash-Funktionen, die den Inhalt eines beliebigen Datenblockes auf eine Byte- bzw. Bitfolge fester Länge abbilden. Einen großen Bekanntheitsgrad unter kryptographischen Hash-Funktionen erlangte der *Secure Hash Algorithm (SHA)*, der inzwischen das Fundament der kryptographischen Sicherheitsarchitektur bildet. SHA existiert in mehreren Varianten. Besonders verbreitet ist *SHA1*, ein Algorithmus, der den Inhalt eines Datenblockes in eine Folge von 160 Bit umwandelt und in DSA (s. Kapitel 3.2) bei der Signierung von Daten zum Einsatz kommt. Die Zukunft von SHA1 ist allerdings in Frage gestellt, da in den letzten zwei Jahren einige Schwachstellen bei dieser SHA-Variante entdeckt worden sind [WYY].

Die DSA-Technologie beinhaltet drei Algorithmen:

- Algorithmus zur Generierung von Schlüsselpaaren
- Algorithmus zur Signierung von Daten
- Algorithmus zur Verifizierung der Signatur

Die Anwendung von DSA ist auf der Abbildung 3.3 veranschaulicht [HC04]. Nach der Generierung eines Schlüsselpaares dienen die Originaldaten und der geheime Schlüssel als Input für den Signieralgorithmus (SHA1). Die erstellte Signatur wird an den Datenblock angehängt, wodurch ein *signierter Datenblock* entsteht. Bei der Verifizierung des signierten Datenblockes wird der öffentliche Schlüssel verwendet. Ist die Verifizierung erfolgreich, so wurde der Datenblock mit dem geheimen Schlüssel signiert, dessen öffentlicher Teil als Eingabe des Verifizierungsalgorithmus' diente.

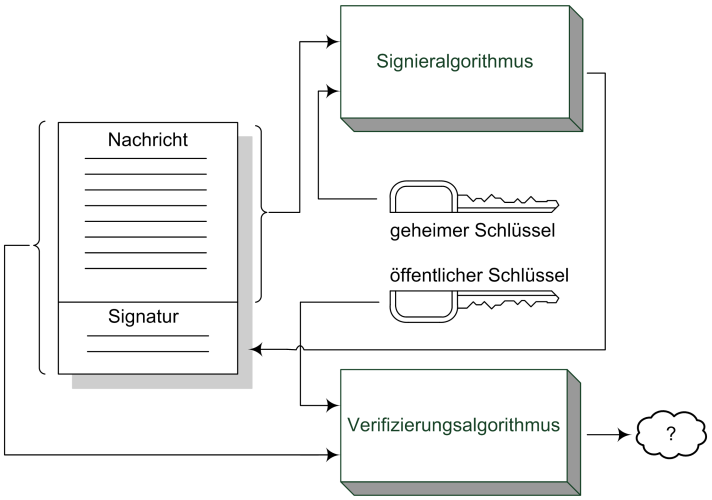

Abbildung 3.3: Signierung und Verifizierung einer Nachricht nach DSA

Signierung von Daten kommt nicht nur bei Nachrichtenübertragung zum Einsatz. Als eine weitere Authentifizierungstechnologie bietet Java Code-Signierung an. Einerseits ermöglicht die Code-Signierung eine flexiblere Verwaltung von Zugriffsrechten, denn diese können je nach Eigentümer des auszuführenden Codes gewährt werden. Andererseits können Manipulationen am Programmcode durch Dritte erkannt werden, was dem System einen Schutz gegen mögliche Schäden bietet, die durch den manipulierten Code verursacht werden können.

Unterzeichnung des Programmcodes auf der Ebene der einzelnen Klassen ist in Java nicht möglich. Damit der Code signiert werden kann, müssen Klassen in einem *Java Archive (JAR)* gekapselt werden [JAR]. JAR ist eine auf dem ZIP-Format basierte Datei, die mehrere Dateien ansammelt und eine *Manifest-Datei* mit Metadaten enthält. Die Manifest-Datei befindet sich immer im Metadaten-Verzeichnis mit dem Namen `META-INF`. Wird ein JAR signiert, so werden im Metadaten-Verzeichnis zwei folgende Dateien erzeugt:

- eine Signatur-Datei mit der Erweiterung `.SF`, die eine Liste mit den Namen der im JAR befindlichen Dateien, der Bezeichnung des verwendeten Digest-Algorithmus' (SHA) sowie die entsprechenden Digests enthält.
- eine Signaturblock-Datei mit dem Zertifikat des Signers und der Signatur der signierten `.SF`-Datei.

Die Verifizierung der signierten JAR-Datei ist erfolgreich, wenn die in der Signaturblock-Datei befindliche Signatur gültig ist und keine der im Archiv

enthaltenen Dateien seit der Erzeugung von Signaturen verändert wurde. Die Verifizierung geschieht in drei Schritten. Zunächst wird die Signatur der `.SF`-Datei verifiziert. Damit wird sichergestellt, dass die in der Signaturblock-Datei angegebene Signatur zu dem geheimen Schlüssel passt, dessen öffentlicher Schlüssel sich in der Signaturblock-Datei befindet, und dass die `.SF`-Datei nicht manipuliert wurde. Im zweiten Schritt wird der im Header der `.SF`-Datei angegebene Digest der Manifest-Datei überprüft, um sicherzustellen, dass die Manifest-Datei nicht manipuliert wurde. Beim dritten und letzten Schritt wird jede in der `.SF`-Datei angegebene Datei aus dem JAR gelesen, und ein Digest für die gelesene Datei erzeugt, der anschließend mit dem in der Manifest-Datei eingetragenen Digest für diese Datei auf Gleichheit geprüft wird. Die Signierung einer JAR-Datei besteht also in der Signierung ihrer einzelnen Entries.

3.4 Tools zur Signierung

Zum Signieren des Bytecodes stellt *Java SDK (Java Standard Development Kit)* zwei Tools zur Verfügung, die jeweils eine Phase der in Kapitel 3.3 am Beispiel von DSA vorgestellten Signierungstechnologie unterstützen. Die Generierung eines Schlüsselpaares erfolgt mit Hilfe des Werkzeuges `keytool`. Keytool ist ein Dienstprogramm zur Verwaltung von Schlüsseln und Zertifikaten sowie zum Administrieren von Keystores, wobei die Erzeugung von Schlüsseln sowohl für symmetrische (DES) als auch für asymmetrische (DSA) Algorithmen unterstützt wird. Die Bedienung des Tools erfolgt über die Kommandozeile. Im Folgenden werden wichtigste Schritte zum Erzeugen einer Keystore und Generieren eines Schlüsselpaares vorgestellt.

Im Vergleich zu Java 5 wurde die Bedienungsschnittstelle von `keytool` in Java 6 etwas verändert. Desweiteren wird die Schnittstelle von Java 5 mit der Angabe der Änderungen in Java 6 beschrieben.

Ein Schlüsselpar lässt sich unter der Verwendung der Option `-genkey` (in Java 6 entspricht dies der Option `-genkeypair`) erzeugen. Als zusätzliche Optionen werden der Alias (`-alias`), für den das Schlüsselpaar erzeugt werden soll, der Pfad zu der Keystore (`-keystore`) sowie der Algorithmus, der zur Generierung des Schlüsselpaares verwendet werden soll, angegeben. Als Algorithmen zum Erzeugen von einem Schlüsselpaar stehen DSA (benutzt SHA1 zur Erstellung von Signaturen) und RSA (erstellt Signaturen unter der Verwendung von MD5) zur Verfügung. Wird der Algorithmus nicht explizit angegeben, so wird DSA verwendet. Die entsprechende Kommandozeile hat folgendes Aussehen:

```
keytool -genkey -alias <alias> -keyalg DSA -keystore <keystore>
```

In darauf folgenden Schritten fordert das Keytool den Benutzer auf, die einzelnen Attribute des Destinguished Name (Kap. 3.2) und das Passwort für die zu erzeugende Key Entry einzugeben. Diese können direkt in der Kommandozeile durch Optionen `-dname` bzw. `-keypass` angegeben werden. Existiert die Keystore unter dem angegebenen Pfad noch nicht, so wird sie automatisch von Keytool erzeugt.

Damit das Zertifikat mit dem öffentlichen Teil des generierten Schlüsselpaares an eine andere Person (bzw. Institution) übertragen werden kann, muss es aus der Keystore exportiert werden. Dies geschieht mit Hilfe der Keytool-Option `-export` (`-exportcert` in Java 6). Ähnlich wie bei der Erzeugung des Schlüsselpaares werden auch beim Export des Zertifikates der Alias und der Keystore-Pfad angegeben. Zusätzlich wird unter der Option `-file` der Name der Datei erwartet, in der das Zertifikat gespeichert werden soll:

```
keytool -export -alias <alias> -keystore <keystore> -file <file>
```

Nach Erhalt des Zertifikates muss es zu weiterer Verwaltung in eine Keystore importiert werden. Analog zur Export-Funktion wird auch das Importieren (Option `-import` bzw. `-importcert` in Java 6) des Zertifikates durchgeführt:

```
keytool -import -trustcacerts -alias <alias> -keystore <keystore>
                              -file <file>
```

Die Option `-trustcacerts` weist darauf hin, dass das zu importierende Zertifikat als *Trusted Certificate* betrachtet werden soll, dem der Benutzer der Keystore vertraut. Der Keystore-Inhalt kann jederzeit unter Verwendung der Option `-list` aufgelistet werden:

```
keytool -keystore akeystore -list
```

Zum Signieren und Verifizieren von JAR-Dateien stellt JDK das Tool `jarsigner` zur Verfügung. `jarsigner` signiert Java-Archive unter Verwendung von geheimen Schlüsseln, die sich in der beim Signieren angegebenen Keystore befinden. Wie auch beim oben vorgestellten Keytool erfolgt die Bedienung des Jarsigners über die Kommandozeile. Das Signieren erfordert die Angabe der Keystore, die den geheimen Schlüssel des Signers beinhaltet (Option `-keystore`) und die Angabe des Signer-Aliases. Möchte man das Originalarchiv nicht überschreiben, so kann mit der Option `-signedjar` der Name des signierten Archives angegeben werden. Eine Kommandozeile zum Signieren einer JAR-Datei mit dem geheimen Schlüssel, der innerhalb der angegebenen Keystore beispielsweise unter dem Alias *abcdef* bekannt ist, hat folgendes Format:

```
jarsigner -keystore <keystore> -signedjar <signedJar>
                <jarToBeSigned> abcdef
```

Im Metadaten-Verzeichnis der signierten JAR-Datei erzeugt der oben angegebene Jarsigner-Aufruf die Signatur-Datei mit dem Namen `ABCDEF.SF` sowie die Signaturblock-Datei namens `ABCDEF.DSA`.

Bei Verifizierung einer JAR-Datei wird die Argument-Liste beim Aufruf von Jarsigner mit der Option `-verify` eingeleitet:

```
jarsigner -verify -verbose -keystore <keystore> <signedJar>
```

Die Option `-verbose` bewirkt eine detaillierte Ausgabe von Verifizierungsergebnissen. Bei der Verifizierung werden die Digests der einzelnen Dateien anhand des in der Signaturblock-Datei befindlichen Zertifikates überprüft. Diese Prüfung kann erweitert werden, indem man eine Keystore angibt (Option `-keystore`), um festzustellen, ob das korrespondierende Zertifikat in dieser Keystore existiert.

Kapitel 4

Szenarien und technische Anforderungen

Beim Vorschlag von *confirmed join points* abstrahiert Harold Ossher in [Oss06] von einer konkreten Programmiersprache. Um zu zeigen, wie sein Konzept in verschiedenen Sprachen realisiert werden kann, wird die AspektJ-Syntax verwendet. Die Idee basiert auf *Pointcuts*, die als Schnittstelle zwischen den Eigentümern der Basisanwendung und den Aspekteigentümern verstanden werden. Diese definieren Mengen von Joinpoints entsprechend den Zugriffsrechten, die an bestimmte Aspektklassen vergeben werden sollen. Diese Technik erfordert allerdings eine syntaktische Erweiterung der Programmiersprache, um Verweise auf die Pointcuts aus der Basisanwendung heraus herzustellen. Bei Ossher sind es Konstrukte *confirms* und *denies*. Die syntaktische Erweiterung der Sprache zieht jedoch eine umfassende Compileränderung nach sich.

Im Rahmen dieser Arbeit sollen dagegen keine syntaktischen Erweiterungen in ObjectTeams zur Realisierung der Kapselung und Freigabe von Joinpoints vorgenommen werden, da ein solcher Ansatz einen erheblichen Eingriff in den ObjectTeams-Compiler erfordert. Der in dieser Arbeit vorgestellte Ansatz basiert auf Java-Sicherheitsmechanismen und reduziert den Eingriff in den Compiler auf das Minimum. Die Änderungen betreffen fast ausschließlich die ObjectTeams-Laufzeitumgebung. Folglich werden technische Anforderungen an die zu entwickelnde Architektur aufgestellt. Nach einer kurzen Beschreibung des Konzeptes werden das Angreifermodell und Angriffsszenarien vorgestellt, gefolgt von Ansatzverfeinerung und der Angabe der Anwendungsszenarien.

4.1 Beschreibung

Das durch die Entwicklung der Architektur verfolgte Ziel ist es, dem Anwender[1] einer Basisanwendung die Gewissheit zu geben, dass Aspekte, die diese Basisanwendung eventuell erweitern, vom Eigentümer dieser Basisanwendung zugelassen sind. Es muss darauf ausdrücklich hingewiesen werden, dass das Konzept nicht die Basisanwendung vor Modifizierungen schützt. Der Anwender soll lediglich die Möglichkeit zur Überprüfung haben, ob das Programm, das er ausführt, von dessen Eigentümer stammt und dass der Programmlauf im Falle eines Fehlers nicht durch unerlaubte Aspekte beeinflusst wurde.

Die Deklaration und Beschreibung der zur Adaptierung freigegebenen Joinpoints erfolgt in einer separaten Policy-Datei, die im Folgenden als *Joinpoint Policy* bezeichnet wird. Im Unterschied zu den aus der Sprache *Java* bekannten Sicherheitspolicies ist Joinpoint Policy ein XML-Dokument, das vom Eigentümer der Basisanwendung erstellt und nur von ihm selbst verändert wird. Eine Veränderung der Policy durch eine andere Person muss erkannt, jedoch nicht verhindert werden. Die Joinpoint Policy enthält Angaben über Joinpoints sowie Aspektcode-Eigentümer, indem den letzteren die vom Basiscode-Eigentümer vergebenen Zugriffsrechte zugeordnet werden. Somit vervollständigt die Joinpoint Policy die in der Classfile-Form bereitgestellte Basisanwendung.

Die Authentisierung der Eigentümer geschieht über digitale Signierung der jeweiligen Produkte (Aspekt- und Basiscode) und Austauschen der Zertifikate. Um dies zu ermöglichen, müssen der Aspekt und die Basisanwendung jeweils in einem *Java Archive* (JAR) gekapselt werden. Die Transformer der ObjectTeams-Laufzeitumgebung müssen beim Weben wissen, für welchen Aspekt sie gerade aktiv sind. Dies wird durch Signierung des Aspektcodes erreicht. Das Ziel ist es, den Aspekteigentümer zu identifizieren, um ihm entsprechende Rechte beim Zugriff auf Joinpoints zu gewähren. Zur Verwaltung von Zertifikaten der Aspekteigentümer enthält das signierte Archiv mit der Basisanwendung und der Joinpoint Policy zusätzlich eine Keystore. Dieses JAR wird im Folgenden als *Basispaket* bezeichnet. Die Identifizierung des Aspekteigentümers erfordert die Verwendung eines asymmetrischen Schlüssels. Mit dem geheimen Teil des Schlüssels wird das Aspekt-Archiv signiert, das Zertifikat mit dem öffentlichen Schlüssel wird in der Keystore des Basispaketes abgelegt. Der Basiscode-Eigentümer benutzt den geheimen Teil seines Schlüssels zur Signierung des Basispaketes und veröffentlicht das Zertifikat mit dem öffentlichen Schlüssel. Diesen öffentlichen Schlüssel übergibt der Anwender an die ObjectTeams-Laufzeitumgebung beim Starten der Basisanwendung, um sicher stellen

[1] Unter dem Begriff *Anwender* ist hier und weiter der Endbenutzer gemeint.

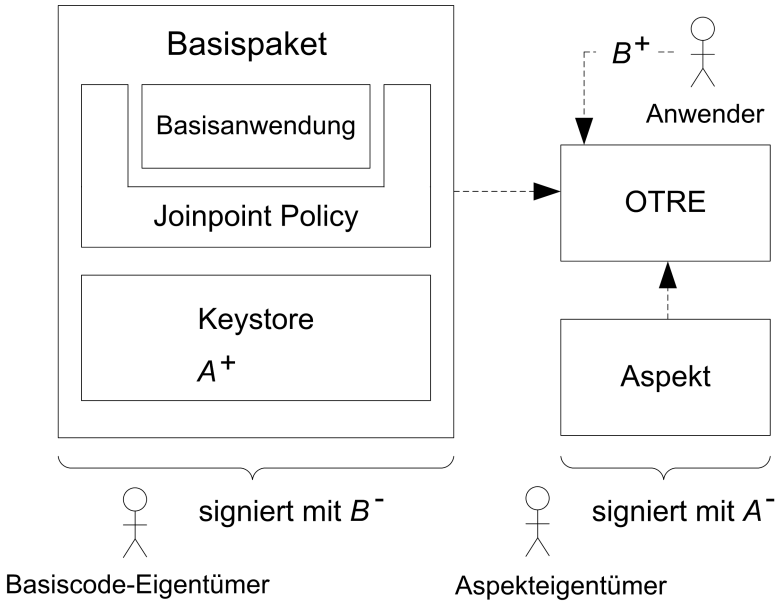

Abbildung 4.1: Architektur

zu können, dass das Basispaket nicht durch Dritte modifiziert wurde. Eine schematische Darstellung der Architektur ist auf der Abbildung 4.1 gegeben.

Die Auswertung der deklarierten Zugriffsrechte sowie die Überprüfung von Zugriffen findet in der ObjectTeams-Laufzeitumgebung zur Webezeit statt. Dazu werden Inhalte von Bytecode-Attributen über Callins und Callouts mit den entsprechenden Einträgen der Joinpoint Policy verglichen. Dies geschieht mit Hilfe eines *Join Point Access Controllers*, dessen Entwicklung in Kapitel 6 ausführlich beschrieben wird.

4.2 Angreifermodell und Angriffsszenarien

Bei der Entwicklung der Architektur zur policy-basierten Zugriffskotrolle für Joinpoints werden zwei der in [FP97] definierten Schutzziele verfolgt:

- **Integrität:** Manipulationen und Fälschungen der durch den Eigentümer der Basisanwendung vergebenen Zugriffsrechte sollen für den Anwender erkennbar sein.

- **Verfügbarkeit:** Aspekte, für die die Zugriffsrechte ausgehandelt wurden, sollen jederzeit von diesen Rechten Gebrauch machen können.

Zur Realisierung der an die Architektur gestellten Schutzziele ist es notwendig, das Bild eines potenziellen Angreifers anzugeben und Annahmen über seine Fähigkeiten zu treffen. Mit dem Begriff *Angreifer* wird hier eine Person bezeichnet, die versucht, sich einen Zugriff auf Joinpoints der Basisanwendung ohne Einwilligung deren Eigentümers zu verschaffen, sodass der Zugriff für den Anwender unerkennbar bleibt.

Es handelt sich dabei in erster Linie um einen Aspektentwickler, dem die Zugriffsrechte verweigert wurden oder er selbst auf deren Aushandlung verzichtet hat. Jeder Aspekteigentümer besitzt bei der Entwicklung seines Aspektes uneingeschränkte Schreib- und Leserechte auf das Basispaket, das vom Entwickler der Basisanwendung zuvor veröffentlicht wurde. Der Angreifer kann die Anwendung ausführen, die Schutzziele beziehen sich jedoch auf das Ausführen der erweiterten Basisanwendung auf dem Rechner des Anwenders. Es wird angenommen, dass der Angreifer die ObjectTeams-Laufzeitumgebung des Anwenders nicht verändern kann. Denn in diesem Fall könnte er aktiv in den Webeprozess eingreifen, was nicht nur die Sicherheit der Joinpoints, sondern auch die Korrektheit des Aspektwebens in Frage stellen würde.

Geht man davon aus, dass die OTRE des Anwenders sicher ist, so bezieht sich die Verschaffung von Zugriffsrechten vor allem auf die Manipulation des Basispaketes. Betroffen sind dabei die Keystore und die Joinpoint Policy. Der Angriff kann allerdings auch von einem autorisierten Aspektentwickler durchgeführt werden, indem er auf Joinpoints-Zugriffe tätigt, die die Joinpoint Policy für ihn nicht vorsieht. Es sind folgende Angriffsszenarien denkbar:

- **Modifizieren der Joinpoint Policy:** Der Angreifer kann die Policy-Datei erweitern, wenn sein Zertifikat mit dem öffentlichen Schlüssel sich bereits in der Keystore des Basispaketes befindet. Dadurch erlangt der Angreifer die fehlenden Zugriffsrechte. Durch Änderung der Policy könnte der Angreifer auch das Ziel verfolgen, bestimmten Aspekten Zugriffsrechte zu entziehen.
- **Modifizieren der Keystore:** Der Inhalt der Keystore wird mit dem Schlüssel des Angreifers erweitert. Diese Modifizierung der Keystore ist allerdings aus der Sicht des Angreifers nur in Kombination mit der Policy-Manipulationen sinnvoll.
- **Identitätstäuschung:** Der Angreifer gibt sich für den Eigentümer der Basisanwendung aus, indem er die Basisanwendung aus dem Originalpaket extrahiert und sein eigenes Basispaket erstellt, das dann die Keystore und die Joinpoint Policy des Angreifers enthält. Damit täuscht der Angreifer vor, Zugriffsrechte vom eigentlichen Eigentümer der Basisanwendung erhalten zu haben.

- **Missachtung der Zugriffsrechte:** Es handelt sich um einen Aspektentwickler, der Zugriffsrechte für eine bestimmte Menge von Joinpoints bereits besitzt, weshalb dessen Schlüssel sich in der Keystore des Basispaketes befindet. Der Aspekt greift jedoch auf Joinpoints zu, die nicht zu der Menge der für ihn freigegebenen Joinpoints gehören.

Darüberhinaus wird angenommen, dass der Angreifer komlexitätstheoretisch beschränkt ist; d. h. er ist nicht in der Lage, in angemessener Zeit den kompletten Schlüsselraum zu durchsuchen, und damit den Schutz der asymmetrischen Verschlüsselungsverfahren zu überwinden.

4.3 Ansatzverfeinerung

Die Aushandlung von Zugriffsrechten zwischen dem Eigentümer der Basisanwendung und einem Aspektentwickler kommt einem Vertragsabschluss zwischen den beiden Seiten gleich. Der Basiscode-Eigentümer erklärt sich dabei bereit, die angefragten Joinpoints zur Adaptierung freizugeben und garantiert deren Verfügbarkeit in den späteren Release-Versionen der Basisanwendung. Der Aspektentwickler verpflichtet sich seinerseits, nur die für ihn bereitgestellten Joinpoints zu benutzen. Zur Authentisierung der beiden Eigentümer und Einhaltung der vereinbarten Nutzung von Joinpoints sind folgende Überprüfungen in der ObjectTeams-Laufzeitumgebung notwendig:

1. *Verifizierung des Basispaketes:* Es wird überprüft, ob das Basispaket mit dem geheimen Teil B^- des vom Anwender an die OTRE übergebenen öffentlichen Teils der Schlüssels B^+ signiert wurde. Es ist anzumerken, dass dies allein nicht die Identität des Basiscode-Eigentümers überprüft. Damit kann man lediglich feststellen, dass das Basispaket nicht nach dessen Signierung modifiziert und von der selben Person signiert wurde, der der öffentliche Teil B^+ gehört. Für die Zugehörigkeit von B^+ zum eigentlichen Basiscode-Eigentümer ist dabei allein der Anwender verantwortlich, der die Anwendung ausführt. Mit anderen Worten, der Anwender muss selbst darauf achten, dass er den öffentlichen Schlüssel des tatsächlichen Basiscode-Eigentümers bezogen hatte. Die Identität des Schlüsselbesitzers kann nachgewiesen werden, wenn das Zertifikat von einer vertrauenswürdigen Organisation ausgestellt wurde.
2. *Verifizierung des Aspektes:* Es handelt sich um die Überprüfung der Identität des Aspektentwicklers. Es wird überprüft, ob der Aspekt mit dem geheimen Teil des Schlüssels A^- signiert wurde, dessen öffentlicher Teil A^+ sich in der Keystore des Basispaketes befindet. War

die Verifizierung des Basispaketes (1. Überprüfung) erfolgreich, so weist die erfolgreiche Verifizierung des Aspektes darauf hin, dass der Aspekt tatsächlich von einem Entwickler stammt, für den die Zugriffsrechte für Joinpoints ausgehandelt wurden. Anders ausgedrückt: Wenn die Verifizierung des Basispaketes fehlschlägt, kann man aus der erfolgreichen Verifizierung des Aspektes nur auf die Gleichheit der Identitäten des Aspekt-Signierers und der Person schließen, deren öffentliche Schlüssel sich in der Keystore des Basispaketes befindet.

3. *Überprüfung der Zugriffsrechte:* Hier werden Zugriffsrechte des Aspekteigentümers überprüft. Dabei werden Inhalte von zusätzlichen Bytecode-Attributen, die Informationen über Callins, Callouts und zu adaptierenden Klassen tragen, mit dem Inhalt der Joinpoint-Policy verglichen. Je nachdem, ob die Verifizierung des Aspektes erfolgreich war, gibt es zwei Klassen von unerlaubten Zugriffen auf Joinpoints. Zu der ersten Klasse gehören Zugriffe von Aspekten, deren Verifizierung nicht erfolgreich war, d. h. wenn die Keystore keinen Freigabe-Schlüssel enthält. Somit sind alle Zugriffe dieses Aspektes als unerlaubt zu betrachten. Zu der zweiten Gruppe gehören Zugriffe von Aspekten, die erfolgreich verifiziert wurden, die entsprechenden Joinpoints befinden sich jedoch laut der Joinpoint-Policy nicht in der Menge der für diesen Aspekt zugelassenen Joinpoints (s. auch Kap. 4.3.2).

In den nachfolgenden Abschnitten werden weitere Gesichtspunkte des Ansatzes diskutiert und verfeinert.

4.3.1 Benutzerkategorien

Die Zugriffsrechte werden vom Eigentümer der Basisanwendung vergeben. Er bestimmt, auf welche Joinpoints überhaupt zugegriffen werden darf und von welchen Eigentümern der Aspektcode dabei stammen muss. Bezeichnet man den Aspektcode-Eigentümer in diesem Kontext als Benutzer, so können Benutzergruppen zur Authentifizierung von Aspekteigentümern ähnlich wie für die Dateirechte in UNIX-Systemen definiert werden:

- **Einzelne Benutzer:** Zugriffe auf einen Joinpoint sind für **einen** konkreten Benutzer erlaubt.
- **Alle Benutzer:** Zugriffe auf einen Joinpoint sind ohne Einschränkung für alle Benutzer erlaubt. Es handelt sich also um öffentliche Joinpoints.

- **Gruppe:** Zugriffe auf einen Joinpoint sind für Benutzer erlaubt, die der angegebenen Gruppe zugehören. Die Entscheidung darüber, welche konkreten Benutzer die Gruppe bilden, wird ausschließlich vom Eigentümer der Basisanwendung getroffen. Gruppenverwaltung soll den Aspekteigentümern verborgen bleiben. Es muss jedoch darauf geachtet werden, dass die Verwaltung von Gruppen die Interessen beider Vertragsparteien nicht außer Acht gelassen werden dürfen. Durch die Entnahme eines Entwicklers aus einer Gruppe sollen die anderen Eigentümer der Gruppe keine Zugriffsrechte verlieren, aber auch keine gewinnen.

4.3.2 Zugriffskategorien von Joinpoints

Aus der Sicht eines Aspektes kann jeder Joinpoint in genau eine der folgenden Zugriffskategorien eingeteilt werden:

- **Denied Joinpoints:** Zu dieser Kategorie gehören Joinpoints, die zu keinem Zeitpunkt für keinen Aspekt zur Freigabe bereit stehen. Jeder Zugriff auf einen *denied* Joinpoint wird als unerlaubt gewertet.
- **Available Joinpoints:** Diese Kategorie bilden Joinpoints, die für den Aspekt zwar nicht zugelassen sind, können aber nach Anfrage des Aspektentwicklers vom Eigentümer der Basisanwendung freigegeben werden.
- **Confirmed Joinpoints:** Das sind Joinpoints, die dem Aspekt nach erfolgten Aushandlung von Zugriffsrechten zur Verfügung stehen. Hat ein Aspekt keine Zugriffsrechte für einen Joinpoint, so können diese ausgehandelt werden, sofern der Joinpoint zur Kategorie *Available Joinpoints* gehört.
- **Free Joinpoints:** Zu dieser Kategorie gehören alle Joinpoints, die zu jedem Zeitpunkt für alle Aspekte freigegeben sind.

Mit der Aufnahme eines Joinpoints in eine der beiden letzteren Kategorien erklärt sich der Basiscode-Eigentümer bereit, diesen Joinpoint auch in den nachfolgenden Release-Versionen anzubieten. Für *Free Joinpoints* muss die Verfügbarkeit **immer** garantiert werden. Gehört ein Joinpoint für einen Aspekt zu der Kategorie *Confirmed Joinpoints*, so sagen wir, dass der Aspekt *exklusive* Zugriffsrechte auf diesen Joinpoint hat. Während die Mengen von *Denied* und *Free Joinpoints* für alle Aspekte gleich sind, werden Available- und Confirmed-Mengen durch die Aushandlung von exklusiven Zugriffsrechten definiert und sind somit aus der Sicht jedes einzelnen Aspektes unterschiedlich.

Falls ein Joinpoint zur Kategorie *Available Joinpoints* gehört, kann er in der neuen Release-Version nur dann entfernt werden, falls er sich für keinen Aspekt unter den *Confirmed Joinpoints* befindet. *Denied Joinpoints* können vom Eigentümer der Basisanwendung zu jedem Zeitpunkt entfernt werden, da man im Allgemeinen davon ausgeht, dass kein Aspekt Joinpoints dieser Kategorie zur Adaptierung verwendet.

An dieser Stelle muss gesagt werden, dass die Verfügbarkeitsgarantie eines Joinpoints im allgemeinen Sinne sehr schwer zu definieren ist. In dieser Arbeit wurde der Begriff *Joinpoint* auf Bezugspunkte für Callins und Callouts konkretisiert. In ObjectTeams sind es Klassen, Methoden und Attribute. Die Verfügbarkeit von diesen Bezugspunkten ist wohldefiniert.

4.3.3 Entwicklungsversion und Release

Oft kann die Menge der benötigten Joinpoints zum Weben eines Aspektes erst während dessen Entwicklung bestimmt werden. Es ist nahezu unmöglich, die endgültige Menge in der Planungsphase anzugeben, dies würde die Entwicklungskosten unnötig in die Höhe treiben. Stellt der Aspektentwickler fest, dass der benötigte Joinpoint nicht zu der Menge von für seinen Aspekt freigegebenen Joinpoints gehört (*confirmed* oder *free*), so wird der Entwicklungsprozess unterbrochen. Die hohen Kosten ergeben sich in diesem Fall aus den Geschäftsprozessen, die stattfinden müssen, um die Freigabe von benötigten Joinpoints beim Eigentümer der Basisanwendung zu erhalten. Erst muss die Anforderung vom Aspekteigentümer gestellt, dann vom Basiscode-Eigentümer bearbeitet und das aktualisierte Anwendungspaket wieder an den Aspekteigentümer übergeben werden. Solche Geschäftsaktionen sind sehr zeitaufwendig. Zur Reduzierung dieses Aufwands ist es sinnvoll, bei der Realisierung der Zugriffskontrolle zwischen einer *Entwicklungsversion* und einer *Release-Version* des Aspektes zu unterscheiden. In der Entwicklungsphase sollen dem Aspektentwickler Zugriffe auf eine viel größere Menge von Joinpoints ermöglicht werden, als für den Aspekt tatsächlich notwendig ist. Dadurch ist der Aspekteigentümer viel flexibler in seinen Entscheidungen und kann ohne großen Zeitverlust erst bei Fertigstellung der Release-Version die Freigabe von benötigten Joinpoints beim Eigentümer der Basisanwendung beantragen. Aus der vorgestellten Differenzierung von Aspektversionsarten wird deutlich, dass jede neue Release-Version eines Aspektes eine neue Release-Version des Basispaketes notwendig macht.

4.3.4 Schlüsselarten

Die Aufteilung von Aspektversionsarten in Entwicklungs- und Release-Versionen impliziert die Differenzierung von Schlüsselarten, die den Aspekteigentümer identifizieren sollen. Demnach gibt es zwei folgende Schlüsselarten, mit denen ein Aspekt signiert werden kann:

- **Developing Key** ist ein Schlüssel für eine Entwicklungsversion eines Aspekts und gibt eine endliche Menge von Joinpoints frei. Diese Menge definiert den Flexibilitätsgrad des Aspektentwicklers und soll so groß wie möglich gehalten werden. Es ist leicht zu erkennen, dass es sich bei dieser Menge (wie oben definiert) um *Available Joinpoints* handelt.

- **Release Key** ist ein Schlüssel für eine Release-Version eines Aspektes und definiert eine endliche Menge von Joinpoints, die für einen konkreten Aspektentwickler zur Adaptierung durch Aspekte freigegeben werden. Die Kardinalität dieser Menge bestimmt den Kontrollierbarkeitsgrad der Basisanwendung durch deren Eigentümer. Je kleiner diese Menge ist, desto mehr Kontolle und somit auch mehr Flexibilität hat der Eigentümer bei Weiterentwicklung der Basisanwendung. Das liegt daran, dass der Basiscode-Eigentümer nach der Aufnahme eines Joinpoints in die Kategorie *Confirmed Joinpoints* für mindestens einen Aspekt die Verfügbarkeit dieses Joinpoints auch in den späteren Release-Versionen der Basisanwendung garantieren muss. Der Release-Schlüssel soll dementsprechend nur die Joinpoints freigeben, die zum Weben des Aspektes tatsächlich benötigt werden.

Bei den Entwicklungsschlüsseln könnte eine Gültigkeitsbeschränkung sinnvoll sein. Dadurch behält der Eigentümer der Basisanwendung den Überblick über alle Aspekte, die sich in der Entwicklungsphase befinden. Mit der Übergabe eines Entwicklungsschlüssels überlässt der Basiscode- Eigentümer dem Aspektentwickler eine bestimmte Zeitspanne zur Entwicklung eines Aspektes. Reicht diese Zeit nicht aus, so muss der Aspektentwickler einen neuen Entwicklungsschlüssel beantragen. Da das Ablaufdatum der Gültigkeit für beide Seiten bekannt ist, kann die Übergabe eines neuen Entwicklungsschlüssels ohne Zeitverlust für den Entwicklungsprozess des Aspektes stattfinden.

4.3.5 Policies

Wie auch die Deklaration von Zugriffsrechten wird die Anfrage des Aspektentwicklers zur Freigabe von Joinpoints in Form einer Policy-Datei formuliert. Während die Joinpoint Policy aus dem Basispaket alle Joinpoints enthält, die die Basisanwendung zum Weben anbietet, reicht es aus, bei Freigabeanfragen eine viel kleinere Menge von Joinpoints anzugeben. In der Anfrage-Datei (*Request Policy*) kann der Aspekteigentümer nicht nur Zugriffsrechte anfordern, sondern auch ablegen, falls ihm zuvor welche durch den Eigentümer der Basisanwendung vergeben wurden, und der Aspektentwickler sich auf diese Zugriffsrechte in Zukunft verzichten kann. Der Verzicht auf erteilte Zugriffsrechte verleiht dem Entwickler der Basisanwendung mehr Flexibilität bei deren Weiterentwicklung, denn dadurch wandern die Joinpoints aus der Sicht des Aspektes wieder in die Kategorie von *Available Joinpoints*. Ist ein Joinpoint für keinen der bekannten Aspekte als *confirmed* deklariert, so kann er vom Basiscode-Eigentümer in der nächsten Release-Version entfernt oder in die Kategorie von *Denied Joinpoints* eingeteilt werden. Neben der Deklaration von Zugriffsrechten entält die Request Policy Angaben über den öffentlichen Schlüssel des Aspektentwicklers.

4.3.6 Ausführungsmodi

Unterscheidung von Joinpoint-Kategorien und Schlüsselarten ermöglicht Ausführung einer durch Aspekte erweiterten Anwendung in unterschiedlichen Modi. Die Entscheidung, die Anwendung in einem bestimmten Modus auszuführen, kann sowohl vom Aspektentwickler als auch vom Anwender des Programms getroffen werden. Jedoch stehen den beiden Stakeholdern jeweils nicht alle Ausführungsmodi zur Verfügung. Das liegt daran, dass einige Ausführungsmodi für die Entwicklungsphase und die anderen für die Release-Version des Aspektes bestimmt sind.

Man unterscheidet zwischen folgenden Ausführungsmodi:

- **Secure Mode:** In diesem Modus erlaubt die OT-Laufzeitumgebung das Weben von Aspekten, die ausschließlich vom Eigentümer der Basisanwendung zugelassene Joinpoints benutzen. Das sind Joinpoints, die entweder für alle Aspekte zur Verfügung stehen (*free*) oder für die die Zugriffsrechte explizit ausgehandelt worden sind (*confirmed*). Der Aspekt muss mit dem entsprechenden Release-Schlüssel signiert werden, um den Aspekteigentümer eindeutig zu identifizieren.

- **Common Mode** ermöglicht das Weben nicht nur an den zugelassenen Joinpoints, sondern auch an denen, die aus der Sicht des Aspektes zu der Kategorie *available* gehören. Es reicht aus, wenn der Aspekt zuvor mit dem Release-Schlüssel signiert wurde, beim Weben werden allerdings auch Joinpoints zugelassen, die der Basiscode-Eigentümer zur Freigabe angeboten hat. Es ist ersichtlich, dass beim fehlerhaften Verhalten der erweiterten Basisanwendung die Verantwortung für dieses Fehlverhalten auf den Anwender übertragen wird. Durch seine Entscheidung, die Anwendung in Common Mode auszuführen, erklärt sich der Anwender bereit, die Einhaltung der zwischen dem Aspektentwickler und dem Eigentümer der Basisanwendung ausgehandelten Zugriffsrechte zu ignorieren.

- **Developing Mode** ist ein Modus, der dem Aspekteigentümer in der Entwicklungsphase des Aspektes zur Verfügung steht. Es handelt sich somit um eine Entwicklungsversion des Aspektes, dementsprechend muss der Aspekt mit dem Entwicklungsschlüssel signiert werden. Beim Weben werden Zugriffe auf alle Joinpoints zugelassen, die vom Eigentümer der Basisanwendung als *free* oder *available* deklariert wurden. Diese beiden Mengen von Joinpoints sind gleich für alle Aspekte, die sich in der Entwicklungsphase befinden und keine exklusiven Zugriffsrechte haben.

- **Request Test Mode:** Dieser Modus ermöglicht einen Testlauf der erweiterten Basisanwendung unter der Verwendung von *Request Policy*. Das Ziel dabei ist es, dem Aspektentwickler den Aufwand bei der Aushandlung von Zugriffsrechten für Joinpoints zu ersparen, indem die Request Policy auf ihre Vollständigkeit überprüft wird. Aufgrund eines erfolgreichen Testlaufs kann der Aspekteigentümer darauf schließen, dass er bei der Erstellung der Request Policy alle Joinpoints angegeben hat, deren Freigabe durch den Eigentümer der Basisanwendung erforderlich ist. Der Grund für die Überprüfung der Request Policy besteht darin, dass die Aushandlung von Zugriffsrechten ein relativ aufwendiger Geschäftsprozess ist. Ist die erstellte Request Policy vollständig, so bleiben weitere Freigabe-Anfragen den beiden Seiten erspart. Die Request Policy wird an die OT-Laufzeitumgebung beim Programmstart übergeben. Verfügt der Aspektentwickler bereits über Zugriffsrechte für einige Joinpoints aus der früheren Release-Versionen des Aspektes, so hat er die Möglichkeit, diese bei Request Test Mode einzubeziehen. Dafür muss der Aspekt mit dem Release-Schlüssel signiert werden. Somit werden bei diesem Modus Zugriffe auf folgende Joinpoints als gültig gewertet: *free*, *confirmed* (Zugriffsrechte aus früheren Release-Versionen des Aspektes) und *available* Joinpoints, die in der Request Policy aufgeführt sind.

- **Unchecked Mode:** In diesem Modus werden alle Zugriffe auf alle Joinpoints als erlaubt gewertet. Somit muss weder die Identität des Basisccode-Eigentümers noch die des Aspekteigentümers geprüft werden. Die Verfügbarkeit der Joinpoint Policy und der Keystore ist ebenfalls nicht vorausgesetzt.

Mit Ausnahme von *Unchecked Mode* sind in allen Modi alle drei zuvor beschriebenen Überprüfungen beim Weben erforderlich: Verifizierung des Basispaketes und des Aspektes sowie Überprüfung der Zugriffsrechte. Die Definition der Ausführungsmodi ist in der Tabelle 4.1 zusammengefasst.

	Verfügbare Joinpoints	**Benötigte Schlüssel**	**Überprüfungen notwendig**
Secure Mode	free confirmed	Release	ja
Common Mode	free confirmed available	Release	ja
Developing Mode	free available	Developing	ja
Request Test Mode	free confirmed available*	Release	ja
Unchecked Mode	alle	–	nein
* Nur in der Request Policy aufgeführte Joinpoints sind verfügbar			

Tabelle 4.1: Ausführungsmodi

4.3.7 Behandlung von Zugriffsrechteverletzungen

Sollte eine Zugriffsrechteverletzung festgestellt werden, bieten sich unterschiedliche Behandlungsmöglichkeiten an:

- Ausgabe einer **Warnung**: Der Programmlauf wird nicht unterbrochen, die Ausgabe einer Warnung dient lediglich dazu, den Anwender (oder Entwickler) über die Zugriffsverletzung zu informieren. Die Ausgabe einer Warnung ist in *Common Mode* und in den beiden Entwicklungsmodi (*Developing* und *Request Test Mode*) geeignet.
- Werfen einer **Exception**: Diese Variante ist unverzichtbar, wenn die erweiterte Basisanwendung als Komponente einer anderen Anwendung eingesetzt wird, deren Entwickler dann die Möglichkeit erhält,

auf die Ausnahmesituation entsprechend zu reagieren. Das Werfen einer Exception ist in allen Ausführungsmodi sinnvoll, in denen die Überprüfung der Zugriffsrechte stattfindet.

- **Beenden** der Anwendung: Ist der Programmweiterlauf nach einer Zugriffsrechteverletzung für den Anwender inakzeptabel, so kann das Programm beendet werden. In Java entspricht dies dem Aufruf von `System.exit()`. Das Beenden des Programms bei einer Verletzung von Zugriffsrechten ist in *Secure* und *Common Mode* sinnvoll.

Die Übersicht über die in unterschiedlichen Ausführungsmodi zur Verfügung stehenden Behandlungsarten wird in der Tabelle 4.2 gegeben.

Ausführungsmodus	Behandlung möglich durch
Secure Mode	Exception, Beenden
Common Mode	Warnung, Exception, Beenden
Developing Mode	Warnung, Exception
Request Test Mode	Warnung, Exception
Unchecked Mode	–

Tabelle 4.2: Zuordnung von Behandlungsarten

4.4 Anwendungsszenarien

Im Lebenszyklus eines Aspektes können zwei Abschnitte hervorgehoben werden. Der erste Abschnitt ist die Entwicklung eines Aspektes. Hinsichtlich der Zugriffskontrolle auf Joinpoints findet in dieser Phase die Aushandlung der Zugriffsrechte statt, an der sich zwei Akteure beteiligen: Aspektentwickler (Eigentümer des Aspektes) und Eigentümer der Basisanwendung. Nachdem die Aspektentwicklung abgeschlossen ist, beginnt die zweite Phase, in der die Basisanwendung und der Aspekt als ein Ganzes ausgeführt werden. In dieser Phase stehen Bezug, Instandsetzen und Ausführen der erweiterten Basisanwendung im Vordergrund. Nachfolgend werden Anwendungsszenarien beschrieben, die in der Abbildung 4.2 in Form eines Sequenzdiagramms gezeigt sind.

4.4.1 Aspektentwicklung

Nach der Entwicklung der Basisanwendung [1] bezieht der Aspektentwickler das signierte Basispaket von dessen Eigentümer und erhält zusätzlich einen Entwicklungsschlüssel, der für eine Freigabe von allen verfügbaren

Joinpoints sorgt und somit einen flexiblen Entwicklungsprozess des Aspektes [2] ermöglicht. Dabei stehen dem Aspektentwickler alle Joinpoints zur Verfügung, die die Joinpoint Policy nicht als *denied* einstuft.

Nachdem Abschluss der Implementierung erstellt der Aspektentwickler eine Request Policy, in der er alle vom Aspekt tatsächlich verwendeten Joinpoints angibt, die aus der Sicht des Aspektes der Kategorie *Available Joinpoints* angehören. Optional kann der Aspekteigentümer Joinpoints angeben, für die er zuvor ausgehandelte Zugriffsrechte ablegen möchte (s. Kap. 4.3.5). Die Request Policy und der öffentliche Schlüssel des Aspektentwicklers wird an den Eigentümer der Basisanwendung übergeben. Dieser fügt den öffentlichen Schlüssel des Aspekteigentümers in die Keystore des Basispaketes ein und aktualisiert die Joinpoint Policy, indem er die in der Request Policy angegebenen Joinpoints für den Aspektentwickler als *confirmed* vermerkt [3]. Hat der Aspektentwickler seine Zugriffsrechte für einen bestimmten Joinpoint abgelegt, so kann der Eigentümer der Basisanwendung diesen Joinpoint in die Kategorie *Denied Joinpoints* einordnen, soweit kein anderer Aspektentwickler exklusive Zugriffsrechte auf diesen Joinpoint besitzt. Die Information dazu kann der Joinpoint Policy entnommen werden. Nach der Aktualisierung der Keystore und der Joinpoint Policy erstellt der Basiscode-Eigentümer ein neues Basispaket und signiert es, wodurch eine neue Release-Version der Basisanwendung entsteht.

Nach der Übergabe der Request Policy und des öffentlichen Schlüssels signiert der Aspekteigentümer den in einem JAR gekapselten Aspekt und veröffentlicht diesen. Damit ist die Entwicklungsphase des Aspektes abgeschlossen.

4.4.2 Ausführung der erweiterten Basisanwendung

Nach Abschluss der Entwicklungsphase müssen Anwendungsszenarien unter Berücksichtigung eines weiteren Stakeholders betrachtet werden. Es handelt sich um den Anwender, der die erweiterte Basisanwendung ausführt. Das kann eine dritte Person sein, die mit der Entwicklungsphase nichts zu tun hat, aber auch einer der beiden Eigentümer: der Aspekteigentümer oder der Eigentümer der Basisanwendung. Der Anwender bezieht das Basispaket, den Aspekt und den öffentlichen Schlüssel des Basiscode-Eigentümers und muss eine Entscheidung treffen, in welchem Modus die Anwendung ausgeführt werden soll [5].

Der Bezug von beiden Produkten (Basisanwendung und Aspekt) steht mit dem Protokoll der Aushandlung von Zugriffsrechten im engen Zusammenhang. Eine Basisanwendung kann von mehreren Aspekten unterschiedlicher Aspekteigentümer erweitert werden, die im Normalfall nichts voneinander wissen.

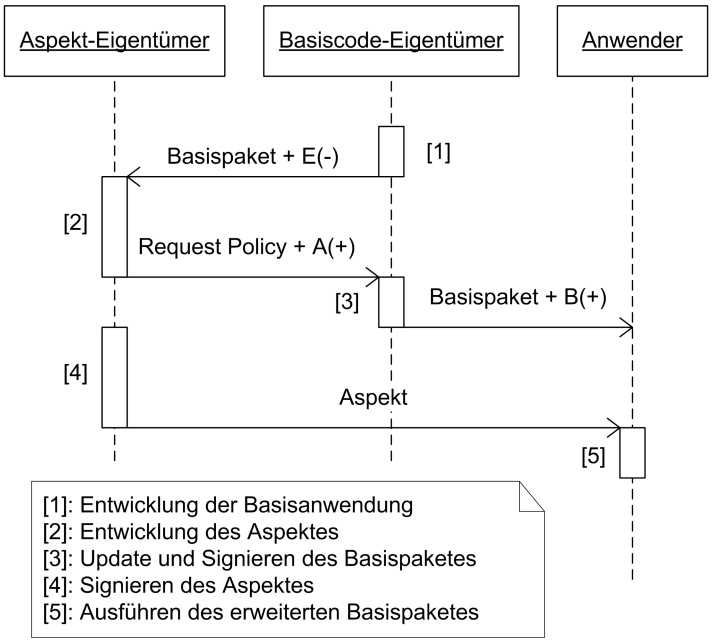

Abbildung 4.2: Anwendungsszenario

ander wissen. Auch wenn keine Weiterentwicklung der Basisanwendung vorgesehen ist, muss deren Eigentümer im Interesse der Anwender der alleinige Anbieter des Basispaketes bleiben. Das liegt daran, dass nur der Eigentümer das Basispaket bei Hinzunahme neuer Aspekte aktualisieren kann. Der Bezug des Basispaketes bei seinem Eigentümer garantiert dem Anwender die Konformität der Joinpoint Policy zu allen bis dato zugelassenen Aspekten.

Mit der Wahl des Ausführungsmodus' gibt der Anwender an, in welchem Maße die Einhaltung der ausgehandelten Zugriffsrechte für ihn wichtig ist. Das Vertrauen des Anwenders bezieht sich dabei nicht auf den Programmcode der beiden Produkte, sondern darauf, ob das Fehlverhalten der Basisanwendung durch den Aspekt verursacht wurde. Wer von den beiden Entwicklern für das Fehlverhalten die Verantwortung trägt, entscheidet sich danach, ob der Aspekt im Rahmen der vereinbarten Regeln auf Joinpoints der Basisanwendung zugegriffen und ob einer dieser Zugriffe zum Fehler geführt hat.

Kapitel 5

Deklaration von Joinpoints

Neben dem Konfigurationsmanagement gehören zu den Anwendungsbereichen von Policies in der IT-Welt hauptsächlich Definition von Zugriffsrechten und Kontrolle von Zugriffen auf diverse Ressourcen. In dieser Hinsicht ist Policy ein Satz von Richtlinien, die spezifizieren, wie sensible Ressourcen benutzt werden sollen. Durch die strenge Einhaltung der definierten Regeln erreicht man einen bestimmten Schutzgrad der Ressource. Eine Policy beschreibt den Gegenstand, für den die Zugriffskontrolle realisiert wird, sowie Objekte (Akteure), die auf den beschriebenen Gegenstand Zugriffe tätigen, und definiert die Zugriffsrelationen.

Zu den Aufgaben einer Joinpoint Policy gehören Deklaration von Joinpoints (Gegenstände, die geschützt werden sollen) und Zuweisung der Zugriffsrechte an einzelne Code-Eigentümer (Akteure, die auf die zu schützenden Joinpoints zugreifen). Somit muss die Policy neben der Deklaration von Joinpoints auch Daten über Code-Eigentümer enthalten, die diese Eigentümer eindeutig identifizieren. Innerhalb der Architektur werden Eigentümer durch ihre öffentlichen Zertifikate dargestellt. Im Prinzip reicht zur Identifizierung des Eigentümers innerhalb der Policy-Datei die Angabe seines Aliases, unter dem das Benutzerzertifikat innerhalb der Basiskeystore bekannt ist, aus, denn Aliases sind innerhalb der Keystore ohnehin eindeutig. Es ist jedoch sinnvoll, alle Daten aus dem Zertifikat in der Poilcy-Datei aufzuführen. Dadurch bleiben Zugriffe auf die Keystore bei der Verwaltung von Policies erspart, die Eigentümer-Daten können direkt aus der Policy entnommen werden.

Als Datenformat zur Deklaration von Joinpoints wurde XML (Extensible Markup Language) gewählt. XML ermöglicht eine strukturierte Darstellung und plattformunabhängige Verarbeitung von Daten. Mit Hilfe von XML können Objekte in Form von Entities genau beschrieben werden. Der ausschlaggebende Punkt bei der Auswahl des Beschreibungsformats war die

Tatsache, dass die Datenverarbeitung innerhalb der in der Programmiersprache Java implementierten ObjectTeams-Laufzeitumgebung geschehen soll. Da Java objektorientiert und plattformunabhängig ist, gestaltet sich die Verarbeitung von Daten in XML-Form sehr flexibel. Außerdem eignet sich XML am besten zur Darstellung von hierarchischen Datenstrukturen. Wie wir später sehen werden, weist die Joinpoint Policy eine solche hierarchische Struktur auf.

Die XML-Spezifikation legt *Wohlgeformtheit* als ein wichtiges Kriterium für XML-Dokumente fest. Ein XML-Dokument ist wohlgeformt, wenn es notwendigen Grundregeln entspricht, die vom W3-Consortium (W3C) spezifiziert wurden [W3XML]. Die XML-Spezifikation beinhaltet einen Satz von Wohlgeformtheitsbeschränkungen, die die Verarbeitung von XML-Dokumenten und die Verwendung von Daten durch unterschiedliche Anwendungsprogramme ermöglichen. Beispiele für Wohlgeformtheitsbeschränkungen sind Existenz von genau einem Wurzelelement in einem XML-Dokument, Übereinstimmung des Namens im End-Tag mit dem Elementtyp im Start-Tag sowie korrekte Verschachtelung von Start- und End-Tags. In der XML-Spezifikation ist mit *Gültigkeit* ein weiteres Kriterium für XML-Dokumente definiert. Ein XML-Dokument ist *gültig*, wenn es wohlgeformt ist, eine Dokumenttyp-Deklaration enthält, die das Format mittels einer Grammatik beschreibt, und die in der Dokumenttyp-Deklaration formulierten Gültigkeitsbeschränkungen einhält. Es handelt sich dabei um Beschränkungen der logischen Struktur des Dokumentes. Das Format kann beispielsweise mittels einer Dokumenttyp-Definition (DTD) oder eines XML-Schemas beschrieben werden. Während die Wohlgeformtheitseinschränkungen strukturelle Anforderungen an das XML-Dokument stellen, legt das Gültigkeitskriterium den inhaltlichen Rahmen des Dokumentes fest. In erster Linie ist das die Definition der Datentypen von Entities und Attributen. Für das Einlesen, Interpretieren und für die Überführung der Elemente in die Objektstruktur sind spezielle Programme bzw. Programmkomponenten, genannt *Parser*, verantwortlich. Das Wohlgeformtheitskriterium ist die Grundlage für eine voll automatisierte Dokumentenverarbeitung.

Bei der Beschreibung von Joinpoints werden keine Gültigkeitsanforderungen an Policy-Dateien gestellt, die Wohlgeformtheit der Joinpoint Policy und der Request Policy reicht vollkommen aus. Daher wird es auf die Angabe der Dokumenttyp-Deklaration verzichtet. In nachfolgenden Abschnitten wird das Format der beiden Policies festgelegt und erläutert. Zuvor muss allerdings der Begriff Joinpoint hinsichtlich der ObjectTeams-Spezifik präzisiert werden.

5.1 Joinpoints in ObjectTeams/Java

In [Her06] werden Joinpoints als Punkte in der statischen Programmstruktur definiert, die den Knoten des abstrakten Syntaxbaums entsprechen. Joinpoints sind also Programmelemente, mit denen der Source- und der Bytecode eines Java-Programms ausgedrückt werden kann. Somit weicht diese Definition von der klassischen Definition ab, wonach Joinpoints Stellen im Kontrollfluss eines Programms sind, und bietet eine statische Sicht auf Joinpoints an. Aus der neuen Definition können unterschiedliche Arten von Joinpnoints (Knoten) hervorgehoben werden: Methodenaufrufe, Instanziierungen, lesende (bzw. schreibende) Zugriffe auf Atribute, Werfen von Exceptions etc. Laut [Her06] können 17 bis 20 Joinpoint-Arten identifiziert werden, die zum Aspektbinden geeignet sind. Aktuell werden in OT/J folgende Arten von Joinpoints zum Weben von Aspekten verwendet:

- Methodenaufruf (`method call`)
- Lesen eines Attributes (`field get`)
- Setzen eines Attributes (`field set`)

In OT/J fungieren die oben genannten Knoten als Bezugspunkte für Callins und Callouts. Per Callouts kann auf Joinpoints von allen drei Arten (Methodenaufrufe sowie das Lesen und Setzen der Attribute) zugegriffen werden. Die Callins sind zurzeit auf Methodenaufrufe eingeschränkt, mit der Einführung von quantifizierten Bindungen sind Callins künftig auch für Attributenzugriffe denkbar.

Knoten von jeder der drei Arten sind über ihre Namen referenzierbar, also über Namen der jeweiligen Programmelemente (Methoden und Attribute). Somit handelt es sich beim Aspektweben in OT/J um zwei Arten von Joinpoint-Zugriffen: Attribut- und Methodenzugriffe. Erweitert man den Knotennamen um die Namen der Klasse und des Paketes, aus der die Klasse stammt, sowie um die Signatur der Methode (bzw. Datentyp des Attributes), so erhält man einen eindeutigen Namen des Knotens innerhalb einer Basisanwendung. Eine weitere Zugriffsart auf Joinpoints beim Aspektweben ist Adaptierung (innerer) Basisklassen, die durch die `playedBy`-Beziehung ausgedrückt wird. Diese Zugriffsart steht allerdings mit mindestens einer der drei oben aufgelisteten Joinpoint-Arten im Zusammenhang. Die Existenz eines Zugriffes auf einen Joinpoint aus einer der drei genannten Gruppen impliziert die Existenz einer `playedBy`-Beziehung zwischen der entsprechenden Basisklasse und mindestens einer Rollenklasse. Daher ist eine explizite Deklaration eines Zugriffes auf die Basisklasse in der Policy-Datei überflüssig.

Allerdings muss dabei ein Sonderfall beachtet werden. Die Adaptierung einer Basisklasse kann den Kontext der Zugriffe auf die darin enthaltenen Joinpoints beeinflussen. Dies ist genau dann der Fall, wenn die Basisklasse für die korrespondierende Rollenklasse (wenn nicht sogar für jede beliebige Klasse) unsichtbar ist. Dabei führt die Adaptierung zur Entkapselung (s. Kap. 2.1.3) der betroffenen Basisklasse und der darin enthaltenen Joinpoints. Der Sonderfall kann bei der Deklaration von Zugriffsrechten dadurch erfasst werden, wenn der Basiscode-Eigentümer die Option bekommt, die Entkapselung generell zu untersagen. Außerdem erhält der Eigentümer dadurch die Möglichkeit, die Decapsulation zentral für alle Aspekte zu deaktivieren und somit die traditionelle Kapselung aus der objektorientierten Welt als Sicherheitskonzept für den Programmcode in der aspektorientierter Anwendung wiederherzustellen. Zur Realisierung einer flexibleren Verwaltung der Klassen-Decapsulation, die die Zugriffe auf einige private Klassen ermöglicht, wäre die Deklaration dieser Klassen in der Joinpoint Policy erforderlich.

In den nächsten Abschnitten wird Definition der Formate für Joinpoint- und Request Policy vorgenommen, und damit die Anforderung aus Kapitel 4.3.5 erfüllt.

5.2 Joinpoint Policy

Definition des Policy-Formats erfolgt durch die Definition der einzelnen Elementtypen. Als Grundlage dienen dabei Anforderungen an die Gesamtarchitektur, die in Kapitel 4 aufgestellt wurden. Um Zugriffsrechte auf Joinpoints einer ObjectTeams-Basisanwendung vollständig zu beschreiben, muss die Policy folgende Datenblöcke enthalten:

- Angaben über die Basisanwendung (Name und Version)
- Angaben über den Basiscode-Eigentümer
- Liste mit Angaben über Aspekteigentümer, denen die Zugriffsrechte gewährt wurden
- Liste mit Angaben über Joinpoints und Relationen zu den entsprechenden Aspekteigentümern

Die dazugehörigen XML-Elemente werden mit den Tags `<BaseApplication>`, `<BaseCodeOwner>`, `<AspectOwnerList>` und `<JoinPointList>` entsprechend eingeleitet. Aus Performance-Gründen müssen die Elemente in der gleichen Reihenfolge wie oben aufgelistet deklariert werden, um eine schnel-

lere Überführung der Daten in die Objektstruktur beim Einlesen der Policy zu erreichen. Die Reihenfolge könnte beispielsweise durch die Deklaration der DTD festgelegt werden; das im Rahmen dieser Arbeit entwickelte Packaging Tool (s. Kapitel 7) hält diese Reihenfolge beim Generieren von Policy-Dateien auch ohne Angabe der DTD ein. Das Element <BaseApplication> enthält zwei untergeordnete Elemente <AppName> und <AppVersion>, die den Namen der Anwendung bzw. deren Versionsnummer deklarieren. Wie bereits erwähnt, korrespondieren die Eigenschaften der Eigentümer-Entities mit den Zertifikat-Attributen (s. Kap. 3.2). Für den Basiscode- und Aspektcode-Eigentümer (<AspectOwner>) sind es dementsprechend folgende untergeordnete Entities:

- <Alias>
- <Name>
- <OrganizationalUnit>
- <Organization>
- <Locality>
- <State>
- <Country>

Bei der Angabe von Zertifikaten können alle Attribute mit Ausnahme von *Alias* leer sein. Zu Verwaltungszwecken ist es jedoch sinnvoll, die Existenz der Attribute *Name*, *Organization*, *Locality* und *Country* in der Joinpoint Policy programmtechnisch zu erzwingen.

Entwicklungsschlüssel werden wie normale Aspekteigentümer behandelt und in der Aspekteigentümerliste beschrieben, sind allerdings mit dem leeren Tag <DevelopingKey/> gekennzeichnet. Damit wird die Anforderung aus Kapitel 4.3.4 erfüllt.

Entsprechend den in Kapitel 5.1 beschriebenen Arten von Joinpoints besteht die Joinpoint-Liste aus einer Kollektion von Elementen <MethodAccess> (für Methoden) und <FieldAccess> (für Attribute). Beide verfügen über Kindelemente <Name>, <Signature> und <Class>, die den Joinpoint innerhalb der Basisanwendung eindeutig identifizieren und somit zwingend sind. Als weitere untergeordnete Entities enthalten <FieldAccess> und <MethodAccess> genau eins der beiden Elemente: <Free/> (soweit es sich um einen *Free Joinpoint* handelt, s. Anforderung in Kapitel 4.3.2) oder <FAPermissionList> (bzw. <MAPermissionList> bei Methodenzugriffen), die Kollektionen von Berechtigungselementen (<FAPermission> bzw. <MAPermission>) einleiten. Falls es sich um einen *Available Joinpoint* handelt, auf den

(noch) keiner der bekannten Aspekteigentümer Zugriffsrechte hat, sind die Kollektionen `<MAPermissionList>` bzw. `<FAPermissionList>` leer.

Die Berechtigungselemente stellen Beziehungen zu den einzelnen Aspekteigentümer-Entities her und ordnen somit diesen die Zugriffsrechte zu jeweiligen Joinpoints zu. `<FAPermission>` und `<MAPermission>` enthalten jeweils eine Referenz auf den Aspekteigentümer (verbindliches Kindelement `<OwnerAlias>`) und deklarieren Daten über die Art des erlaubten Zugriffs. Bei Methodenzugriffen sind es Elemente `<Callin>` und `<CalloutToMethod/>`, für Attributenzugriffe ist nur die Angabe von `<CalloutToField>` möglich. Die Callin-Arten `before`, `after` und `replace` genauso wie CalloutToField-Arten `get` und `set` werden als Werte der entsprechenden Entity-Elemente angegeben. Als Alternative bietet sich die Deklaration von Callin-Arten als Attribut des Tags `<Callin>`, sodass die Angaben unter Verwendung einer Format-Definition (DTD) beim Parsieren validierbar sind. Da die Gültigkeit der Policy-Dateien in dieser Arbeit nicht gefordert wird, kann die Überprüfung der Deklaration von Callin-Arten in der Parser-Klasse stattfinden. Dafür ist die Angabe der Callin-Arten als Werte der Entity-Elemente ausreichend. Bei der Deklaration von zwei oder mehr Werten werden diese durch Kommata voneinander getrennt:

```
<Callin>before,after</Callin>
```

Callouts auf Methoden werden durch ein leeres Tag deklariert:

```
<CalloutToMethod/>
```

In der Policy müssen alle Joinpoints deklariert werden, die zu den Kategorien *Free* und *Available* gehören. Ist ein Joinpoint in der Policy nicht aufgeführt, so handelt es sich um einen *Denied Joinpoint*. Sollte die Joinpoint Policy die Decapsulation deaktivieren, um die traditionelle Kapselung aus der objektorientierten Welt aufrechtzuerhalten, muss das Element `<JoinPointList>` das leere Tag `<SpecialAccessDenied/>` deklarieren. Die in der Joinpoint-Liste enthaltenen Zugriffsberechtigungen, die die Kapselung verletzen, sollen bei der Auswertung von Zugriffsrechten ignoriert werden. Somit werden die betroffenen Joinpoints für *denied* erklärt, auch wenn deren Deklaration in der Joinpoint-Liste verfügbar ist. Dies ist die Behandlung des im Abschnitt 5.1 genannten Sonderfalls.

Bei der Angabe von Elementwerten muss darauf geachtet werden, dass einige Zeichen Teil des XML-Markups sind. Aus diesem Grund müssen diese Zeichen durch entsprechende *Entity-Referenzen* ersetzt werden. In XML sind folgende Zeichen für Markups vordefiniert:

- Spitze Klammer < und > (müssen durch `<` bzw. `>` ersetzt werden)
- Das Anführungszeichen " (`&quot;`)
- Das Apostroph ’ (`&apos;`)
- Das kaufmännische Und (`&`)

Die Struktur einer Joinpoint Policy-Datei lässt sich am besten an einem Beispiel erläutern.

Beispiel

Gegeben sei eine Basisklasse `Person`:

```
package org.xyz.util;
public class Person {
    private String name;
    private String address;

    public void setName(String n) {
        name = n;
    }
    public void register(int id) {
        ...
    }
    public void print() {
        ...
    }
    ...
}
```

Die Daten des Eigentümers der Basisklasse lauten:

- Alias: xyzcomp
- Name: XYZ Company
- OrganizationalUnit: Unit A
- Organization: XYZ Company
- Locality: Denver
- State: Colorado
- Country: US

Ein Aspekteigentümer mit den Daten

- Alias: abcit
- Name: Frank Mustermann
- Organization: ABC-IT GmbH
- Locality: Berlin
- Country: DE

besitzt exklusive Zugriffsrechte für das Attribut `address` (Callout, lesender Zugriff) und für die Methode `setName` (`before`- und `after`-Callin). Bei der Methode `print` handelt es sich um einen *Free Joinpoint*. Es seien keine weiteren Aspekteigentümer bekannt, die über Zugriffsrechte auf Joinpoints der Klasse `Person` verfügen. Eine entsprechende Joinpoint Policy sieht dann wie folgt aus:

```
<?xml version='1.0' encoding='us-ascii'?>
<JoinPointPolicy>
    <BaseApplication>
        <AppName>Simple Administrator</AppName>
        <AppVersion>0.1</AppVersion>
    </BaseApplication>
    <BaseCodeOwner>
        <Alias>xyzcomp</Alias>
        <Name>XYZ Company</Name>
        <OrganizationalUnit>Unit A</OrganizationalUnit>
        <Organization>XYZ Company</Organization>
        <Locality>Denver</Locality>
        <State>Colorado</State>
        <Country>US</Country>
    </BaseCodeOwner>
    <AspectOwnerList>
        <AspectOwner>
            <Alias>abcit</Alias>
            <Name>Frank Mustermann</Name>
            <Organization>ABC-IT GmbH</Organization>
            <Locality>Berlin</Locality>
            <Country>DE</Country>
        </AspectOwner>
    </AspectOwnerList>
    <JoinPointList>
        <FieldAccess>
            <Name>address</Name>
            <Signature>Ljava/lang/String;</Signature>
            <Class>org.xyz.util.Person</Class>
            <FAPermissionList>
                <FAPermission>
                    <OwnerAlias>abcit</OwnerAlias>
```

```
                    <CalloutToField>get</CalloutToField>
                </FAPermission>
            </FAPermissionList>
        </FieldAccess>
        <MethodAccess>
            <Name>setName</Name>
            <Signature>(Ljava/lang/String;)V</Signature>
            <Class>org.xyz.util.Person</Class>
            <MAPermissionList>
                <MAPermission>
                    <OwnerAlias>abcit</OwnerAlias>
                    <Callin>before,after</Callin>
                    <CalloutToMethod/>
                </MAPermission>
            </MAPermissionList>
        </MethodAccess>
        <MethodAccess>
            <Name>register</Name>
            <Signature>(I)V</Signature>
            <Class>org.xyz.util.Person</Class>
            <MAPermissionList></MAPermissionList>
        </MethodAccess>
        <MethodAccess>
            <Name>print</Name>
            <Signature>()V</Signature>
            <Class>org.xyz.util.Person</Class>
            <Free/>
        </MethodAccess>
    </JoinPointList>
</JoinPointPolicy>
```

Das angegebene XML-Dokument ist wohlgeformt [Vld]. Wie man erkennen kann, gehört das Attribut `name` zur Kategorie *Denied Joinpoints*, da es nicht in der Joinpoint-Liste aufgeführt ist. Bei der Methode `register` handelt es sich um einen *Available Joinpoint*, für den keiner der bekannten Aspekteigentümer Zugriffsrechte ausgehandelt hatte. Die Berechtigungsliste `<MAPermissionList>` ist dementsprechend leer. Das Format der Joinpoint Policy ist im Anhang A.1 dieser Arbeit in EBNF veranschaulicht.

Nach dem vorgestellten Format wird jeder bekannte Aspekteigentümer und jeder verfügbare Joinpoint genau einmal in der Policy-Datei deklariert. Die Zugriffsrechte-Beziehung wird dabei in der Deklaration des Joinpoints durch die Referenz auf den Alias des Aspekteigentümers gesetzt; die Policy-Datei ist also nach Joinpoints strukturiert. Als Alternative könnte die Strukturierung nach Aspekteigentümern in Betracht kommen, bei der Joinpoints in der Deklaration von Aspekteigentümern referenziert werden. Diese Technik würde sich allerdings negativ auf der Größe der XML-Datei auswirken,

da zum Referenzieren eines Joinpoints Angaben des Klassennamens, Elementnamens und der Signatur (bzw. des Typs bei Attributen) erforderlich ist. Somit ensteht im Vergleich zum Referenzieren des Aspekteigentümers in der Joinpoint-Deklaration (nur die Angabe dessen Aliases notwendig) ein spürbarer Overhead.

5.3 Request Policy

Während die Joinpoint Policy persistente Haltung von Zugriffsrelationen realisiert, dient die Request Policy zur Aushandlung von Zugriffsrechten. Dementsprechend ist der Lebenszyklus einer Request Policy auf die Dauer des Aushandlungsvorgangs beschränkt. Dabei bezieht sich die Policy-Datei auf eine bestimmte Basisanwendung und ist demzufolge fest einem bestimmten Basiscode-Eigentümer sowie einem konkreten Aspektcode-Eigentümer zugeordnet. Der Inhalt der Anfrage-Datei steht mit dem Inhalt der Joinpoint Policy im engen Zusammenhang, denn die Anfragen werden vom Inhalt der Joinpoint Policy abgeleitet und beinhalten Daten, um welche die Joinpoit Policy erweitert und/oder reduziert werden soll.

Die Request Policy besteht aus fünf Abschnitten:

- Angaben über die Basisanwendung (Name und Version)
- Angaben über den Basiscode-Eigentümer
- Angaben über den Aspekteigentümer
- Liste mit Berechtigungsanfragen für Joinpoint-Zugriffe
- Liste mit Berechtigungen, auf die der Aspekteigentümer verzichtet

Die ersten beiden Abschnitte (`<BaseApplication>` und `<BaseCodeOwner>`) werden unverändert aus der Joinpoint Policy übernommen. Dadurch kann der Basiscode-Eigentümer die eingegangene Anfrage-Datei der entsprechenden Joinpoint Policy zuordnen. Im Falle wiederholter Berechtigungsanfrage stimmt auch der Inhalt des Elementes `<AspectOwner>` mit dem aus dem gleichen Abschnitt der Joinpoint Policy überein. Die Aktualisierung der Aspekteigentümer-Daten ist nicht zulässig.

Die Listen werden durch die Tags `<RequestList>` für Berechtigungsanfragen bzw. `<RelinquishmentList>` für abgelegte Berechtigungen eingeleitet und bestehen aus Elementen `<FieldAccess>` und `<MethodAccess>` ähnlich wie bei einer Joinpoint Policy. Wie auch die gleichnamigen Elemente in der Joinpoint Policy enthalten `<FieldAccess>` und `<MethodAccess>`

Daten über Joinpoints (Kindelemente <Name>, <Signatur> und <Class>). Da die Request Policy einem bestimmten Aspekteigentümer eindeutig zugeordnet ist, können Zugriffsarten von Joinpoints direkt als untergeordnete Elemente von <FieldAccess> und <MethodAccess> angegeben werden. Diese untergeordneten Elemente entsprechen den Elementen <Callin>, <CalloutToMethod/> und <CalloutToField> aus der Joinpoint Policy. Die Referenz auf den Aspekteigentümer (<OwnerAlias>) ist überflüssig, da dieser innerhalb der Request Policy global eindeutig ist.

Die Berechtigungsanfagen können nur Joinpoints betreffen, die in der Joinpoint Policy aufgeführt sind und nicht in die Kategorie *Free Joinpoints* eingeteilt wurden. Anfragen über die Einteilung eines Joinpoints in eine andere Kategorie sind nicht zulässig. Aus diesem Grund ist die Verwendung von <Free/> als untergeordnetes Element von <FieldAccess> und <MethodAccess> in der Request Policy nicht gestattet. Zur Veranschaulichung der Struktur der Request Policy wird das Beispiel aus dem Kapitel 5.2 erweitert.

Beispiel

Gegeben sei die Basisklasse Person, Eigentümerdaten und eine Joinpoint Policy aus dem vorigen Beispiel (Kap. 5.2). Der Aspekteigentümer mit dem Alias abcit möchte eine Berechtigungsanfrage für die Zugriffe auf die Methoden register (*CalloutToMethod*) und setName (*replace-Callin*) stellen. Zusätzlich legt er die zuvor ausgehandelten Zugriffsrechte für das Attribut address (lesender *CalloutToField*) ab. Die dazugehörige Request Policy muss wie folgt definiert werden:

```
<?xml version='1.0' encoding='us-ascii'?>
<JoinPointRequest>
    <BaseApplication>
        <AppName>Simple Administrator</AppName>
        <AppVersion>0.1</AppVersion>
    </BaseApplication>
    <BaseCodeOwner>
        <Alias>xyzcomp</Alias>
        <Name>XYZ Company</Name>
        <OrganizationalUnit>Unit A</OrganizationalUnit>
        <Organization>XYZ Company</Organization>
        <Locality>Denver</Locality>
        <State>Colorado</State>
        <Country>US</Country>
    </BaseCodeOwner>
    <AspectOwner>
        <Alias>abcit</Alias>
        <Name>Frank Mustermann</Name>
        <Organization>ABC-IT GmbH</Organization>
        <Locality>Berlin</Locality>
        <Country>DE</Country>
```

```
    </AspectOwner>
    <RequestList>
        <MethodAccess>
            <Name>setName</Name>
            <Signature>(Ljava/lang/String;)V</Signature>
            <Class>org.xyz.util.Person</Class>
            <Callin>replace</Callin>
        </MethodAccess>
        <MethodAccess>
            <Name>register</Name>
            <Signature>(I)V</Signature>
            <Class>org.xyz.util.Person</Class>
            <CalloutToMethod/>
        </MethodAccess>
    </RequestList>
    <RelinquishmentList>
        <FieldAccess>
            <Name>address</Name>
            <Signature>Ljava/lang/String;</Signature>
            <Class>org.xyz.util.Person</Class>
            <CalloutToField>get</CalloutToField>
        </FieldAccess>
    </RelinquishmentList>
</JoinPointRequest>
```

Die angegebene Request Policy ist ein wohlgeformtes XML-Dokument [Vld]. Die Bestätigung der Request Policy durch den Basiscode-Eigentümer impliziert die Aktualisierung der Joinpoint Policy. Dabei hat der Basiscode-Eigentümer die Option, bestimmte Joinpoints einer anderen Zugriffskategorie zuzuordnen. Wie aus dem Beispiel ersichtlich ist, kann das Attribut `address` in die Kategorien *Free* oder *Denied Joinpoints* eingeteilt werden, da nach der Bearbeitung der oben aufgeführten Anfrage-Datei kein Aspekteigentümer über exklusive Zugriffsrechte auf diesen Joinpoint verfügt. Die Format-Definition der Request Policy ist im Anhang A.2 in EBNF angegeben.

Kapitel 6

Erweiterung der ObjectTeams/Java-Laufzeitumgebung

Dieses Kapitel beschäftigt sich mit dem ersten Teil der praktischen Arbeit und beschreibt die Implementierung des *Join Point Access Controllers (JPAC)*. JPAC erweitert die Funktionalität der OT-Laufzeitumgebung, indem er sich in den Webeprozess einklinkt, um Zugriffsrechte auf beim Weben verwendete Joinpoints zu überprüfen. Die Erweiterung der OT-Laufzeitumgebung erstreckt sich auf zwei Punkte. Zunächst muss eine Infrastruktur entwickelt werden, die das Parsen von Policies und deren Überführung in die Objektstruktur zum Verwalten von Zugriffsrechten ermöglicht. Dies ist die Grundlage für den zweiten Punkt, der sich mit der eigentlicher Zugriffskontrolle befasst. In den nachfolgenden Abschnitten wird die Realisierung der beiden genannten Punkte dargestellt.

6.1 Policy-Parser und Zugriffsrechteverwaltung

Zum Parsen von Policies wird die in Java unterstützte *SAX (Simple API for XML)* verwendet. Bei der Verarbeitung von XML-Dokumenten verfolgt SAX einen event-basierten Ansatz, der sich von den anderen Verarbeitungstechniken durch etwas schnelleres und speicherschonendes Vorgehen unterscheidet. Dies erweist sich insbesondere bei der Verarbeitung von großen XML-Dateien als vorteilhaft und spricht für den Einsatz von SAX beim Parsen von Joinpoint Policies, da diese bei einer großen Anzahl von Joinpoints in der Basisanwendung sehr groß sein können.

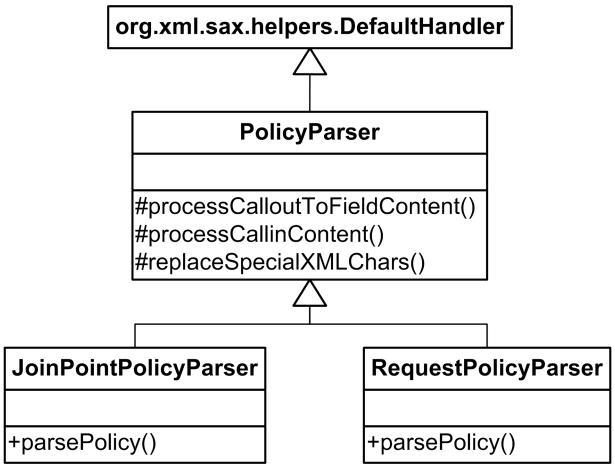

Abbildung 6.1: Die Parser-Klassen

6.1.1 Parser-Klassen

Kapitel 4.3.5 definiert zwei Arten von Policies (Joinpoint Policy und Request Policy), dementsprechend wurde für jede Policy jeweils eine Parser-Klasse (`JPPolicyParser` und `RequestPolicyParser`) im Paket `org.objectteams.transformer.jpac.parser` implementiert. Diese erweitern die Klasse `PolicyParser`, die Methoden zur Verarbeitung von Entity-Inhalten sowie zur Behandlung der für Markups reservierten Zeichen bereitstellt. (Abb. 6.1). Die Dateien werden aus einem Datenstrom gelesen, der an die Funktion `parsePolicy` der jeweiligen Parser-Klasse übergeben wird.

6.1.2 Infrastruktur zur Verwaltung von Zugriffsrechten

Das Paket `org.objectteams.transformer.jpac.util` stellt eine Infrastruktur zur Verwaltung von aus den Policies eingelesenen Informationen bereit. Die Infrastruktur ist in der Abbildung 6.2 in Form eines UML-Klassendiagramms veranschaulicht.

Datenstrukturen

Zur Speicherung von Objekten werden folgende Datenstrukturen verwendet:

- `HashMap` dient zur unsortierten Speicherung von Objekten anhand des übergebenen Schlüssels. Ist ein Objekt unter angegebenem

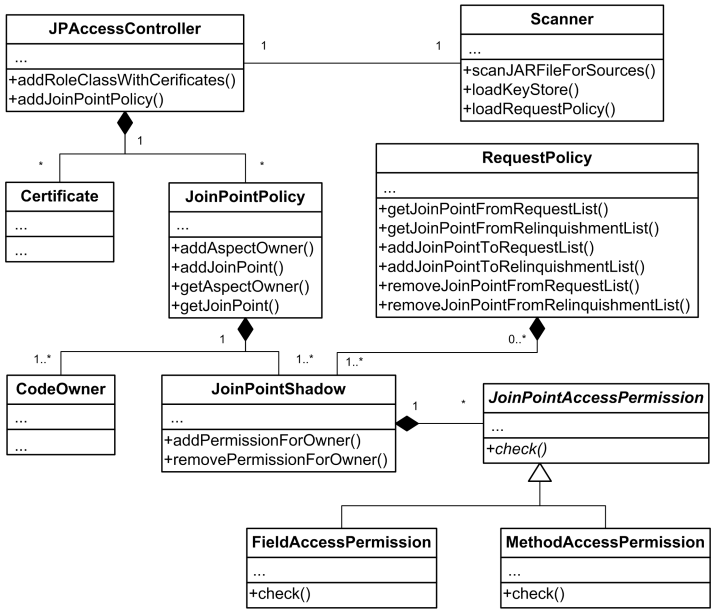

Abbildung 6.2: Infrastruktur zur Verwaltung von Zugriffsrechten

Schlüssel in der `HashMap` bereits vorhanden, so wird es mit dem neuen Objekt ersetzt. `HashMap` dient zur Speicherung der Permissions in einem `JoinpointPointShadow`-Objekt sowie der Policies und Namen der bekannten Klassen innerhalb des JPAccessControllers.

- `HashList` wurde im Infrastruktur-Paket implementiert und enthält zur Performance-Verbesserung die redundante Datenstruktur vom Typ `LinkedList`. Die Datenstruktur speichert beliebige Objekte gleichen Typs unter Verwendung von `String`-Instanzen als Schlüssel. Im Unterschied zu `HashMap` ist eine wiederholte Speicherung von Objekten unter dem gleichen Schlüssel nicht möglich, d. h. die Einträge können nicht überschrieben werden. Ist ein Objekt unter dem angegebenen Schlüssel in der Hash-Liste bereits enthalten, so werden erneute Speicherungsversuche ignoriert. Die Funktion `put(String key, <T> value)` zum Einfügen der Elemente liefert `true`, falls das übergebene Element gespeichert werden konnte, `false` sonst. `HashList` wird zum Speichern von `JoinPointShadow`- und `CodeOwner`-Instanzen in der Policies verwendet.

- `ListValueHashMap` wurde im Paket `org.objectteams.transformer.util` implementiert und kann Objekte mit dem gleichen Schlüssel in Form einer Liste speichern. Wie auch `HashList` enthält `ListValueHashMap` eine redundante Datenstruktur vom Typ `LinkedList`. Als Schlüssel werden ebenfalls `String`-Objekte verwendet. `ListValueHashMap` dient

zur Speicherung der Bezeichnungen von eingescannten Klassenelementen (s. Kap. 6.2.3) sowie Zertifikaten in JPAC.

Repository

Die zentrale Rolle in der Infrastruktur spielt die Klasse `JPAccessController`, die neben der eigentlichen Zugriffskontrolle (s. Kapitel 6.2) Aufgaben eines Repository für eingelesene Joinpoint Policies und Zertifikate[1] erfüllt. Analog zu der `AccessController`-Klasse aus dem Sicherheitspaket von Java kann die Klasse `JPAccessController` nicht instantiiert werden und enthält nur statische Members (Methoden und Attribute). Die darin gespeicherten `JoinpointPolicy`- und `Certificate`-Objekte werden intern verwendet, dementsprechend bietet `JPAccesController` keine Getter-Methoden für diese Objekte. Das Format zur Speicherung von oben genannten Objekten sieht wie folgt aus:

- Joinpoint Policies in `HashMap<String, JoinPointPolicy> policies`: Als Schlüssel (vom Typ `String`) dienen die absoluten Pfade von Basispaketen, aus der die Policy-Datei stammt.
- Zertifikate in `ListValueHashMap<Certificate> roleClassCertificates`: Als Schlüssel (vom Typ `String`) fungiert der vollständige Name der Rollenklasse; die gespeicherten Werte sind Zertifikate der Klasseneigentümer (in Form einer Liste). Die Relationen zwischen Klassennamen und absoluten Pfaden von Paketen, in denen diese Klassen enthalten sind, werden in der Datenstruktur `HashMap<String, String> scannedRoleClassNames` gespeichert.

Source Scanning

Unmittelbar nach der Aktivierung des JPAC wird der Java-Classpath nach Basispaketen und Aspekten durchsucht, um die Menge von Joinpoints der potentiellen Basisanwendung und die Signer-Zertifikate zu bestimmen. Der Klassenpfad wird der System-Property `java.class.path` entnommen und ist ein aus den einzelnen Pfaden konkatenierter String. Nach der Zerlegung dieses Strings werden irrelevante Pfade ausgefiltert. Als irrelevant werden folgende Pfade eingestuft:

- Alle Pfade, die nicht auf eine JAR-Datei verweisen

[1] Es handelt sich um Zertifikate, die die Signer von Rollenklassen identifizieren.

- JAR-Dateien, die in der OT-Laufzeitumgebung intern verwendet werden, sowie das Paket `rt.jar` mit Java-Kernklassen

Als Rest ergibt sich dann eine Menge aus Pfaden zu einzelnen JAR-Dateien. Diese Pfade haben für den JPAC ähnliche Bedeutung wie Code-Ressourcen für den Access Controller aus dem Java-Sicherheitspaket (s. Kapitel 3.1).

Das Scannen jeder einzelnen JAR-Datei übernimmt die Klasse `Scanner`, die für alle Zugriffe auf das lokale Dateisystem verantwortlich ist. Beim Einlesen der JAR-Datei wird überprüft, ob diese in ihrem Hauptverzeichnis eine Joinpoint Policy enthält. Ist dies der Fall, registriert JPAC alle in dem Archiv enthaltenen Klassen als Basisklassen durch die Eintragung der entsprechenden Relation in die Datenstruktur `HashMap<String, JoinPointPolicy> baseClassCandidates`. Als Schlüssel dient dabei der Name der Basisklasse, als Wert wird die `JoinPointPolicy`-Instanz angegeben, die die Zugriffsrechte auf Joinpoints dieser Klasse deklariert. Neben den potenziellen Basisklassen müssen auch die Rollenklassen eingelesen werden, um deren Eigentümer zu bestimmen. Das Ziel ist es, die Zertifikate der Sígner zu ermitteln und diese unter Angabe des Klassennamens im JPAC abzuspeichern. Die Rollenklassen lassen sich nur anhand des Wertes vom OT-Bytecode-Attribut `OTClassFlags` erkennen, somit muss der Scanner bei jeder Klasse prüfen, ob das Bytecode-Attribut verfügbar ist und, falls ja, welchen Wert das Attribut hat. Handelt es sich bei einer Klasse um eine Rolle, so bestimmt der Scanner die Zertifikate deren Eigentümers und registriert diese beim JPAC.

Besteht eine Anwendung aus mehreren JAR-Dateien, so kann es vorkommen, dass unterschiedliche Klassen unter dem gleichen Namen in mehreren Archiven implementiert sind. Beim Ausführen der Anwendung wird mit diesem Namen immer die Klasse adressiert, deren JAR-Datei-Pfad im Java-Classpath als Erster (d. h. am weitesten links) aufgeführt ist. Der JPAC erlaubt keine wiederholte Registrierung einer Klasse mit einem bereits bekannten Namen. Dementsprechend muss die Reihenfolge bei der Bearbeitung der einzelnen Pfade während des Scannens beachtet werden. Damit wird sicher gestellt, dass JPAC immer die Rechte der tatsächlich adressierten Klasse enthält.

Eindeutigkeit von Namen

Wie bereits erwähnt, werden Klassen und Joinpoints beim JPAC anhand ihrer Namen registriert, die dabei als Schlüssel bei der Eintragung in die entsprechenden Datenstrukturen dienen. Dies setzt die Eindeutigkeit des Namens innerhalb der Basisanwendung voraus. Die Voraussetzung ist durch folgende Konventionen erfüllt:

- Jede Klasse wird durch ihren vollständigen Namen identifiziert. Dieser resultiert sich aus dem Namen des Paketes und dem eigentlichen Namen der Klasse (Namensteile werden durch Punkte getrennt).
- Jeder Joinpoint wird durch die Konkatenation des Klassennamens, des eigenen Namens innerhalb der Klasse (Attribut- bzw. Methodenname) und seiner Signatur (bzw. des Typs bei Attributen) identifiziert. Zur Konkatenation wird der String `$$$` verwendet. Damit wird die Verwechslungsgefahr mit einem Teil des Methoden- bzw. Klassennamens vermieden, denn die Wahrscheinlichkeit des Auftretens von `$$$` in einem Namen als relativ gering eingestuft werden kann. Beispiel für eine Methode:

  `Klassenname$$$Methodenname$$$Methodensignatur`

Werden in einer Applikation mehrere Klassenlader verwendet, so ist der Name der Klasse innerhalb der Anwendung nicht mehr eindeutig, denn jedes `ClassLoader`-Objekt genau einen Namensraum definiert. In diesem Fall wäre die Angabe des Klassenladers im Gesamtstring notwendig, beispielsweise die Voranstellung der String-Repräsentation (Methode `String Object.toString()`) vom entsprechenden `ClassLoader`-Objekt an den Namen der Klasse.

Erweiterbarkeit der Infrastruktur

Die Arten der Permissions korrespondieren mit den Joinpoint-Arten. Die entsprechenden Permissions werden durch die Klassen `FieldAccessPermission` bzw. `MethodAccessPermission` repräsentiert, die jeweils die abstrakte Klasse `JoinPointAccessPermission` erweitern. Jede konkrete Permission-Klasse implementiert die abstrakte Funktion `check` aus der Klasse `JoinPointAccessPermission`, indem sie den booleschen Wert des entsprechenden Permission-Flags zurückliefert. Eine solche Struktur ist auf eine mögliche Erweiterung der Sprache durch Hinzunahme von neuen Joinpoint-Arten ausgelegt.

6.2 Join Point Access Controller

Die Analogie zwischen dem *Join Point Access Controller (JPAC)* und dem *Access Controller (AC)* aus dem Sicherheitspaket von Java wurde bereits im letzten Abschnitt angesprochen. Die oben beschriebene Infrastruktur von JPAC, zu der auch die Klasse `JPAccessController` zählt, kann als eine eigenständige Programmkomponente betrachtet werden, die bei Bedarf

aktiviert werden kann. Allerdings weisen die beiden Controller funktionale Unterschiede auf. Während der AC seine Aufgaben mit dem Sicherheitsmanager teilt, ist JPAC sowohl für Authentisierung als auch für Autorisierung der Eigentümer zuständig. Nachfolgende Abschnitte befassen sich mit der Aktivierung des JPAC sowie mit der Realisierung der beiden Sicherheitsmechanismen.

6.2.1 Aktivierung

Die Funktionalität von JPAC stützt sich darauf, dass alle Daten über die Aspektbindung zu Beginn des Webeprozesses bekannt sind. Der zweite (abschließende) Phase des Webeprozesses findet zur Ladezeit der Anwendung statt (s. Kap. 2.2). Somit müssen die Informationen darüber, ob und in welcher Konfiguration JPAC aktiv sein soll, bereits beim Start der OT-Laufzeitumgebung verfügbar sein. Diese Anforderung kann durch das Setzen der System-Properties erfüllt werden. Zur Benachrichtigung der Laufzeitumgebung wurden folgende System-Properties definiert:

- `ot.controller.runmode` aktiviert den JPAC und legt fest, in welchem Modus die Anwendung ausgeführt werden soll. Gemäß in Kapitel 4.3.6 definierten Ausführungsmodi sind folgende Werte zulässig: `secure`, `common`, `developing`, `requesttest` und `unchecked`. Bei `unchecked` wird JPAC nicht aktiviert.
- `ot.controller.handling` bestimmt die Behandlungsart der Zugriffsrechteverletzungen (s. Kapitel 4.3.7). Zulässige Werte sind `exception`, `warning` und `exit`. Ist die Property nicht angegeben, so wird `exception` als vorgegebener Wert gesetzt.
- Unter `ot.controller.keystore` übergibt man die Adresse der Benutzer-Keystore mit dem Zertifikat des Basiscode-Eigentümers bzw. mit mehreren Zertifikaten, falls die Basisanwendung aus mehreren Komponenten besteht, die von unterschiedlichen Eigentümern stammen.
- Mit Hilfe von `ot.controller.keypass` wird das Passwort der Benutzer-Keystore angegeben.
- `ot.controller.requestpolicy` bestimmt die Adresse der Request Policy, falls die Anwendung in Request Test Mode ausgeführt werden soll.

Bei der Angabe von Werten der ersten beiden Properties wird zwischen Klein- und Großschreibung nicht unterschieden.

Die Aktivierung des JPAC erfolgt über den Aufruf der Methode `startAccessController`. Um die rechtzeitige Aktivierung zu gewährleisten, wird die Startmethode unmittelbar vor dem Scannen von Bytecode-Attributen der ersten geladenen Klasse aufgerufen, also aus der Methode `scanClassOTAttributes` der Tranformer-Oberklassse `ObjectTeamsTransformation` heraus. Der Aktivierungsprozess besteht aus zwei Aktionen:

- Einlesen der Property-Werte und Konfigurierung des JPAC
- Scannen des Klassenpfades (s. Kapitel 6.1.2)

Die eingelesenen Property-Werte bestimmen, ob die zweite Aktion ausgeführt werden soll. Um eine mehrfache Aktivierung des JPAC beim Laden jeder neuen Klasse zu verhindern, muss die Tranformer-Oberklasse die JPAC-Methode `getStartChecked` vor der JPAC-Aktivierung aufrufen. Diese liefert einen booleschen Wert zurück, der anzeigt, ob Versuche zum Starten des JPAC bereits unternommen und somit dessen Konfigurierung durchgeführt wurde. Dementsprechend enthält die Methode `scanClassOTAttributes` zur Aktivierung des JPAC den folgenden Aufruf:

```
if(!JPAccessController.getStartChecked()) {
    JPAccessController.startAccessController();
}
```

6.2.2 Verifizierung des Basispaketes

In diesem Abschnitt wird die erste der drei in Kapitel 4.3 beschriebenen Überprüfungen behandelt. Authentisierung des Basiscode-Eigentümers und Verifizierung des Basiscodes findet beim Scannen des Basispaketes und somit bei der Aktivierung des Join Point Access Controllers statt. Java-API bietet die Möglichkeit, die einzelnen JAR-Entries direkt beim Einlesen der Archiv-Datei zu verifizieren. Dafür muss das `verify`-Flag beim Instanziieren der Klasse `JarFile` gesetzt werden:

```
JarFile jar = new JarFile(jarFileName, true);
```

Wird eine JAR-Datei als Basispaket erkannt, so findet zunächst die Verifizierung der Policy-Datei und der Basiskeystore statt gefolgt von iterativer Verifizierung der einzelnen Classfiles. Zur Überprüfung der Identität des Basiscode-Eigentümers implementiert die Klasse `JPAccessController` die Funktion `verifySourceFromBasePackage`, die folgende Eingaben erwartet:

- Array mit Zertifikaten, die den Signer der Datei identifizieren. Diese

ermittelt man beim Einlesen der Entry durch der Aufruf der Funktion `getCertificate` der Klasse `JarEntry`.

- Alias des Basicode-Eigentümers, der die im Basispaket enthaltene Joinpoint Policy angibt.
- Name der `JarEntry` (Dateiname), der allerdings nicht beim Verifizieren, sondern lediglich bei der Ausgabe eventueller Fehlermeldung verwendet wird.

Bei der Überprüfung wird das Zertifikat unter Angabe des übergebenen Aliases der Benutzer-Keystore entnommen und liefert den öffentlichen Schlüssel des Basiscode-Eigentümers. Dieser wird dann bei der Verifizierung der ermittelten Signer-Zertifikate eingesetzt, indem er als Parameter bei Aufrufen der Funktion `verify` jedes einzelnen `Certificate`-Objektes dient. Die Funktion `verify` liefert `true`, falls der Schlüssel des Basicode-Eigentümers mit dem Signer-Zertifikate korrespondiert. Die Verifizierung ist erfolgreich, wenn der Schlüssel zu einem der Zertifikate passt, andernfalls wird eine `JoinPointAccessException` geworfen.

Wie bereits erwähnt, wird bei der Entnahme des Basiscode-Eigentümer-Zertifikates aus der Benutzer-Keystore der Alias aus der Joinpoint Policy verwendet. Aus diesem Grund muss der Benutzer darauf achten, dass das Zertifikat des Basiscode-Eigentümers unter dem gleichen Alias in der Benutzer-Keystore gespeichert ist, wie ihn die Policy-Datei des Basispaketes angibt.

6.2.3 Verifizierung der Aspekte und Zugriffsrechtekontrolle

Die zweite und dritte Überprüfungen (s. Kapitel 4.3) realisieren die Sicherheitsmechanismen *Authentisierung* und *Autorisierung* und erfolgen beim Scannen von Bytecode-Attributen jeder einzelnen Team- bzw. Rollenklasse. Als Quellen dienen die in Kapitel 2.2.1 vorgestellten OT-Bytecode-Attribute `CalloutMappings`, `CallinMethodMappings` und `OTSpecialAccess`.

Überprüfungsfunktionen

Analog zu den `check`-Funktionen des Klasse `SecurityManager` aus dem Sicherheitspaket von Java stellt die Klasse `JPAccessController` zwei Funktionen zur Überprüfung von Zugriffsrechten auf Joinpoints bereit:

- `checkJoinPointAccess` prüft die Zugriffsrechte für alle Arten von Join-

point-Zugriffen: `before-`, `after-` und `replace-`Callins sowie Callouts auf Methoden und Attribute (`get` und `set`). Als Eingaben werden folgende Daten erwartet:

  - Name der Basisklasse
  - Name der Basismethode (bzw. des Attributes)
  - Signatur der Basismethode (bzw. Typ des Attributes)
  - Name der Rollenklasse
  - Art des Joinpoint-Zugriffes
  - Typ des Joinpoints (Methode oder Attribut)

- `checkJoinPointSpecialAccess` wertet Zugriffe auf entkapselte Joinpoints aus. Dies sind Methoden und Felder aus der Basisklasse, die im Bytecode-Attribut `OTSpecialAccess` aufgeführt wurden. Da das eventuelle Verbot der Decapsulation sich auf alle Joinpoints der Basisanwendung und alle Aspekteigentümer erstreckt, ist die Authentisierung der Aspekteigentümer bei solchen Zugriffen nicht nötig. Die Prüfung reduziert sich auf die Abfrage des entsprechenden Flags beim Policy-Objekt. Ist dieses gesetzt, so wird eine Zugriffsverletzung registriert. Eingabe-Parameter der Funktion:

  - Name der Basisklasse
  - Name der Basismethode (bzw. des Attributes)
  - Signatur der Basismethode (bzw. Typ des Attributes)
  - Art des Joinpoint-Zugriffes

Die Authentisierung des Aspekteigentümers in der Funktion `checkJoinPointAccess` verläuft ähnlich wie die des Basiscode-Eigentümers (Kap. 6.2.2). Dabei werden die bei der Aktivierung des JPAC ermittelten Signer-Zertifikate der betroffenen Rollenklasse mit öffentlichen Schlüsseln aller innerhalb der Keystore des Basispaketes bekannten Aspekteigentümer verglichen. Wurde die Rollenklasse von mehreren Aspekteigentümern signiert, so werden deren Zugriffsrechte bei der Autorisierung *'verodert'*. Bei der ODER-Strategie ist der Zugriff erlaubt, wenn mindestens ein Aspekteigentümer Zugriffsrechte auf den Joinpoint besitzt. Dies bietet zwar eine Lücke für den Missbrauch von Zugriffsrechten, wenn beispielsweise ein Aspekteigentümer, der über Zugriffsrechte auf bestimmte Joinpoints verfügt, den Aspekt eines Fremden *'mitsigniert'*, um diesem die eigenen Zugriffsrechte zu gewähren. Ein solcher Missbrauch ist aber eher eine rechtliche Angelegenheit, die mit dem Missbrauch von Software-Lizenzen verglichen werden kann, und soll somit in dieser Arbeit nicht weiter thematisiert werden.

Die Autorisierung erfolgt über die Einstufung des Joinpoints in eine Zugriffskategorie aus der Sicht der verfügbaren Aspekteigentümer-Zertifikaten. Bei

der ODER-Strategie ist das die Kategorie aus der Sicht des Eigentümers, der die meisten Zugriffsrechte auf den Joinpoint hat. Damit wird die Anforderung aus Kapitel 4.3.2 erfüllt.

Einbindung der Überprüfungsfunktionen

Die beiden `check`-Funktionen werden direkt nach dem Einlesen von Daten des entsprechenden Joinpoints aus dem Bytecode-Attribut (`CalloutMappings`, `CallinMethodMappings` oder `OTSpecialAccess`) aufgerufen. Erst nach der erfolgreichen Überprüfung von Zugriffsrechten für alle Bindungen wird die zweite Phase des Webeprozesses (s. Kap. 2.2) eingeleitet. Je nach angegebenem Behandlungstyp von Zugriffsrechteverletzungen (Werfen einer Exception oder Beenden der Anwendung) kann das Weben von Callin-Bindungen verhindert werden. Sollte als Behandlung einer Zugriffsrechteverletzung eine Ausgabe der entsprechenden Warnung erfolgen, wird der Webeprozess nicht unterbrochen. Für Callout-Bindungen (mit Ausnahme der Entkapselung von Attributen, Methoden und Klassen) ist dagegen der OT-Compiler zuständig, somit sind diese zum Zeitpunkt der Überprüfung zum Teil verwoben.

Wie auch bei der Aktivierung des JPAC sind die Aufrufe der beiden `check`-Funktionen in der Methode `scanClassOTAttributes` implementiert. Somit ist diese Methode die einzige Stelle, an der die Einbindung des JPAC in die OT-Laufzeitumgebung stattfindet.

Aktionen bei Zugriffsrechteverletzungen

Stellt eine der beiden oben beschriebenen `check`-Funktionen eine Zugriffsverletzung fest, so wird die Funktion `handleAccessViolation` aufgerufen, die die Klasse `JPAccessController` als Behandlungsmethode für Zugriffsrechteverletzungen bereitstellt. Diese setzt die Anforderung aus Kapitel 4.3.7 um und ist für folgende Eingabewerte definiert:

- Ausführungsmodi: `secure`, `common`, `developing`
  Bei `unchecked` ist der JPAC nicht aktiv. Nach der Aktualisierung der Joinpoint Policy (s. unten) im `requesttest`-Modus werden Zugriffsrechteverletzungen wie im `secure`-Modus behandelt.

- Zugriffskategorien: `available`, `denied`
  Für Joinpoints der Kategorien `free` und `confirmed` sind Zugriffsrechteverletzungen nicht definiert.

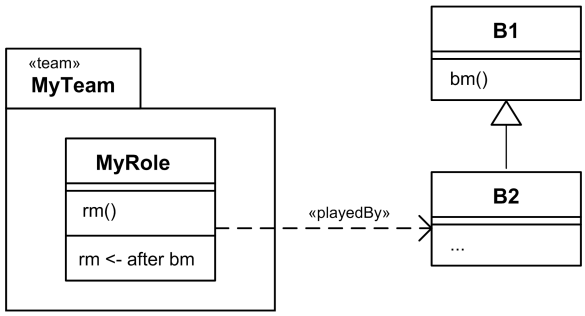

Abbildung 6.3: Methodendeklaration in der Oberklasse

Als Ergebnis der Funktion wird gemäß der aufgestellten Anforderung genau eine der folgenden Aktionen ausgeführt:

- Ausgabe einer Warnung auf den Ausgabestrom der Klasse `System`
- Werfen einer `JoinPointAccessException`
- Aufruf der Methode `System.exit()`

Aktualisierung der Joinpoint Policy

Der Ausführungsmodus `requesttest` sieht die Erweiterung der in der Joinpoint Policy definierten Zugriffsrechte vor. Zu diesem Zweck implementiert die Klasse `JoinPointPolicy` die Methode `updateWithRequestPolicy`, die als Eingabe ein `RequestPolicy`-Objekt erwartet. Die Aktualisierung der Policy erfolgt bei der Aktivierung des JPAC, falls die entsprechenden Properties (s. Kap. 6.2.1) beim Programmstart gesetzt wurden. Unmittelbar nach der Aktualisierung wird der Ausführungsmodus auf `secure` umgeschaltet.

Deklarierende Basisklassen

Bei der Vergabe der Zugriffsrechte auf Joinpoints wird *deklarative* Strategie verfolgt. Ein Joinpoint wird in der Policy unter der Angabe der Basisklasse aufgeführt, die den Joinpoint deklariert. Dadurch wird der bei der Klassenvererbung entstehende Overhead vermieden. Dieser würde sich nämlich in der Größe der Policy-Datei resultieren, falls die Joinpoint Policy alle vererbten Joinpoints enthalten sollte. Die deklarative Strategie reduziert diesen Aufwand, erfordert allerdings bei der Autorisierung die Ermittlung der deklarierenden Basisklasse. Ein Beispiel dafür ist auf der Abbildung 6.3 dargestellt. Die Klasse `B1` deklariert die Methode `bm`, die in der Subklasse `B2`

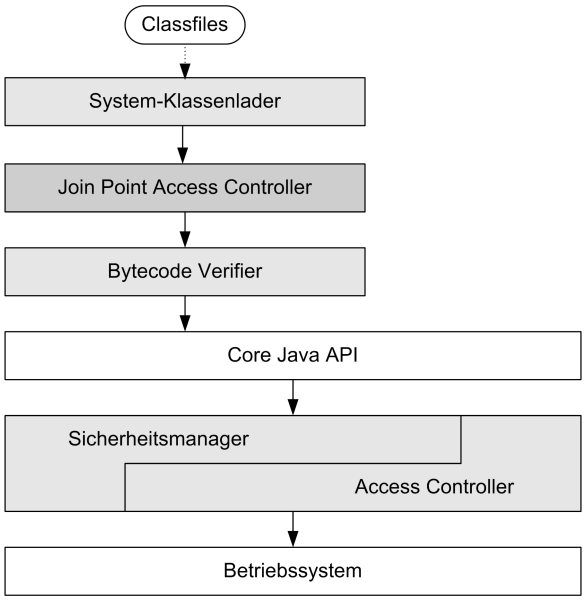

Abbildung 6.4: Einordnung des JPAC in die Sicherheitsarchitektur

an die Rollenmethode `rm` gebunden wird. In der Joinpoint Policy wird `bm` als Joinpoint der Klasse `B1` aufgeführt. Die Ermittlung der deklarierenden Basisklasse setzt die Speicherung des Vererbungsbaums für die Basisanwendung voraus. Dieser wird beim Scannen des Basispaketes während der JPAC-Aktivierung erzeugt, indem für jede eingescannte Klasse ein Tupel der Form

(Name_der_Klasse, Name_der_Oberklasse)

in der Datenstruktur `superClassNames` vom Typ `HashMap<String,String>` als Kante des Vererbungsbaums gespeichert wird. Da Java die Mehrfachvererbung nicht unterstützt, sind die Schlüsselwerte innerhalb der Datenstruktur (der erste Parameter des Tupels) eindeutig, sodass der Weg entlang der Hierarchie nach oben deterministisch ist.

Die Bestimmung der deklarierenden Klasse übernimmt die Funktion `getDeclaringClassName` in der Klasse `JPAccessController`. Diese benutzt zwei Datenstrukturen `scannedMethodNames` und `scannedFieldNames` vom Typ `ListValueHashMap<String>`, die unter den Klassennamen als Schlüssel Namen und Signaturen der deklarierten Methoden bzw. Namen und Typen der Attribute enthalten, und liefert der Namen der deklarierenden Basisklasse zurück. Ist der Joinpoint außerhalb des Basispaketes deklariert, wird `null` zurückgeliefert, worauf die Funktion `checkJoinPointAccess` den Zugriff als erlaubt wertet. Erst wenn der Name der deklarierenden Basis-

klasse bekannt ist, können die Zugriffsrechte auf den betroffenen Joinpoint geprüft werden.

JPAC und Linking

Die Aktivität des JPAC erstreckt sich ausschließlich auf den Linking-Prozess, indem alle JPAC-Aktionen unmittelbar nach dem Laden der Klasse durchgeführt werden, bevor der Bytecode-Verifier die Überprüfung des Codes auf seine Konformität zur Sprachdefinition aufnimmt (s. Abbildung 6.4).

Kapitel 7

Das Packaging Tool

Der zweite Teil der praktischen Arbeit behandelt die Implementierung des *Packaging Tools* zur Unterstützung des Code-Eigentümers bei der Erstellung von Policy-Dateien sowie Erzeugung und Verwaltung gültiger Basispakete. Die nächsten Abschnitte befassen sich mit den einzelnen Entwicklungsphasen: Aufstellen der Anforgerungen (Kapitel 7.1), Entwurf (Kap. 7.2) und Realisierung (Kap. 7.3 und 7.4).

7.1 Anforderungen

Das Tool soll folgende Anforderungen erfüllen:

A. Nicht-funktionale Anforderungen:

   1. Plattformunabhängigkeit
   2. Realisierung als Standalone-Applikation
   3. Bedienung über ein grafisches Interface

B. Funktionale Anforderungen:

   1. Authentisierung der Code-Eigentümer
   2. Erstellen und Verwalten der Joinpoint Policies
   3. Erstellen und Verwalten von Request Policies
   4. Erstellen und Verwalten der in den Basispaketen enthaltenen Keystores.
   5. Signieren der Basispakete

Je nachdem, ob es sich beim Benutzer um einen Aspekt- oder Basiscode-Eigentümer handelt, soll das Tool zwei Interaktionsmodi bieten. Während die Aspekteigentümer das Tool lediglich zur Erstellung von Request Policies nutzen, verfügt jeder Basiscode-Eigentümer über administrative Rechte auf sein Basispaket. Administrierung des Basispaketes schließt folgende Interaktionen mit ein:

- Anzeige von *denied* Joinpoints der im Basispaket befindlichen Klassen: Dadurch ist der Basiscode-Eigentümer in der Lage, Mengen von allen Joinpoint-Kategorien zu verwalten. Die Anzeige von Joinpoints für einen Aspekteigentümer soll auf die Mengen *free*, *confirmed* und *available* beschränkt werden.

- Entfernen und Einfügen von Schlüsseln aus der bzw. in die Basiskeystore: Während Aspekteigentümer den Inhalt der Basiskeystore nur sehen dürfen, können die Basiscode-Eigentümer diesen auch verändern.

- Anzeige aller Berechtigungen für *confirmed* Joinpoints: Der Eigentümer des Basispaketes soll jederzeit die Informationen über **alle** vergebenen Rechte abrufen können. Für sonstige Benutzer beschränkt sich die Anzeige nur auf ihre eigenen Berechtigungen.

- Aktualisierung von Joinpoint Policies durch Bestätigung der in den Request Policies deklarierten Anfragen.

7.2 Entwurf

Alle nicht-funktionalen Anforderungen können bereits in der Entwurfsphase erfüllt werden. Das Packaging Tool wird in Form einer *Java-Swing-Applikation* (Anforderungen A.2 und A.3) realisiert. Swing ist ein API zum Implementieren von grafischen Benutzeroberflächen in der Programmiersprache Java, deren Plattformunabhängigkeit die Plattformunabhängigkeit der Applikationen impliziert (A.1).

Bei der Verwaltung von Basispaketen verwendet das Packaging Tool Klassen aus der in Kapitel 6 vorgestellten Infrastruktur. Dazu gehören Parser-Klassen sowie die Implementierungen von Datenstrukturen, Policies, Joinpoints, Berechtigungen und Code-Eigentümern.

7.2.1 Model-View-Controller

Um ein flexibleres Programmdesign zu gewährleisten, wird das Packaging Tool als interaktive Anwendung nach dem Architekturmuster *Model-View-Controller* [Re79] realisiert. Die Klasse `gui.GUI` ist für die Präsentation der Inhalte von Policies und Keystores in Tabellenform zuständig (s. Kap. 7.2.2), wobei jede Tabelle ihr eigenes Datenmodell als eine Instanz der Klasse `PolicyModel` bzw. `RequestPolicyModel` oder `KeyStoreModel` verwendet. Das Verhalten der Applikation wird in der Klasse `Controller` implementiert, zu deren Aufgaben Bearbeitung der Benutzeraktionen und Realisierung der Anwendungslogik zählen.

7.2.2 Die Benutzeroberfläche

Ausgehend von den Anforderungen B.1, B.2, B.3 und B.4 lässt sich die Benutzeroberfläche (Klasse `gui.GUI`), in vier folgende Gruppen von grafischen Komponenten einteilen, die sich jeweils mit einer der folgenden Aufgaben beschäftigen:

- Darstellung und Verwaltung der Joinpoint Policy
- Darstellung und Verwaltung der Keystore aus dem Basispaket (Basiskeystore)
- Darstellung und Verwaltung der Request Policy
- Darstellung der Benutzer-Keystore, die die privaten Schlüssel zur Authentisierung des Benutzers sowie Zertifikate zur Verwaltung der Basiskeystore beinhaltet.

Jede Gruppe wird durch einen Tab dargestellt, wobei die Auflistung von Inhalten in Tabellenform erfolgt (Klasse `javax.swing.JTable`). Tabellen zur Darstellung von Policies beschreiben Joinpoints durch die Angabe deren Namen, Signaturen, Typen (zurzeit Attribute oder Methoden) sowie die Namen der Klassen, in denen diese Joinpoints deklariert sind. Keystore-Tabellen stellen die in den Keystores enthaltenen Zertifikate dar, indem sie alle Attribute des *Destinguished Name* (s. Kapitel 3.2) sowie den Alias des jeweiligen Zertifikates auflisten. Ein zusätzliches Tabellenattribut gibt bei Request Policies an, ob es sich um ein Zertifikat des Entwicklungsschlüssels handelt (s. Anforderung in Kap. 4.3.4).

Zur Steigerung der Benutzerfreundlichkeit werden Schlüsseltypen und Zugriffskategorien von Joinpoints durch farbigen Hintergrund der entsprechenden Tabellenzeilen dargestellt. Für die Zuordnung von Farben sind die im

Paket `gui` implementierten *Renderer*-Klassen verantwortlich, die das Layout der Tabellenzellen bestimmen. Renderer stützen sich auf Daten der in Kapitel 7.2.1 genannten Modell-Klassen, somit verfügt jede Modell-Klasse über eine eigene Renderer-Klasse.

Sind für einen Joinpoint Zugriffsrechte deklariert, so können diese in einem separaten Dialog-Fenster visualisiert werden (Klasse `PermissionDialog`). Die Darstellung von Berechtigungen erfolgt ebenfalls in Tabellenform. Die Tabellenzeilen enthalten jeweils den Alias des berechtigten Eigentümers sowie alle Callin- und Callout-Arten.

7.2.3 Die `Controller`-Klasse

Die Klasse `Controller` ist in erster Linie für die Bearbeitung der Benutzeraktionen zuständig und bildet somit die zentrale logische Einheit des Tools. Aufgrund der in Kapitel 7.1 aufgestellten Anforderungen gehören folgende Funktionen zur Schnittstelle der Controller-Klasse:

- `loadPackage(File file)` liest die JAR-Datei aus dem übergebenen `File`-Objekt. Auf der Basis der geladenen JAR-Datei wird ein neues Basispaket erstellt, dementsprechend erzeugt die Funktion unmittelbar nach dem Laden der Datei eine neue Keystore und ein neues `JoinPointPolicy`-Objekt, die die JAR-Datei zum Basispaket vervollständigen.

- `loadBasePackage(File file, boolean verify)` liest das Basispaket aus der übergebenen `File`-Instanz und verifiziert dieses, falls der Parameter `verify` gesetzt ist.

- `saveBasePackage(File file)` speichert das Basispaket im übergebenen `File`-Objekt.

- `newRequestPolicy()` erzeugt ein neues `RequestPolicy`-Objekt. Da jede Request Policy immer von einer Joinpoint Policy abgeleitet wird und die Angabe des Basispaket-Eigentümers erfordert, kann diese Funktion nur dann ausgeführt werden, wenn ein (Basis-)Paket und die Benutzer-Keystore bereits geladen worden sind.

- `loadRequestPolicy(File file)` liest die Request Policy aus dem übergebenen `File`-Objekt. Für diese Funktion gelten die gleichen Voraussetzungen wie beim Erzeugen einer neuen Request Policy.

- `saveRequestPolicy(File file)` speichert die Request Policy im angegebenen `File`-Objekt.

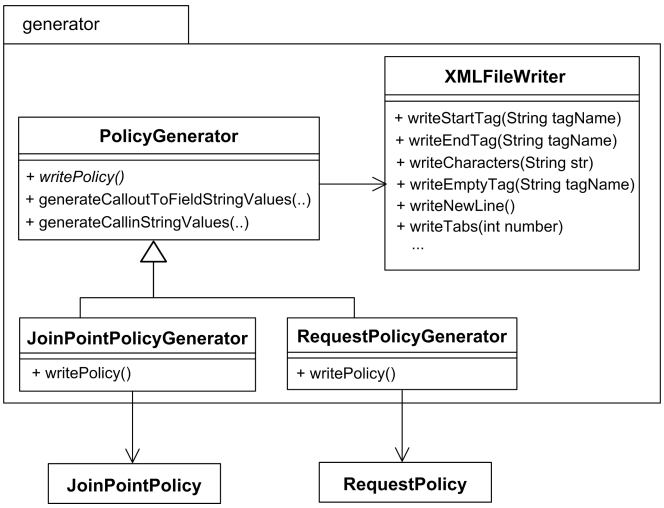

Abbildung 7.1: Das Paket `generator`

- `loadUserKeyStore(File file)` liest die Keystore aus dem übergebenen `File`-Objekt.

7.2.4 Generierung der Policy-Dateien

Wie bereits erwähnt, verwendet das Packaging Tool beim Einlesen von Policies und deren Überführung in die Objektstruktur die in Kapitel 6.1.1 vorgestellten Parser. Die Erstellung von XML-Dateien aus Policy-Objekten übernehmen die im Paket `generator` zur Verfügung stehenden Generator-Klassen. Ein Generator nimmt einen `OutputStream` entgegen und schreibt in diesen Stream die Werte des übergebenen Policy-Objektes unter Verwendung eines der in Kapitel 5 definierten Policy-Formate. Analog zu den Parser-Klassen ist jedem Policy-Typ (Joinpoint- und Request-Policy) eine Generator-Klasse zugeordnet. Abbildung 7.1 gibt einen Überblick über das Paket `generator` und die darin enthaltenen Klassen. Die generalisierte (abstrakte) Klasse `PolicyGenerator` implementiert Funktionen `generateCalloutToFieldStringValues` und `generateCallinStringValues`, die die Entity-Werte der XML-Elemente `<CalloutToField>` bzw. `<Callin>` generieren (s. A.1 und A.2 im Anhang). Die speziellen Klassen `JoinPointPolicyGenerator` und `RequestPolicyGenerator` implementieren jeweils die abstrakte Funktion `writePolicy()`, die zur Erstellung der XML-Dateien das Format des entsprechenden Policy-Typs verwendet. Die Eingaben für die Funktion (`OutputStream`-Instanz sowie das Policy-Objekt) werden bereits beim Instanziieren der Generator-Klasse an den Konstruktor übergeben. Die Klasse `XMLFileWriter` bietet eine etwas abstraktere Sicht auf die Struktur einer

XML-Datei und ist für die Konvertierung der für den XML-Markup reservierten Zeichen zuständig.

7.3 Verwaltung von Basispaketen

Dieser Abschnitt behandelt die Kernpunkte der Implementierungsphase. Im Mittelpunkt stehen die Verwaltung und Visualisierung der einzelnen Komponenten von Basispaketen. Da die Anwendungslogik überwiegend in der Klasse `Controller` realisiert ist, beschreiben die nachfolgenden Abschnitte die Implementierung des Controllers.

Wrapping von Zertifikaten

Die Klasse `Certificate` aus der Java API bietet keine Möglichkeit zur Abfrage von einzelnen Zertifikat-Attributen. Der gesamte String des *Destinguished Name* kann nur über den Aufruf der Methode `toString` bestimmt werden. Um auf die einzelnen Attribute zu kommen, muss der String geparst werden. Das Parsen des *Destinguished Name* übernimmt die im Paket `main` implementierte Klasse `CertificateWrapper`, die die `Certificate` Klasse 'einkapselt' und die Getter-Methoden für die einzelnen Attribute zur Verfügung stellt. Ferner enthält der Wrapper zusätzliche Keystore-bezogene Werte des korrespondierenden Zertifikats wie Alias oder Entry-Typ sowie ein Flag, das anzeigt, ob das aus der Basiskeystore entnommene Zertifikat als Träger des Entwicklungsschlüssels in der Joinpoint Policy aufgeführt ist. Die `Certificate`-Instanz und die zusätzlichen Daten werden an den Konstruktor des Wrappers übergeben, der dann das Parsen des Zertifikats ansteuert.

Scannen der Basispakete

Die in einer Joinpoint Policy enthaltenen Daten über Joinpoints reichen nicht aus, um die in Kapitel 7.1 aufgestellten Anforderungen zu erfüllen. Zum einen ist für die Administrierung erforderliche Menge von *denied* Joinpoints in der Policy-Datei nicht deklariert. Zum anderen ist die Beschreibung von Joinpoints in der Policy-Datei nicht detailliert genug, um die Verwaltung von Joinpoints flexibel zu gestalten. Das in Kapitel 5.2 definierte Format sieht beispielsweise die Angabe von Modifiers nicht vor; diese können jedoch die Interaktionen erheblich vereinfachen, indem der Benutzer die Möglichkeit bekommt, Mengen der zu bearbeitenden Joinpoints durch

Markierung nach deren Attributwerten, beispielsweise alle statischen (privaten, konstanten usw.) Members, zu definieren.

Die fehlenden Informationen werden durch das Scannen der im Basispaket befindlichen Classfiles gewonnen. Dies ist insbesondere bei der Erstellung von neuen Basispaketen notwendig. Das Scanning vervollständigt die Menge der in der Joinpoint Policy deklarierten Joinpoints um *denied* Joinpoints und initialisiert alle `JoinPointShadow`-Instanzen mit zusätzlichen Werten, die die Methode bzw. das Attribut charakterisieren:

- Sichtbarkeit des Members (`public`, `protected`, `private` oder `default`)
- Änderbarkeit (`final`)
- Zugehörigkeit des Members zur Klasse bzw. zum Objekt (`static`)

Das in Kapitel 6.1.2 beschriebene Source Scanning (Klasse `Scanner`) ist auf Bedürfnisse des JPAC ausgelegt, daher implementiert die `Controller`-Klasse eigene Funktionen zum Scannen von Basispaketen. Die Methode `scanJARFileForJoinPoints` ruft für jede Java-Klasse aus der in Form eines `File`-Objektes übergebenen JAR-Datei die Funktion `scanJavaClass` auf. Diese filtert die fürs Aspektweben irrelevanten Annotation-Klassen aus und leitet das Scannen jedes einzelnes Elementes der Klasse (Attribut bzw. Methode) durch den Aufruf der Methode `getScannedJoinPoint` ein. Das Classfile-Format definiert zwei Arten von speziellen Methoden:

- Methoden mit dem Namen `<init>` sind Konstruktoren der Klasse. Diese sollen beim Scannen genau so wie die anderen Methoden behandelt werden, denn die Sprachdefinition von ObjectTeams lässt indirekte Aufrufe von Konstruktoren der Basisklassen zu (§ 2.4.2(b) in [OTLD]). Solche Aufrufe ist ein Teil des Decapsulation-Konzeptes, dementsprechend müssen die Basiskonstruktoren durch Policies geschützt werden.
- Methoden mit dem Namen `<clinit>` initialisieren die statischen Attribute der Klasse und sind für Team- und Rollenklassen nicht sichtbar. Aus diesem Grund müssen `<clinit>`-Methoden beim Scannen von Classfiles ausgefiltert werden.

`getScannedJoinPoint` erzeugt für jeden Joinpoint ein `JoinPointShadow`-Objekt und initialisiert dieses mit eingelesenen Modifier-Werten. Die eingescannten Joinpoints werden in der Datenstruktur `scannedJoinPoints` vom Typ `ListValueHashMap<JoinPointShadow>` gespeichert, wobei deren Klassennamen als Schlüssel verwendet werden.

Authentisierung der Code-Eigentümer

Zum Nachweisen der eigenen Identität muss der Benutzer (Basiscode- oder Aspekteigentümer) eine Keystore beim Tool registrieren (laden), die seinen geheimen Schlüssel enthält. Die Authentisierung des Eigentümers ist erfolgreich, wenn beide folgende Punkte erfüllt sind:

1. Die Daten des Eigentümers sind in der Joinpoint Policy deklariert.
2. Der öffentliche Schlüssel des Eigentümers ist in der Keystore des Basispaketes enthalten.

Die Authentisierung des Eigentümers ist nur beim Verwalten des Basispaketes erforderlich; bei der Erstellung eines neuen Basispaketes oder einer neuen Request Policy muss der Benutzer den geheimen bzw. den öffentlichen Teil seines Schlüssels lediglich angeben. Dieser enthält Daten über die Identität des Eigentümers des Basispaketes bzw. der Request Policy.

Die Identitätsprüfung ist in der Methode `checkCodeOwnerAuth` realisiert. Die Funktion erwartet folgende Eingaben:

- Das `CodeOwner`-Objekt des Eigentümers, deren Identität geprüft werden soll
- Ein boolescher Wert, der anzeigt, ob die Person als Basiscode-Eigentümer vermutet wird.
- Ein Array von `Certificate`-Objekten, die beim Einlesen des Basispaketes als Signer-Zertifikate erkannt wurden.

Im Falle erfolgreicher Authentisierung registriert die Funktion das `CodeOwner`-Objekt sowie die beiden Schlüsselteile des entsprechenden Eigentümers global bei der Controller-Klasse und liefert `true` an den Aufrufer zurück. Es gibt zwei Situationen, in denen `checkCodeOwnerAuth` aufgerufen wird:

- Automatische Identitätsprüfung erfolgt beim Laden des Basispaketes, wenn eine Benutzer-Keystore mit mindestens einem geheimen Schlüssel beim Tool bereits registriert wurde.
- Zur expliziten Authentisierung implementiert die Klasse `Controller` die Methode `loginCert`, die das entsprechende `CertificateWrapper`-Objekt von der `GUI`-Klasse erwartet. Die Abmeldung des Eigentümers erfolgt über den Aufruf der Methode `logoutCert`.

Schlüsselverwaltung

Die zentrale Rolle in der Verwaltung von Schlüsseln spielt die Keystore des Basispaketes. Diese wird beim Laden der für das spätere Basispaket als Quelle dienenden JAR-Datei erzeugt und beim Speichern des Basispaketes in das Archiv unter dem Namen `jp.keystore` eingefügt. Die Schlüsselverwaltung wird durch drei Methoden der `Controller`-Klasse unterstützt. `addPublicCertToKeystore` fügt ein `Certificate`-Objekt in die übergebene Keystore ein, durch den Aufruf der Methode `removePublicCertFromKeystore` wird das zur übergebenen `CertificateWrapper`-Instanz korrespondierende Zertifikat aus der Keystore entfernt. Die Methode `getCertificateFromKeyStore` liefert das unter dem angegebenen Alias enthaltene Zertifikat in Form eines `CertificateWrapper`-Objektes zurück.

Die vorgestellten Funktionen erfordern die Angabe des Keystore-Passworts. Handelt es sich um eine Benutzer-Keystore, so wird der Benutzer beim Laden der Keystore über die grafische Benutzeroberfläche aufgefordert, das Passwort anzugeben. Die Basiskeystore enthält nur öffentliche Zertifikate, somit hat das Keystore-Passwort nur einen repräsentativen Charakter und kann veröffentlicht werden. Das Packaging Tool verwendet (und erwartet) beim Verwalten der Basiskeystores den Passwort-String mit dem Wert `objectteams`.

Da jede Key Entry mit dem geheimen Schlüssel innerhalb der Keystore durch ein zusätzliches Passwort geschützt ist, muss auch dieses zur Authentisierung vom Benutzer eingegeben werden. Somit sind zur Authentisierung des Eigentümers die Angabe von zwei Passwörtern erforderlich.

7.4 Signieren von Java-Archiven

Java API beinhaltet keine Infrastruktur zum programmtechnischen Signieren von Java-Archiven und bietet lediglich die Möglichkeit der Verifizierung von digital unterschriebenen Daten. Zur Signierung von Daten muss das in Kapitel 3.4 beschriebene externe Tool `jarsigner` verwendet werden. Dies stellt ein Hindernis bei der Realisierung der automatisierten Signierung von JAR-Dateien dar. Es existieren mehrere Ansätze, die das programmtechnische Signieren von JAR-Dateien ermöglichen.

In [Kr01] wird die Implementierung des Signiersalgorithmus' vorgestellt. D. Lyon erörtert in [Ly04] die Mängel dieses Ansatzes und schlägt seine eigene Lösung vor. Diese basiert auf der Verwendung des Signierungstools *JarSigner* aus der *Sun*-Bibliothek `sun.security.tools` über eine nach dem Design Pattern *Facade* entwickelte Schnittstelle. Da `sun.security.tools`

sich zwar im Installationsverzeichnis von *JDK (Java Development Kit)* befindet, aber keine öffentliche Bibliothek ist, wird in dieser Arbeit auf den Einsatz dieses Verfahrens verzichtet. Stattdessen wird das in Kapitel 3.4 vorgestellte Tool `jarsigner` aus dem JDK über die Methode `exec` der Laufzeitumgebung (Klasse `java.lang.Runtime`) aufgerufen.

Das Signieren von Basispaketen übernimmt die Funktion `signJar` in der Klasse `Controller`. Diese konstruiert einen Kommandozeilenstring anhand folgender Eingaben:

- Adresse der Keystore mit dem geheimen Schlüssel des Signers
- Passwort der Keystore
- Adresse der zu signierenden JAR-Datei
- Alias des Signers innerhalb der Keystore
- Passwort für die Key Entry, die den geheimen Schlüssel des Signers enthält

Die Funktion wird unmittelbar nach dem Speichern der Basispaketes aus der Funktion `saveBasePackage` aufgerufen. Die Eingaben werden zum String der Form

```
jarsigner -keystore <keystore> -storepass <storepassword>
      -keypass <keyentrypassword> <jarfile> <alias>
```

konkateniert, der dann an die Methode `Runtime.exec(String)` übergeben wird. Beim einem solchen Aufruf müssen allerdings zwei Punkte berücksichtigt werden:

- Die Passwörter bleiben im System in ungeschützter Form erhalten und sind somit leicht erreichbar
- Die Verfügbarkeit von JDK im System des Benutzers ist vorausgesetzt, wobei dessen `bin`-Verzeichnis im System-Classpath aufgeführt sein muss.

Hinsichtlich des ersten Punktes überlässt das Packaging Tool dem Eigentümer die Entscheidung, ob das Basispaket signiert oder als unsigniertes JAR gespeichert werden soll. Dafür wird ein entsprechender Dialog angezeigt, in dem der Eigentümer die Entscheidung treffen muss. Ist der Eigentümer sich sicher, dass die Voraussetzung des zweiten Punktes nicht erfüllt ist, so kann er auf das Unterzeichnen des Paketes mit Hilfe des Tools ebenfalls verzichten und dieses dann manuell signieren.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

In der Arbeit wurde ein alternativer Ansatz zu Osshers *confirmed join points* vorgestellt [Oss06] und durch Implementierung des Join Point Access Controllers (JPAC) als Erweiterung der ObjectTeams-Laufzeitumgebung praktisch umgesetzt. Das Konzept bietet sowohl den Entwicklern als auch den Benutzern die gewünschten Vorteile. Der Basiscode-Eigentümer erhält mehr Kontrolle über seinen Programmcode dank der Möglichkeit, seine Anwendung vor nicht-sanktionierten Adaptierungen durch ObjectTeams-Aspekte zu schützen. Der Aspekteigentümer gewinnt bei der Entwicklung des Aspektes an Flexibilität durch exklusive Zugriffsrechte auf eine feste Menge von verfügbaren Joinpoints. Dem Benutzer der durch ObjectTeams-Aspekte erweiterten Anwendung garantiert der vorgestellte Ansatz, dass zwischen den Eigentümer der Aspekte und dem Basiscode-Eigentümer eine Vertrauensbasis in Hinblick auf die Adaptierung besteht. Auch wenn nicht alle für eine flexible Zugriffskontrolle notwendigen Anforderungen umgesetzt wurden, ist der im Rahmen dieser Arbeit entwickelte Join Point Access Controller (JPAC) einsatzbereit.

Im theoretischen Teil der Arbeit wurde Kategorisierung der Joinpoints hinsichtlich der Zugriffsrechteverwaltung vorgenommen. Bei der Erarbeitung des Konzeptes wurde sowohl die Entwicklung als auch die Postentwicklungsphase einer ObjectTeams-Anwendung berücksichtigt, indem Benutzerkategorien, Schlüsselarten und Programmversionen sowie unterschiedliche Ausführungsmodi und Behandlungsarten von Zugriffsrechteverletzungen definiert wurden. All dies diente als Grundlage bei der Entwicklung des

Beschreibungsformates für Zugriffsrechte auf Joinpoints, das sich an die ObjectTeams-spezifische Aspektbindung orientiert.

Bei der Realisierung des Konzeptes wurden Standard-Sicherheitsmechanismen und Strategien aus der Programmiersprache Java verwendet. Dazu zählen Beschreibung von Zugriffsrechten in Form von Policy-Dateien sowie digitale Signierung und Verifizierung des Programmcodes, wofür die Kapselung der Anwendungskomponenten in Java-Archiven erforderlich ist. In der Entwurfsphase des JPAC wurde dessen funktionale Ähnlichkeit zum Sicherheitsmanager bzw. Access Controller aus dem Java-Sicherheitspaket beachtet, was auch zur strukturellen Ähnlichkeit des Join Point Access Controllers mit den beiden Sicherheitskomponenten führte.

JPAC nutzt die in den Bytecode-Attributen von Team- und Rollenklassen gespeicherten Informationen über die Callin- und Callout-Bindungen aus, um die Menge der im Webeprozess beteiligten Joinpoints zu ermitteln. Die Überprüfung findet beim Einlesen der Bytecode-Attribute unmittelbar vor dem Weben statt, sodass die Durchführung des Webeprozesses beim Auftreten einer Zugriffsrechteverletzung verhindert werden kann. Der Schutz vor Manipulationen des Programmcodes gehört nicht zu den Zielen der policy-basierten Zugriffskontrolle, die Verifizierung des Codes ermöglicht lediglich die Erkennung solcher Manipulationen. Das Konzept basiert darauf, dass die Überprüfung von Zugriffsrechten im System des Benutzers stattfindet, denn der Benutzer ist letztendlich derjenige, der die korrekte Programmausführung erwartet und dementsprechend die Vorgaben für die Verifizierung in Form von einzelnen Anwendungskomponenten sowie der Identität des Basiscode-Eigentümers liefert.

Zur Unterstützung der Basiscode-Eigentümer bei der Verwaltung von Zugriffsrechten sowie der Aspekteigentümer beim Formulieren von Zugriffsrechteanfragen wurde das Packaging Tool entwickelt. Dieses stellt den Eigentümern ein grafisches Interface zur Verfügung und automatisiert die Erstellung der Basispakete und Policy-Dateien.

Bewertung des policy-basierten Ansatzes

Deklaration von Zugriffsrechten außerhalb der Basisanwendung verleiht dem Ansatz einen bedeutsamen Vorteil im Vergleich zu den von H. Ossher in [Oss06] vorgestellten *confirmed join points*. Zum einen behält der Code der Basisanwendung die Java-Konformität, d. h. die Anwendung ist (soweit keine Erweiterung durch ObjectTeams-Aspekte stattfindet) innerhalb der Java-Standard-Laufzeitumgebung lauffähig. Zum anderen gestaltet sich die Verwaltung der Zugriffsrechte viel flexibler, da der Code der Basisanwendung bei der Rechtevergabe von den anfallenden Änderungen

unbetroffen bleibt.

Ein weiterer Vorteil des in dieser Arbeit erarbeiteten Konzeptes gegenüber *confirmed join points* wird durch die digitale Signierung des Programmcodes erreicht. Dies bietet einen besseren Schutz für die gekapselten Joinpoints, da neben den Zugriffsrechten auch die Identität des Aspekteigentümers überprüft wird. An der Stelle muss allerdings angemerkt werden, das die Zuweisung von Zugriffsrechten in [Oss06] sich auf die Pointcut-Deklarationen und somit auf die Namen der Aspektklassen und Methoden stützt, während beim policy-basierten Ansatz die Identität des Aspekteigentümers ausschlaggebend ist, d. h. die Zugriffsrechte werden nicht an Aspekte, sondern an deren Eigentümer vergeben.

Zu den Nachteilen der policy-basierten Zugriffskontrolle gehört in erster Linie der Ladezeit-Overhead, der beim Starten einer ObjectTeams-Anwendung entsteht. Dieser ergibt sich aus der Zeit, die zum Parsen der im Basispaket enthaltenen Joinpoint Policy sowie zum Scannen von Java-Archiven zur Ermittlung von Aspektklassen und Zertifikaten deren Eigentümer erforderlich ist. Insbesondere wenn die Basisanwendung und die Anzahl der bekannten Aspekteigentümer sehr groß sind, kann es zu erheblichen Verzögerungen beim Anwendungsstart kommen. Die Verzögerungen betreffen allerdings nur den Ladeprozess, da der JPAC zur Programmlaufzeit nicht aktiv ist.

8.2 Ausblick

Im Rahmen dieser Arbeit wurden die Grundanforderungen zur Kontrolle der Zugriffe auf Joinpoints realisiert, dennoch kann das vorgestellte Konzept erweitert und verfeinert werden. Die Notwendigkeit der Erweiterung entsteht vor allem durch die Evolution von ObjectTeams als Programmiersprache. Die nachfolgenden Abschnitte geben eine kurze Übersicht über konzeptuelle Erweiterungsmöglichkeiten und fassen offene bzw. nicht-realisierbare Punkte auf.

Erweiterung durch neue Joinpoint-Arten

Zurzeit werden in ObjectTeams drei Arten von Joinpoints unterstützt (s. Kap 5.1). Joinpoints sind Elemente im Programmcode und können somit durch Namen und weitere Eigenschaften der korrespondierenden Programmelemente beschrieben werden. Das Konzept der policy-basierten Zugriffskontrolle basiert auf der Beschreibung der Joinpoints in Form von Bytecode-Attributen. Wird die Sprache um eine neue Joinpoint-Art erwei-

tert, so müssen die Eigenschaften der entsprechenden Programmelemente in Classfiles angegeben werden. Dies kann über die Definition eines neuen bzw. über die Aktualisierung eines bereits bestehenden OT-Bytecode-Attributes erfolgen. Die Erweiterung des in dieser Arbeit vorgestellten Konzeptes schließt folgende Punkte mit ein:

- Aktualisierung des *Policy-Formats* geschieht durch die Definition eines neuen untergeordneten Elements der Entity `<JoinPointList>` in der Joinpoint Policy bzw. `<Requestlist>` und `<RelinquishmentList>` in der Request Policy.
- Erweiterung der *JPAC-Infrastruktur* durch Implementierung eines neuen Subtyps der Klasse `JoinPointAccessPermission` sowie durch Aktualisierung der Parser- und Generator-Klassen.

Je nachdem, auf welche Programmelemente sich die neue Joinpoint-Art bezieht, muss das im Packaging Tool realisierte *Classfile-Scannen* zur Bestimmung von potenziellen Joinpoints verfeinert werden. Momentan ist das Scannen des Bytecodes auf die Ermittlung der in der Klasse deklarierten Methoden und Attributen beschränkt. Die Hinzunahme von neuen Joinpoint-Arten könnte die Inspektion der Methodenimplementierungen zur Erkennung von bestimmten Butecode-Mustern notwendig machen.

Quantifiziertes Matching

Aktuell wird in ObjectTeams ausschließlich das explizite Matching unterstützt, d. h. jede Methodenbindung wird explizit im Quellcode des Aspektes angegeben. Demgegenüber steht das quantifizierte Matching, das die Deklaration von mehreren Bindungen mit Hilfe eines regulären Ausdrucks ermöglicht. Der Ansatz der policy-basierten Zugriffskontrolle ist mit dem quantifizierten Matching verträglich unter der Voraussetzung, dass das Ergebnis der Auswertung des regulären Ausdrucks in der Laufzeitumgebung zu Beginn des Webeprozesses verfügbar ist. Die Verträglichkeit ergibt sich aus der Tatsache, dass der Join Point Access Controller nicht die Deklaration von Bindungen, sondern deren durch den OT-Compiler ausgewertete Form (Informationen in den Bytecode-Attributen) bei der Realisierung der Zugriffskontrolle verwendet. Da der Weber jede konkrete Bindung einzeln verarbeitet, ist diese unmittelbar vor der Transformation des Bytecodes bekannt, sodass die `check`-Methode des JPAC aufgerufen werden kann. Also würde dieser Aufruf im Vergleich zur aktuellen Lösung zwar etwas später, aber trotzdem früh genug stattfinden, um eine eventuelle Zugriffsrechteverletzung erkennen und darauf reagieren zu können. Das Beschreibungsformat für Joinpoints in den Policy-Dateien bleibt dabei unverändert.

Verantwortung beim Fehlverhalten der Anwendung

Durch die Gewährleistung der Vertrauensbasis definiert das vorgestellte Konzept die Verantwortlichkeit der Eigentümer für die Korrektheit des Programmcodes. Demnach ist beim Ausführen der Anwendung in *Secure Mode* der Basiscode-Eigentümer für aufgetretene Fehler verantwortlich, in allen anderen Fällen kann auch der Eigentümer des Aspektes zur Verantwortung gezogen werden. In Wirklichkeit ist eine solche Verantwortungsteilung zu grob und kann in der Praxis nur bedingt angewandt werden. Bei der Bestätigung der angefragten Zugriffsrechte ist der Basiscode-Eigentümer normalerweise nicht in der Lage, eine Aussage über Zustände zu treffen, in die seine Anwendung durch den Aspekt versetzt werden kann.

Die Schuld am Fehlverhalten einer durch Aspekte erweiterten Anwendung lässt sich erst in einem konkret aufgetretenen Fall klären. Der Benutzer muss das inkorrekte Verhalten der Anwendung gegenüber beiden Eigentümern nachweisen können; die Umsetzung eines Nachweisverfahrens ist allerdings nicht trivial. Zum einen reicht der durch die Laufzeitumgebung gelieferte Stacktrace zur Beschreibung des Fehlers nicht aus, dafür sind Daten über die Konfiguration des Gesamtsystems sowie ein Protokoll über die Benutzeraktionen erforderlich, die zum Fehlverhalten geführt haben. Diese Informationen können zwar durch das Logging verfügbar gemacht werden, dieses wird aber in vielen Systemen aus unterschiedlichen Gründen nicht unterstützt. Zum anderen muss die Glaubwürdigkeit des Identitätsnachweises gewährleistet werden. In Java kann dies durch die digitale Signierung der relevanten Daten erreicht werden, als Signer muss jedoch eine Instanz fungieren, deren Beteiligung am Interessenkonflikt zwischen dem Benutzer und Code-Eigentümern ausgeschlossen ist. Da das System, in dem die Anwendung ausgeführt wird, sich unter der vollen Kontrolle des Benutzers (kann ebenfalls in den Interessenkonflikt involviert sein) befindet, kann die Echtheit der signierten Daten nicht gewährleistet werden. Somit ist eine flexiblere Verantwortlichkeitsteilung innerhalb des in dieser Arbeit vorgestellten Konzeptes nicht realisierbar.

Gültigkeit von Entwicklungsschlüsseln

Aus der Definition des Entwicklungsschlüssels (s. Kap. 4.3.4) wird deutlich, dass dieser in der Entwicklungsphase dem Aspekt mehr Freiheit bei Zugriffen auf Joinpoints einräumen kann als der Schlüssel des Aspekteigentümers (Release Key). Die im gleichen Kapitel angesprochene Gültigkeitseinschränkung bietet keinen effektiven Schutz vor Missbrauch der Entwicklungsschlüssel in den Release-Versionen, da diese durch die Verstellung des aktuellen Datums im System des Benutzers leicht umgangen

werden kann. Aus der Sicht des Benutzers entspricht die Menge der durch die Verwendung des Entwicklungsschlüssels zugeteilten Zugriffsrechte der Menge der Zugriffsrechte beim Ausführen der Anwendung im *Common Mode*. Die Gültigkeitseinschränkung wirkt sich dagegen benachteiligend auf den Eintwicklungsprozess des Aspektes aus, da beim Ablauf des Gültigkeitsdatums ein zusätzlicher Aufwand zum Erhalt eines neuen Entwicklungsschlüssels entsteht. Infolge dessen wurde auf die Realisierung der Gültigkeitseinschränkung im vorgestellten Konzept verzichtet.

Entkapselungskontrolle

Das in dieser Arbeit beschriebene Konzept realisiert die Kontrolle der Decapsulation nur im eingeschränkten Maße, indem die Zugriffe auf nicht sichtbare Klassenelemente global für die ganze Basisanwendung erlaubt bzw. unterbunden werden können. Die Entkapselung einer Klasse kann allerdings allein durch die Deklaration einer `playedBy`-Beziehung notwendig sein, auch wenn die korrespondierende Rollenklasse keine Bindungen an die Elemente der Basisklassen definiert. Der Verzicht auf die Deklaration von Basisklassen als Bezugspunkte für Adaptierung in den Policy-Dateien lässt eine flexiblere Verwaltung der zu entkapselnden Programmelemente nicht zu, da die Sichtbarkeit von Methoden und Attributen unter anderem auf die Sichtbarkeit der umschließenden Klasse bezogen ist. An dieser Stelle besteht Potenzial für die Erweiterung des Konzeptes, wodurch eine gezielte Decapsulation von einzelnen Klassenelementen ermöglicht werden kann. Dafür müsste die Deklaration von Zugriffsrechten auf Klassen in der Joinpoint Policy möglich sein, somit würden Basisklassen quasi eine neue Jonipoint-Art bilden. Demnach findet der Zugriff auf einen solchen Joinpoint durch die Deklaration der `playedBy`-Beziehung in der Rollenklasse statt. Die Gewährung der Zugriffsrechte auf eine bestimmte Methode macht dann die Vergabe der Rechte auf die Adaptierung der umschließenden Basisklasse erforderlich. Somit würde die Joinpoint-Policy-Struktur die Klassenstruktur der Basisanwendung nachbilden: die Methoden- und Attributenzugriffe sowie innere Klassen werden innerhalb der Klassendeklaration angegeben. Zur Unterstützung der Eigentümer bei der Erstellung von Policies könnte auch die Funktionalität des Packaging Tools um die Berechnung der für die Entkapselung relevanten Menge von Joinpoints erweitert werden.

Sicheres Linking

Die Zugriffskontrolle für Joinpoints zur Programmladezeit wirft die Frage auf, inwiefern der Linking-Prozess gegen Angriffe von außen geschützt ist.

Das Paketsystem von Java bietet keine ausreichende Sicherheit im Hinblick auf das Laden von Klassen. Ein Angreifer kann beispielsweise eine Klasse implementieren, die sich selbst für einen Teil der zu schützenden Anwendung ausgibt, um Zugriff auf private Klassen des Paketes zu bekommen. In Java ist dies durch die Angabe des vollständigen Pfades im Klassennamen möglich. Dies wurde beim Entwurf des Konzeptes in dieser Arbeit noch nicht berücksichtigt. Als Lösung zum Schutz der Anwendung gegen Angriffe solcher Art schlägt [BAF03] die Verwendung von Beschreibungsdateien in Java-Archiven vor. Das Java-Archiv wird demnach als Programm-Modul betrachtet, das eine Liste der darin enthaltenen Klassen deklariert. Die Liste kategorisiert die Klassen nach derer Sichtbarkeit außerhalb des Moduls durch explizite Angabe der vertraulichen Klassen sowie Klassen, die die Import- und Export-Schnittstellen des Moduls bilden. Die vollständige Deklaration des Modul-Inhaltes macht die Verhinderung des Linkings fremder Klassen mit der vertraulichen Anwendung möglich. Im policy-basierten Konzept, das im Rahmen dieser Arbeit ausgearbeitet wurde, verfügen die Basispakete bereits über eine Beschreibungsdatei (Joinpoint Policy), was dem Konzept Erweiterungsmöglichkeiten zur Realisierung des sicheren Linkings bietet.

Anhang A

Dateiformate für Policies

A.1 Joinpoint Policy (EBNF)

```
JoinPointPolicy = '<JoinPointPolicy>' BaseApplication BaseCodeOwner
                  AspectOwnerList JoinPointList '</JoinPointPolicy>';
BaseApplication = '<BaseApplication>' AppName AppVersion
                  '</BaseApplication>';
AppName         = '<AppName>' CHARS '</AppName>';
AppVersion      = '<AppVersion>' CHARS '</AppVersion>';
BaseCodeOwner   = '<BaseCodeOwner>' OwnerAlias OwnerName OrgUnit
                  Organization Locality State Country
                  '</BaseCodeOwner>';
OwnerAlias      = '<Alias>' CHARS '</Alias>';
OwnerName       = '<Name>' CHARS '</Name>';
OrgUnit         = '<OrganizationalUnit>' CHARS '</OrganizationalUnit>';
Organization    = '<Organization>' CHARS '</Organization>';
Locality        = '<Locality>' CHARS '</Locality>';
State           = '<State>' CHARS '</State>';
Country         = '<Country>' CHARS '</Country>';
AspectOwnerList = '<AspectOwnerList>' {AspectOwner}
                  '</AspectOwnerList>';
AspectOwner     = '<AspectOwner>' OwnerAlias OwnerName [OrgUnit]
                  Organization Locality [State] Country
                  ['<DevelopingKey/>'] '</AspectOwner>';
JoinPointList   = '<JoinPointList>' ['<SpecialAccessDenied/>']
                  {MethodAccess | FieldAccess} '</JoinPointList>';
MethodAccess    = '<MethodAccess>' MemberName MemberSignature Class
                  (MAPermList | '<Free/>') '</MethodAccess>';
MemberName      = '<Name>' CHARS '</Name>';
MemberSignature = '<Signature>' CHARS '</Signature>';
Class           = '<Class>' CHARS '</Class>';
MAPermList      = '<MAPermissionList>' {MAPerm} '</MAPermissionList>';
MAPerm          = '<MAPermission>' OAlias [Callin] [CalloutMeth]
                  '</MAPermission>';
OAlias          = '<OwnerAlias>' CHARS '</OwnerAlias>';
Callin          = '<Callin>' ['before'] ['after'] ['replace']
                  '</Callin>';
(*Entity-Werte 'before', 'after' und 'replace' werden
    durch ',' voneinander  getrennt*)
CallouthMeth    = '<CalloutToMethod/>'
```

```
FieldAccess    = '<FieldAccess>' MemberName MemberSignature Class
                 (FAPermList | '<Free/>') '</FieldAccess>';
FAPermList     = '<FAPermissionList>' {FAPerm} '</FAPermissionList>';
FAPerm         = '<FAPermission>' OAlias CalloutField
                 '</FAPermission>';
CalloutField   = '<CalloutToField>' ['get'] ['set']
                 '</CalloutToField>';
(*Entity-Werte 'get' und 'set' werden
    durch ',' voneinander  getrennt*)
CHARS          = {#x20 | #x5B | #x5D | [a-xA-X0-9] | [./;_]}
```

A.2 Request Policy (EBNF)

```
JoinPointRequest= '<JoinPointRequest>' BaseApplication BaseCodeOwner
                  AspectOwner RequestList RelinquishList
                  '</JoinPointRequest>';
BaseApplication = '<BaseApplication>' AppName AppVersion
                  '</BaseApplication>';
AppName         = '<AppName>' CHARS '</AppName>';
AppVersion      = '<AppVersion>' CHARS '</AppVersion>';
BaseCodeOwner   = '<BaseCodeOwner>' OwnerAlias OwnerName OrgUnit
                  Organization Locality State Country
                  '</BaseCodeOwner>';
OwnerAlias      = '<Alias>' CHARS '</Alias>';
OwnerName       = '<Name>' CHARS '</Name>';
OrgUnit         = '<OrganizationalUnit>' CHARS '</OrganizationalUnit>';
Organization    = '<Organization>' CHARS '</Organization>';
Locality        = '<Locality>' CHARS '</Locality>';
State           = '<State>' CHARS '</State>';
Country         = '<Country>' CHARS '</Country>';
AspectOwner     = '<AspectOwner>' OwnerAlias OwnerName [OrgUnit]
                  Organization Locality [State] Country
                  ['<DevelopingKey/>'] '</AspectOwner>'
RequestList     = '<RequestList>' {MethodAccess | FieldAccess}
                  '</RequestList>';
MethodAccess    = '<MethodAccess>' MemberName MemberSignature Class
                  [Callin] [CalloutMeth] '</MethodAccess>';
MemberName      = '<Name>' CHARS '</Name>';
MemberSignature = '<Signature>' CHARS '</Signature>';
Class           = '<Class>' CHARS '</Class>';
Callin          = '<Callin>' ['before'] ['after'] ['replace']
                  '</Callin>';
(*Entity-Werte 'before', 'after' und 'replace' werden
    durch ',' voneinander  getrennt*)
CallouthMeth    = '<CalloutToMethod/>'
FieldAccess     = '<FieldAccess>' MemberName MemberSignature Class
                  CalloutField '</FieldAccess>';
CalloutField    = '<CalloutToField>' ['get'] ['set']
                  '</CalloutToField>';
(*Entity-Werte 'get' und 'set' werden
    durch ',' voneinander  getrennt*)
RelinquishList  = '<RelinquishmentList>' {MethodAccess | FieldAccess}
                  '</RelinquishmentList>';
CHARS           = {#x20 | #x5B | #x5D | [a-xA-X0-9] | [./;_]}
```

Literaturverzeichnis

[AJ] AspectJ Homepage. `http://www.eclipse.org/aspectj/`

[BAF03] Lujo Bauer, Andrew W. Appel, Edward W. Felten. Mechanisms for secure modular programming in Java. Software-practice and experience, Vol. 33, pp. 461-480, 2003.

[Fl06] Michael Flüh. Schemaerhaltende Bytecodetransformationen zum Aspektweben zur Programmlaufzeit. Diplomarbeit. Technische Universität Berlin, 2006.

[FP97] Hannes Federrath, Andreas Pfitzmann. Bausteine zur Realisierung mehrseitiger Sicherheit. In: Günter Müller, Andreas Pfitzmann (Hrsg.): Mehrseitige Sicherheit in der Kommunikationstechnik, Addison-Wesley-Longman 1997, 83-104

[HC04] Cay S. Horstmann, Gary Cornell. Core Java 2, Volume II - Advanced Features, 7th Edition. Prentice Hall, 2004.

[Her02] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. *Proceedings of Net.ObjectDays*, Erfurt, 2002.

[Her06] Stephan Herrmann. Are Pointcuts a FirstClass Language Feature? Paper, Technische Universität Berlin, 2006.

[HHP06] Stephan Herrmann, Christine Hundt, Carsten Pfeiffer. Eclipse plugin adaptation with Equinox and ObjectTeams/Java. Paper, Technische Universität Berlin, 2006.

[Hu03] Christine Hundt. Bytecode-Transformation zur Laufzeitunterstützung von aspekt-orientierter Modularisierung mit ObjectTeams/Java. Diplomarbeit. Technische Universität Berlin, 2003.

[JAR] Sun Homepage. JAR File Specification. `http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html`

[JLS] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. Java Language Definition, 3rd Edition, Addison-Wesley. 2005.

[JVM] Tim Lindholm, Frank Yellin. The Java Virtual Machine Specification, 2nd Edition. Prentice Hall PTR, 1999.

[Kr01] Raffi Krikorian. Programmatically Signing JAR Files. 2001. `http://www.onjava.com/lpt/a/761`

[LSS03] David Larochelle, Karl Scheidt, Kevin Sullivan. Join Point Encapsulation. *Proceedings of the SPLAT workshop*, 2003.

[Ly04] Douglas Lyon. Project Initium: Programmatic Deployment. Journal Of Object Technology, ETH Zurich, Chair of Software Engineering, 2004.

[Oss06] Harold Ossher. Confirmed Join Points. *Proceedings of the SPLAT workshop at AOSD'06*, Bonn, 2006.

[OT] Object Teams Homepage. `http://www.objectteams.org`

[OTLD] Object Teams Language Definition. `http://www.objectteams.org/def/`

[PFS] Sun Homepage. Default Policy Implementation and Policy File Syntax. `http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html`

[Pf00] Andreas Pfitzmann. Sicherheit in Rechnernetzen: Mehrseitige Sicherheit in verteilten und durch verteilte Systeme. Skript zu den Vorlesungen Datensicherheit und Kryptographie, Dresden, 2000.

[Re79] Trygve Reenskaug. Models-Views-Controllers. Xerox PARC technical note, 1979.

[SHS] Secure Hash Standard, Federal Information Processing Standards Publication 180-1. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`

[SO01] Scott Oaks. Java Security, 2nd Edition. O'Reilly, 2001.

[St05] William Stallings. Cryptography and Network Security, 4th Edition, Chapt. 8-13, Prentice Hall, 2005.

[TvS] Andrew Tanenbaum, Marten van Steen. Verteilte Systeme, 1. Auflage, Kap. 8. Pearson Studium, 2003.

[Vld] Validome Homepage. W3C-standardkonformes Validierungstool für XML-Dokumente. `http://www.validome.org/xml/validate/`

[W3XML] W3C Hompage. Extensible Markup Language (XML) 1.0 (Fourth Edition). *http://www.w3.org/TR/2006/REC-xml-20060816/*

[WYY] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu. Finding Collisions in the Full SHA-1. Advances in Cryptology - Crypto'05, p. 17-36, 2005.

Erklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, 10. April 2007

Unterschrift: