SAARLAND UNIVERSITY
COMPUTER SCIENCE

Proseminar "13 Wochen - 13 Sprachen"

Prof. Dr. Andreas Zeller, Jun.-Prof. Dr. Sebastian Hack
Kontakt: Martin Burger, Christoph Mallon

13. Aufgabenblatt – AspectJ

Ausgabe: 01. Juli 2009 Abgabe: 14. Juli 2009 Revision: 3396

1 AspectJ: Aspektorientierte Programmierung

Manche Funktionen eines Systems lassen sich schwer an einem einzelnen Ort zusammenfassen. Ein typisches Beispiel sind *Logging*-Ausgaben, die typischerweise kreuz und quer über den Code verstreut sind – aber auch Konsistenzprüfungen für bestimmte Objekte, die an allen Orten stattfinden müssen, wo diese Objekte benutzt werden.

Die Aspektorientierte Programmierung versucht, dieses Problem mit Hilfe sogenannter *Aspekte* zu lösen. Ein Aspekt ist eine Sammlung von Codestücken (sog. *Advices*), die an zahlreichen Stellen (sog. *Joinpoints*) ins Programm eingeflochten werden. So kann man etwa einmal eine Logging-Anweisung definieren, und diese dann an zahlreichen Stellen anwenden. Die populärste aspektorientierte Sprache ist *AspectJ*, eine aspektorientierte Erweiterung von Java.[1]

2 Ein einfacher Aspekt

Als Beispiel betrachten wir zunächst eine einfache Klassenhierarchie in Java, definiert in der Datei `Test.java`:

```
class Figure {
    public void setXY(int x, int y) {}
}

class Point extends Figure {
    public void setX(int x) {}
    public void setY(int y) {}
}

class Line extends Figure {
    public void setP1(Point p1) {}
    public void setP2(Point p2) {}
}
```

[1] Unter `http://www.eclipse.org/aspectj/` finden Sie alles über AspectJ. Betrachten Sie insbesondere die Einführung unter "Docs → Getting Started".

```
class Test {
    static public void main(String[] args) {
        System.out.println("start");

        Point p1 = new Point();
        p1.setX(10);
        p1.setY(20);

        Point p2 = new Point();
        p2.setXY(30, 30);

        Line l = new Line();
        l.setP1(p1);
        l.setP2(p2);

        System.out.println("done");
    }
}
```

Zur Übersetzung finden Sie in den CIP-Pools den `AspectJ`-Compiler (s. Abschnitt 6). Rufen Sie den AspectJ-Compiler auf mit

```
$ ajc Test.java
```

und rufen Sie das Programm auf mit

```
$ java Test
```

Um den Lauf zu verfolgen, wollen wir nun einen Aspekt hinzufügen. Die Datei `Logging.aj` definiert einen Aspekt zum Logging der Bewegungs-Operationen. Sie besteht aus

- einer Aspekt-Deklaration (`aspect`), die dem Aspekt seinen Namen gibt;
- einem Pointcut (`pointcut`), der die Stellen (*join points)* definiert, an der der Advice eingepflegt wird;
- dem Advice – also dem Codestück, das an verschiedenen Stellen zum Einsatz kommen soll:

```
aspect Logging {
  // Der Pointcut 'move' umfasst alle Methoden,
  // in denen eine Figur bewegt wird:
  pointcut move():
    call(void Figure.setXY(int,int)) ||
    call(void Point.setX(int))       ||
    call(void Point.setY(int))       ||
```

```
    call(void Line.setP1(Point))     ||
    call(void Line.setP2(Point));

  // Nachdem eine dieser Methoden zurückkehrt,
  // soll eine Ausgabe erfolgen:
  after() returning: move() {
    System.out.println("just moved");
  }
}
```

Um das `Test`-Programm zusammen mit dem eingewebten `Logging`-Aspekt zu übersetzen, rufen Sie den AspectJ-Compiler auf mit

```
$ ajc Test.java Logging.aj
```

und rufen Sie das Programm auf mit

```
$ java Test
```

Wie verändert sich jetzt die Ausgabe?

3 Ihre Aufgabe

Ihre Aufgaben drehen sich rund um die oben beschriebene Klassenhierarchie:

1. Das explizite Aufzählen der Methoden, in denen der Advice ausgeführt werden soll, ist auf Dauer lästig. Definieren Sie den `move`-pointcut so um, dass er mit Hilfe eines *Patterns* alle Methoden abfängt, deren Name mit `set` beginnt.

2. Geben Sie einen zweiten Advice an, der *vor* Aufruf einer `set`-Methode "about to move" ausgibt.

3. Definieren Sie einen zusätzlichen Aspekt `DisplayUpdate`, der bei jedem Aufruf einer `set`-Methode ein Flag `must_update_display` setzt; die Methode `update_display()` löscht es wieder.

4. Definieren Sie einen Aspekt `Checking`, der bei allen `set`-Methoden, die eine `int`-Koordinate erwarten, prüft, ob die angegebene Koordinate nicht-negativ ist; falls nicht, soll sie eine *Exception* werfen.

5. Wenn Sie alle drei Aspekte `DisplayUpdate`, `Logging` *und* `Checking` einbinden: In welcher Reihenfolge werden die Aspekte ausgeführt? Können Sie dies kontrollieren?

6. (Für Enthusiasten) Definieren Sie einen Aspekt, der die `Figure.setXY()`-Methode durch eine komplett andere Methode ersetzt, ohne die ursprüngliche Methode aufzurufen.

4 Diskussion

Diskutieren Sie folgende Fragestellungen auf etwa einer DIN A4 Seite:

1. Welche Vorteile bringt Aspekt-Orientierung? Welche Nachteile sind damit verbunden?
2. Sollten Aspekte sich auf *optionale* Programmfunktionalitäten beschränken, oder sollten große Systeme von vorneherein in einzelne Aspekte zerlegt werden?

5 Abgabe Ihrer Lösung

Drucken Sie Ihre Lösung aus und werfen Sie den Ausdruck bis zum **14. Juli 2009 um 9:00 Uhr** in den Briefkasten des Lehrstuhls für Softwaretechnik[2]. Vergessen Sie dabei nicht, Ihren Ausdruck mit Ihrem Namen und Ihrer Matrikelnummer zu versehen.

Bringen Sie bitte zum Proseminar ebenfalls einen Ausdruck mit, so dass Sie ihn bei etwaigen Diskussionen vorliegen haben.

6 Links und Hinweise

Zum Erstellen und Bearbeiten Ihres Programms können Sie einen gewöhnlichen Texteditor benutzen. Der AspectJ-Compiler ist (optionaler) Bestandteil vieler Open Source-Distributionen. Im CIP-Pool müssen Sie zunächst zwei Umgebungsvariablen setzen, um ihn aufzurufen:

```
$ export PATH=/installer/import/linux/compilers/aspectj/bin:$PATH
$ export CLASSPATH=/installer/import/linux/compilers/aspectj/lib/\
  aspectjrt.jar:$CLASSPATH
```

Danach können Sie den Compiler aufrufen mit

```
$ ajc Program.java Aspect1.aj Aspect2.aj ...
```

Das entstehende Programm können Sie dann so ausführen:

```
$ java Klasse
```

Mit Hilfe der Standard-Ausgabe `System.out` können Sie den Ablauf Ihres Programms verfolgen.

[2] Gebäude E1 1, neben dem InfoPoint des Rechenzentrums.