Modellierung und Auswertung von aspektorientierten Pointcuts mit der UML und OCL

Robert Woll, 197801
Betreuerin: Dr. Katharina Mehner
Prüfer: Prof. Dr. Stefan Jähnichen
TU-Berlin, Fachgebiet Softwaretechnik

6. Juni 2006

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den

Unterschrift

Inhaltsverzeichnis

Kapitel 1

Einleitung

Software wird heutzutage immer komplexer und ihre Planung und Entwicklung immer aufwendiger. Die Modellierung von Software spielt dabei neben anderen wichtigen Einflussfaktoren wie eingesetzten Programmiersprachen, Hardware und Werkzeugunterstützung eine immer wichtigere Rolle bei der Lösung dieser Herausforderung. Ohne die Modellierung von Software wären viele Softwareprojekte kaum noch überschaubar. Dazu muss jedoch die eingesetzte Modellierungsnotation Möglichkeiten bieten, die Softwarearchitektur, die durch sie beschrieben wird und die implementiert wird oder bereits ist, geeignet darzustellen. Die Unified Modeling Language (UML) [34] ist eine Modellierungssprache, die sich dank ihrer umfangreichen Ausdrucksmöglichkeiten als die am häufigsten eingesetzte Modellierungsnotation für Softwaresysteme durchgesetzt hat. Sie stellt jedoch keine geeigneten Ausdrucksmöglichkeiten für die Abbildung aspektorientierter Techniken bereit und keiner der vorgestellten Lösungsansätze für dieses Problem hat sich bisher als Standard durchsetzen können. Es wird deshalb noch eine Erweiterung der UML um aspektorientierte Ansätze benötigt.

Die Aspektorientierung ist ein Ansatz, der die Strukturierung komplexer Softwaresysteme erleichtern soll. Eine bessere Trennung verschiedener Funktionalitäten im System durch Aspekte verspricht nicht nur die Übersichtlichkeit, sondern auch die Wartbarkeit und Wiederverwendbarkeit einzelner Bereiche des Systems zu verbessern. Da die Probleme, welcher sich die Aspektorientierung annimmt, nämlich die unklare Trennung verschiedener Funktionalitäten, auf der Implementierungsebene am deutlichsten werden, wurden aspektorientierte Ansätze auch zuerst in Form neuer Programmiersprachen oder Erweiterungen existierender Programmiersprachen vorgestellt. Der anfängliche Fokus auf die Implementierungsebene hatte zur Folge, dass aspektorientierte Ansätze in der Modellierung bisher noch nicht ausreichend

behandelt wurden. Während die Aspektorientierung also auf Programmiersprachenebene ausgereift ist, fehlt es an einer geeigneten Modellierungsnotation für aspektorientierte Systemelemente. Die Entwicklung einer geeigneten Modellierungsnotation für aspektorientierte Systemelemente ist deshalb ein Wegbereiter für den zukünftig verstärkten Einsatz aspektorientierter Ansätze im Rahmen komplexer Softewareprojekte.

Ein möglicher Grund dafür, dass sich keine der bisher vorgestellten Modellierungsnotationen für aspektorientierte Systeme durchsetzen konnte, sind ihre eingeschränkten Ausdrucksmöglichkeiten für die Modellierung von Pointcuts. Diese sind meist zu undetalliert, unterstützen nicht die Modellierung wiederverwendbarer Pointcuts oder enthalten bereits plattformspezifische Informationen, die ihre Implementierung nur auf ganz bestimmte aspektorientierte Programmiersprachen einschränkt.

Es wird deshalb eine Modellierungsnotation für den aspektorientierten Entwurf benötigt, welche die Modellierung von Pointcuts ohne Bindungen zu einem Basissystem erlaubt, so dass die modellierten Pointcuts wiederverwendbar sind. Eine solche Modellierungsnotation sollte ausserdem plattformunabhängig und die mit ihr modellierbaren Pointcuts sollten ausreichend detailliert ist, um aus ihr Implementationen in verschiedenen aspektorientierten Programmiersprachen abzuleiten. Wie ausserdem später noch begründet wird, ist es sinnvoll eine solche Modellierungsnotation zusätzlich durch eine Auswertungsfunktion zu unterstützen, welche die Auswertung von Pointcuts auf Modellebene erlaubt.

Um dieses Ziel zu erreichen, wird in dieser Arbeit ein UML-Profil für die *Unified Modeling Language (UML)* vorgestellt, welches einen Fokus auf der *Modellierung von Pointcuts* hat. Um zu ermitteln welche Eigenschaften von Pointcuts in das UML-Profil aufgenommen werden sollen, werden verschiedene existierende, aspektorientierte Programmiersprachen untersucht. Die Modellierungsnotation soll möglichst viele verschiedene Ansätze aus den verschiedenen Sprachen miteinander vereinen, so dass sich Modelle für möglichst viele Implementierungssprachen mit ihr bilden lassen, ohne jedoch gleichzeitig plattformspezifische Informationen einzusetzen.

Es wird ausserdem eine Auswertungsfunktion vorgestellt, mit der Pointcuts, die mithilfe des vorgestellten UML-Profils erstellt wurden, ausgewertet wer-

den können. Diese soll es ermöglichen, die Join-Point-Shadows eines in einem Modell ausgewählten Pointcuts zu ermitteln. Unter den Join-Point-Shadows werden hier statische Systemelemente verstanden, die zur Laufzeit in Join-Points involviert sein können. Dabei sollen aus den modellierten Pointcuts Ausdrücke in der *Object Constraint Language (OCL)* abgeleitet werden, die dann als Modellabfragen eingesetzt werden.

Um das UML-Profil und die Auswertungsfunktion in der Praxis einsetzen zu können, sollen diese für ein konkretes UML-CASE-Werkzeug implementiert werden. Ob die plattformunabhängige Modellierung wiederverwendbarer Pointcuts mithilfe des implementierten UML-Profils möglich ist, soll anhand einer Machbarkeistsstudie überprüft werden. Es soll auch überprüft werden, ob die OCL als Modellabfragesprache geeignet ist und in wiefern sie dadurch Anforderungen an eine Pointcutsprache erfüllt.

Eine interessante Nebenrolle in dieser Arbeit spielt der Ansatz der Model Driven Architecture (MDA) [25]. Wenn die plattformunabhängige Modellierung von Pointcuts mithilfe des vorgestellten UML-Profils möglich ist, so ermöglicht diese auch die Implementierung sogenannter Transformatoren, mithilfe derer sie auf plattformspezifische Modelle abgebildet und aus denen eventuell eine Implementierung generiert werden kann. Damit würde das vorgestellte UML-Profil noch eine weitere vielversprechende Eigenschaft besitzen, welche in den bisher vorgestellten Modellierungsnotationen für aspektorientierte Systeme nicht zu Verfügung steht.

Die vorliegende Arbeit ist grob in mehrere Abschnitte unterteilt, die im Folgenden vorgestellt werden: Im Kapitel „Aspektorientierte Modellierung“ werden Grundlagen der Modellierung aspektorientierter Systeme vorgestellt. Danach wird in dem Kapitel „Verwandte Arbeiten“ auf existierende Modellierungsnotationen für aspektorientierte Systeme und auf Auswertungsmechanismen für Pointcuts eingegangen. Auf die dort vorgestellten Arbeiten aufbauend, wird ein konkreter Handlungsbedarf formuliert, der die Motivation für die hier vorliegende Arbeit liefert. In dem darauf folgenden Kapitel „Die entwickelte Modellierungsnotation“ werden das UML-Profil und die dazugehörige Auswertungsfunktion für Pointcuts auf Modellebene vorgestellt, welche im Rahmen dieser Arbeit entstanden sind. Um zu überprüfen, ob die gesetzten Ziele mithilfe dieser enstandenen Werkzeuge erreicht werden können, wird im Kapitel „Machbarkeitsstudie“ erklärt, wie die Modellie-

rungsnotation und die Auswertungsfunktion für ein konkretes UML-CASE-Werkzeug implementiert wurden, um dann ein Beispielmodell mit diesen zu erstellen, welches dann ausgewertet wird. Schliesslich wird im letzten Kapitel noch mal eine Zusammenfassung aller Ergebnisse gegeben und es werden weiterführende Arbeiten für die vorgestellte Thematik vorgeschlagen.

Kapitel 2

Aspektorientierte Modellierung

Im Folgenden werden die Konzepte der Aspektorientierung, der UML-Modellierung und der MDA vorgestellt.

2.1 Aspektorientierung

Ein wichtiges Ziel bei der Entwicklung komplexer Softwaresysteme ist die Aufteilung eines Systems in verschiedene, physikalische Entitäten. Dabei wird ein System in logisch voneinander abgrenzbare Funktionalitäten, sogenannte Belange, unterteilt. Die Trennung von Belangen erleichtert das Verständnis des Systems und erleichtert gezielte Änderungen einzelner Belange und die Wiederverwendung einzelner Funktionalität in anderen Systemen. Als ein kleines Beispiel für verschiedene Belange stelle man sich eine einfache Webseite für ein Online-Geschäft vor. Man könnte hier von verschiedenen Belangen wie Benutzeroberfläche, Sicherheitsmechanismen, Datenspeicherung und Ausführung von Einkaufsvorgängen sprechen.

In der Praxis hat sich herausgestellt, dass es immer wieder Belange gibt, die in sehr vielen anderen Bereichen des Systems, in denen andere Belange behandelt werden, mitwirken, sich deshalb physikalisch schwer herauslösen lassen und so über das System verstreut bleiben. Solche sogenannten querschneidenden Belange werden in der Aspektorientierung mittels eines neuen Ansatzes physikalisch vom Rest des Systems getrennt und als Aspekte bezeichnet. Mittels sogenannter Webemechanismen wird die Funktionalität von Aspekten mit der Funktionalität der anderen Systembestandteile verknüpft. Dabei kann aspektorientierter Programmcode zur Kompilierzeit in den anderen Programmcode eingefügt werden, es können zur Laufzeit eines Programms Überwachungsmechanismen implementiert werden, die an den richtigen Stellen die Ausführung vom aspektorientiertem Code bewir-

ken, oder man kann bereits auf Modellebene aspektorientierte mit nicht-aspektorientierten Modellbestandteilen zusammen führen, so dass sich daraus ein nicht-aspektorientiertes Modell ergibt.

Im Rahmen dieser Arbeit wurden verschiedene aspektorientierte Programmiersprachen untersucht, die als Grundlage der hier vorgestellten Arbeit dienen. Untersucht wurden dabei die aspektorientierte Sprachen AspectJ [1, 21], ObjectTeams [16], JBossAOP [19], AspectWerkz [2], QSoul[14] und LogicAJ [36] und die Pointcutsprache Alpha [27].

Eine gute Einführung in die Konzepte der Aspektorientierung bieten Elrad et al. in [9]. Im Folgenden werden diese zentralen Konzepte und solche, die in der oben genannten Arbeit nicht genauer erläutert werden, vorgestellt:

- **Aspekt**: Ein Aspekt ist ein querschneidender Belang, der aus dem System herausgetrennt wurde und über Webemechanismen später wieder in das System eingefügt wird. Ein Aspekt kapselt Funktionalität und die dafür relevanten Daten. Ein Beispiel für einen Aspekt ist ein Logging-Modul, welches bei bestimmten Ereignissen in einem System Dateien erstellt, in denen der Programmablauf dokumentiert wird.

- **Advice**: Ein Advice ist ein Bestandteil eines Aspekts und beinhaltet die genaue Beschreibung eines Teils der Funktionalität des Aspekts. Die Gesamtheit aller Advice eines Aspekts machen seine gesamte Funktionalität aus. Advice können an Pointcuts gebunden sein, die festlegen, an welchen Stellen im System die Advice ausgeführt werden sollen. Ein Beispiel für einen Advice wäre eine Methode in einem Logging-Modul, die, immer wenn ein bestimmtes Ereignis im System auftritt, das Logging-Modul eine neue Datei schreiben lässt.

- **Crosscutting**: Crosscutting ist der englische Begriff für querschneidend. Ein querschneidender Belang (Crosscutting-Concern) steht nicht „neben“ anderen Belangen in einem System, sondern erstreckt sich „quer“ über mehrere andere Belange. So findet sich in anderen Belangen Funktionalität wieder, die nicht zu diesen gehört. Der querschneidende Belang „schneidet“ sozusagen eine logische Lücke in fremde Belange. Wird ein querschneidender Belang in Form eines Aspekts aus dem System herausgelöst, so muss er dennoch später wieder durch einen Webemechanismus in das System eingefügt werden. Die Beziehungen

zwischen einem Aspekt und den Stellen, an denen er eingewebt wird, nennt man querschneidende Beziehungen (Crosscutting-Relationships). Diese Beziehungen werden in UML-Diagrammen oft als Verbindungen zwischen einem Pointcut und seinen Join-Point-Shadows dargestellt.

- **Introduction**: Introductions sind Felder oder Operationen, die einem quergeschnittenen Element durch einen Aspekt hinzugefügt werden. Introductions werden in dieser Arbeit nicht weiter behandelt.

- **Join-Point**: Als Join-Point bezeichnet man Ereignisse in einem System, die einen Aspekt auslösen. Sie stehen in einer querschneidenden Beziehung zu einem Aspekt.

- **Join-Point-Shadow**: Als Join-Point-Shadow bezeichnet man die statischen Systemelemente, die in ein als Join-Point identifiziertes Ereignis involviert sind. Dieser Begriff wird auch in [23] verwendet. Wurde beispielsweise ein Methodenaufruf als Join-Point identifiziert, so kann die aufgerufene Methode in einem statischen UML-Diagramm wie dem Klassendiagramm als Join-Point-Shadow markiert werden.

- **Pointcut**: Ein Pointcut beschreibt, wann, wo und unter welchen Bedingungen ein Aspekt wirkt, falls einer seiner Advice mit diesem Pointcut verbunden ist. Jedes Ereignis in einem System, das auf diese Beschreibung passt, wird als Join-Point bezeichnet. Ein Pointcut formuliert dazu Eigenschaften, die zur Ermittlung von Join-Points dienen. In einigen Fällen werden Pointcuts allerdings auch vereinfacht als eine Aufzählung konkreter Ereignisse realisiert.

Das Paradigma der Aspektorientierung baut nicht direkt auf dem Paradigma der objektorientierten, der funktionalen oder der logischen Programmierung auf. Jedes dieser Paradigmen realisiert die Trennung von Belangen auf eine andere Weise. Die Aspektorientierung ist der Lösungsansatz dafür, wie Belange, die nicht klar aus anderen Belangen herausgetrennt werden können, behandelt werden sollen und kann in Kombination mit all den genannten Paradigmen verwendet werden. Die Aspektorientierung kann daher aber auch nicht alleine existieren, sondern ist immer abhängig von einem anderem Paradigma. In dieser Arbeit werden aspektorientierte Ansätze im Folgenden nur im Rahmen des objektorientierten Ansatzes betrachtet, da die Mehrheit

aller existierenden aspektorientierten Programmiersprachen auf objektorientierten Programmiersprachen aufbaut und die Mehrheit aller existierenden Modellierungsnotationen auf der objektorientierten UML aufbaut.

Die Konzepte der Aspektorientierung werden in aspektorientierten Programmiersprachen durch Erweiterungen existierender Programmiersprachen um neue Sprachelemente realisiert. Jede aspektorientierte Programmiersprache muss dabei neue Sprachelemente für die Formulierung von Pointcuts zur Verfügung stellen. Die Syntax für die Formulierung von Pointcuts und ihre Semantik wird im Folgenden als Pointcutsprache bezeichnet. Da es allerdings keine fundierte Definition des Begriffs Pointcutsprache gibt, wird die folgende Definition für die hier vorliegenden Arbeit angenommen:

> Mit einer Pointcutsprache werden Ereignisse in einem Softwaresystem beschrieben. Sie umfasst Ausdrucksmöglichkeiten für statische Informationen über die Elemente des Softwaresystems, die in dieses Ereignis involviert sind, und für dynamische Informationen über Charakteristika des Ereignisses selbst. Anhand der Informationen, die sie beschreibt, werden bestimmte Ereignisse in einem Softwaresystem als sogenannte Join-Points identifiziert. Die statischen Elemente des Softwaresystems, die an solchen Ereignissen beteiligt sind, werden Join-Point-Shadows genannt.

Im Folgenden werden nur die Ausdrucksmöglichkeiten für Pointcuts existierender aspektorientierter Programmiersprachen und nicht die der existierenden Modellierungsnotationen beschrieben. Der Grund dafür ist, dass die Ausdrucksmöglichkeiten der existierenden Programmiersprachen die Ausdrucksmöglichkeiten der Modellierungsnotationen bei weitem übertreffen. Auf die eingeschränkten Ausdrucksmöglichkeiten der existierenden Modellierungsnotationen wird im Kapitel „Verwandte Arbeiten“ genauer eingegangen.

Im vorangegangenen Kapitel wurde zwischen zwei Sorten von Pointcuts unterschieden, solchen Pointcuts, die eine Menge konkreter Ereignisse aufzählen, und solchen Pointcuts, die eine Menge von Eigenschaften von Ereignissen definieren. Bei letzteren muss also jedes Ereignis im System daraufhin überprüft werden, ob es die angegebenen Eigenschaften besitzt. Dahinter steckt ein Ansatz, den Filman in [12] als Quantifikation bezeichnet. Dieser besagt: Aspekte sollen bei all jenen Ereignissen wirken, die die in einem Pointcut

beschriebenen Eigenschaften besitzen. Der Vorteil, den man sich von diesem Ansatz verspricht, ist die Entkopplung der Beschreibung der Wirkungsstellen der Aspekte von einem bestimmten Basissystem. Indem man nicht bestimmte Ereignisse in einem bestimmten Basissystem auszählt, sondern nur Eigenschaften von relevanten Ereignissen, kann man diese Beschreibung mit jedem beliebigen Basissystem wiederverwenden.

Betrachtet man zunächst nur die Ansätze, bei denen die von einem Pointcut zu überwachenden Ereignisse konkret aufgezählt werden, so stellt sich die Frage, wie man diese eindeutig identifizieren kann. Dazu müssen alle Eigenschaften eines zu überwachenden Ereignisses, die zu seiner eindeutigen Identifizierung vonnöten sind, angegeben werden. Gibt man aber nicht Eigenschaften an, die nur das eine bestimmte Ereignis besitzt und kein anderes, so könnten auch andere Ereignisse auf diese Eigenschaftsbeschreibungen zutreffen. Im Folgenden werden zunächst typische Eigenschaften von Ereignissen aufgezählt, die bei der Auswertung von Pointcuts von Bedeutung sein können.

Die wichtigste Eigenschaft eines Ereignisses ist sein Ereignistyp. Mögliche Ereignistypen sind:

- Methodenaufruf,
- Methodenausführung,
- Konstruktoraufruf,
- Konstruktorausführung,
- Attributzugriff (lesen oder schreibend),
- das Eintreten von Exceptions oder Errors und
- das Erreichen einer bestimmten Anweisung innerhalb eines Ausführungsblocks (zum Beispiel Auswahlanweisungen oder Iterationsanweisungen).

Keine der im Rahmen dieser Arbeit untersuchten, aspektorientierten Programmiersprachen erlaubt die Verwendung aller der oben aufgeführten Ereignistypen. Insbesondere das Erreichen einer bestimmen Anweisung ist in keiner der Arbeiten als möglicher Ereignistyp vorgesehen. Die Sprache AspectJ

besitzt ausser für diesen Ereignistyp beispielsweise Ausdrucksmöglichkeiten für alle weiteren oben aufgeführten Ereignistypen.

Jedes Ereignis wird für gewöhnlich mit bestimmten statischen Elementen im System in Zusammenhang gebracht. Während der Ereignistyp eine dynamische Information darstellt, ist ein Modul eine statische, also eine strukturbezogene Information. So kann beispielsweise ausgedrückt werden, dass das Ereignis Konstruktorausführung nur in einer bestimmten Klasse betrachtet werden soll. Typische strukturbezogene Informationen sind Klassennamen oder Paketpfade. Soll ganz bewusst nicht ein ganz bestimmtes Ereigniss beschrieben werden, so werden für diese beiden Angaben oft Textmuster verwendet. In AspectJ könnte man beispielsweise schreiben „*com.programmer.project.* *Impl*“. Dieser Ausdruck ist ein Bestandteil eines AspectJ-Pointcutausdrucks und gibt vor, dass sich der gesuchte Ort in einem Unterpackage des Pfades „*com.programmer.project*“ und in einer Klasse, deren Name mit „*Impl*“ endet, befindet.

Je nachdem, um was für einen Ereignistyp es sich handelt, der von einem Pointcut überwacht wird, gibt es oft noch weitere relevante strukturelle Informationen, die für die Ermittlung von Ereignissen wichtig sein können. Einige Ereignisse treten an Attributen von Klassen auf, andere an Methoden oder Konstruktoren. Typische Informationen über diese betroffenen Elemente sind:

- Metatyp (zum Beispiel Methode, Konstruktor, Attribut, etc.),
- Bezeichner (Name einer Methode oder eines Attributs beispielsweise),
- Rückgabetyp, Parameter und auslösbare Exceptiontypen (wenn es sich zum Beispiel um eine Methode handelt),
- Typ (wenn es sich um ein Attribut handelt) und
- Modifier (private, public, protected, etc.).

Wie bei Pfadangaben oder Bezeichnern von Klassen, können auch hier wieder Muster für Bezeichner eines solchen Bestandteils eingesetzt werden (Beispiel: eine Methode mit dem Namen „*set**“). Zudem ist es wichtig zu verstehen, dass bestimmte strukturelle Informationen nicht für alle Ereignistypen relevant sind: die Festlegung von Parametertypen macht nur Sinn, wenn ein

Ereignistyp angegeben wurde, der mit einer Methode oder einem Konstruktor zu tun hat. Aus diesem Grund variiert die Auswahl möglicher, festlegbarer Eigenschaften in Abhängigkeit des Ereignistyps.

Mit den bisher vorgestellten Ausdrucksmöglichkeiten sind bisher nur Angaben über den Ereignistyp und die Eigenschaften des Elements, an dem ein Ereignis auftritt, möglich. Manchmal ist es aber auch nötig, strukturelle Nebenbedingungen zu formulieren. Ein Beispiel dafür wäre: „Betrachtet werden alle Konstruktorausführungen in der Klasse A, wenn sich im gleichen Paketpfad auch eine Klasse mit dem Namen B befindet.“. Die Existenz einer B genannten Klasse im gleichen Paketpfad ist eine strukturelle Nebenbedingung. Sie bezieht sich weder auf eine Eigenschaft der Klasse A noch auf Eigenschaften ihrer Konstruktoren, sondern auf davon „unabhängige“ Elemente im Modell. Die Möglichkeit solche strukturellen Nebenbedingungen zu formulieren wird beispielsweise von Sprachen wie Alpha, LogicAJ oder QSoul, die sich alle logischer Metaprogrammierung bedienen, ermöglicht, indem man zusätzliche Prädikate mit einem Pointcutausdruck logisch verundet.

Bisher wurden nur die Eigenschaften der Pointcuts selbst betrachtet. Noch mehr Ausdrucksmöglichkeiten erhält man durch Bindungen zwischen Pointcuts und anderen Elementen. Besondere Bedeutung kommt beispielsweise der Bindung zwischen einem Pointcut und den Advice, die mit ihm verknüpft sind, zu. Ein Advice kann mit mehreren Pointcuts verknüpft sein und ein Pointcut mit mehreren Advice. Die Grundidee dieser Bindung ist: der Pointcut gibt ein Signal an die Advice, die mit ihm verknüpft sind, weiter. Die Advice reagieren mit einer Aktion. In der Praxis wird das System zu dem Zeitpunkt, wo das von einem Pointcut betrachtete Ereignis gerade beginnen soll, unterbrochen. Für jeden Advice, der mit diesem Pointcut verknüpft ist, kann nun angegeben werden, ob er vor, nach oder anstatt des Ereignisses seine Aktion beginnen soll. Der Pointcut gibt jedem seiner Advice jeweils zu dem gewünschten Zeitpunkt das Signal für die Aktion weiter. Wenn die Advice ihre Aktionen ausgeführt haben, läuft das Basissystem weiter.

Soll ein Advice nur unter bestimmten Bedingungen, die nur zur Laufzeit bekannt sind, ein Signal von einem Pointcut erhalten, so kann die Bindung zwischen Pointcut und Advice zusätzlich um eine sogenannte Guardbedingung ergänzt werden. Die Guardbedingung enthält Aussagen über Werte von Systemelementen zur Laufzeit. Treffen diese Aussagen zu dem Zeitpunkt, an

dem der Pointcut sein Signal an den Advice weitergeben müsste, nicht zu, so wird das Senden des Signals unterdrückt. Während strukturelle Nebenbedingungen also zu dem Pointcut selbst gehören, gehören dynamische Nebenbedingunen zu der Bindung zwischen einem Pointcut und einem anderen Element. Strukturelle Nebenbedingungen können so meist nachgeahmt werden, indem alle Bindungen zwischen einem Pointcut und seinen Advice den gleichen Guard besitzen. Die aspektorientierte Programmiersprache Object-Teams erlaubt das Formulieren solcher Guards.

Schließlich sei die Bindung von Pointcut zu Pointcut erwähnt. Während ein Pointcut sich, wie bisher beschrieben, normalerweise auf nur ein Ereignis bezieht, ist es auch möglich, dass ein Pointcut erst dann sein Signal an die Advice weiterleitet, wenn mehrere Ereignisse aufgetreten sind. Ein Pointcut kann sich also auch auf mehrere Ereignisse beziehen und Relationen zwischen diesen betrachten. Diese Relationen beziehen sich zum Beispiel auf eine zeitliche Abfolge (Beispiel Präzendenzrelation: Ereignis B tritt nach Ereignis A auf) oder auf eine Verschachtelung im Programmablauf (Beispiel: Ereignis B tritt innerhalb von Ereignis A auf). Zeitliche Abfolgen sind beispielsweise möglich in Alpha oder dem von Walker [35] vorgestellten Ansatz. Die Verschachtelung von Ereignissen wird oft mithilfe des sogennanten cFlow Pointcuts ausgedrückt, der neben AspectJ auch in Sprachen wie Alpha benutzt werden kann. Verschachtelung ist nur bei Ereignissen möglich, die Programmabläufe enthalten (Methodenausführung oder Konstruktorausführung). Um einen Pointcut auf mehrere Ereignisse zu beziehen, ist es das Einfachste ihn aus Pointcuts zusammen zu setzen. Man unterscheidet einfach zwischen einfachen und zusammengesetzten Pointcuts. Herrmann schlägt in [18] vor, diese Begrifflichkeiten stärker voneinander zu trennen, indem einfache Pointcuts als „Join-Point-Queries“ und nur komplexe Pointcuts tatsächlich als „Pointcuts“ bezeichnet werden. Diese Bezeichnungen werden auch später in der Modellierungsnotation verwendet. Wenn dort also von „Join-Point-Queries“ die Rede ist, so sind damit einfache Pointcuts gemeint, und „Pointcuts“ entsprechen dann zusammengesetzten Pointcuts.

Bezieht sich ein Pointcut auf verschiedene Ereignisse, so ist es oft wünschenswert, sich in den Beschreibungen der Ereignisse auf gleiche Elemente zu beziehen. Beispiele hierfür sind Aussagen wie: „*Der Prozeduraufruf B findet nach dem Prozeduraufruf A statt und beide wurden aus derselben Methode einer anderen Klasse aus aufgerufen*“ oder „*Der Wert a muss während des*

Prozeduraufrufs B größer sein, als er während des Prozeduraufrufs von A war“. Damit Bezugswerte zweier verschiedener Ereignisse in Zusammenhang gesetzt werden können, bieten viele aspektorientierte Programmiersprachen geeignete Identifikationsmechanismen für solche Bezugswerte an. Sprachen wie Alpha, LogicAJ, QSoul bedienen sich der Konzepte logischer Metaprogrammiersprachen und erlauben damit logische Unifikation, um Bezugswerte, durch Variablen dargestellt, in verknüpften Ausdrücken wiederzuverwenden. In AspectJ werden parametrisierte Pointcuts und die Pointcuts this, target und args zur Verfügung gestellt, um Bezugswerte in verschiedenen Teilausdrücken wieder zu verwenden. Man nennt diesen Ansatz „Context Exposure“. Das Konzept der Context Exposure oder der Unifikation von Pointcutvariablen aus Sprachen wie LogicAJ oder Alpha ermöglicht nicht die Definition struktureller Nebenbedinungen wie sie oben beschrieben wurden. Es dient vielmehr der Definition von Bedingungen, die direkt für die beschriebenen Ereignisse und die darin involvierten statischen Systemelemente gelten sollen. Strukturelle Nebenbedingungen beziehen sich auf Systemelemente, die nicht direkt in eines dieser Ereignisse involviert sind.

Die Ausdrucksmöglichkeiten von Pointcuts lassen sich zusammengefasst anhand der folgenden Fragestellungen bestimmen:

- Welche Ereignistypen können formuliert werden,
- welche strukturellen Informationen können in Mustern festgelegt werden? Sind auch Nebenbedingungen möglich,
- können Bindungen zwischen Pointcuts und Advice mit vorher-, nachher- oder anstatt-Informationen und Guardbedingungen versehen werden und
- können Pointcuts mit Pointcuts verbunden werden, um komplexe Pointcuts zu formen?

Schliesslich sei angemerkt, dass viele aspektorientierte Sprachen eine separate Pointcutsprache enthalten. Da die meisten aspektorientierten Sprachen auf objektorientierten Sprachen aufbauen, wie beispielsweise AspectJ auf Java, wird eine Trennung zwischen einer Spracherweiterung der objektorientierten Sprache für die Programmierung von Aspekten und Advice und einer eigenen Pointcutsprache vollzogen.

2.2 Die Unified Modeling Language

Die Unified Modeling Language (UML) ist eine visuelle Modellierungssprache, die zur Darstellung komplexer Anwendungsstrukturen von der Object Management Group (OMG) [26] entwickelt wurde. Sie hat sich mittlerweile als industrieweiter Standard in der Softwareentwicklung durchgesetzt und wird mittlerweile auch in anderen Branchen zum Modellieren von Geschäftsprozessen und Datenstrukturen eingesetzt.

Die UML basiert auf der ebenfalls von der OMG vorgestellten Meta Object Facility (MOF) [24], einer Metabeschreibungssprache für andere Beschreibungssprachen, für die eine Vielzahl von Werkzeugen zur Speicherung oder zum Datenaustausch existieren. Neben einem theoretischen, syntaktischen Fundament besitzt die UML dadurch zudem auch eine Menge wichtiger unterstützender Funktionalität.

2.2.1 Diagrammtypen der UML

Die UML definiert insgesamt 13 verschiedene Diagrammtypen, die den folgenden Kategorien zugeordnet sind:

- Strukturdiagramme: Klassen-, Objekt-, Komponenten-, Kompositionsstruktur-, Paket- und Deploymentdiagramm,
- Verhaltensdiagramme: Anwendungsfall-, Aktivitäts- und Zustandsmaschinendiagramm und
- Interaktionsdiagramme: Sequenz-, Kommunikations-, Timing- und Interaktionsübersichtsdiagramme.

Die bekanntesten und am häufigsten eingesetzten Diagrammtypen sind das Klassen-, das Anwendungsfall-, das Sequenz-, das Aktivitäts- und das Zustandsmaschinendiagramm.

Vorgehensmodelle bestimmten, welche Diagramme in welcher Reihenfolge erstellt werden, in welchen Beziehungen sie zueinander stehen und welche Prozesse vonnöten sind, um die in den Diagrammen modellierten Informationen zu sammeln und zusammen zu stellen. Die UML selbst gibt kein Vorgehensmodell vor. Das wohl bekannteste Vorgehensmodell, das sich der UML

bedient, ist der Rational Unified Process (RUP) [31]. Die Wahl eines geeigneten Vorgehensmodell hängt stark von der Komplexität und dem Kontext des zu entwickelnden Softwaresystems ab.

Die UML selbst besteht zu großen Anteilen aus einer Auflistung verschiedener Metaklassen, die mit der MOF beschrieben sind. Diese Metaklassen sind in verschiedene Pakete sortiert. Ein wichtiger Bestandteil ist dabei ein Package namens „profile“, das festlegt, wie die UML durch sogenannte UML-Profile um zusätzliche Elemente erweitert werden kann. Neben der Definition der UML in Form von Metaklassen besteht die UML aus Vorgaben für die Darstellung der verschiedenen Diagrammtypen und definiert ausserdem noch die Object Constraint Language (OCL), die im folgenden Unterkapitel genauer beschrieben wird.

2.2.2 Die Object Constraint Language

Neben der Definition der verschiedenen UML-Diagrammtypen enthält die UML-Spezifikation zudem eine Definition der Object Constraint Language (OCL). Diese dient dazu, Modelle um Bedingungen zu ergänzen, die deren Verwendung einschränken. Dazu können für ein Modellelement eine beliebige Anzahl sogenannter Constraints (Einschränkungen) definiert werden. Ein Constraint legt Bedingungen für alle Instanzen des Modellelements fest, auf das der Constraint angewendet wird.

Constraints können eingeschränkte Wertebereiche für Eigenschaften von Modellelementen in Form sogenannter Invarianten festlegen. Ein Beispiel dafür wäre eine Metaklasse „EinsZuEinsAssoziation“, die von der Metaklasse „Assoziation“ erbt und für die immer gelten muss, dass ihre Kardinalitäten immer gleich 1 sind. Sehr oft werden OCL Constraints auch eingesetzt, um Vor- und Nachbedingungen für die Operationen von Klassen festzulegen.

Constraints können entweder so eingesetzt werden, dass sie die Bildung von Instanzen verhindern, die nicht mit den Constraints vereinbar sind, oder eine Instanz eines Modells kann im Nachhinein auf ihre Korrektheit in Bezug auf ihre Constraints hin validiert werden. Werden Constraints so verwendet, dass sie die Bildung unerlaubter Instanzen noch vor ihrer Bildung verhindern, so

kann man sie auch als „Live-Constraints“ bezeichnen [1]. Diese Funktionalität ist aber nur mit geeigneter Werkzeugunterstützung möglich. Üblich ist eigentlich die nachträgliche Validierung der Instanzen von Modellen.

Eine andere Einsatzmöglichkeit der OCL ist ihr Einsatz als Abfragesprache auf UML-Modellen wie von Hanenberg et al. in [32] beschrieben. Da die OCL dazu entworfen wurde, Wertebereiche von Eigenschaften zu formulieren, kann man OCL-Ausdrücke auch als Muster für die Ermittlung passender Modellelemente einsetzen. Das Modell wird dabei nach allen Elementen durchsucht, deren Eigenschaften zu den vorgegebenen Bedingungen passen. Die Ermittlung passender Elemente ist nicht Bestandteil der UML Spezifikation und kann, wie der Einsatz von Live-Constraints, nur durch Werkzeugunterstützung ermöglicht werden.

2.2.3 Erweiterungsmechanismen der UML

Die UML stellt für die Modellierung eine Vielzahl von Metaklassen zur Verfügung. Wird ein Modell entworfen, so ist jedes Modellelement eine Instanz einer dieser Metaklassen. Steht für eine bestimmte Information, die modelliert werden soll, keine geeignete Metaklasse zur Verfügung, so kann man die UML erweitern. Es gibt zwei Möglichkeiten dies zu tun: man kann auf der MOF aufbauend das UML-Metamodell selbst verändern oder man kann ein sogenanntes UML-Profil erstellen.

Die Erweiterung des Metamodells bedeutet, dass neue Metaklassen definiert werden. Diese neuen Metaklassen stehen dann beim Modellieren zur Verfügung. Für den Modellierer ist diese Art der Erweiterung die Einfachste. Sie hat jedoch einen Nachteil: da UML-Modelle meist mithilfe von UML-CASE-Werkzeugen erstellt werden, muss auch das verwendete UML-CASE-Werkzeug erweitert werden. Da ein UML-CASE-Werkzeug für die zur Verfügung stehenden Metaklassen der UML jeweils eine grafische Repräsentation, eine Repräsentation im Datenmodell, etc. besitzt, muss für ein neue Metaklasse häufig tief in die Implementation des verwendeten UML-CASE-Werkzeuges eingegriffen werden.

Die Erweiterung der UML durch ein UML-Profil funktioniert anders. Das

[1] Diese Bezeichnung wird im IBM Rational Software Architect verwendet

Metamodell wird nicht um neue Metaklassen erweitert, sondern um sogenannte Stereotypen. In der UML Superstructure Specification wird ein Stereotyp folgendermassen definiert:

> Stereotype is a kind of Class that extends Classes through Extensions. Just like a class, a stereotype may have properties, which may be referred to as tag definitions. When a stereotype is applied to a model element, the values of the properties may be referred to as tagged values. [...] A stereotype is a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends. Each stereotype may extend one or more classes through extensions as part of a profile. Similarly, a class may be extended by one or more stereotypes.

Diese Definition besagt, dass Stereotypen spezielle Klassen sind, die existierende Klassen durch einen sogenannten Erweiterungspunkt erweitern. Mit Klassen sind hier Metaklassen gemeint. Bekannte Beispiele für Instanzen von Metaklassen sind Klassen, Anwendungsfälle und Assoziationen. Stereotypen definieren neue Eigenschaften, die einer Instanz einer Metaklasse, die durch einen Stereotyp erweitert wird, hinzugefügt werden. Die Werte dieser Eigenschaften können für jede Instanz einer erweiterten Metaklasse separat festgelegt werden und können sich deswegen von Instanz zu Instanz unterscheiden. Die Werte dieser Eigenschaftsdefinition heißen Tagged Values. Für jeden Stereotyp muss ausserdem festgelegt werden, welche Metaklassen durch ihn erweitert werden können.

Weitere Eigenschaften von Stereotypen, die nicht aus der obigen Defition hervorgehen, werden im Folgenden aufgezählt:

- Die Definitionen der Eigenschaften eines Stereotypen dürfen nur eine Menge vorgegebener primitiver Datentypen oder Enmuerationstypen aus dem gleichen UML-Profil besitzen. Deshalb können Stereotypen nicht miteinander oder mit Instanzen von Metaklassen assoziiert werden.

- Instanzen von Metaklassen, denen Instanzen von Stereotypen zugewiesen wurden, werden in UML-Diagrammen dadurch gekennzeichnet, dass vor ihrem Namen der Name des erweiternden Stereotypen in eckigen Klammern steht (Beispiel: ≪StereotypeName≫ ElementName).

Der Vorteil von UML-Profilen ist, dass die Erweiterung von UML-CASE-Werkzeugen um Stereotypen einfacher ist als deren Erweiterung um neue Metaklassen. Der Nachteil ist, dass Stereotypen immer von Metaklassen abhängen und es nicht immer leicht fällt zu entscheiden, auf welche Metaklassen ein Stereotyp angewendet werden können soll. Problematisch ist auch die Einschränkung der Definitionen von Stereotypeigenschaften auf primitive Datentypen und Enumerationstypen.

Der Erweiterungsmechanismus durch UML-Profile wurde von der OMG genau für den Zweck entworfen, dass die Erweiterung von UML-CASE-Werkzeugen einfacher ist und dass Fehler, die durch den Eingriff in das UML-Metamodell entstehen, vermieden werden. In der Praxis ist deshalb die Erweiterung der UML durch UML-Profile gängiger als die Erweiterung des UML-Metamodells selbst.

2.3 Der Model Driven Architecture Ansatz

Die Modellierung eines Softwaresystems dient nicht der Dokumentation seiner Implementation, sondern bestimmt diese. Das Modell ist eine Art Bauzeichnung für das zu implementierende System. Je genauer es ist, desto mehr wird die Implementation auch dem Modell entsprechen. Ist das Modell genau genug, kann die Implementation sogar daraus generiert werden, so dass eine Implementierung der Programmbestandteile, die im Modell beschrieben sind, durch einen Programmierer überflüssig wird. Dadurch werden Kosten eingespart und eine höhere Qualität des Codes kann gewährleistet werden.

Ein Modell, das so genau ist, dass eine Implementation daraus generiert werden kann, muss allerdings immer auf eine Zielplattform hin abgestimmt sein. Unter einer Zielplattform wird in diesem Zusammenhang meistens eine konkrete Programmiersprache verstanden.

Der Ansatz der Model Driven Architecture (MDA) unterscheidet zwischen den verschiedenen Abstraktionsstufen von Modellen. Die sehr detaillierten Modelle, von denen gerade die Rede war, werden plattformabhängige oder plattformspezifische Modelle (PSM) genannt. Man stelle sich ein UML-Modell vor, in dem beispielsweise Java-spezifische Informationen enthalten sind. Abstrahiert man nun von allen plattformspezifischen Informationen in einem

Modell, so erhält man ein Modell, das auf verschiedene Zielplattformen hin wieder spezialisiert werden kann. Dieses Modell wird plattformunabhängiges Modell genannt (PIM, für platform-independent model). Ein typisches objektorientiertes Klassendiagramm ist beispielsweise mit den meisten objektorientierten Programmiersprachen wie Java oder C# kompatibel, enthält aber häufig nicht unbedingt genug Informationen, um daraus Code in einer konkreten Programmiersprache generieren zu können. So unterscheidet die UML in Klassendiagrammen beispielsweise nicht zwischen Konstruktoren und Methoden, sondern erlaubt nur die Modellierung von Operationen.

Das PIM und das PSM verwenden eine genaue, unterschiedlich detaillierte Syntax und Semantik, die befolgt werden muss, damit das Modell später für einen Computer aussagekräftig ist. Eine Abstraktionsstufe höher sieht die MDA deshalb noch das computerunabhängige Modell (CIM, für computer-independent model) vor. Dabei handelt es sich um die gröbste Stufe des Modells, ein Modell, das zumeist nur aus natürlichsprachlichen Bezeichnungen und nichtstandardisierten Schaubildern besteht.

Die MDA soll die Arbeit im Entwicklungsprozess erleichtern, indem sogenannte Transformatoren eingesetzt werden, Programme, die Modelle in feinere Modelle oder ausreichend feine Modelle in Implementationen umsetzen. Die Transformation von Modellen in feinere Modelle dient dazu PIMs in PSMs umzuwandeln. Die Umsetzung eines PSM in Programmcode wird auch als Tranformation bezeichnet. Die Trennung zwischen PIM und PSM wurde vorgesehen, um es Entwicklern zu ermöglichen, ein System zunächst plattformunabhängig zu entwerfen. Da die plattformspezfischen Anforderungen an ein System sich weitaus häufiger verändern als die Anforderungen an die im PIM modellierten fachlichen Geschäftsprozesse, bleibt das System dadurch anpassbarer und die im PIM modellierten Informationen können wiederverwendet werden.

Kapitel 3

Verwandte Arbeiten

3.1 Modellierugnsnotationen

Wie in der Motivation dieser Arbeit bereits angesprochen, wurde in den letzten Jahren eine Vielzahl neuer Ansätze vorgestellt, aspektorientierte Elemente mithilfe der UML darzustellen, wovon im Folgenden einige diskutiert werden sollen.

Unter den Autoren der hier vorgestellten Arbeiten herrscht durchweg grosse Einigkeit darüber, dass die Modellierung von Aspekten ein notwendiger Schritt ist, um den Einsatz aspektorientierter Techniken in der Praxis voranzutreiben. Elrad et al. haben dies in [8] schön beschrieben und gerechtfertigt:

> Recent work in AO design has demonstrated the need to deploy this technology early in the software life cycle in order to utilize the technology to its full potential. Once an initial decomposition of the problem identifies the software components and the corresponding aspectual properties that cut through these components, we would like to be able to express and model this initial decomposition in a formal way and carry it to the next phase of the development life cycle. Aspect-Oriented modeling should have the same relationship to AO Programming as Object-Oriented modeling has to OO Programming. Since UML is „the“ standard modeling language for OO it is natural to use it (and perhaps extend it) for AO. Most of the research in AO modeling has taken this approach and this was also agreed upon in the workshops proceedings of the first AOSD conference (2002) [...] we demonstrated that when aspects are identified at an early stage of the development life cycle, their design components are made more reusable, and automatic code generation is made

> possible for AOP systems.[7] [...] it is shown that applying AO design to the development life cycle can help maintain consistency between requirements, design, and code.

Elrad et al. argumentieren also unter anderem, dass die Modellierung von Aspekten deren Wiederverwendbarkeit und die Konsistenz zwischen Anforderungen, Entwurf und Code fördert und Code-Generierung ermöglicht.

Weitere Gemeinsamkeiten der untersuchten Ansätze seien im Folgenden genannt:

- Aspekte werden als spezielle Classifier [4, 22], oft sogar als spezielle Klassen [29, 37, 8, 15], angesehen. Werden UML-Profile als UML-Erweiterungsmechanismus verwendet, werden Aspekte in fast allen Fällen als spezielle Klassen angesehen. Die einzige Ausnahme bildet hier der Ansatz von Herrmann [17], in dem Aspekte spezielle Packages sind.

- Advice werden in der Regel als spezielle Operationen angesehen [29, 37, 8, 13]. In Verbindung mit dem Advice steht stets die Angabe, wie der Advice im Verhältnis zu dem Punkt, an dem er einsetzt, steht. Üblich sind hier die Angaben „before“, „after“, „replace“ oder „around“. In einem Fall wird ein Advice auch als spezieller Classifier betrachtet [22], in einem anderen Fall auch als Element [15]. Gemein ist allen diesen Ansätzen, dass Advice immer fest an einen Aspekt gebunden sind, entweder als Operation bzw. Methode oder über eine Assoziation.

Der Kern, in dem große Einigkeit herrscht, lautet zusammengefasst: Aspekte sind eigenständige Module, deren Funktionalität in speziellen Operationen, den Advice festgelegt ist.

Unterschiede zwischen den verschiedenen Ansätzen offenbaren sich, wenn es darum geht, wie Aspekte im Modell mit einem Basissystem in Verbindung gebracht werden sollen. In einigen Ansätzen werden querschneidende Beziehungen direkt, in Form von Verbindungen zwischen Advice oder Aspekten und statischen oder dynamischen Systemelementen, modelliert. In anderen Ansätzen wiederum werden Pointcuts in Form von Entitäten modelliert, die eine Menge von Informationen enthalten, die Ereignisse beschreiben. Werden querschneidende Beziehungen direkt modelliert, so werden sie meist in Klassendiagrammen als Assoziationen oder als Abhängigkeiten zwischen Aspekten und Elementen im Basissystem modelliert. In anderen Ansätzen werden

die querschneidenden Beziehungen in dynamischen Modelldiagrammen der UML modelliert. In der hier vorliegenden Arbeit wird die Modellierung von Aspekten und Pointcuts in dynamischen UML-Modellen und -Diagrammen jedoch nicht weiter betrachtet. Der Fokus liegt hier auf dem statischen UML-Klassendiagramm.

Gemein ist allen Ansätzen, bei denen die querschneidenden Beziehungen statisch im Klassendiagramm modelliert werden, dass querschneidende Beziehungen immer zwischen Aspekten und Klassen oder Packages aus dem Basissystem repräsentiert werden. Beispiele hierfür sind [29, 37, 8]. Die Annahme, die diesen Ansätzen zugrunde liegt, ist die, dass Aspekte oft als Erweiterung für ein konkretes Basissystem entworfen werden und man die beabsichtigten Stellen für die querschneidenden Beziehungen deshalb bereits kennt. Es liegt also nahe, diese Bindungen direkt zu modellieren. Viele aspektorientierte Programmiersprachen wie zum Beispiel AspectJ verfolgen jedoch den Ansatz, nicht querschneidende Beziehungen aufzuzählen, sondern Pointcuts als Beschreibungen von Ereignissen einzusetzen. Es werden also keine konkreten Bindungen aufgezählt, sondern die Eigenschaften formuliert, die ein Ereignis im System besitzen muss, damit es als Join-Point identifiziert werden kann und dessen Join-Point-Shadows zu einem Aspekt eine querschneidende Beziehung besitzen. Indem Pointcuts nur die gemeinsamen Eigenschaften aller möglichen quergeschnittenen Stellen beschreiben, sind sie generische Beschreibungen derselbigen. Dem bereits vielfach formulierten Bedarf nach generischen Beschreibungen der quergeschnittenen Stellen in einem System [12] werden die Ansätze, in denen die Beziehungen zu diesen Stellen direkt modelliert werden, folglich nicht gerecht. Zudem entsteht durch die fehlende Modellierung von Pointcuts eine Informationslücke zwischen Modell und Implementierung, die schnell zu Fehlern in der Implementation führen kann [28].

Um die Lücke zwischen Modell und Implementierung zu schliessen, sollten Pointcuts bereits auf Modellebene geeignet beschrieben werden. Betrachtet man nun all jene Ansätze, bei denen die Modellierung von Pointcuts als generische Ereignisbeschreibungen vorgesehen ist, so lassen sich zwei Gruppen von Ansätzen identifizieren: bei den einen enthalten die Pointcuts gar keine Informationen [37, 8, 11], bei den anderen enthalten Pointcuts einen Pointcutausdruck in einer bestimmten Pointcutsprache (üblicherweise der Pointcutsprache von AspectJ)[15, 22]. Die ersteren Ansätze weisen darauf hin, dass man Pointcuts gerne modellieren möchte, nur noch nicht genau weiß wie. Die

Letzteren sind pragmatische Lösungen, die jedoch das Problem mit sich bringen, dass daraus auch nur Implementierungen abgeleitet werden können, die sich der gleichen Pointcutsprache bedienen. Es fehlt also noch ein Ansatz bei dem Pointcuts sprachunabhängig modelliert werden können.

Schliesslich gibt es noch Unterschiede zwischen den bisher vorgestellten Ansätzen in Hinsicht auf die technische Realisierung der UML-Erweiterung. Hier ist die Wahl des Erweiterungsmechanismus für die UML ein wichtiges Charakteristikum. In einigen Ansätzen werden neue Metamodelle vorgestellt [4, 15], die auf der Meta Object Facility (MOF) basieren, in der verbleibenden Mehrzahl der Ansätze [8, 37, 13, 20] werden jedoch UML-Profile eingesetzt. Eine ausführliche Diskussion und Gegenüberstellung beider Ansätze steht dringend aus. Chavez et al. [4] argumentieren zwar beispielsweise, dass UML-Profile nicht geeignet sind, um die UML um UML-Elemente zu erweitern, können sich dabei jedoch nicht auf konkrete Studien oder Argumentationen stützen. Die Entscheidung für einen der beiden Erweiterungsmechanismen aufgrund von Ergebnissen verwandter Arbeiten kann also nicht erfolgen.

3.2 Auswertung von Pointcuts

Wie im vorangegangenen Kapitel beschrieben, fehlt noch ein Ansatz, bei dem Pointcuts als Ereignisbeschreibungen geeignet modelliert werden können. Würde man nun aber, wie in den bisher vorgestellten Ansätzen, die Modellierung von Pointcuts und die Modellierung querschneidender Beziehungen gleichzeitig zulassen, so könnten schnell Inkonsistenzen entstehen. Querschneidende Beziehungen lassen sich nämlich aus einem konkreten Pointcut in Kombination mit einem konkreten Basissystem ableiten. Durch die Modellierung von Pointcuts als Ereignisbeschreibungen wird die Modellierung querschneidender Beziehungen durch den Entwickler also überflüssig. Dennoch sollten aus diesen modellierten Pointcuts in Kombination mit einem Basissystem unbedingt die querschneidenden Beziehungen abgeleitet werden. Denn nur so ist es möglich, die Korrektheit der entworfenen Pointcuts zu überprüfen. Man erinnere sich: ein korrekter Pointcut beschreibt genau die beabsichtigten quergeschnittenen Stellen im System und keine anderen. Man braucht also einen Mechanismus, um querschneidende Beziehungen zwischen modellierten aspektorientierten Elementen, den Pointcuts, und Elementen aus dem Basissystem zu ermitteln.

Ein Ansatz, der in einigen Arbeiten vorgestellt wurde, ist der, Aspekte in einem separaten Aspektmodell zu entwerfen und dieses dann in ein objektorientiertes Modell eines konkreten Basissystems einzuweben. Im Vordergrund steht hierbei oft auch die Idee, durch das Einweben eines Aspektmodells in ein objektorientiertes Modell wieder ein rein objektorientiertes Ergebnismodell zu erhalten, weil man mit der Implementierung objektorientierter Entwürfe mehr Erfahrung hat und über bessere Werkzeugunterstützung verfügt. Auch hier geht es also darum, Verknüpfungspunkte zwischen modellierten aspektorientierten Elementen und Elementen aus dem Basissystem zu ermitteln, nämlich um die Webestellen zu ermitteln. Dieser Ansatz ist aber für die hier vorliegende Arbeit nur in Bezug auf den Mechanismus der Ermittlung von Webestellen von Interesse. In dieser Arbeit sollen Aspekte und Basissystem auch in einem gemeinsamen Modell modelliert werden können.

Jezequel et al. [20] stellen einen Ansatz vor, bei dem Contracts (aus dem Kontext Quality of Service) mithilfe eines UML-Profils als Aspekte modelliert werden. Die modellierten aspektorientierten Contracts können dann in ein objektorientiertes Modell eingewoben werden. Um die Verknüpfungspunkte ermitteln zu können, werden auf Profilebene Eigenschaften für die Stereotypen festgelegt, die als Abfragekriterien auf dem Modell das Basismodells eingesetzt werden können. Für die Ermittlung werden aus diesen Informationen OCL-Ausdrücke erstellt, die dann als Modellabfragen verwendet werden.

Auch Boellert et al. [3] stellen einen Ansatz vor, bei dem die OCL eine wichtige Rolle spielt. In dem vorgestellten Ansatz geht es zwar um den Hyperebenenansatz aus HyperUML und die Verschmelzung verschiedener Hyperebenen, doch das ähnelt konzeptuell dem Zusammenweben von Aspekten mit einem Basissystem. Die Bedingungen, die Verschmelzpunkte zwischen Hyperebenen zu erfüllen haben, sind hier mittels OCL-Ausdrücken festgelegt. Wie diese Punkte genau ermittelt werden, wird nicht genau beschrieben. Grundsätzlich gilt aber wie bei Jezequel et al.: die Eigenschaften der Verknüpfungspunkte werden mithilfe der OCL ausgedrückt.

Es stellt sich also heraus, dass die OCL scheinbar geeignet ist, um die Eigenschaften von Modellelementen aus dem Basissystem, die als Verknüpfungspunkte dienen sollen, zu beschreiben und auch um diese zu ermitteln. Jezequel et al. merken in ihrer Arbeit allerdings auch an, dass der Einsatz der

OCL als Abfragesprache problematisch ist, weil man das UML-Metamodell sehr genau kennen muss, um zu wissen wie man OCL-Ausdrücke richtig formuliert und weil diese Ausdrücke schnell sehr komplex werden können. Sie schlagen deshalb vor, das Schreiben dieser Ausdrücke nicht dem Entwickler zu überlassen, sondern diese aus Informationen aus dem aspektorientierten Modell abzuleiten. Stein et al. stellen in [32] ebenfalls fest, dass die OCL zwar sehr gut als Abfragesprache, gerade auch für den Zweck der Ermittlung von quergeschnittenen Elementen in einem Modell, geeignet ist, doch eine grafische UML-Notation zur Umschreibung von OCL-Abfrageausdrücken nötig ist.

3.3 Handlungsbedarf

Aus der Studie der existierenden Arbeiten zum Thema Modellierung von aspektorientierten Systemen geht hervor, dass es einer Modellierungsnotation bedarf, welche den plattformunabhängigen Entwurf von Pointcuts ermöglicht. Wie dort auch erwähnt wurde, muss in einer solchen Modellierungsnotation auf die expliziten Bindungen zwischen aspektorientierten Systemelementen und nicht-aspektorientierten Elementen verzichtet werden. Aus diesem Grund bedarf es ausserdem einer Auswertungsfunktion, welche es ermöglicht, solche Bindungen aus den modellierten Pointcuts im Modell abzuleiten, um die Korrektheit modellierter Pointcuts überprüfen zu können. Um schliesslich nachzuweisen, dass Pointcuts tatsächlich plattformunabhängig modelliert werden können und eine Auswertungsfunktion für die Modellebene möglich ist, muss zudem in einer Machbarkeitsstudie der praktische Einsatz der Modellierungsnotation und der Auswertungsfunktion im Rahmen eines konkreten UML-CASE-Werkzeuges ermöglicht werden. Diese Anforderungen werden im Folgenden genauer erläutert.

3.3.1 Erweiterte Pointcutmodellierung

Eine Modellierungsnotation ist nur dann nützlich, wenn es auch Implementierungssprachen gibt, mit denen mit ihr erstellte Modelle implementiert werden können. Da im Bereich der Aspektorientierung die Entwicklung von Implementierungssprachen fortgeschrittener ist als die Entwicklung von geeigneten Modellierungsnotationen, orientiert sich die hier vorgestellte Modellierungsnotation an den existierenden aspektorientierten Programmiersprachen. Sie

soll keine neuen Konzepte enthalten, sondern die existierenden Konzepte aus den Implementierungssprachen übernehmen. Dazu werden die im Rahmen dieser Arbeit untersuchten aspektorientierten Programmiersprachen als Grundlage herangezogen.

Die Modellierungsnotation soll zudem bei der Namensgebung so weit wie möglich auf wohldefinierte Begrifflichkeiten zurückgreifen. Herrmann untersucht und definiert in [18] einige zentrale Begriffe aus der Aspektorientierung. Da sich die Ergebnisse dieser Arbeit gut mit den Ergebnissen der hier vorliegenden Arbeit decken, soll die Modellierungsnotation auf die Begrifflichkeiten aus [18] zurückgreifen. Die in dieser Arbeit vorgestellten Konzepte werden dabei wie folgt auf die Begrifflichkeiten aus [18] abgebildet.

- Einfache Pointcuts heißen bei Herrmann „Join-Point-Queries“. Ein „Join-Point-Query“ ist eine Abfrage, die aus einem „Scope“ (Gültigkeitsbereich), einem „Join-Point-Kind“ und optionalen „Constraints“ besteht. Die Angabe eines Klassennamens und eines Paketpfades lässt sich auf das Konzept des Gültigkeitsbereichs abbilden. Die Angabe des Ereignistyps passt zu der Festlegung eines „Join-Point-Kinds“. Die Angabe weiterer struktureller Eigenschaften (wie zum Beispiel Parameter einer Methode) lässt sich auf das „Constraint“-Konzept [1] abbilden.

- Der Ansatz, bestimmte Ereignisse im System zum Zeitpunkt ihres Eintretens zu unterbrechen und an dieser Stelle (vor, nach oder anstattdessen) andere Aktionen „einzubauen“, wird bei Herrmann „Join-Point-Interception“ genannt.

- Der Ansatz, „Join-Point-Interception“ nur unter bestimmten Bedingungen, die zur Laufzeit ausgewertet werden, zu ermöglichen, wird bei Herrmann als „Event-Filtering“ bezeichnet.

- Das Konzept zusammengesetzter Pointcuts, also Pointcuts, die sich auf mehrere Ereignisse beziehen, wird bei Herrmann „Pointcut“ genannt.

Die folgenden Konzepte aus [18] sollen in die Modellierungsnotation aufgenommen werden:

- Join-Point-Query,

[1] Das Constraint Konzept aus [18] ist nicht mit dem Konzept der OCL-Constraints gleichzusetzen.

- Join-Point-Kind,
- Scope,
- (Static) Constraints,
- Join-Point-Interception,
- Event-Filter und
- Pointcut.

Neben den oben aufgezählten Konzepten sollen die folgenden weiteren Konzepte modellierbar sein:

- Join-Point-Shadow,
- Aspect,
- Advice,
- Team,
- Role,
- RoleMethod und
- Crosscutting-Relationship.

Die Konzepte Join-Point-Shadow und Crosscutting-Relationship werden für die zu erstellende Auswertungsfunktion benötigt. Die Konzepte Aspect, Advice, Team, Role und RoleMethod werden für die Modellierung aspektorientierter Module benötigt, die an Pointcuts gebunden werden.

Die Modellierungsnotation soll durch ein UML-Profil realisiert werden. Dadurch soll es einfacher sein, diese für UML-CASE-Werkzeuge zu implementieren.

Um bei der Arbeit mit dem UML-Profil den Entwurf semantisch falscher Modelle zu verhindern, soll das UML-Profil mit geeigneten OCL-Constraints versehen werden, die eine Validierung seiner Modelle ermöglicht.

Es soll möglich sein, mithilfe der Modellierungsnotation PIMs im Sinne des MDA-Ansatzes zu modellieren. Diese Anforderung deckt sich mit der Anforderung, Konzepte verschiedener aspektorientierter Programmiersprachen modellieren zu können.

3.3.2 Pointcutauswertung auf Modellebene

Unter der Auswertung von Pointcuts wird in dieser Arbeit verstanden, mithilfe der in den Pointcuts festgelegten Informationen Join-Point-Shadows im System zu ermitteln. Dazu sollen alle Werte aller Eigenschaften, die in einem modellierten Pointcut festgelegt sind und die sich auf statische Eigenschaften von Modellelementen beziehen, benutzt werden, um all jene Modellelemente zu ermitteln, die genau diese Werte aufweisen. Typische Werte sind hier Name oder Typ des Modellelements. Die ermittelten Elemente werden dann mit dem Join-Point-Stereotyp belegt.

Die Auswertungsfunktion muss festlegen, wie anhand von Modellinformationen Modellabfragen abgeleitet werden können, mithilfe derer Join-Point-Shadows ermittelt werden können. Dazu bedarf es eines Algorithmus und einer Abbildung von Informationen aus dem Modell auf eine konkrete Abfragesprache.

In [33] gehen Tarr et al. auf das Problem ein, welches entsteht wenn verschiedene Belange eines Systems in verschiedenen Diagrammen visualisiert werden. In komplexen Modellen wird es dadurch oft schwer, die Zusammnhänge zwischen diesen Belangen zu erkennen. Die Auswertungsfunktion von Pointcuts soll eben dieses Problem lösen.

3.3.3 Machbarkeitsstudie

Das UML-Profil und die Auswertungsfunktion sollen für ein konkretes UML-CASE-Werkzeug implementiert werden. Dieses UML-CASE-Werkzeug soll auch die Implementierung von Transformatoren ermöglichen und konform zu dem offiziellen UML-2-Standard sein. Es muss zudem einen visuellen UML-Editor zur Vefügung stellen, in dem aspektorientierte Modellelemente geeignet grafisch hervorgehoben werden können. Wenn möglich, soll das UML-CASE-Werkzeug während der Modellierung die mithilfe des UML-Profils er-

stellten Modelle auf ihre semantische Korrektheit hin überprüfen, indem die OCL-Constraints aus dem UML-Profil ausgewertet werden. Dies soll immer dann geschehen, wenn ein Modellelement mit einem Stereotypen aus dem UML-Profil belegt werden soll. Würde die Zuweisung des Stereotyps ein semantisch falsches Modell erzeugen, so soll diese Zuweisung verboten werden, so dass kein inkorrektes Modell zustande kommen kann.

Anschliessend soll mithilfe der Implementation des UML-Profils mit dem ausgewählten UML-CASE-Werkzeug ein Beispielmodell entworfen werden, in welchem mehrere Pointcuts enthalten sind. Es soll nachgewiesen werden, dass das Beispielmodell plattformunabhängig ist, also theoretisch in verschiedenen aspektorientierten Programmiersprachen implementiert werden kann. Ebenso soll aufgezeigt werden, dass es mithilfe der Auswertungsfunktion möglich ist, die quergeschnittenen nicht-aspektorientierten Modellelemente, welche durch die modellierten Pointcuts beschrieben werden, zu ermitteln und geeignet zu visualisieren.

Kapitel 4

Die entwickelte Modellierungsnotation

Die erstellte Modellierungsnotation besteht aus einem UML-Profil für die Modellierung von aspektorientierten Systemen und einem Konzept für eine Auswertungsfunktion für modellierte Pointcuts auf der Modellebene. Diese beiden Ergebnisse werden im Folgenden vorgestellt.

4.1 Das UML-Profil

Das hier beschriebene UML-Profil ist nur für die Verwendung in UML-Klassendiagrammen bestimmt. Aus Komplexitätsdründen wurde auf die Miteinbeziehung dynamischer UML-Diagrammtypen in die Modellierungsnotation verzichtet.

Das UML-Profil wird im Folgenden durch Profildiagramme dokumentiert, aus denen die Vererbungsbeziehungen zwischen den Stereotypen, ihre Extensionsklassen und ihre Tagged-Values hervorgehen. Kursiv geschriebene Stereotypnamen bedeuten, dass es sich bei diesen Stereotypen um abstrakte Stereotypen handelt, die nicht direkt verwendet werden können. Enumerationstypen aus dem UML-Profil und ihre Literale sind ebenfalls den Profildiagrammen zu entnehmen. Zu jedem Profildiagramm werden die Tagged-Values all jener Stereotypen erklärt, die mindestens eine Tagged-Value besitzen. Die Constraints aller angegebenen Stereotypen werden ebenfalls aufgezählt.

Es muss darauf hingewiesen wird, dass die UML keine Profildiagramme spezifiziert. Die Metaklassen eines UML-Metamodells können in einem UML-Klassendiagramm dargestellt werden. Ein UML-Profil enthält hingegen Stereotypen und bei diesen handelt es sich nicht um vollwertige Metaklassen.

Zudem definiert die UML keinen Metarelationstyp, der die Extensionsbeziehung zwischen einem Stereotyp und einer Metaklasse darstellen kann. Die vorgestellten Profildiagramme stellen aus den oben genannten Gründen Stereotypen wie Metaklassen, aber mit einem Stereotyp „Stereotyp“ dar. Metaklassen werden mit dem Stereotyp „Metaclass“ gekennzeichnet. Vererbungsbeziehungen zwischen einer Metaklasse und einem Stereotypen stellen Extensionsbeziehungen dar, die nicht mit den Vererbungsbeziehungen zwischen Stereotypen zu verwechseln sind.

Für jedes der in den Anforderungen an eine Modellierungsnotation genannten Konzepte gibt es Gegenstücke in dem hier vorgestellten UML-Profil. Das UML-Profil wird deshalb nicht in nur einem, sondern in mehreren Diagrammen dokumentiert, welche die verschiedenen konzeptionelle Bereiche des Modells darstellen und so die Übersicht erleichtern sollen. Zunächst wird die Realisierung der Konzepte Join-Point-Query und Join-Point-Kind in Abbildung 4. 1 dargestellt:

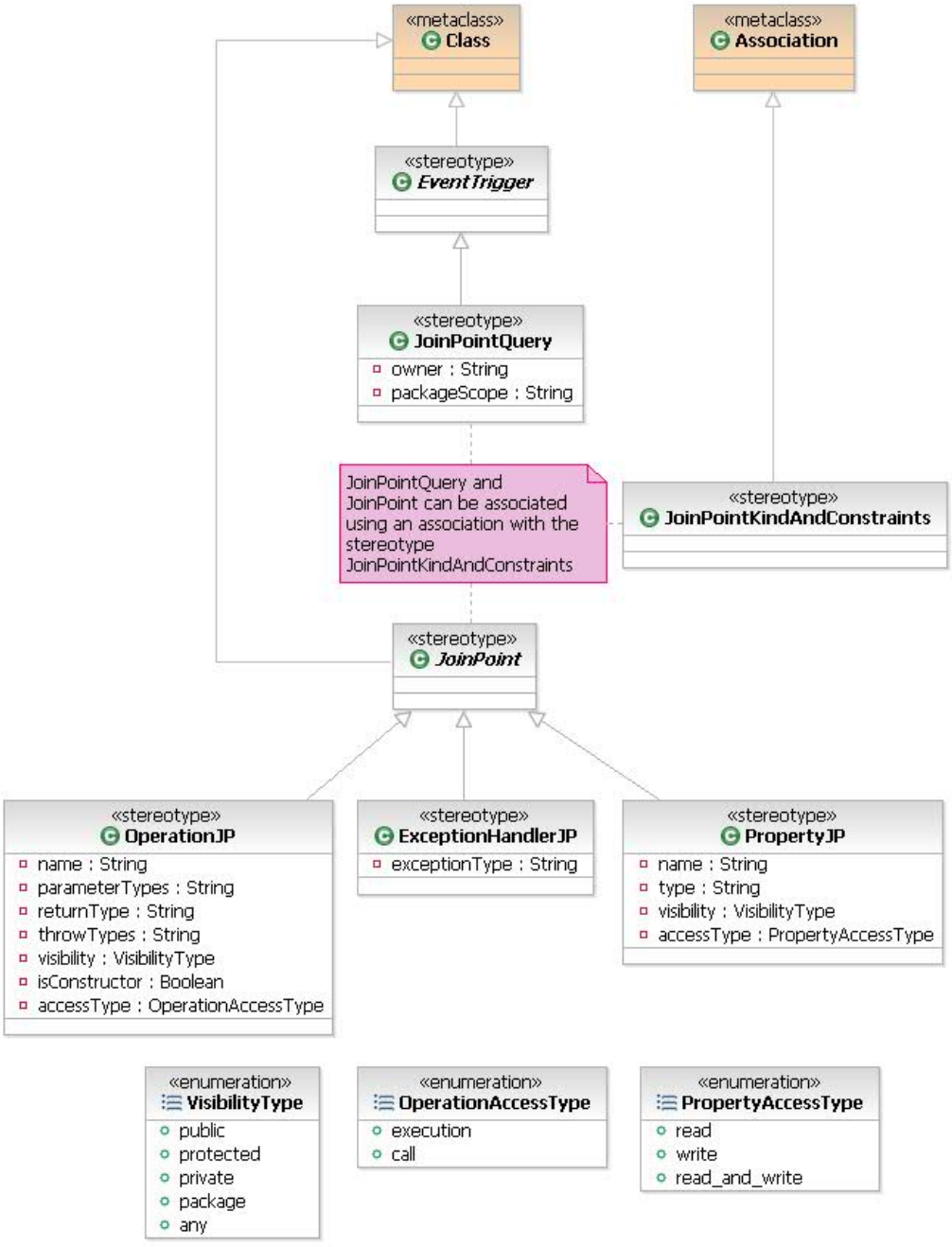

Abbildung 4.1: Join-Point-Queries und Join-Point-Kinds

Ein Join-Point-Query ist eine eigenständige Entität und erbt deswegen von dem Stereotyp „EventTrigger“, der später noch genauer erläutert wird, einen Extensionspunkt zu der Metaklasse „Class“. Er zeichnet sich durch eine Beschreibung seines Scopes und durch eine Angabe seines Join-Point-Kinds aus. Er beschreibt Ereignisse im System, die von diesem ermittelt werden sollen. Der Scope kann dabei die Ermittlung solcher Ereignisse auf bestimmte Systembereiche einschränken, indem Einschränkungen für Paketpfade und Namen in einem Join-Point-Query angegeben werden können. Der Stereotyp „JoinPointQuery“ besitzt deshalb die folgenden zwei Tagged-Values:

- owner -
  Typ: String
  Verwendet in Auswertungsfunktion: ja
  Semantik: Diese Tagged-Value wird als Namensmuster angegeben. Ein Ereignis an einem Feature kann nur dann als Join-Point identifiziert werden, wenn der Name der Klasse, in der sich dieses Feature befindet, zu diesem Namensmuster passt. Wird ein leerer String angegeben, so kann kein Join-Point identifiziert werden.

- packageScope -
  Typ: String[]
  Verwendet in Auswertungsfunktion: ja
  Semantik: Diese Tagged-Value wird als geordneter Array von Strings angegeben. Jede der Strings wird als Namensmuster verwendet. Da Packagepfade sich als geordnete Liste von Namen darstellen lassen, muss der Packagepfad der Klasse, in der sich ein Feature befindet, mindestens genau so lang sein wie der hier angegebene Array. Jeder Name eines Packages in diesem Pfad muss dem angegebenen Muster aus packageScope gleichen. Nur unter diesen Voraussetzungen kann ein Feature als Join-Point-Shadow identifiziert werden. Ist der String-Array leer, so kann kein Join-Point identifiziert werden.

Die genaue Beschreibung der betrachteten Ereignisse und der Eigenschaften, die diese besitzen, wird in dem Join-Point-Kind angegegen, welches einem Join-Point-Query immer zugeordnet sein muss. Um diese Zuordnung im Modell zu erreichen wird der Stereotyp „JoinPointKindAndConstraints“ zur

Verfügung gestellt, der einen Extensionspunkt zu der Metaklasse „Association“ besitzt. Für diesen Stereotyp gelten die folgenden Bedingungen:
Eine Assoziation mit dem Stereotyp „JoinPointKindAndConstraints“ darf nur Klassen mit dem Stereotyp „JoinPointQuery“ mit Klassen mit einem Stereotyp der von „JoinPoint“ erbt verbinden. Ausserdem darf es von einer Klasse mit dem Stereotyp „JoinPointQuery“ aus nur eine Assoziation mit dem Stereotyp „JoinPointKindAndConstraints“ geben.

Für das Konzept der Join-Point-Kinds stellt das UML-Profil den bereits erwähnten Stereotyp „JoinPoint“ zur Verfügung. Dieser kann ebenfalls als eigentständige Entität modelliert werden und besitzt deshalb einen Extensionspunkt zu der Metaklasse „Class“. Indem Join-Point-Kinds als eigenständige Entitäten modelliert werden können, können sie von verschiedenen Join-Point-Queries, die verschiedene Scopes vorgeben, wiederverwendet werden. Der Stereotyp „JoinPoint“ ist abstrakt und es gibt drei Stereotypen, die von diesem erben: „ExceptionHandlerJP“, „OperationJP“ und „PropertyJP“. Jeder dieser Join-Point-Kind-Typen bezieht sich auf verschiedene Ereignistypen und definiert die Constraints für diese. Dabei handelt es such um Angaben über die statischen Eigenschaften der Modellelemente, die mit dem Ereignis zu tun haben. Der Ereignistyp bestimmt sich über den Namen des Stereotyp.

Der Stereotyp „ExceptionHandlerJP“ beschreibt als Ereignis die Auslösung einer Exception. Dazu definiert er die folgende Tagged-Value:

- exceptionType -
  Typ: String
  Verwendet in Auswertungsfunktion: nein
  Semantik: Mit diesem Wert kann ein Namensmuster für alle Exceptions festgelegt werden, deren Auslösung als Join-Point identifiziert werden soll.

Der Stereotyp „OperationJP“ beschreibt als Ereignis den Aufruf einer Methode oder eines Konstruktors. Dazu definiert er die folgenden Tagged-Values:

- accessType -
  Typ: OperationAccessType
  Verwendet in Auswertungsfunktion: nein
  Semantik: Dieser Wert bestimmt, ob eine Operation zur Laufzeit zum Zeitpunkt ihres Aufrufs oder zum Zeitpunkt ihrer Ausführung als Join-Point identifiziert werden soll.

- isConstructor -
  Typ: Boolean
  Verwendet in Auswertungsfunktion: nein
  Semantik: Diese Tagged-Value kann verwendet werden, um in detaillierteren Modellen oder auf Implementierungsebene, auf der zwischen Methoden und Konstruktoren unterschieden werden kann, festzulegen, dass es sich um einen Konstruktor handeln muss. Ist der Wert dieser Tagged-Value false so bedeutet dies, dass es sich bei der beschriebenen Operation um eine Methode handeln soll.

- name -
  Typ: String
  Verwendet in Auswertungsfunktion: ja
  Semantik: Der Wert dieser Tagged-Value ist ein Namensmuster. Die Auswertungsfunktion kann nur dann Ereignis an einer Operation als Join-Point identifizieren, wenn dessen Name zu dem angegebenen Namensmuster passt. Wird ein leerer String angegeben, so kann kein Join-Point identifiziert werden.

- parameterTypes -
  Typ: String[]
  Verwendet in Auswertungsfunktion: ja
  Semantik: Diese Tagged-Value wird als geordneter Array von Namensmustern angegeben. Da die Parameterliste einer Operation geordnet ist, muss eine Operation die gleiche Anzahl von Parametern besitzen und die Namen der Parametertypen müssen zu den angegebenen Namensmustern passen, damit die Auswertungsfunktion ein Ereignis an dieser Operation als Join-Point identifizieren kann. Ist der String-Array

leer, so kann kein Join-Point identifiziert werden.

- returnType -
  Typ: String
  Verwendet in Auswertungsfunktion: ja
  Semantik: Diese Tagged-Value wird als Namensmuster angegeben. Der Name des Rückgabetyps einer Operation muss zu diesem Namensmuster passen, amit die Auswertungsfunktion ein Ereignis an dieser Operation als Join-Point identifizieren kann. Wird ein leerer String angegeben, so kann kein Join-Point identifiziert werden.

- throwTypes -
  Typ: String[]
  Verwendet in Auswertungsfunktion: ja
  Semantik: Diese Tagged-Value wird als geordneter Array von Namensmustern angegeben. Für jedes der angegebenen Namensmuster muss eine Operation mindestens einen Exceptiontyp in der Liste ihrer möglicherweise auslösenbaren Exceptions besitzen, dessen Name zu diesem Namensmuster passt, damit die Auswertungsfunktion ein Ereignis an dieser Operation als Join-Point identifizieren kann. Ist der String-Array leer, so kann kein Join-Point identifiziert werden.

- visibility -
  Typ: VisibilityType
  Verwendet in Auswertungsfunktion: ja
  Semantik: Eine Operation kann nur dann als Join-Point identifiziert werden, wenn ihre Visibility der in dieser Tagged-Value angegebenen Visibility entspricht. In dem Fall, in dem targetOperationVisibility den Wert any besitzt, wird diese Tagged-Value nicht ausgewertet.

Der Stereotyp „PropertyJP“ beschreibt als Ereignis den Zugriff auf ein Attribut einer Klasse. Dazu definiert er die folgenden Tagged-Values:

- accessType -
  Typ: PropertyAccessType
  Verwendet in Auswertungsfunktion: nein
  Semantik: Dieser Wert bestimmt, ob nur lesende, schreibende oder beide Arten von Zugriffen auf eine Property als Join-Point identifiziert werden können.

- name -
  Typ: String
  Verwendet in Auswertungsfunktion: ja
  Semantik: Der Wert dieser Tagged-Value ist ein Namensmuster. Die Auswertungsfunktion kann nur dann ein Ereignis an einer Property als Join-Point identifzieren, wenn dessen Name zu dem angegebenen Namensmuster passt. Wird ein leerer String angegeben, so kann kein Join-Point identifiziert werden.

- type -
  Typ: String
  Verwendet in Auswertungsfunktion: ja
  Semantik: Der Wert dieser Tagged-Value ist ein Namensmuster. Die Auswertungsfunktion kann nur dann ein Ereignis an einer Property als Join-Point identifizieren, wenn der Name des Typs dieser Property zu dem angegebenen Namensmuster passt. Wird ein leerer String angegeben, so kann kein Join-Point identifiziert werden.

- visibility -
  Typ: VisibilityType
  Verwendet in Auswertungsfunktion: ja
  Semantik: Ein Ereignis an einer Property kann nur dann als Join-Point identifiziert werden, wenn ihre Visibility der in dieser Tagged-Value angegebenen Visibility entspricht. In dem Fall, in dem propertyVisibility den Wert any besitzt, wird diese Tagged-Value nicht ausgewertet.

Für die Angabe einiger Tagged-Values definiert das UML-Profil zudem die

Enumerationstypen „OperationAccessType“, „PropertyAccessType“ und „VisibilityType“. Der Stereotyp „EventTrigger“ wird in der Beschreibung von Abbildung 4.2 näher erläutert. Die Realisierung der Konzepte Join-Point-Interception, Event-Filter und Pointcut ist dort zu sehen.

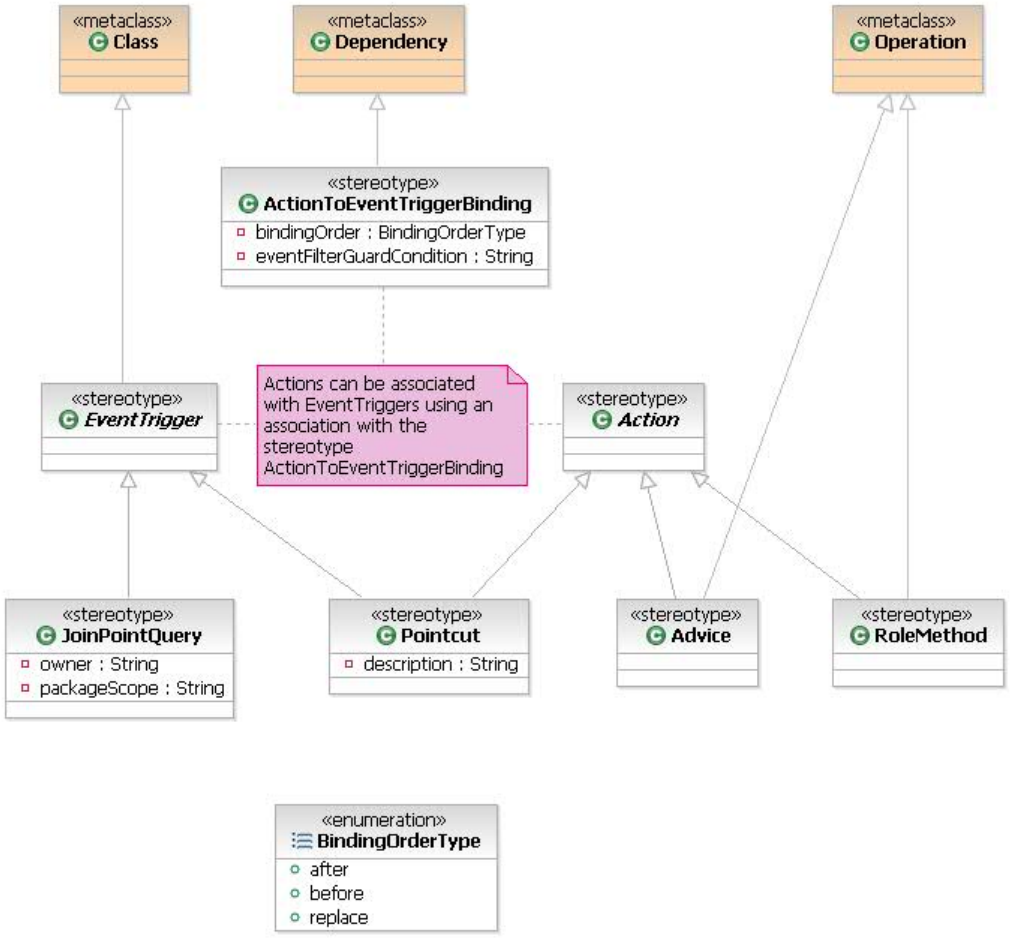

Abbildung 4.2: Join-Point-Interception, Event-Filter und Pointcut

Für das Konzept der Join-Point-Interception gibt es keine gleichnamige Metaklasse. Die Stereotypen „EventTrigger“, „Action“ und „ActionToEventTriggerBinding“ realisieren dieses Konzept gemeinsam. Join-Point-Interception bedeutet, dass zu dem Zeitpunkt des Eintretens eines Ereignisses, das als

Join-Point identifiziert wurde, eine Aktion ausgelöst wird. Die Identifikation von Join-Points wird von Event-Triggern übernommen. Die Aktionen, die durch die Identifikation eines Ereignisses ausgelöst werden können, heißen Actions. Join-Point-Interception geschieht immer dann, wenn eine Action über ein Binding an einen Event-Trigger gebunden ist und dieser Event-Trigger einen Join-Point identifiziert.

Event-Trigger sind eigenständige Entitäten. Deshalb besitzt der Stereotyp „EventTrigger“ auch einen Extensionspunkt zu der Metaklasse „Class“. Da dieser Stereotyp abstrakt ist, kann man nur die beiden von ihm erbenden Stereotypen „JoinPointQuery“ und „Pointcuts“ verwenden. Join-Point-Queries identifizieren, wie bereits erklärt, nur ein Ereignis. Pointcuts identifizieren das Eintreten eines Ereignisses in Abhängigkeit anderer zuvor eingetretener Ereignisse und beziehen sich somit auf mehrere Ereignisse. Jedes mal, wenn eines der relevanten Ereignisse auftritt, wird überprüft ob alle relevanten Ereignisse in einer bestimmten Konstellation aufgetreten sind. Das letzte aller relevanten Ereignisse wird als Join-Point identifiziert, wenn alle vorangegangenen Ereignisse in der vorausgesetzten Konstellation aufgetreten sind. Diese Überprüfung wird als Aktion interpretiert, weshalb ein Pointcut gleichzeitig auch eine Aktion ist. Das Eintreten der relevanten Ereignisse erkennt ein Pointcut, indem er mit einem Binding an andere Event-Trigger gebunden wird. Um den Zusammenhang der zwischen den einzelnen Ereignissen, die relevant für einen Pointcut sind, besteht, besitzt der Stereotyp „Pointcut“ die folgende Tagged-Value:

- description -
  OwningStereotype: Pointcut
  Typ: String
  Verwendet in Auswertungsfunktion: nein
  Semantik: Da ein Pointcut mehrere Ereignisse in einer bestimmten Konstellation identifizieren soll, soll der Zusammenhang, der zwischen diesen Ereignisses besteht, modelliert werden können. Die verschiedenen Ereignisse werden durch die Bindings eines Pointcuts zu anderen Event-Trigger bestimmt, die Zusammenhänge zwischen dieser werden natürlichsprachlich in der Tagged-Value description beschrieben.

Der Stereotyp „Action“ beschreibt Aktionen, die durch Event-Trigger ausgelöst werden. Da er abstrakt ist, können nur die von ihm erbenden Stereotypen Pointcut“, „Advice“ und „RoleMethod“ verwendet werden. Die Stereotypen „Advice“ und „RoleMethod“ werden an späterer Stelle beschrieben.

Der Stereotyp „ActionToAdviceTriggerBinding“ kann dazu eingesetzt werden, Event-Trigger mit Actions über eine Dependency zu verbinden. Für jedes Binding zwischen einem Event-Trigger und einer Action kann angegeben werden, wie die Aktion in dem Fall, dass der Event-Trigger einen Join-Point identifiziert hat, ausgelöst werden soll: vor, nach oder anstatt dem als Join-Point identifizierten Ereignis. Ausserdem ist das Konzept des Event-Filters in Form der Tagged-Value „eventFilterGuardCondition“ in dem Stereotyp „ActionToAdviceTriggerBinding“ realisiert. Der Stereotyp „ActionToEventTriggerBinding“ besitzt die folgenden Tagged-Values:

- bindingOrder -
  OwningStereotype: ActionToEventTriggerBinding
  Typ: BindingOrderType
  Verwendet in Auswertungsfunktion: nein
  Semantik: Dieser Wert bestimmt, wann zur Laufzeit eine Action, die an einen Event-Trigger gebunden ist, ausgeführt wird, wenn ein passender Join-Point für den Event-Trigger gefunden wird.

- eventFilterGuardCondition -
  OwningStereotype: ActionToEventTriggerBinding
  Typ: String
  Verwendet in Auswertungsfunktion: nein
  Semantik: Ein Event-Filter kann zur Geltung kommen, wenn zur Laufzeit durch einen Event-Trigger ein Ereignis als Join-Point identifiziert wurde und eine Action über ein Binding mit diesem Event-Trigger verbunden ist. Die eventFilterGuardCondition kann ein Ausdruck in einer beliebigen Beschreibungssprache oder in natürlicher Sprache sein, der wiedergibt, welche Eigenschaften das Ereignis aufweisen muss, damit die Action ausgelöst wird.

Für den Stereotyp „ActionToEventTriggerBinding“ gilt ausserdem die folgende Bedingung:

- Eine Dependency mit dem Stereotyp „ActionToEventTriggerBinding“ darf nur Klassen mit einem Stereotyp der von „Action“ erbt mit Klassen mit einem Stereotyp der von „EventTrigger“ erbt verbinden.

Die Konzepte der Advice und der RoleMethods, für die es eigene Stereotypen gibt, sind in Abbildung 4.3 zu sehen, welche die Stereotypen darstellt, die für die Modellierung aspektorientierter Komponenten benötigt werden.

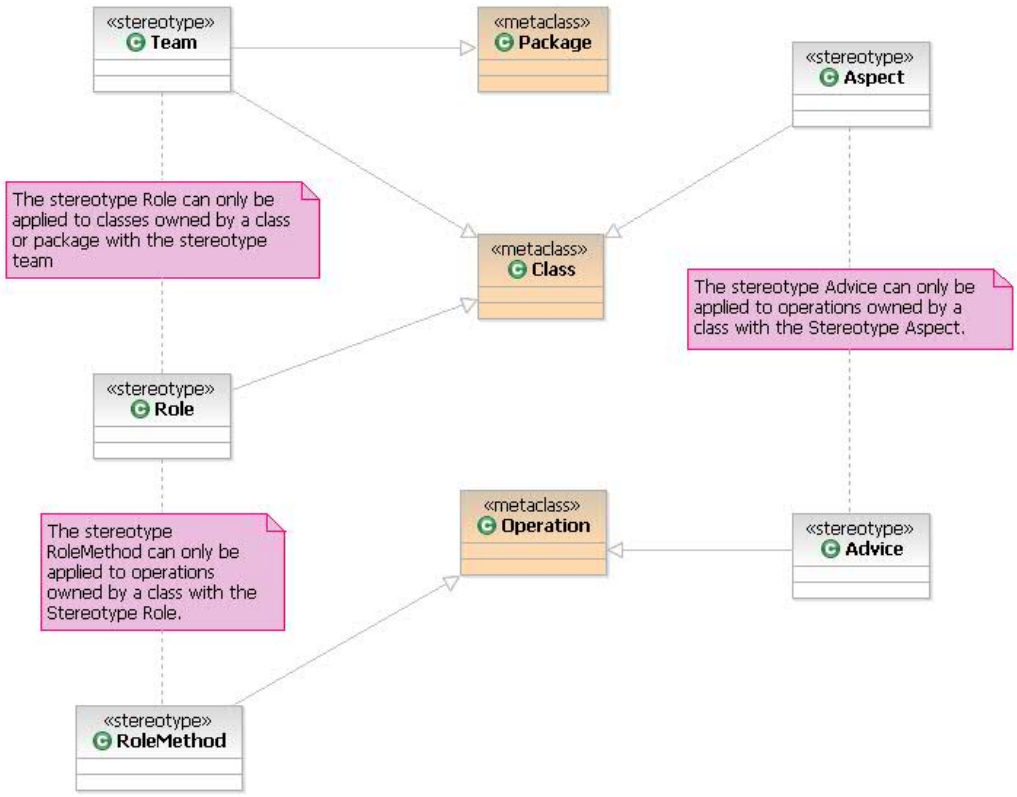

Abbildung 4.3: Aspekt Komponenten

Die meisten aspektorientierten Programmiersprachen sehen die Programmierung von Aspekten als spezielle Klassen und von Advice als spezielle Methoden vor. Der Stereotyp „Aspect“ besitzt deshalb einen Extensionspunkt zu der Metaklasse „Class“, um wie eine Klasse modelliert werden zu können. Der Stereotyp „Advice“ besitzt dementsprechend einen Extensionspunkt zu der Metaklasse „Operation“, so dass Advice als spezielle Operationen modelliert werden können. Für Advice gilt zudem die folgende Bedingung:

- Eine Operation darf nur dann den Stereotyp „Advice“ besitzen, wenn sie sich in einer Klasse mit dem Stereotyp „Aspect“ befindet.

Einige aspektorientierte Programmiersprachen wie ObjectTeams unterstützen das Aspect- und das Advice-Konzept nicht. Die Verwendung der Stereotypen „Aspect“ und „Advice“ garantiert also keine vollständige Plattformunabhängigkeit. Dies wird auch später in dem Fallbeispiel aufgezeigt. Um auch geeignete Modelle für ObjectTeams mithilfe des hier vorgestellten UML-Profils modellieren zu können, wurde das UML-Profil um die Stereotypen „Team“, „Role“ und „RoleMethod“ erweitert. Diese Stereotypen enthalten keine Tagged-Values, es gelten aber die folgenden Bedinungen für ihre Verwendung:

- Eine Operation darf nur dann den Stereotyp „RoleMethod“ besitzen, wenn sie sich in einer Klasse mit dem Stereotyp „Role“ befindet und
- eine Klasse darf nur dann den Stereotyp „Role“ besitzen, wenn sie sich in einer Klasse oder einem Package mit dem Stereotyp „Team“ befindet.

Mithilfe der nun vorgestellten Stereotypen sollten sich genug Informationen modellieren lassen, um auf Modellebene bereits Join-Point-Shadows ermitteln zu können. Damit diese auch im Modell gekennzeichnet werden können, enthält das UML-Profil schliesslich noch mehrere Stereotypen, die dafür benötigt werden und die in Abbildung 4.4 gezeigt werden.

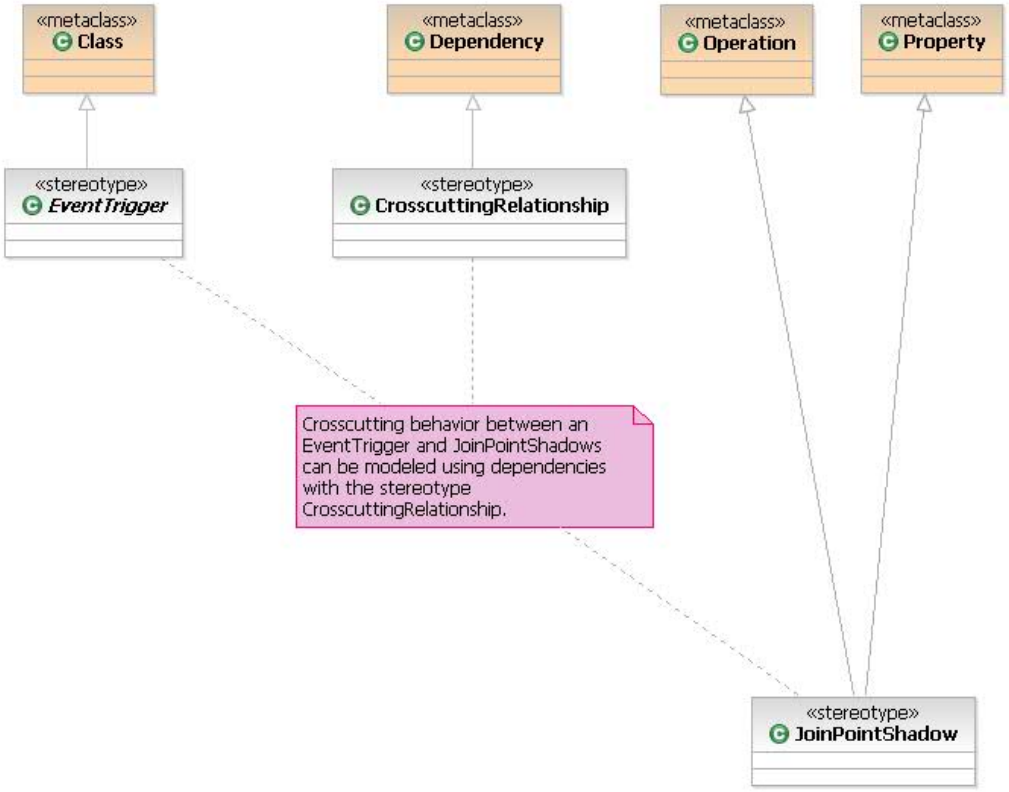

Abbildung 4.4: Join-Point-Shadows

Der Stereotyp „JoinPointShadow“ dient der Markierung ermittelter Operationen oder Properties und besitzt deswegen Extensionspunkte zu den Metaklassen „Operation“ und „Property“.

Damit die Zugehörigkeit eines Join-Point-Shadow zu einem Event-Trigger verdeutlicht werden kann, gibt es zudem den Stereotyp „CrosscuttingRelationship“. Für diesen Stereotyp gilt die folgende Bedingung:

- Eine Dependency mit dem Stereotyp „CrosscuttingRelationship“ darf nur Klassen mit einem Stereotyp der von „EventTrigger“ erbt mit Operationen oder Properties mit dem Stereotyp „JoinPointShadow“ verbinden.

Die beiden Stereotypen „JoinPointShadow“ und „CrosscuttingRelationship“

sollten nicht direkt vom Benutzer verwendet werden, sondern sind für die Auswertungsfunktion bestimmt. Sie können dennoch vom Benutzer verwendet werden, um ein Modell verständlicher zu machen. Die Ergebnisse der Auswertungsfunktion werden durch die Verwendung dieser Stereotypen durch den Benutzer nicht beeinflusst.

Abschliessend muss noch einmal genauer auf die Semantik der Tagged-Values eingegangen werden, die durch Strings oder StringArrays angegeben werden. Diese Tagged-Values werden mit Aussnahme von den beiden Tagged-Values „description“ des Stereotyp „Pointcut“ und „eventFilterGuardCondition“ des Stereotyp „ActionToEventTriggerBinding“ von der Auswertungsfunktion als Namensmuster interpretiert. Die Semantik für solche Namensmuster wird im Folgenden beschrieben:

- der String „**“ passt zu jedem beliebigen String,
- der String „*Text“ passt zu jedem String, dessen Wert mit „Text“ endet,
- der String „Text*“ passt zu jedem String, dessen Wert mit „Text“ beginnt,
- der String „*Text*“ passt zu jedem String, der an einer beliebigen Stelle in seinem Wert den Substring „Text“ enthält und
- der String „*Text*“ passt nur zu einem String, der auch den Wert „Text“ besitzt.

Der Wert „Text“ kann durch beliebige Werte ersetzt werden. Nimmt man als Beispiel den Methodennamen „customerSetID“ an, so gilt:

Das Muster „**“ passt zu „customerSetID“,
das Muster „**“ passt auch zu „“,
das Muster „*SetID“ passt zu „customerSetID“,
das Muster „*Set“ passt NICHT zu „customerSetID“,
das Muster „customer*“ passt zu „customerSetID“,
das Muster „Set*“ passt NICHT zu „customerSetID“,
das Muster „*customerSet*“ passt zu „customerSetID“,
das Muster „*SetID*“ passt zu „customerSetID“,
das Muster „*customerSetID*“ passt zu „customerSetID“,

das Muster „customerSetID“ passt zu „customerSetID“ und
das Muster „customerSet“ passt NICHT zu „customerSetID“.

4.2 Pointcutauswertung auf Modellebene

Unter der Auswertung von Pointcuts wird, wie bereits beschrieben, die Ermittlung von sogenannten Join-Point-Shadows verstanden. Dem hier verwendeten Begriff Pointcut entspricht in dem vorgestellten UML Profil der Stereotyp „EventTrigger“ oder genauer gesagt die beiden von diesem erbenden Stereotypen „Pointcut“ und „JoinPointQuery“, da der Stereotyp „EventTrigger“ abstrakt ist. Ermittelt werden dabei Features, also Attribute oder Operationen von Klassen, deren statische Eigenschaften den in den EventTriggern beschriebenen statischen Informationen entsprechen.

Im Folgenden wird zunächst der Auswertungsalgorithmus beschrieben. Bei der Auswertung werden OCL-Ausdrücke als Modellabfragen eingesetzt. Die Umsetzung von Informationen aus dem Modell zu OCL-Ausdrücken, die als Abfragen eingesetzt werden können, wird dann in einem eigenen Unterkapitel näher beschrieben.

4.2.1 Auswertungsalgorithmus

Bei der Auswertung von Event-Triggern muss zwischen der Auswertung eines Join-Point-Queries und der Auswertung eines Pointcuts unterschieden werden. Da ein Pointcut aus mehreren Event-Triggern zusammengebaut ist, lässt sich die Auswertung eines Pointcuts rekursiv auf die Auswertung einer Menge von Join-Point-Queries zurückführen. Die Vereinigung aller ermittelten Join-Point-Shadows aller ausgewerteten Join-Point-Queries eines Pointcuts ergibt die Menge seiner Join-Point-Shadows. Der Auswertungsalgorithmus für einen Pointcut lautet also wie folgt:

1. Alle Join-Point-Queries eines Pointcuts werden ermittelt,
2. jeder einzelne dieser Join-Point-Queries wird ausgewertet und
3. die Ergebnisse dieser Auswertungen werden vereint und ergeben gemeinsam die Menge der Join-Point-Shadows des Pointcuts.

Da ein Pointcut aus Join-Point-Queries und Pointcuts zusammengesetzt sein kann, ergibt sich die Menge all seiner Join-Point-Queries aus allen direkt zugeordneten Join-Point-Queries und den Join-Point-Queries aller seiner zugeordneten Pointcuts. Die Ermittlung dieser Menge wird folglich durch eine rekursive Funktion realisiert.

Die Auswertung eines Join-Point-Queries verwendet Informationen aus den Tagged-Values des Stereotyp „JointPointQuery“ und Informationen aus Stereotypen, die von „JoinPoint“ erben. Jeder Klasse mit dem Stereotyp „JoinPointQuery“ ist auch eine Klasse mit einem Stereotypen, der von „JoinPoint“ erbt, zugeordnet. Diese Zuordnung erfolgt über eine stereotypisierte Assoziation. Stereotypen, die von „JoinPoint“ erben, enthalten die eigentlichen Informationen der zu ermittelnden Join-Point-Shadows. Der Stereotyp „JoinPointQuery“ besitzt nur Tagged-Values, die den Suchraum einschränken sollen.

Bei der Auswertung eines Join-Point-Queries wird entweder eine Menge von Operationen oder eine Menge von Properties gesucht. Ob es sich dabei um Properties oder Operationen handelt, geht aus dem konkreten Join-Point-Kind-Stereotyp hervor. Im Groben lässt sich der Algorithmus für die Auswertung eines Join-Point-Queries wie folgt beschreiben:

1. Der Join-Point-Query und sein dazugehöriges Join-Point-Kind werden ermittelt,
2. die Werte der Tagged-Values beider Stereotypen werden ermittelt,
3. die passenden Operationen oder Properties werden ermittelt und
4. die gefundenen Operationen oder Properties werden als Join-Point-Shadows gekennzeichnet und mit dem Join-Point-Query über eine Crosscutting-Relationship verbunden.

Der auszuwertende Join-Point-Query oder Pointcut wird vom Benutzer im Modell markiert. Das zu einem Join-Point-Query zugehörige Join-Point-Kind kann ermittelt werden, indem nach einer Assoziation dieser Klasse mit dem Stereotyp „JoinPointKindAndConstraints“ gesucht wird. Das durch diese Assoziation zugeordnete Modellelement ist eine Klasse mit einem Stereotyp, der von „JoinPoint“ erbt. Wurden beide Stereotypen ermittelt, lassen sich die

Werte ihrer Tagged-Values direkt auslesen.

Um anhand der Werte von Tagged-Values Operationen oder Properties im Modell zu ermitteln, werden die Werte der Tagged-Values in OCL-Ausdrücke eingebaut, die dann als Modellabfragen auf dem Modell ausgeführt werden. Wie das genau funktioniert wird im folgenden Unterkapitel beschrieben.

Die Kennzeichnung der ermittelten Join-Point-Shadows erfolgt, indem die ermittelten Operationen oder Properties mit Stereotypen versehen werden. Um ihre Zugehörigkeit zu ihrem Join-Point-Query zu verdeutlichen werden sie diesem zudem über eine stereotypisierte Dependency zugeordnet. Dabei wird eine Dependency von einem Join-Point-Query hin zu jedem betroffenen Join-Point-Shadow erstellt, die den Stereotyp „CrosscuttingRelationship“ besitzt.

4.2.2 Modellabfragen mit der OCL

Die Wahl der OCL für die Ausdrücke erfolgte, weil sie bereits fester Bestandteil der UML ist und genau wie Pointcuts dem Zweck dient, Bedingungen zu formulieren. Pointcuts formulieren Bedingungen, die eine Stelle in einem System erfüllen muss, damit ein Aspekt an ihr wirkt. Die OCL formuliert, sehr viel allgemeiner, Bedingungen, die Objekte in einem Modell erfüllen müssen. Da OCL-Ausdrücke auch als Modellabfragen eingesetzt werden können, eignen sie sich deshalb optimal als Pointcutausdrücke für die Modellebene. OCL-Ausdrücke als Modellabfragen einzusetzen bedeutet dabei, alle Modellelemente zu ermitteln, die die in dem Ausdruck beschriebenen Bedingungen erfüllen.

In Anlehnung an das vorherige Unterkapitel werden nur für Join-Point-Queries und ihre Join-Point-Kinds OCL-Ausdrücke erstellt. Dabei wird folgendermassen vorgegangen:

1. Erstellung eines einzelnen Ausdrucks für jede relevante Tagged-Value.
2. Verknüpfung der Ausdrücke für die festgelegten Attribute durch einen logischen UND-Operator.

Für fast alle Tagged-Values aus dem UML-Profil lässt sich ein bestimmter OCL-Ausdruck angeben, der nur in Abhängigkeit des Werts der Tagged-

Value variiert. Im Folgenden werden die einzelnen Tagged-Values und die dazu passenden OCL-Ausdrücke vorgestellt:

- JoinPointQuery.owner

  ```
  self.class_.name.contains('TAGGED_VALUE')
  ```

- JoinPointQuery.packageScope

  Für jeden Packagenamen Ausdruck der Art

  ```
  self.class_[.package]*.name.contains('TAGGED_VALUE[*]')
  ```

  erstellt. Diese Ausdrücke ergeben miteinander logisch verundet einen Gesamtausdruck. Ein Beispiel dafür wäre:

  ```
  self.class_.package.package.package.contains('de')
  and self.class_.package.package.name.contains('gebit')
  and self.class_.package.name.contains('topprax')
  and self.class_.name.contains('**')
  ```

- PropertyJP.name, OperationJP.name, ExceptionHandlerJP.exceptionName

  ```
  self.name.contains('TAGGED_VALUE')
  ```

- OperationJP.ParameterTypes

  Diese Tagged-Value kann leider aufgrund fehlender OCL-Operationen nicht ausgewertet werden.

- OperationJP.returnType, PropertyJP.type

  ```
  self.type.name.contains('TAGGED_VALUE')
  ```

- OperationJP.throwTypesClassNames

  Diese Tagged-Value kann leider aufgrund fehlender OCL-Operationen nicht ausgewertet werden.

- OperationJP.isConstructor

  Da auf der UML-Modellebene, auf der diese Auswertungsfunktion arbeitet, bei Operationen noch nicht zwischen Konstruktoren und Methoden unterschieden wird, kann diese Tagged-Value nicht explizit ausgewertet werden.

- OperationJP.visibility, PropertyJP.visibility

  ```
  self.visibility = VisibilityKind::TAGGED_VALUE
  ```

In den oben angeführten OCL-Ausdrücken steht der Eintrag „TAGGED_VALUE“ für den Wert der Tagged-Value, wie er im Modell angegeben ist. Die Schreibweise „TAGGED_VALUE[*]“ weist auf Werte von Tagged-Values hin, die als Arrays angegeben werden, wie zum Beispiel der Packagepfad. Das „[*]“ steht für den Index dieses Arrays.

Die in manchen der oben genannten Ausdrücke verwendete String-Operation

```
boolean contains(String substring)
```

ist nicht Bestandteil der OCL-Spezifikation. Sie sollte deshalb in einer Implementierung der OCL ergänzt werden.

Ein fertiger zusammengesetzter OCL-Ausdruck schliesslich kann dann als Modellabfrage verwendet werden, wobei implizit der folgende OCL-Ausdruck verwendet wird:

```
allInstances->select(AUSDRUCK)
```

Das bedeutet, es werden alle Elemente im Modell ermittelt, für die der angegebene Ausdruck zu true ausgewertet werden kann. Ein Beispiel für einen Gesamtausdruck ist:

```
self.class_.name.contains('*FlightBooking*')
and self.oclIsKindOf(uml::Operation)
and self.name.contains('*bookFlight*')
and self.visibility = VisibilityKind::public
```

Kapitel 5

Machbarkeitsstudie

Für die Machbarkeitsstudie wurden das UML-Profil und die Auswertungsfunktion als Plugins für IBM Rational Software Architect [30] implementiert. Der RSA erfüllt alle der gestellten Anforderungen an ein UML-CASE-Werkzeug und besitzt eine bedeutende Marktstellung. Der RSA baut zudem auf der weit verbreiteten und bekannten freien Entwicklungsplattform Eclipse [5] auf. Die Erweiterung um zusätzliche Funktionalität über den Eclipse-Plugin-Mechanismus schliesslich macht den Einstieg für Eclipse-erfahrene Benutzer leicht.

Im Folgenden wird zunächst die Erweiterung des RSA beschrieben. Anschliessend wird ein Beispielmodell vorgestellt und schliesslich erläutert, in wiefern das UML-Profil und die Auswertungsfunktion die an sie gestellten Anforderungen erfüllen. Danach wird noch darauf eingegangen, inwiefern sich die OCL als Pointcutsprache einsetzen lässt.

5.1 Implementierung für den RSA

Um zu verstehen, wie man den RSA um zusätzliche Funktionalität erweitern kann, muss man zunächst die Eclipse-Plattform, auf die der RSA aufbaut, verstehen. Die Eclipse-Plattform besteht aus verschiedenen Bausteinen, sogenannten Plugins, die beim Starten von der Plattform durch einen speziellen Mechanismus zusammengebaut werden. Bis auf die Eclipse-internen Basisplugins baut jedes Plugin immer auf einem sogenannten Erweiterungspunkt auf. Die Eclipse-internen Basisplugins stellen eine Anzahl von Erweiterungspunkten zur Verfügung. Ein Erweiterungspunkt definiert dabei, wie die Funktionalität eines Plugins, welches auf diesem Erweiterungspunkt aufbaut, in die Benutzeroberfläche eingebunden wird. Der einfachste Erweiterungspunkt von Eclipse beispielsweise erstellt für ein Plugin, welches diesen

verwendet, einen Menüeintrag, über dessen Anklicken die Funktionalität des Plugins gestartet wird.

Neben der Verwendung von Erweiterungspunkten kann jedes Plugin auch die Bibliotheken aller anderen Plugins importieren. Erweiterungspunkte zu verwenden bedeutet also nicht Klassenbibliotheken zu importieren, sondern sich bei anderen Plugins für bestimmte Einträge in der Benutzeroberfläche zu registrieren.

Der RSA besteht aus der Eclipse-Plattform und einer Vielzahl zusätzlicher Plugins. Einige der Plugins des RSA bauen direkt auf Erweiterungspunkten der Standard-Eclipse-Plugins auf und erweitern dadurch die Standardbenutzeroberfläche. Der RSA bietet dadurch beispielsweise neue Projekttypen, neue Perspektiven und Views. Doch viele der Plugins des RSA stellen selbst wiederum Erweiterungspunkte zur Verfügung, die zum Beispiel die Ergänzung der neu definierten Views erlauben.

Ein neues Plugin, welches die Funktionalität des RSA ergänzen soll, muss um gestartet werden zu können einen Erweiterungspunkt verwenden. Entweder wird dazu der Erweiterungspunkt eines Standard-Eclipse-Plugins oder der eines RSA Plugins verwendet. Zusätzlich muss es von Bibliotheken verschiedener RSA- und oft auch Eclipse-Plugins Gebrauch machen. Im Folgenden wird ein Überblick über die erstellten Plugins und die von ihnen verwendeten Erweiterungspunkte gegeben:

- Das vorgestellte UML-Profil wird mithilfe RSA-eigener Plugins erstellt. Der RSA bietet dazu einen neuen Projekttyp namens „UML Profile Project“ an. Dadurch wird das UML-Profil automatisch in einem Format erstellt, welches später von anderen RSA-Plugins, die für die Modellierung zuständig sind, verwendet werden kann. Um das UML-Profil dann in den RSA einzubinden, muss es allerdings noch als Plugin kompiliert werden. Dabei baut es auf dem Erweiterungspunkt com.ibm.xtools.uml2.msl.UMLProfiles auf. Dieser Erweiterungspunkt erlaubt es, ein darin enthaltenes UML-Profil automatisch bei der UML-Modellierung mit dem RSA zur Verfügung zu stellen.

- Die Funktionalität für die Auswertung von Pointcuts wird in einem eigenen Plugin realisiert, welches einen Erweiterungspunkt des Eclipse-Standard-Plugins org.eclipse.ui_3.0.1 verwendet. Dieser Erweiterungs-

punkt erlaubt das Hinzufügen eines neuen Menüpunkts mit einem neuen Menüeintrag in der Menüleiste, durch dessen Klicken das neue Plugin gestartet werden kann.

- Die Funktionalität für die Entfernung von Join-Point-Shadow-Markierungen wird genau wie das Plugin für die Auswertung von Plugins in den RSA eingebunden.

Wichtige Bibliotheken, die im Rahmen der hier vorgestellten Plugins von besonderer Bedeutung sind:

- org.eclipse.uml2 - Die Eclipse UML 2 API [10] baut auf EMF auf und bildet die Metaklassen aus der UML 2 Spezifikation in Form von Klassen ab. Mit Ihr können beispielsweise Klassen im Modell Attribute zugewiesen werden und Assoziationen abgefragt werden.

- org.eclipse.emf - Das Eclipse Modeling Framework (EMF) [6]bildet die Grundlage für die Eclipse UML2 API . Viele Schnittstellen der RSA-Plugins liefern und erhalten Werte von Typen aus der EMF, die aber zu UML2 Typen gecastet werden können.

- com.ibm.xtools.modeler.UMLModeler - Dieses Paket erlaubt den Zugriff auf das Modell, welches vom Benutzer editiert wird.

- com.ibm.xtools.emf.query.ocl - Dieses Paket wertet OCL Queries auf dem von dem Benutzer editierten Modell aus und liefert passende Modellelemente zurück.

- com.ibm.xtools.emf.ocl.engine_6.0.0 - Dieses Paket wird bei der Auswertung von OCL Ausdrücken verwendet. Da die Quellen offen gelegt sind, kann an dieser Stelle in den Auswertungsmechanismus selbst eingegriffen und die OCL Standard Library erweitert werden.

Es wurden drei neue Plugins für den RSA mithilfe der oben angeführten Mechanismen erstellt:

- ein UML-Profil Plugin, welches das vorgestellte UML-Profil implementiert,

- ein Plugin, welches die Auswertung von ausgewählten Event-Triggern im Modell realisiert und dabei die beschriebene Auswertungsfunktion implementiert, und
- ein Plugin, mit welchem Stereotypen, die bei der Auswertung in das Modell eingetragen wurden, wieder entfernt werden können.

5.2 Beispielmodell

Das in Abbildung 5.1 vorgestellte Beispielmodell zeigt ein imaginäres Softwaresystem, welches mithilfe des im Rahmen dieser Arbeit erstellten UML-Profils erstellt wurde. Auf der linken Seite sind aspektorientierte und auf der rechten Seite nicht-aspektorientierte Modellelemente zu sehen. Man stelle sich die Elemente auf der rechten Seite als Ausschnitt eines größeren, komplexeren Flugbuchungssystem vor, die eigentlich auch mehr Features als im Diagramm angegeben enthalten. Die Elemente auf der linken Seite stellen eine aspektorientierte Erweiterung des Systems um ein Bonusmeilenprogramm dar, welches ebenfalls nur stark vereinfacht dargestellt ist.

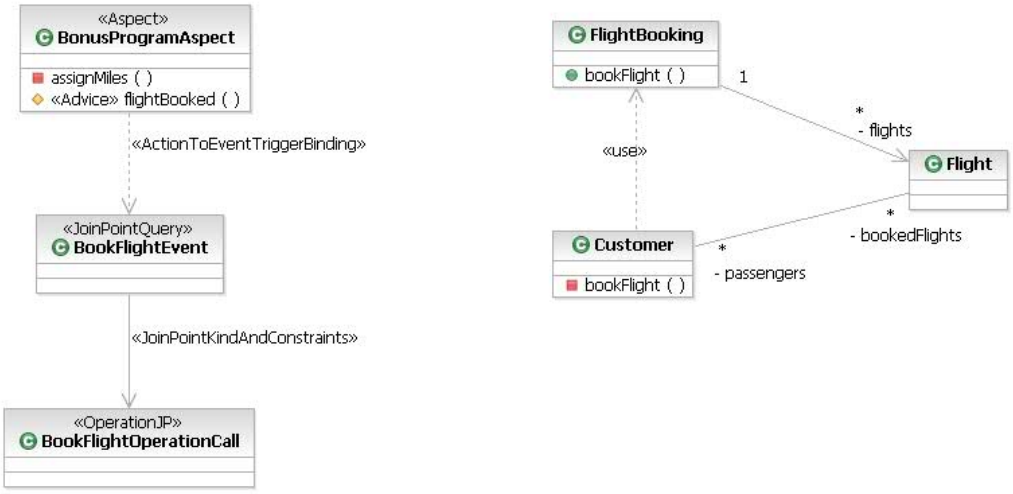

Abbildung 5.1: Flugbuchungssystem und Bonusmeilenprogramm

Es ist deutlich zu sehen, dass keine expliziten Verbindungen zwischen den aspektorientierten und den nicht-aspektorientierten Modellelementen beste-

hen. Dennoch werden in der Klasse „BookFlightEvent“ und der dazugehörigen Klasse „BookFlightOperationCall“ Eigenschaften beschrieben, die auf die Operation „bookFlight“ der Klasse „FlightBooking“ zutreffen. In der Grafik ist auch eine Schwäche des Einsatzes von UML-Profilen ersichtlich: die Tagged-Values der Stereotypen können in einem UML-Diagramm nicht visualisiert werden, obwohl es sich dabei um wichtige Informationen für das Verständnis des Modells handelt. Sie können nur in der Eigenschaftsansicht eines Modellelements im RSA eingesehen und verändert werden. Die Werte des Join-Point-Query „BookFlightEvent“ wurden nicht mit Werten versehen, was bedeutet, dass sich dieses Event in jeder beliebigen Klasse im Modell ereignen kann. Die Beschreibung des beobachteten Operationsaufrufs „BookFlightOperationCall“ wird durch die folgenden Tagged-Values des Stereotypen „OperationJP“ festgelegt:

- accessType = OperationAccessType.call ,
- isConstructor = false ,
- name = „bookFlight“ ,
- parameterTypes = keine Angabe ,
- returnType = keine Angabe ,
- throwTypes = keine Angabe und
- visibility = VisibilityType.public .

Es sollen also alle Aufrufe von öffentlichen (visibility = VisibilityType.public) Methoden (isConstructor = false) namens „bookFlight“ (name = „bookFlight“) zum Zeitpunkt ihres Aufrufs (accessType = OperationAccessType.call) als Join-Point erkannt werden. Parameter, möglicherweise ausgelöste Exceptions und Rückgabewert spielen dabei keine Rolle (keine Angaben für diese Kriterien).

Die Abbildung 5.2 zeigt, wie das Modell durch die Auswertungsfunktion verändert wird.

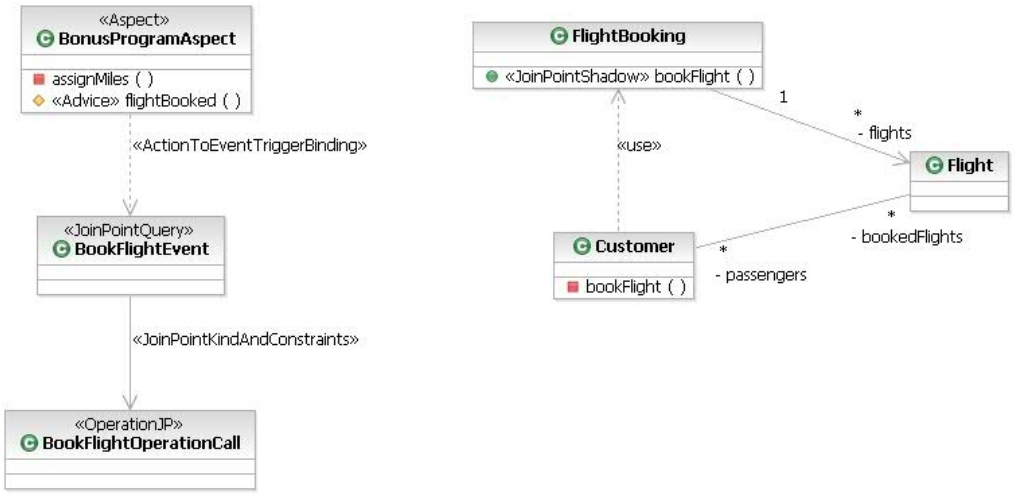

Abbildung 5.2: Modell nach Auswertung

Die querschneidende Beziehung zwischen dem Join-Point-Query „BookFlight-Event“ und der Operation „bookFlight“ konnte, wie man in der Grafik erkennen kann, mithilfe der Auswertungsfunktion aus diesen Informationen abgeleitet werden. Die Auswertungsfunktion erstellt auch eine Dependency zwischen dem „BookFlightEvent“ und „bookFlight“. Der RSA kann diese Dependency, wenn sie nicht über den Diagrammeditor angelegt wurde, nicht im Diagramm visualisieren. Fügt man sie allerdings manuell im Diagramm ein, so ergibt sich eine Ansicht, die in Abbildung 5.3 dargestellt wird.

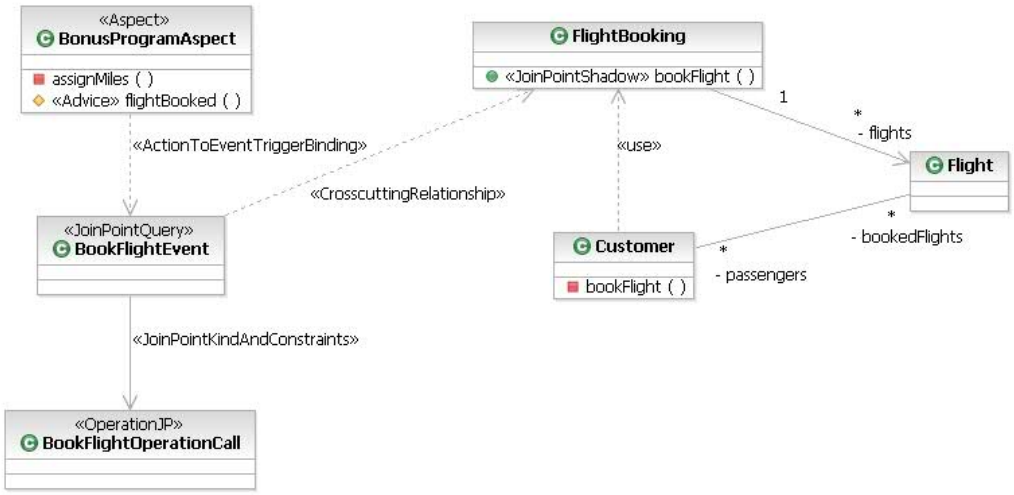

Abbildung 5.3: Modell nach Auswertung mit Dependencies

Es handelt sich dabei um ein technisches Problem, welches allerdings keine Auswirkung auf die Feststellung hat, dass in dem aspektorientierten Modell genügend Informationen modelliert werden können, um die richtigen querschneidenden Beziehungen daraus zu ermitteln.

Aus diesem Beispiel geht allerdings der Vorteil der impliziten Beschreibung der Join-Point-Shadows noch nicht klar hervor. Man stelle sich nun vor, man verwendet die gleichen aspektorientierten Modellelemente mit einem ebenfalls stark vereinfachten Buchungssystem eines Reisebüros und verwendet wieder die Auswertungsfunktion. Dann ergibt sich das in Abbildung 5.4 dargestellte Diagramm.

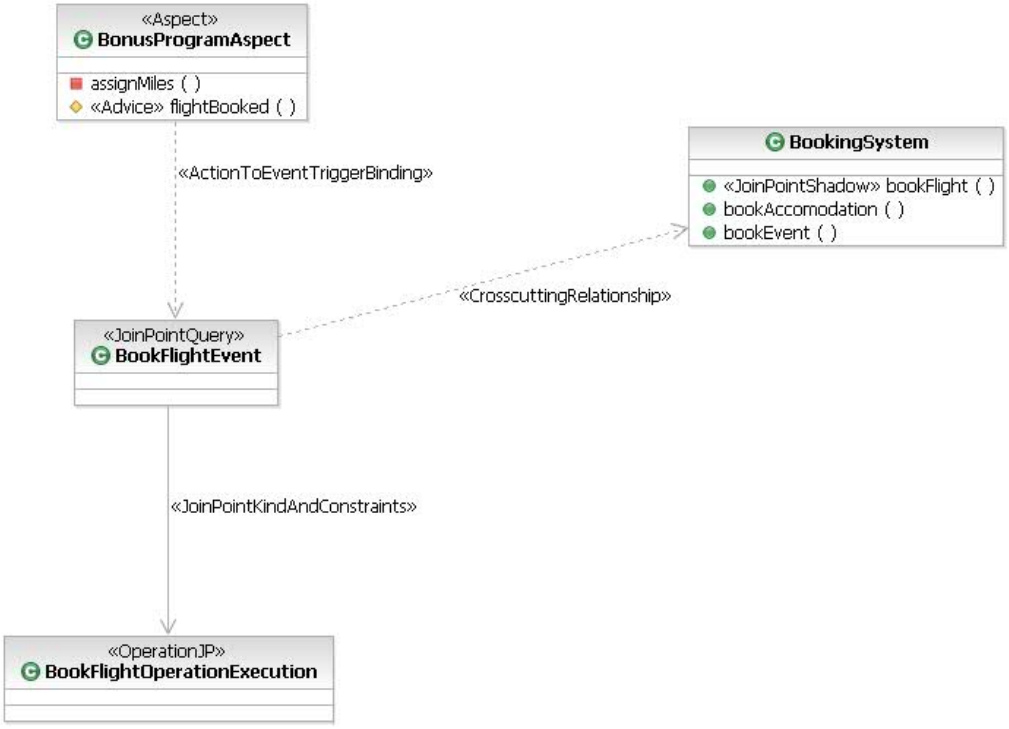

Abbildung 5.4: Variante mit anderem Basissystem

Die Dependency zwischen „BookFlightEvent“ und „bookFlight“ wurde hier der Verständlichkeit halber im Diagramm manuell eingetragen. Man sieht: dadurch, dass die Informationen des Join-Point-Queries implizit angegeben sind und nicht direkt querschneidende Beziehungen modelliert wurden, ist der Join-Point-Query „BookFlightEvent“ wiederverwendbar.

Mit den obigen Beispielen wurde zunächst nachgewiesen, dass die Modellierung von Pointcuts möglich ist und alle notwendigen Informationen für die Ermittlung querschneidender Beziehungen modelliert werden können. Das aspektorientierte Modell ist ausserdem wiederverwendbar in Kombination mit anderen Systemen, die es erweitern soll. Im Folgenden wird nun gezeigt, wie zusammengesetzte Pointcuts, die in dem vorgestellten UML-Profil einfach als „Pointcuts“ bezeichnet werden, modelliert werden können. Dazu wird wieder das erste Beispiel des Flugbuchungssystems herangezogen und eine Änderung an dem aspektorientierten System vorgenommen. Im der Ab-

bildung 5.5 ist das komplexere Beispiel nach seiner Auswertung zu sehen, wobei die Dependencies wieder manuell im Diagramm eingefügt wurden.

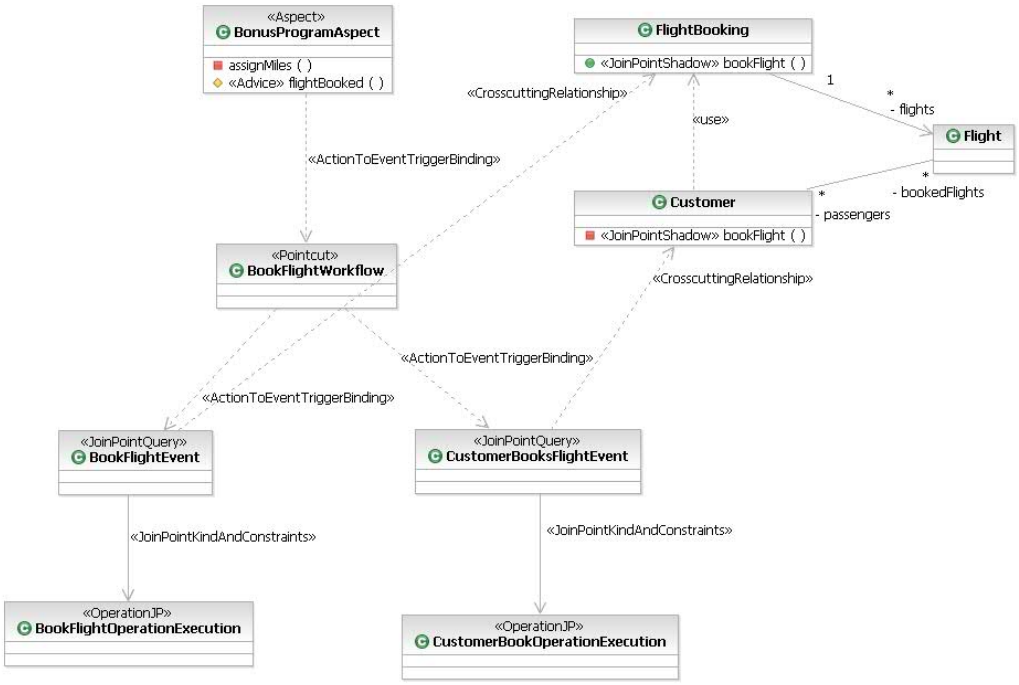

Abbildung 5.5: Flugbuchungssystem mit komplexem Pointcut

Der Advice ist nicht mehr mit dem „BookFlightEvent“ verbunden, sondern mit dem Pointcut „BookFlightWorkflow“. „BookFlightWorkflow“ wiederum verbindet zwei Join-Point-Queries. Die Relation zwischen diesen ist in der Tagged-Values „description“ angegeben und besagt, dass das in „BookFlight-Event“ beschriebene Ereignis innerhalb des Ereignisses, welches durch „CustomerBookFlightEvent“ beschrieben wird, stattfinden muss. Die Idee, die hinter diesem Pointcut steckt, ist: es sollen nur solche Flugbuchungen betrachtet werden, die von einem Customer durchgeführt wurden. In dem ersten Beispiel des Flugbuchungssystems wurden hingegen alle Flugbuchungen betrachtet.

Dieses Beispiel soll verdeutlichen, dass auch die Modellierung zusammengesetzter Pointcuts geeignete Informationen für die Ermittlung von Join-Point-

Shadows zur Verfügung stellt. Nun gilt es nachzuweisen, dass das vorgestellte Modell plattformunabhängig ist. Um dies zu verdeutlichen, soll das Modell in den zwei konzeptuell und syntaktisch stark unterschiedlichen Sprachen AspectJ und ObjectTeams implementiert werden. Auch wenn dadurch kein Beweis erbracht werden kann, dass die Abbildung des Beispielmodells auf beliebige andere aspektorientierte Programmiersprachen möglich ist, so lässt sich doch an dieser Stelle anmerken, dass die meisten Sprachen eine Syntax für Pointcuts verwenden, die der von AspectJ stark ähnelt, weshalb auch für diese Sprachen eine solche Abbildung möglich sein sollte. Beispiele für solche Sprachen sind JBossAOP, LogicAJ oder AspectWerkz.

Um die gleich folgenden Codebeispiele besser verständlich zu machen, werden noch einmal die Tagged-Values der dafür relevanten Modellelemente aufgezählt. Die Werte des Join-Point-Query „BookFlightEvent“ wurden nicht mit Werten versehen. Im Join-Point-Query „CustomerBooksFlightEvent“ ist hingegen die Tagged-Value „owner“ mit dem String „Customer“ festgelegt. „BookFlightEventExecution“ enthält die folgenden Tagged-Values:

- accessType = OperationAccessType.execution ,
- isConstructor = false ,
- name = „bookFlight“ ,
- parameterTypes = keine Angabe ,
- returnType = keine Angabe ,
- throwTypes = keine Angabe und
- visibility = VisibilityType.public .

Und „CustomerBookOperationExecution“ hat die Tagged-Values:

- accessType = OperationAccessType.execution ,
- isConstructor = false ,
- name = „bookFlight“ ,
- parameterTypes = keine Angabe ,

- returnType = keine Angabe ,
- throwTypes = keine Angabe und
- visibility = VisibilityType.private .

In dem Pointcut „BookFlightWorkflow“ hat die Tagged-Value „description“ den folgenden Wert:

```
The method described in "BookFlightEvent" is executed during
the execution of the method described in
"CustomerBooksFlightEvent".
```

Diese natürlichsprachliche Aussage entspricht einem Pointcuttyp, der in den meisten Pointcutsprachen „cflow“ genannt wird. Man erkennt an diesem Beispiel, dass ein Programmierer, der das vorgestellte Modell implementieren soll, aus dieser Beschreibung selbstständig erkennen muss, dass es sich um einen „cflow“-Pointcut handelt, der hier beschrieben wird. Das UML-Profil gibt für den Wert der Tagged-Value „description“ keine Konvention vor. Man könnte den Wert ebenfalls folgendermassen formulieren:

```
cflow(@BookFlightEvent) && @CustomerBooksFlightEvent
```

Diese Syntax lehnt sich grob an eine Syntax aus Programmiersprachen wie JBossAOP oder AspectWerkz an, in denen an anderer Stelle beschriebene Pointcuts mit einem „@“ Zeichen referenziert werden. Um festzulegen, wie die Value von „description“ zu formulieren ist, sollte vor der Modellierung eine entsprechende Richtlinie festgelegt werden. Möchte man die Plattformunabhängigkeit des Modells sicherstellen, sollte man sich dabei auf natürlichsprachliche Aussagen festlegen.

Im Folgenden wird nun eine Implementierung des Modells mit den Stereotypen „Aspect“ und „Advice“ in AspectJ vorgestellt:

```
aspect BonusProgramAspect {

  private void assignMiles () {
      //...
  }
```

```
  protected void flightBooked() {
      //...
  }

  after (): bookFlightWorkflow() {
      flightBooked();
  }

  pointcut bookFlightWorkflow:
      cflow(
        execution( private * Customer.bookFlight(..) ) ) &&
        execution (public * bookFlight(..))

}
```

Möchte man aus dem Besipielmodell eine ObjectTeams Implementation ableiten, so ergibt sich das Problem, dass die Stereotypen „Aspect“ und „Advice“ nicht gut geeignet sind, um Teams aus dem ObjectTeams-Ansatz zu modellieren. Das Modell wird deshalb paralell noch einmal für ObjectTeams Zwecke folgendermassen angepasst. Abbildung 5.6 zeigt das angepasste Modell.

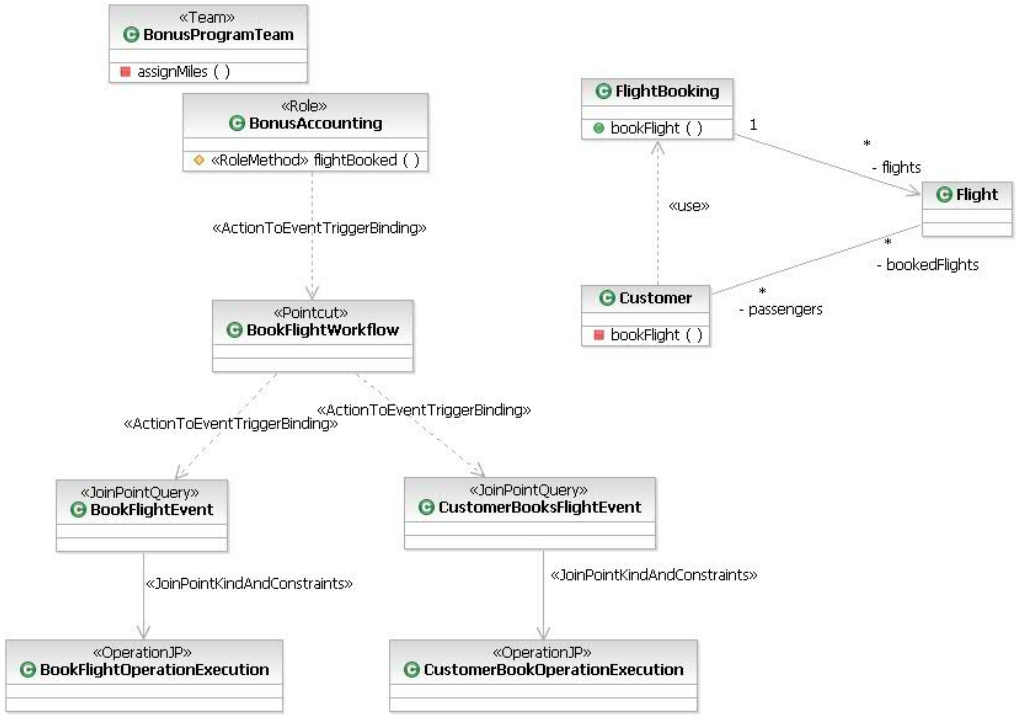

Abbildung 5.6: Angepasstes Modell für ObjectTeams

Der ObjectTeams-Code für das für ObjectTeams angepasste Modell sähe folgendermassen aus:

```
team class BonusProgramTeam {

    private void assignMiles () {
        //...
    }

    class BookFlightWorkflow extends Pointcut {

        boolean enabled = false;
        callin void bookFlightWorkflowCflow ( ){
            boolean oldVal = enabled;
            enabled = true;
            base.customerBooksFlightEvent ( );
```

```
            enabled = oldVal;
        }

        query Set<Method> customerBooksFlightEvent() {
            system.packages*.types.methods[isPrivate &&
              name.equals("Customer.bookFlight")]
        }
        bookFlightWorkflowCflow <-
          replace query customerBooksFlightEvent;

        query Set<Method> bookFlightEvent() {
            system.packages*.types. methods[isPublic &&
              name.equals("bookFlight")]
        }
        fire <- replace query bookFlightEvent when (enabled);

    }

    class BonusAccounting {

        void flightBooked ( ) {
            //....
        }

        flightBooked <- after BookFlightWorkflow.fire;
    }

}
```

Der hier angeführte ObjectTeams-Code enthält noch Sprachfeatures, welche die derzeitige ObjectTeams-Implementation noch nicht bereit stellt und die deshalb noch Änderungen unterliegen können. Wichtig ist, dass aus dem Beispiel hervorgeht, dass eine Abbildung der Informationen aus dem Beispielmodell auf ObjectTeams-Sprachfeatures möglich ist, auch wenn die oben angebene Syntax sich noch ändern kann.

Aus den Implementationsbeispielen lässt sich erkennen, dass aus den gleichen modellierten Pointcuts im Modell genug Informationen für die Implementie-

rung dieses Modells in verschiedenen aspektorientierten Programmiersprachen enthalten sind. Die Modellierungsnotation ist also in Hinsicht auf die mit ihr modellierbaren Pointcuts plattformunabhängig. Eine Schwäche des UML-Profils ist die freie Verwendung der Tagged-Value „description“ des Stereotyp „Pointcut“. Hier sollte, wie bereits erwähnt, darauf Wert gelegt werden, keine plattformspezifische Syntax zu verwenden.

Das ObjectTeams-Codebeispiel zeigt, dass es nicht immer möglich ist, die Konzepte Aspect und Advice auf alle verschiedenen aspektorientierten Sprachen abzubilden. Während die meisten aspektorientierten Programmiersprachen wie AspectJ, JBossAOP oder AspectWerkz direkt die Konzepte Aspect und Advice unterstützen, finden sich diese Konzepte in anderen Sprachen wie ObjectTeams nicht wieder. Für ObjectTeams beispielsweise ist eine plattformspezifische Anpassung dieser Konzepte auf Teams, Rollen und Rollenmethoden notwendig. Die Pointcutkonzepte des vorgestellten UML-Profils lassen sich allerdings auf alle der hier genannten aspektorientierten Programmiersprachen abbilden.

5.3 Auswertung

Die Implementierung der Plugins für den RSA und der Entwurf des vorgestellten Beispielmodells mithilfe dieser Plugins haben mehrere wichtige Ergebnisse für diese Arbeit geliefert, die im Folgenden erläutert werden.

Die wichtigste Feststellung ist, dass das Beispielmodell gezeigt hat, dass die beiden zentralen Ziele der Arbeit erreicht wurden. Die plattformunabhängige Modellierung von Pointcuts mithilfe des vorgestellten UML-Profils ist möglich und die mit ihr modellierbaren Pointcuts lassen sich auf Modellebene auswerten, so dass eine Überprüfung ihrer Korrektheit in Hinsicht auf ihre Join-Point-Shadows durchgeführt werden kann. Die im Kapitel „Handlungsbedarf“ formulierten Anorderungen werden somit durch das vorgestellte UML-Profil und die Auswertungsfunktion erfüllt.

Beim praktischen Einsatz des vorgestellten UML-Profils hat sich herausgestellt, dass die Verwendung eines UML-Profils gegenüber der Verwendung eines erweiterten UML-Metamodells einige Einschränkungen mit sich bringt, die im Folgenden aufgezählt werden:

- Um die semantische Korrektheit von Modellen, bei denen das UML-Profil verwendet wird, zu garantieren, müssen viele Stereotypen durch Constraints eingeschränkt werden. Diese Einschränkungen offenbaren sich dem Benutzer nur, wenn er einen Stereotypen „falsch“ einsetzen möchte und das Modell durch sein UML-CASE-Werkzeug hinsichtlich seiner Constraints validiert wird. Es wäre besser, wenn die semantisch falsche Verwendung von Modellelementen gar nicht möglich wäre.

- Die Attribute eines Stereotypen, seine Tagged-Values, sind keine regulären Attribute des Modellelements, auf das der Stereotyp angewendet wurde. Daher kann man diese Werte in den meisten UML-CASE-Werkzeugen, wie dem RSA, nicht in der Diagrammansicht eines Modells visualisieren. Da bei der Modellierung von Pointcuts aber gerade die Tagged-Values der Stereotypen eine sehr wichtige Rolle spielen, wäre es wünschenswert, diese Angaben auch in der Diagrammansicht visualisieren zu können.

- Stereotypen können immer nur auf existierende Modellelemene angewendet werden. In der hier vorliegenden Arbeit, werden in vielen Fällen Modellelemente erstellt, die nur dazu bestimmt sind, als Platzhalter für einen Stereotyp zu dienen. Ihre gesamte Semantik ist alleine in dem angewendeten Stereotyp enthalten. In einem solchen Fall, wäre es wünschenswert, nicht mit Stereotypen zu arbeiten, sondern direkt Metaklassen verwenden zu können. Ein Beispiel dafür ist der Stereotyp „JoinPointQuery“.

Bei der Entwicklung der Constraints für die Stereotypen, hat sich herausgestellt, dass sich keiner der Constraints der Stereotypen des vorgestellten UML-Profils geeignet mithilfe der OCL formulieren lässt. Aus diesem Grund wurden sogenannte Java-Constraints implementiert, eine Technik, die der RSA zur Verfügung stellt. Java-Constraints sind leider im Gegensatz zu OCL-Constraints nicht wiederverwendbar im Rahmen anderer UML-CASE-Werkzeuge.

Um die Komplexität der Modellierungsnotation in Grenzen zu halten, wurden bewusst einige mögliche Features einer Modellierungsnotation für aspektorientierte Modelle ausgespart oder in stark vereinfachter Form in die hier beschriebene Modellierungsnotation aufgenommen. Im Folgenden werden diese

Einschränkungen aufgeführt und begründet:

- Es gibt keinen Join-Point-Kind für das Erreichen bestimmter Anweisungen innerhalb eines Anweisungsblocks. Dieses sehr spezielle Feature wird von keiner der im Rahmen dieser Arbeit untersuchten Pointcutsprachen zur Verfügung gestellt. Außerdem ist es mit der vorliegenden Modellierungsnotation und dem gegebenen UML-Metamodell nicht möglich in einem Modell Informationen über Anweisungen innerhalb von Ausführungsblöcken darzustellen.

- Die Beziehungen, die zwischen einzelnen Event-Triggern eines Pointcuts bestehen, lassen sich nur in einem String, der Tagged-Value „description“ des Stereotypen „Pointcut“, ausdrücken. Temporalrelationen, Veroderungen von Ereignissen oder Context Exposure beispielsweise können deshalb nicht mithilfe dafür vorgesehener Modellelemente ausgedrückt werden. Möchte man zum Beispiel einen Pointcut definieren, der zwei Ereignisse in einer bestimmten zeitlichen Reihenfolge und unter der Bedingung, dass beide Ereignisse die gleiche Operation aufrufen, beschreiben, so ist dieser Zusammenhang nur natürlichsprachlich oder in Form einer vom Benutzer ausgewählten Ausdruckssprache formulierbar.

- Event-Filter wie sie von Herrmann in [18] beschrieben werden, lassen sich ebenfalls nur natürlichsprachlich oder in einer vom Benutzer ausgewählten Ausdruckssprache formulieren.

- Join-Point-Kinds können nicht direkt zwischen einer Methode oder einem Konstruktor unterscheiden. Diese Differenzierung kann der Benutzer nur über die Tagged-Value „operationType“ eines Stereotypen, der sich auf Operationen bezieht, erzielen. Während diese Differenzierung für die Implementation von Bedeutung ist, kann auf der hier verwendeten UML-Modellebene nicht zwischen Methoden oder Konstruktoren unterschieden werden.

Die Auswertungsfunktion hat ebenfalls ein paar Einschränkungen, die bei ihrem Einsatz beachtet werden müssen.

- Join-Point-Queries werden bei der Auswertung durch stereotypisierte Dependencies mit ihren Join-Point-Shadows verbunden. Diese Dependencies werden aber leider nicht vom RSA im Diagramm angezeigt,

sondern können nur im ModelExplorer des RSA eingesehen werden. Das ist auf eine technische Einschränkung des RSA zurück zu führen. Während Änderungen im Modell wie die Umbenennung von Features oder Klassen oder die Anwendung von Stereotypen automatisch im Diagramm übernommen und visualisiert werden, ist dies für Dependencies nicht der Fall.

- Wird ein Pointcut ausgewertet, so ergeben sich seine Join-Point-Shadows immer durch die Vereinigung aller Join-Point-Shadows der Event-Trigger, von denen er abhängt. In der „description“ eines Pointcuts können allerdings Bedingungen formuliert werden, die die Menge der möglichen Join-Point-Shadows einschränken. Ein Beispiel: in einem Pointcut wird formuliert, dass mehrere Attributzugriffe innerhalb ein under derselben Operation ausgeführt werden. Dadurch schränkt sich die Gesamtmenge aller möglichen Operationen, um die es sich dabei handeln kann, auf die Schnittmenge der Operationen, in denen die verschiedenen Attributzugriffe auftreten können, ein. Die Auswertungsfunktion kann diese Einschränkungen allerdings nicht interpretieren und liefert als Ergebnis dennoch die Vereinigung dieser Mengen als Ergebnis.

- Die Implementierung einer contains-Operation für die OCL-Implementation des RSA war nicht möglich. Deshalb konnten alle OCL-Ausdrücke, die im Kapitel über den Einsatz der OCL als Querysprache vorgestellt wurden, die die contains-Operation enthalten, nicht eingesetzt werden. Der Einsatz der OCL als Querysprache ist also stark von den Gegebenheiten des verwendeten UML-CASE-Werkzeuges abhängig.

Zusammenfassend lässt sich also feststellen:

- der Umgang mit einem UML-Profil gegenüber dem Umgang mit einem erweiterten UML-Metamodell hat sich als umständlich erwiesen,

- die Formulierung von OCL-Constraints, die sich auf Stereotypen beziehen ist nicht möglich und

- einige Features mussten in dem vorgestellten UML-Profil eingespart werden, um dessen Komplexität gering zu halten.

5.4 Die OCL als Pointcutsprache

Die OCL wurde in dieser Arbeit als Querysprache auf dem Modell eingesetzt. Dazu wurden mithilfe der OCL-Ausdrücke formuliert, welche die Eigenschaften von Join-Point-Shadows beschreiben. Ein Plugin des RSA wertet diese Ausdrücke dann aus und ermittelt Modellelemente, welche die beschriebenen Eigenschaften besitzen. Lässt sich also behaupten, die OCL sei als Pointcutsprache verwendet worden? Im Folgenden wird begründet, warum die OCL nur bedingt als Pointcutsprache bezeichnet werden kann und welche praktischen Probleme sich im Umgang mit ihr ergeben können.

Mithilfe der OCL können Eigenschaften von Modellelementen der UML beschrieben werden. Die UML umfasst Beschreibungsmöglichkeiten für statische und für dynamische Elemente eines Softwaresystems. Da die OCL theoretisch die Eigenschaften für alle existierende Modellelemente der UML formulieren kann, können mit ihr Informationen über sowohl statische als auch dynamische Modellelemente formuliert werden. Da OCL-Ausdrücke zudem als Abfragen auf einem Modell eingesetzt werden können, die alle Elemente ermitteln, die den Angaben in dem OCL-Ausdruck entsprechen, kann ein solcher Ausdruck zum Identifizieren von Ereignissen verwendet werden. Da die Definition einer Pointcutsprache zudem nicht festlegt, ob es sich bei dem Softwaresystem um dessen Modell oder dessen Implementierung handelt, erfüllt die OCL damit die Kriterien für eine Pointcutsprache.

Die meisten Pointcutsprachen werden wohlgemerkt nicht dazu eingesetzt Join-Points im Modell eines Softwaresystems zu ermitteln, sondern in dessen Implementierung, und zwar meist zur Laufzeit. Da mit der OCL nur Ausdrücke über Modellelemente formuliert werden können, ist sie deshalb theoretisch nicht als Pointcutsprache für die Implementierung eines Softwaresystems geeignet. Dies könnte allerdings möglich sein, wenn jedem Modellelement in der Implementierung ein eindeutiges Element in der Implementierung zugeordnet wird. Ob dies wiederum möglich ist, hängt von den Laufzeitmodellen der verwendeten Implementierungssprache ab. Zusammengefasst lässt sich feststellen: die OCL kann als Pointcutsprachen verwendet werden, mit der Einschränkung, dass dies zunächst nur auf Modellebene eines Softwaresystems möglich ist.

Eine weitere, wichtige Feststellung ist, dass die möglichen Aussagen, die

mit der OCL formuliert werden können, von dem konkreten UML-Modell abhängen, auf welches sie sich beziehen. Enthält ein UML-Modell keine dynamischen Informationen, so lassen sich mithilfe der OCL auch keine Aussagen über die Dynamik von Elementen des Softwaresystems formulieren. Soll die OCL also als Pointcutsprache eingesetzt werden, die auch Aussagen über die Dynamik eines Softwaresystems formuliert, so muss das UML-Modell des Softwaresystems auch dynamische Elemente enthalten. Die OCL kann also, der oben aufgeführten Definition folgend, nicht als Pointcutsprache für Softwaresysteme, deren UML-Modell keine dynamischen Elemente enthält, angesehen werden.

Bei der Verwendung der OCL als Constraint- und als Querysprache im Rahmen dieser Arbeit, haben sich Probleme beim praktischen Einsatz der OCL als Pointcutsprache offenbart. Die verwendete Implementierung der OCL, die vom RSA bereit gestellt wird, schränkt den Einsatz von OCL-Ausdrücken ein.

Die offizielle OCL-Spezifikation definiert Grundtypen, welche die Basisbibliothek der OCL bilden und die allen Implementierungen einer OCL-Library gemein sind. Da die OCL dazu dienen soll, Aussagen über UML-Modelle zu formulieren, ihre Basisbibliothek aber keine Typen aus der UML vordefiniert, muss die Basisbibliothek der OCL zunächst um UML-Metaklassen erweitert werden. Für die Implementierung der OCL-Bibliothek gilt dasselbe.

Die OCL-Implementierung des RSA enthält zwar alle wichtigen UML-Metatypen, aber viele wichtige Operationen dieser Metatypen stehen nicht zur Verfügung. Ein wichtiges Beispiel dafür sind Stereotypen: mithilfe der OCL-Implementierung des RSA ist es nicht möglich herauszufinden, welche Stereotypen auf ein Element angewendet wurden. Die Implementierung der UML-Typen innerhalb der OCL-Typbibliothek ist nicht vollständig. Der Einsatz von OCL-Constraints und auch OCL-Modellqueries ist also in hohem Maße von der Implementierung der OCL abhängig, die eingesetzt wird.

Das Problem der fehlenden Ausdrucksmöglichkeiten über Stereotypen von Modellelementen ergab sich im Rahmen der Formulierung von Constraints für Stereotypen aus dem UML-Profil. Es wurde gelöst, indem diese Constraints nicht als OCL-Constraints formuliert wurden, sondern eine RSA-eigene Implementierung, sogenannte Java-Constraints verwendet wurden.

Ein weiteres praktisches Problem, das sich im Zusammenhang mit der OCL als Pointcutsprache herausgestellt hat, ist dass der primitive OCL-Typ String über keine Operation verfügt, die das Ermitteln von Substrings erlaubt. Eine solche Operation ist aber zwingend nötig, um Namensmuster auswerten zu können. Die OCL-Implementierung des RSA wurde aus diesem Grund um eine solche Operation für den Typ String erweitert. Aber auch dabei haben sich unerwartete Probleme ergeben. Alle OCL-Ausdrücke, welche die neu erstellte Operation benutzen sollten, wurden deshalb nicht eingesetzt. Die Namensmuster wurden deshalb erst im Nachhinein ausgewertet.

Es lässt sich also zusammenfassen.

- Die OCL kann nur als Pointcutsprache für UML-Modelle eines Softwaresystems eingesetzt werden, nicht aber für deren Implementation,
- die OCL kann nur dann Aussagen über Dynamik formulieren, wenn das UML-Modell, auf dass sie sich bezieht, auch dynamische Elemente enthält. Das ist keine Schwäche der OCL, muss aber bedacht werden, wenn man OCL-Ausdrücke über dynamische Systemeigenschaften formulieren möchte. Und
- die Ausdrucksmöglichkeiten der OCL hängen von der konkreten OCL-Implementation die verwendet wird ab.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurden existierende Ansätze zur Modellierung von aspektorientierten Systemen untersucht. Dabei hat sich herausgestellt, dass den untersuchten Ansätzen in Bezug auf die Modellierung von Pointcuts wichtige Ausdrucksmöglichkeiten fehlen, dass sich keine wiederverwendbaren Pointcuts mit ihnen modellieren lassen oder sie plattformabhängig sind. Daraus wurde die Zielsetzung abgeleitet, eine plattformunabhängige Modellierungsnotation zu erstellen, mit der wiederverwendbare Pointcuts modelliert werden können, die genug Informationen enthalten, um aus ihnen Implementationen in konkreten aspektorientierten Programmiersprachen ableiten zu können.

Die Modellierungsnotatin wurde in Form eines UML-Profils realisiert. Um die praktische Einsetzbarkeit dieses UML-Profils nachzuweisen, wurde es als Erweiterung des UML-CASE-Werkzeuges IBM Rational Software Architect (RSA) implementiert und ein Beispielmodell vorgestellt und ausgewertet.

Es wurde zudem eine Vorgehensweise vorgestellt, wie man anhand modellierter Pointcuts Join-Point-Shadows auf Modellebene ermitteln kann. Dadurch kann beim Entwurf bereits geprüft werden, ob die statischen Informationsbestandteile eines modellierten Pointcuts die beabsichtigte Elemente im Basis-Softwaresystem beschreiben. Diese Auswertungsfunktion wurde ebenfalls als Erweiterung des RSA implementiert.

Wie die Machbarkeitsstudie gezeigt hat, ist die plattformunabhängige Modellierung wiederverwendbarer Pointcuts mithilfe des vorgestellten UML-Profils möglich. Es hat sich allerdings auch herausgestellt, dass der Umgang mit einem UML-Profil teilweise unkomfortabel ist, weil Tagged-Values in Diagrammen nicht sichtbar sind und Stereotypen immer von Extionspunkten abhängen. Auch ist die Formulierung von OCL-Constraints für Stereotypen

in dem UML-Profil, die Aussagen über andere Stereotypen formulieren sollen, unmöglich. In Bezug auf die OCL hat sich auch herausgestellt, dass sie nur bedingt als Pointcutsprache angesehen werden kann und dass ihre Verwendung als Modellabfragesprache ihre Erweiterung um zusätzliche Operationen erfordert. Eine solche Erweiterung hat sich im Rahmen der hier vorgestellten Implementation zudem als problematisch herausgestellt.

Aufgrund der Ergebnisse der Fallstudie im Umgang mit dem UML-Profil und mit der Auswertungsfunktion für Pointcuts und in Hinsicht auf ihre Verwendung im Rahmen des MDA-Ansatzes, ergeben sich schliesslich mehrere interessante Anknüpfpunkte für weiterführende Arbeiten, die im Folgenden genannt werden sollen:

Um die in dieser Arbeit beschriebenen Probleme mit dem Umgang mit dem UML-Profil zu lösen, wäre es denkbar die Modellierungsnotation als Erweiterung des UML-Metamodells für das verwendete UML-CASE-Werkzeug zu implementieren. Die Stereotypen des vorgestellten UML-Profils liessen sich ebenso als Metaklassen eines UML-Metamodells interpretieren. Die Implementatierung eines solchen UML-Metamodells für ein konkretes UML-CASE-Werkzeug wie den RSA ist allerdings weitaus umständlicher als die Implementierung eines UML-Profils.

Bleibt man bei bei dem Einsatz eines UML-Profils, so kann man das Problem, dass Tagged-Values in Diagrammen nicht sichtbar sind, lösen, indem man das verwendete UML-CASE-Werkzeug um zusätzliche Visualisierungsfunktionen für Tagged-Values von Stereotypen erweitert.

Eine sehr interessante, mögliche Ergänzung der vorgestellten Modellierungsnotation, wäre ihre Ergänzung um dynamische Modellelemente. In dem vorgestellten UML-Profil gibt es nur Stereotypen für Modellelemente aus statischen Modellen, die typischerweise in UML-Klassendiagrammen visualisiert werden. Um dynamische Informationen modellieren zu können, müsste das UML-Metamodell genauer auf seine dynamischen Elemente hin untersucht und die Modellierungsnotation entsprechend um weitere Stereotypen für dynamische UML-Metamodellelemente erweitert werden.

Betrachtet man die hier vorgestellte Arbeit im Kontext des MDA-Ansatzes, wäre beispielsweise eine Transformation eines Modells, das mithilfe der vor-

gestellten Modellierungsnotation erstellt wurde, in ein plattformspezifisches Modell, zum Beispiel in ein Modell das speziell auf AspectJ ausgerichtet ist, denkbar. Für ein plattformspezifisches Modell könnte zudem ein Transformator implementiert werden, der die Generierung konkreten Programmcodes ermöglicht. Auch ist es denkbar, mithilfe von Transformatoren einen Weaving-Mechanismus auf plattformunabhängiger Modellebene zu realisieren, der Aspekte in ein nicht-aspektorientiertes Modell einwebt und ein verfeinertes nicht-aspektorientiertes Modell liefert.

In der Machbarkeitsstudie wurde erwähnt, dass es keine Join-Point-Kinds für Anweisungen innerhalb von Ausführungsblöcken gibt, weil die UML solche Informationen nicht zur Verfügung stellt. Es wäre allerdings denkbar, mithilfe einer Profilerweiterung die Anreichungen von Operationen um Informationen, welche Anweisungen sie enthalten, zu erzielen. Mithilfe von Reverse-Engineering-Methoden wäre es vielleicht möglich, diese Informationen aus der Implementation eines Modells auszulesen und das Modell rückwirkend um diese zu bereichern.

Zusammenfassend ergeben sich also die folgenden Vorschläge für weiterführende Arbeiten:

- Das UML-CASE-Werkzeug könnte um die Funktionalität, Tagged-Values von Stereotypen in UML-Diagrammen zu visualisieren, erweitert werden,
- anstatt eines UML-Profils könnte man die Modellierungsnotation auch als UML-Metamodell definieren und implementieren, um den Umgang zu erleichtern,
- die Modellierungsnotation könnte um die Möglichkeit ergänzt werden, Pointcuts auch mithilfe dynamischer UML-Modellbestandteile zu modellieren,
- es könnten Transformatoren für die Umsetzung eines mit dem vorgestellten UML-Profils erstellen Modells in ein plattformspezifisches Modell und für die Umsetzung eines plattformspezifischen Modells in Programmcode implementiert werden,
- es könnten Transformatoren implementiert werden, die aspektorientierte Elemente aus einem mit dem vorgestellten UML-Profil erstellten Mo-

dell derart in die nicht-aspektorientierten Modellbestandteile einweben, dass sich daraus wieder ein plattformunabhängiges, rein objektorientiertes Modell ergibt und

- mithilfe von Reverse-Engineering-Methoden und einer geeigneten Erweiterung des vorgestellten UML-Profils, wäre es vielleicht möglich, auch Join-Point-Kinds für Anweisungen in Ausführungsblöcken zu ermöglichen, die dann auch auf Modellebene ausgewertet werden könnten.

Das Ziel der vorliegenden Arbeit, eine neue Modellierungsnotation für die Modellierung aspektorientierter Systeme, die eine plattformunabhängige Modellierung wiederverwendbarer Pointcuts ermöglicht, bereit zu stellen, ist erreicht. Dennoch stellte sich heraus, dass der Umgang mit dem vorgestellten UML-Profil teilweise nicht zufriedenstellend ist. Die Verwendung einer Implementierung eines UML-Metamodells anstatt eines UML-Profils erscheint für die Praxis komfortabler zu sein. Die Implementierung eines solchen Metamodells erfordert allerdings größeren Aufwand bei der Einbettung in ein konkretes UML-CASE-Werkzeug. Der Umgang mit der OCL als Modellabfragesprache unterliegt mehr Einschränkungen als zunächst vermutet. Die Möglichkeiten ihres Einsatzes als Modellabfragesprache hängt stark von den Gegebenheiten der verwendeten Implementationen der OCL selbst ab. Die OCL sollte deshalb und auch aus weiteren aufgeführten Gründen nicht uneingeschränkt als Pointcutsprache verstanden werden.

Es stellt sich also heraus, dass hinsichtlich der in dieser Arbeit vorgestellten Thematik noch akuter Handlungsbedarf besteht. Es konnten diesbezüglich eine Vielzahl weiterführender Arbeiten aufgezeigt werden.

Literaturverzeichnis

[1] Aspectj. Eclipse Foundation. www.eclipse.org/aspectj.

[2] Aspectwerkz. aspectwerkz.codehaus.org.

[3] Kai Boellert, Ilka Philippow, and Matthias Riebisch. The hyper/uml approach for feature based software design. In *4th International Workshop on Aspect-Oriented Modeling*, 2003.

[4] Christina Chavez and Carlos Lucena. A metamodel for aspect-oriented modeling. In *1st International Workshop on Aspect-Oriented Modeling*, 2002.

[5] Eclipse development platform. Eclipse Foundation. www.eclipse.org.

[6] Eclipse modeling framework. Eclipse Foundation. www.eclipse.org/emf.

[7] Tzilla Elrad, Omar Aldawud, and Atef Bader. Aspect-oriented modeling: Bridging the gap between implementation and design. In *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, Pittsburgh, PA, USA, Proceedings*, 2002.

[8] Tzilla Elrad, Omar Aldawud, and Atef Bader. A uml profile for aspect oriented modeling. In *3rd International Workshop on Aspect-Oriented Modeling*, 2003.

[9] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[10] Emf-based uml 2 metamodel implementation. Eclipse Foundation. www.eclipse.org/uml2.

[11] M.E. Fayad and Anita Ranganath. Modeling aspects using software stability and uml. In *4th International Workshop on Aspect-Oriented Modeling*, 2003.

[12] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000.

[13] Iris Groher and Stefan Schulze. Generating aspect code from uml models. In *3rd International Workshop on Aspect-Oriented Modeling*, 2003.

[14] Kris Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In *Second Workshop on Aspect-Oriented Software Development, Bonn, Germany, 2002*, 2002.

[15] Yan Han, Günter Kniesel, and Armin B. Cremers. A meta model for aspectj. Technical report iai-tr-2004-3, Computer Science Department III, University of Bonn, 10 2004.

[16] Stephan Herrmann. Objectteams. www.objectteams.org.

[17] Stephan Herrmann. Composable designs with ufa. In *1st International Workshop on Aspect-Oriented Modeling*, 2002.

[18] Stephan Herrmann. Are pointcuts a first-class language feature? In *Foundations of Aspect-Oriented Languages 2006*, 2006.

[19] Jboss aop. The JBoss Open Source Federation. www.jboss.org/products/aop.

[20] Jean-Marc Jézéquel, Noël Plouzeau, Torben Weis, and Kurt Geihs. From contracts to aspects in uml designs. In *1st International Workshop on Aspect-Oriented Modeling*, 2002.

[21] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[22] Jean Marie Lions, Didier Simoneau, Gilles Pitette, and Imed Moussa. Extending opentool/uml using metamodeling: An aspect-oriented programming case study. In *2nd International Workshop on Aspect-Oriented Modeling*, 2002.

[23] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs, 2002.

[24] Meta-object facility. Object Management Group (OMG). www.omg.org/technology/cwm.

[25] Model driven architecture. Object Management Group (OMG). www.omg.org/mda.

[26] Object management group. www.omg.org.

[27] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[28] Johan Ovlinger. From aspect-orientied model to implementation. In *3rd International Workshop on Aspect-Oriented Modeling*, 2003.

[29] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. A metamodel for aspect-oriented modeling. In *1st International Workshop on Aspect-Oriented Modeling*, 2002.

[30] Rational software architect. Internationel Business Machines (IBM). www-306.ibm.com/software/awdtools/architect/swarchitect.

[31] Rational unified process. Rational Software Corporation. www-306.ibm.com/software/awdtools/rup.

[32] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Query models. In *Seventh International Conference on the Unified Modeling Language - the Language and its applications*, 2004.

[33] Peri Tarr, Harold Ossher, and Johannes Henkel. Visualization as an aid to compositional software engineering. In *Workshop on Advanced Separation of Concerns, OOPSLA 2001*, 2001.

[34] Unified modeling language. Object Management Group (OMG). www.uml.org.

[35] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 2004.

[36] Tobias Windeln. Logicaj - eine erweiterung von aspectj um logische meta-programmierung. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2003.

[37] Aida Atef Zakaria, Hoda Hosny, and Amir Zeid. A uml extension for modeling aspect-oriented systems. In *2nd International Workshop on Aspect-Oriented Modeling*, 2002.