AspectJ - AOP für Java

Georg Blaschke (georg.blaschke@stud.informatik.uni-erlangen.de)

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Betriebssysteme und Verteilte Systeme

3. Mai 2003

Aspekt-Orientierte Programmierung, ist eine relativ junge Programmiertechnik, für welche es momentan noch wenig Unterstützung seitens der Programmiersprachen gibt. Eine dieser wenigen Programmiersprachen, die den aspekt-orientierten Ansatz implementieren, ist AspectJ.

1 Motivation und Einleitung

Mit herkömmlichen Programmiermethoden (OOP, POP,...) ist es schwierig Eigenschaften, die sich durch das gesamten Softwaresystem hindurchziehen, getrennt von der eigentlichen Funktionalität der Softwarekomponenten zu modellieren und zu implementieren. Diese Eigenschaften – auch *crosscutting concerns* genannt – wirken in der Praxis häufig an vielen Stellen im Quellcode und sind somit umständlich zu warten. Eine Lösung bietet hierbei die Aspekt-orientierte Programmierung (AOP), die eine Trennung des Aspekt-Codes vom Komponenten-Code in einem Software-System ermöglicht. Ein Aspekt kapselt crosscutting concerns ausserhalb einer herkömmlichen Software-Komponente. Die Trennung des Aspekt-Codes kann auf vielerlei Weisen geschehen. Eine Möglichkeit wird mit AspectJ in dieser Arbeit vorgestellt. AspectJ ist eine Erweiterung der Programmiersprache Java und unterstützt den AOP Ansatz durch neue Konzepte und Sprachkonstrukte.
Zunächst wird auf die grundlegende Syntax und Semantik der zusätzlichen Sprach-Elementen von AspectJ eingegangen, worauf dann abschließend kurz der praktische Einsatz von AspectJ behandelt wird.
Um ein besseres, allgemeines Verständnis für AOP zu erlangen, welches hier vorausgetzt wird, stellt *Aspect-Oriented Programming* [1] eine sehr empfehlenswerte Lektüre dar.

2 Die Erweiterung von Java zu AspectJ

Um die Programmierung von Aspekt-Code zu unterstützen, bietet AspektJ einige neue Sprach-Konstrukte, die in Java selbst nicht zu finden sind. In den folgenden Unterkapiteln wird ein Überblick über einige diese Konstrukte gegeben.

2.1 Pointcut

Wenn man Aspekt-Code mit Komponenten-Code zusammenführen (*weben*) will, muss in irgendeiner Weise beschrieben werden, an welchen Stellen der Aspekt-Code zur Geltung kommen soll. Zu diesem Zweck gibt es sogenannte *Pointcuts*, die einen oder auch mehrere *Join Points* in einem Programm identifizieren. Joint Points sind die Stellen im Komponenten-Code, an denen der Aspekt-Code mit dem Komponenten-Code verwebt wird (siehe auch [1]) . Zum Beispiel identifiziert dieser Pointcut

```
execution(void ChatClient.send(String))
```

die Ausführung der Methode send(String) einer Instanz der Klasse ChatClient. Das eben gezeigte Beispiel ist einer von vielen *primitiven* Pointcuts; d.h. er besteht aus nur einer Anweisung (**execution**()). Weitere primitive Pointcuts sind zum Beispiel **args**() , **call** () , **target**() ,... .
Eine komplette Übersicht aller verfügbaren primitiven Pointcuts und ihre Beschreibung findet man in [2]. Wie das Attribut „primitiv“ schon vermuten läßt, gibt es auch nicht-primitive Pointcuts, die durch Komposition von mehreren Pointcuts generiert werden. Hier ein Beispiel, das aus der „Veroderung“ zweier primitiver Pointcuts besteht:

```
execution(void ChatClient.send(String)) ||
execution(void ChatServer.send(String))
```

Es ist auch möglich Pointcuts mit Namen zu versehen, sie also wie Variablen zu deklarieren. Diese Anweisung

```
pointcut send():execution(void ChatClient.send(String)) ||
               execution(void ChatServer.send(String)) ;
```

deklariert den Pointcut send(). Mit den primitiven Pointcuts **this**() , **target**() und **args**() ist es auch möglich Werte von Variablen weiter zu reichen, wie es in einem späteren Beispiel illustriert wird.

Mit den bisher gezeigten Möglichkteiten, kann man bereits gezielt verschiedene Komponenten verteilte Stellen im Softwaresystem ansprechen (crosscutting concerns !)

Möchte man jedoch mit einem Pointcut alle Methoden einer bestimmten Klasse identifizieren, ist es etwas mühsam jede einzelne Methode explizit aufzuführen.

Deshalb bietet AspectJ auch die Möglichkeit, bei der Definition der Pointcuts Wildcard-Pattern einzusetzen. So sieht ein Pointcut, der die Ausführung aller Methoden der Klasse ChatClient identifiziert, folgendermaßen aus:

```
execution(* ChatClient .* (..) )
```

2.2 Advice

Ein *Advice* definiert, welcher Code vor, nach oder anstelle eines Pointcuts ausgeführt werden soll. So wird ein *Before Advice* ausgeführt bevor ein Pointcut erreicht wird und ein *After Advice* nachdem der Code, der durch den Pointcut identifiziert wird, abgeschlossen ist. Ein kurzes Beispiel:

```
pointcut myPointcut(): execution(* ChatClient .* (..) );

before():myPointcut(){
  System.out. println ("Eine Methode der Klasse ChatClient wird gleich
      ausgefuehrt");
}

after():myPointcut(){
  System.out. println ("Eine Methode der Klasse ChatClient wurde ausgefuehrt")
      ;
}
```

Natürlich könnte man in diesem Beispiel auch die Deklaration des Pointcuts myPointcut() weglassen, und den primitiven Pointcut direkt hinter den jeweiligen Advice setzen; also:

```
before():execution(* ChatClient .* (..) ){
  /* ... */
}
```

Die dritte Klasse von Advices stellt das *Around Advice* dar und es wird **anstelle** des durch den Pointcut identifizierten Codes ausgefürt. Da hierbei unter Umständen eine Methode vollkommen ersetzt werden kann, muss immer ein Rückgabewert spezifiert werden. Zum Beispiel:

```
void around(String string):execution(void ChatClient.send(String))&&
                           args(string){
  System.out. println ("Die Zeichenkette: "+string+" soll gesendet werden.");
}
```

2.3 Introduction

Mit den Sprachkonstrukten, die durch Advices und Pointcuts zu Verfügung gestellt werden, ist es nun möglich das Ablauf-Verhalten eines Programms zu beein-

flussen. Will man jedoch eine Komponente statisch verändern, d.h. neue Variablen oder Methoden einführen, oder die Klassenhierachie modifizieren, ist das mit den bisherigen Techniken nicht möglich.
Deshalb gibt es das Konzept der *Introduction* welches die eben genannten Modifikationen ermöglicht.
Es folgen nun einige Beispiele wie statische Veränderungen realisiert werden könnten:

- ```
  declare parents: ChatClient implements Observer;
  ```

  Mit der Anweisung **declare** parents: verändert man die Beziehungen zwischen Datentypen.

- ```
  private Subject ChatClient.chatServer;
  ```

  Der Klasse ChatClient wird die private Membervariable chatServer hinzugefügt, welche vom Typ Subject ist.

- ```
  public void ChatClient.update(String s){
          this.display(s);
  }
  ```

  Der Klasse ChatClient wird die öffentliche Methode update hinzugefügt. Achtung! Die Referenz **this** bezieht sich hierbei auf die jeweilige Instanz der Klasse ChatClient. Bei Advice-Code muss die instanz immer explizit angegeben werden, welche mit dem primitiven Pointcut **target**() ermittelt werden kann. Siehe dazu [2].

Anzumerken bleibt noch, dass sich jede statische Veränderung einer Klasse auf alle von ihr abgeleiteten Klassen auswirkt, da Modifikationen mit Introdutions beim Kompilieren berücksichtigt werden.

2.4 Aspect

Jetzt bleibt nur noch die Frage, wo man als Programmierer Introductions, Pointcuts und Advices überhaupt hinschreiben soll. Dafür gibt es in AspectJ den Datentyp **aspect**. Dieser kann –genau wie **class**– abgeleitet werden, ein Interface implementieren oder auch mit Hilfe des Schlüsselworts **abstract** einen einen abstrakten Datentyp erzeugen. Die Deklaration eines aspect-Datentyps könnte also folgendermaßen aussehen:

```
public aspect Tracing{
  after():execution(* ChatClient.* (..) ){
    System.out.println("--> "+ thisJoinPointStaticPart.getSignature() );
  }
}
```
```

Dieses Beispiel zeigt eine simple Implementierung des Aspekts *Tracing*, wobei hierbei mit Hilfe des speziellen Ausducks **thisJoinPointStaticPart**.getSignature() die Methoden-Signatur ausgegeben wird. Genauere Information zu solchen Ausdrücken findet man in [2].

3 AspectJ in der Praxis

Da nun die grundlegende Syntax klar ist, wird noch kurz auf die praktische Anwendung von AspecJ eingegangen.

3.1 Compiler und Tools

Die in Kapitel 2 vorgestellten Spracherweiterungen werden vom Java-Compiler(javac) natürlich nicht unterstützt, weswegen ein spezieller Compiler(ajc) benötigt wird, um AspectJ-Code zu übersetzen. Nach der Übersetzung jedoch, kann das Programm in der standardmäsigen Java-Laufzeitumgebung gestartet werden.
Die Handhabung des AspectJ-Compilers ist, von einigen Optionen abgesehen nahezu identisch mit der des Standard-Java-Compilers; was bedeutet, dass er die Quelldateien übersetzt, die ihm als Argumente übergeben wurden. Um einen Aspekt auf ein Software-System anzuwenden ist es also ausreichend den Aspect-Code zusammen mit dem Komponenten-Code zu kompilieren. In den folgenden Beispielen befindet sich der Aspekt-Code in der Datei `Tracing.java`.

1. Übersetung der Klasse `ChatClient` **mit** dem Aspekt *Tracing*

```
$ ajc chat/client/ChatClient.java chat/Tracing.java
```

2. Übersetung der Klasse `ChatClient` **ohne** Aspekte

```
$ ajc chat/client/ChatClient.java
```

Wie beim Java-Compiler läßt sich die Übergabe der Quelldatei-Argumente in Form einer Textdatei bewerkstelligen, was die Anwendung von Aspekten sehr übersichtlich machen kann. Dabei werden dann zeilenweise die verwendeten Quell-Dateien aufgeführt.
Eine solche Textdatei(*build configuration file*) ist sogar erforderlich, wenn der AspectJ-Browser(ajbrowser) verwendet wird, da er als Argument eben jene erwartet. Der AspectJ-Browser wird standardmäßig mit dem AspectJ-Compiler ausgeliefert. Mit Hilfe einer interaktiven grafischen Benutzeroberfläche ermöglicht er dem Programmierer die Auswirkung von Aspket-Code auf den Komponenten-Code anschaulich darzustellen.

Einige integrierte Entwicklungsumgebungen unterstützen bereits die Programmierung von Aspekten unter Java mit AspectJ. Zu ihnen gehören unter anderem Eclipse[1] und der JBuilder[2]. Ohne Unterstützung einer Entwicklungsumgebung ist es relativ schwierig die Auswirkungen von Aspekt-Code im Überblick zu behalten, weswegen der Aufwand der Programmierung mit AspectJ direkt von den Entwicklungswerkzeugen abhängt.

3.2 Aspekte als Unterstützung bei der Sofware-Entwicklung

Wie weiter oben schon angedeutet kann AspectJ sehr hilfreich bei der Entwicklung von Software sein. Dabei muss das Gesamtprojekt nicht einmal im Sinne des aspekt-orientierten Ansatzes geplant sein. Zum Beispiel würde mit herkömmlichen Java die Programmierung von Tracing(verfolgen von Funktionsaufrufen) für alle Methoden einer Klasse in viel Tipparbeit ausarten, da man in jedem Methodenrumpf ein `System.out.println()` benötigen würde.
Mit AspectJ braucht man dafür nur wenige Zeilen.
Aber neben Tracing sind in AspectJ auch andere Techniken wie Profiling oder Logging ohne großen Aufwand als Aspekte realisierbar.

3.3 Aspekte als Grundlage für das Sofware-Desing

Da AspectJ entwickelt wurde um AOP für Java zu ermöglichen, lassen sich auch die Vorteile von AOP unter AspectJ ausnutzen. Durch den Einsatz von Aspekten lassen sich Komponenten-übergreifende Eigenschaften kapseln, was die einzelnen Komponenten in Hinblick auf ihre Wiederverwendbarkeit,Wartbarkeit und Erweiterbakeit positiv beeinflusst, da sie ja nun von „fremdartigen“ Code befreit sind.

Literatur

[1] Gregor Kiczales, et al., *Aspect-Oriented Programming*, Springer-Verlag LNCS 1241, 1997.

[2] The AspectJ Team, *The AspectJ Programming Guide*, Xerox Cooperation, 2002.

[1] http://www.eclipse.org/
[2] http://www.borland.com/