

Javassist

Java Programming Assistant



Seminararbeit zum Thema AOP
von Christian Menz

Inhaltsverzeichnis

Voraussetzungen.....	3
Ziel.....	3
Einführung.....	3
Download und Installation.....	4
Beispiel Applikation.....	4
Wichtige Klassen.....	5
ClassPool.....	5
CtClass.....	5
CtMethod.....	5
CtField.....	5
Eine neue Klasse erstellen.....	5
Bestehende Klassen manipulieren.....	6
Der ClassPool.....	6
Anwendungsbereich AOP.....	7
Before und After.....	7
Add Catch.....	9
Auf Parameter zugreifen.....	10
Mit Expressions arbeiten.....	11
Low Level API.....	12
GluonJ.....	13
Fazit und persönliche Meinung.....	13
Weiterführende Informationen.....	14

Voraussetzungen

- Gutes Java Verständnis ist vorhanden
- Kenntnisse des Java Reflection API von Vorteil
- Grundverständnis und Grundbegriffe von AOP sind bekannt

Ziel

Ziel dieser Arbeit ist es, die frei verfügbare Bibliothek Javassist vorzustellen und aufzuzeigen, wie damit AOP betrieben werden kann. Der Leser dieser Einführung sollte nach gründlichem Studium in der Lage sein, selber einfache Manipulationen mit Javassist durchzuführen.

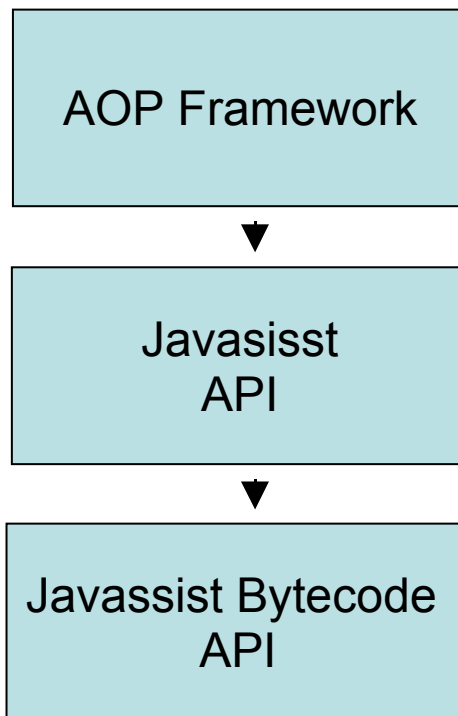
Ausserdem soll dieses Dokument frische Ansätze zur Lösung von immer wieder auftauchenden Problemen in der Praxis liefern.

Einführung

Javassist ist eine kleine aber feine Bibliothek, welche es ermöglicht Java Bytecode zu manipulieren. Das spezielle an Javassist ist, dass man eine Klasse unter anderem zur Laufzeit verändern kann. Mit Javassist kann man z.B. zur Laufzeit eine bestehende Klasse ändern (unter anderem für AOP) und neue Klassen erzeugen. Das API ist leicht verständlich, wie wir an diversen Beispielen sehen werden.

Das Thema der Seminararbeit ist AOP, allerdings muss vorweggenommen werden, dass Javassist kein AOP Framework ist. Es ist wie bereits erwähnt eine Bibliothek, um zur Laufzeit Java Bytecode nach unseren Wünschen anzupassen.

Das folgende Bild sollte verdeutlichen, wo Javassist eingeordnet werden kann:



Das API bietet sowohl die Möglichkeit auf der Ebene von Klassen, Methoden und Attributen zu arbeiten als auch direkt mit dem Maschinencode.

Doch bevor wir uns zu sehr in Details verlieren, schauen wir uns ein erstes Beispiel an, um ein Gefühl dafür zu bekommen, was alles möglich ist.

Download und Installation

Javassist ist eine sehr kleine Bibliothek, um sie zu verwenden muss lediglich ein einzige JAR in den Klassenpfad hinzugefügt werden. Die offizielle Website von Javassist findet man unter der folgenden Adresse:

<http://www.csg.is.titech.ac.jp/~chiba/javassist/>

Die Installation ist denkbar einfach!

1. ZIP entpacken
2. javassist.jar in den Klassenpfad aufnehmen

Beispiel Applikation

Ein Beispielprogramm (Java Quelltext) zum selber experimentieren liegt dieser Einführung bei.

Wichtige Klassen

ClassPool

Startpunkt für unsere Interaktionen mit dem API, es ist eigentlich nichts weiter als eine Sammlung von CtClass Objekten.

CtClass

Repräsentiert eine Java Klasse, welche verändert werden kann

CtMethod

Stellt im Kontext von Java eine normale Java Methode dar, welche dann z.B. an eine CtClass angehängt werden kann. Die Klasse CtMethod erlaubt uns diverse Manipulationen einer Methode wie insertBefore() und insertAfter().

CtField

In Java kennen wir Klassen, Methoden und Attribute. Die Klasse CtField repräsentiert dabei ein Attribut, welches auch manipuliert werden kann.



Die Klassen, mit denen man üblicherweise arbeitet, beginnen jeweils mit einem Ct, was für „compile time“ steht. Wenn man das Reflection API bereits kennt, geht die Umstellung ohne grossen Aufwand.

Eine neue Klasse erstellen

```
ClassPool cp = ClassPool.getDefault();  
CtClass cl = cp.makeClass("javassist.Hello");  
Class c = cl.toClass();  
Object o = c.newInstance();
```

Damit haben wir zur Laufzeit eine neue Klasse namens javassist.Hello erstellt und eine Instanz erzeugt. Einfach oder? Die neue Klasse kann nach bedarf auch im Dateisystem gespeichert werden und z.B. von einem anderen Programm als normale Java Klasse verwendet werden:

```
cl.writeFile();
```

Ist bisher noch nicht sonderlich spannend, aber man kann gut erkennen, wie einfach dieses API zu verwenden ist. Man kann nun im Prinzip über ein ähnlich einfaches API Felder (Attribute) und Methoden hinzufügen.



Siehe KlasseErstellen.java in der Beispielapplikation

Bestehende Klassen manipulieren

Anstatt eine Klasse neu zu definieren, ist es mit Javassist auch möglich, bestehende Klassen zu laden und zu manipulieren, was besonders für die Aspekt orientierte Programmierung (AOP) interessant ist, doch dazu später.

In dem folgenden Beispiel wird die Vererbungshierarchie über Javassist verändert.

```
ClassPool cp = ClassPool.getDefault();  
  
CtClass cl = cp.get("javassist.Main");  
CtClass superClass = cp.get("javax.swing.JFrame");  
  
cl.setSuperclass(superClass);  
cl.writeFile();
```

Hat es funktioniert? Dies können wir relativ einfach mit javap überprüfen:

```
C:\javassist>javap Main  
  
Compiled from "Main.java"  
public class javassist.Main extends javax.swing.JFrame{  
    public javassist.Main();  
    public static void main(java.lang.String[]) throws java.lang.Exception  
    ;  
}
```



Javap ist ein kleines Tool, welches üblicherweise mit dem JDK ausgeliefert wird. Es ist ein einfacher Disassembler für Java Klassen. Es ermöglicht so, eine kompilierte Klasse bis auf die Ebene von Opcodes zu analysieren.

Ziemlich cool oder? Und vor allem sehr einfach zu verwenden und zu verstehen. Ein Blick in das API lohnt sich, man entdeckt immer wieder interessante Methoden.



Siehe BestehendeKlasse.java in der Beispielapplikation

Der ClassPool

Sicher ist aufgefallen, dass wir die Klasse `ClassPool` bisher jeweils als Startpunkt für unsere Interaktionen mit dem API verwendet haben. Der `ClassPool` ist nichts weiter als eine Sammlung von `CtClasses`. Möchte man eine neue Klasse erstellen oder eine bestehende laden, so muss man dies über die Methoden des `ClassPools` bewerkstelligen. Üblicherweise sucht der `ClassPool` im Klassenpfad der aktuell laufenden VM nach bereits existierenden Klassen.

Anwendungsbereich AOP

Before und After

Mit Javassist kann man nicht nur die Vererbungshierarchie verändern, sondern auch bestehende Felder und Methoden manipulieren oder gar neu definieren. Für AOP ist es besonders interessant, dass man eine bestehende Methode auf einfache Art und Weise verändern kann. Betrachten wir ein einfaches Beispiel.

Nehmen wir an, wir möchten die Methode `sayHello()` der nachfolgenden Klasse dahingehend abändern, dass „Hello World“ anstelle von „Hello“ ausgegeben wird.

```
package javassist;

/**
 *
 * @author cme
 */
public class Hello {

    /** Creates a new instance of Hello */
    public Hello() {
    }

    public void sayHello() {
        System.out.println("Hello");
    }
}
```

Mit Javassist kann man nun die Methode `sayHello()` z.B. neu definieren, um somit zum Beispiel eine schnellere Implementierung bereitzustellen.

```
package javassist;

import java.io.IOException;

/**
 *
 * @author cme
 */
public class Main {
```

```
/** Creates a new instance of Main */  
public Main() {  
}  
  
/**  
 * @param args the command line arguments  
 */  
public static void main(String[] args) throws Exception {  
    ClassPool cp = ClassPool.getDefault();  
  
    CtClass cl = cp.get("javassist.Hello");  
    CtMethod method = cl.getDeclaredMethod("sayHello");  
  
    method.setBody("System.out.println(\"Hello World\");");  
  
    Class c = cl.toClass();  
  
    Hello hello = (Hello) c.newInstance();  
  
    hello.sayHello();  
}  
}
```

Hello World

Die Klasse CtMethod bietet Methoden, die uns aus dem Bereich AOP bekannt vorkommen dürften: insertBefore(), insertAfter() und addCatch()

```
ClassPool cp = ClassPool.getDefault();  
  
CtClass cl = cp.get("javassist.Hello");  
CtMethod method = cl.getDeclaredMethod("sayHello");  
  
method.insertBefore("System.out.println(\"This is a simple\");");  
method.insertAfter("System.out.println(\"World\");");  
  
Class c = cl.toClass();  
  
Hello hello = (Hello) c.newInstance();  
  
hello.sayHello();
```

This is a simple
Hello
World

Anhand des oben gezeigten Beispiels sollte deutlich werden, wie einfach gewisse Teile einer Applikation verändert werden können. Nur um Klarheit zu verschaffen, Javassist ist kein AOP Framework à la AspectJ. Es ist lediglich eine Bibliothek, welche es erlaubt Java Klassen zur Laufzeit zu manipulieren. Allerdings kann Javassist die Basis für ein AOP Framework bilden, wie wir später noch sehen werden.

Für einfache Aufgaben, für welche z.B. eine AspectJ Lösung zu schwergewichtig ist, kann Javassist allerdings sehr hilfreich sein.



Siehe BeforeAfter.java in der Beispielapplikation

Add Catch

Möchte man eine Methode mit einem try/catch Block versehen, bietet sich die Methode `addCatch()` an. Nehmen wir an wir hätten eine Klasse `BadCode` mit einer Methode `test()`. Diese Methode hat leider einen Bug, sie wirft dummerweise ständig `NumberFormatException`s. Mit `addCatch` können wir dieses Problem beheben:

```
package test;

import javassist.*;

/**
 *
 * @author cme
 */
public class BadCode {

    /** Creates a new instance of Hello */
    public BadCode() {
    }

    public void test() {
        int n = Integer.parseInt("hallo, das klappt nicht");
    }
}
```

Und so sieht unsere Lösung aus:

```
package test;

import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;

/**
 *
 * @author cme
 */
public class Main {

    /** Creates a new instance of Main */
    public Main() {
    }

    /**
     * @param args the command line arguments
     */
}
```

```

public static void main(String[] args) throws Exception {
    ClassPool cp = ClassPool.getDefault();

    CtClass cl = cp.get("test.BadCode");
    CtMethod method = cl.getDeclaredMethod("test");

    method.addCatch("return;", cp.get("java.lang.Exception"));

    Class c = cl.toClass();

    BadCode bc = (BadCode) c.newInstance();

    bc.test();
}
}

```

Lässt man das Programm nun laufen, kann man feststellen, dass die Ausnahme nun nicht mehr geworfen wird – wir haben dafür gesorgt, dass sie einfach ignoriert wird. Achtung, das ist keine gute Programmierpraxis und soll lediglich als Beispiel dienen.

Wichtig zu beachten ist bei addCatch(), dass der catch block entweder mit einem return oder throw Statement enden muss.

Möchte man auf die Ausnahme zugreifen, kann man dies über die spezielle Variable \$e tun. Wie das mit diesen Parametern funktioniert werden wir gleich sehen.

```

throw $e; // Beispiel

```

Auf Parameter zugreifen

Verändert man eine Methode, möchte man normalerweise auch auf die übergebenen Parameter zugreifen. Dies kann man über so genannte spezielle Variablen bewerkstelligen:

\$0,..\$n	Zeiger auf this und restliche Parameter
\$args	Array der Parameter (wobei [0] nicht this entspricht)
\$\$	Alle Parameter im Format \$1,\$2,\$3,..

Es gibt noch eine Reihe von weiteren speziellen Variablen, doch üblicherweise sind die oben genannten interessant.

Die ursprüngliche Methode:

```

public void test(int n) {
    System.out.println("n=" + n);
}

```

Erste Variante, wir setzen den Parameter n immer auf 0:

```
method.insertBefore("n=0;"); // wenn wir wissen, dass der Parameter n heisst
```

Zweite, bessere Variante (wir müssen den Namen des Parameters nicht kennen):

```
method.insertBefore("$1=0;"); // besser
```

Und das sollte bei einem Aufruf der Methode passieren:

Aufruf:
test.test(10);

Output:
n=0



Siehe AccessParam.java in der Beispielapplikation

Mit Expressions arbeiten

Ähnlich wie die Pointcuts in einem üblichen AOP System kann man mit Expressions Java Code an bestimmten Stellen und bei bestimmten Ereignissen verändern. Das tolle dabei ist, dass man wirklich sehr genau ausprogrammieren kann, wann man einen Call abfängt oder nicht. Der grosse Nachteil ist, dass die ganze Geschichte hart kodiert werden muss.

```
CtMethod method = cl.getDeclaredMethod("test");

// die Methode, welche wir verändern möchten
method.instrument(new ExprEditor() {

    // wird in der methode eine andere methode aufgerufen, wird diese methode gerufen
    public void edit(MethodCall c) throws CannotCompileException {
        // betrifft der Aufruf eine bestimmte Klasse, kann der Methodenaufruf durch einen anderen ersetzt
        // werden, oder
        // man verhindert z.B. den Aufruf basierend auf einem if/else.
        if (c.getClassName().equals("test.Hallo")) {
            c.replace("if (false) { $proceed($$); }"); // Methoden auf Objekten der Klasse Hallo wollen wir
            // nicht ausführen
            //c.replace(""); // Alternativ..
        }
    }
});
```

Die Methode edit(MethodCall call) wird nun jedes Mal aufgerufen, wenn in der Methode eine andere Methode aufgerufen wird. Deshalb haben wir eine Bedingung eingefügt, welche sicherstellt, dass nur Methoden auf der Klasse test.Hallo abgefangen werden. Den Aufruf der Methode kann man nun durch etwas anderes ersetzen. Mit \$proceed(\$\$) wird der ursprüngliche Methodenaufruf fortgesetzt, mit allen Parametern (\$\$).

Man kann nicht nur Aufrufe auf Methoden überwachen sondern auch:

- Konstruktor Aufrufe
- Zugriffe auf Felder/Attribute
- Objekt Erzeugung (new ..)
- Neues Array
- Instanceof Aufrufe
- Type Casts
- Handler, ein Catch-Block Aufruf



Siehe Expressions.java in der Beispielapplikation

Low Level API

Javassist bietet auch die Möglichkeit, auf der Ebene von Opcodes zu arbeiten. Um Java wirklich schnell zu machen, ist es unabdingbar zu verstehen, wie der Java Bytecode aussieht.



Java Bytecode ist sozusagen das Assembler der JVM. Zum Beispiel mit javap kann man den Maschinencode anschauen. Es gibt aber auch Compiler, welche direkt Java Assembler in voll funktionsfähige Java Klassen umsetzen.

Ich möchte aber keinesfalls dazu raten, den Optimierungsprozess auf der Ebene von Opcodes anzufangen. Meist ist langsamer Java Code einfach schlecht programmiert und durch ein paar Handgriffe im Java Quelltext ist viel zu erreichen.

Doch genug der Theorie, schauen wir uns an einem Beispiel an, wie dieses API funktioniert:

```
ClassFile cf = new ClassFile(new DataInputStream(Main.class.getResourceAsStream("/test/Main.class")));  
ConstPool pool = cf.getConstPool();  
Bytecode byteCode = new Bytecode(pool);  
byteCode.add(Bytecode.ICONST_3);  
byteCode.addReturn(CtClass.doubleType);  
CodeAttribute ca = byteCode.toCodeAttribute();
```

Der ByteCode den wir erzeugt haben entspricht der folgenden Opcode Folde:

```
iconst_3  
ireturn
```

Tiefer in diese Materie einzusteigen würde den Rahmen dieser Arbeit sprengen. Allerdings ist es gut zu wissen, dass man auch mit Opcodes arbeiten kann, wenn man denn möchte.

GluonJ

Auf Basis von Javassist ist ein einfaches AOP System namens GluonJ entstanden. Es ist ein auf Annotations basierendes System, welches sehr schnell erlernt werden kann. Zudem funktioniert es (wie wir das von Javassist kennen) sowohl zur compile-time als auch zur load-time. Die Transformation von Java Bytecode nennt man übrigens Bytecode-Weaving, ein bekanntes Tool, welches dieses Verfahren anwendet ist der Retroweaver, welche für Java 5.0 kompilierte Klassen in für Java 1.4 lauffähig umwandelt – aber dies nur nebenbei.

Als einführendes Beispiel im Bereich von AOP bietet sich ein Logging Aspekt an. Mit GluonJ sieht ein solcher Aspekt folgendermassen aus:

```
import javassist.gluonj.*;

@Glue
public class Logging {
    @Before("{ logger.log(\"Die Transfer Methode wird aufgerufen!\"); }")
    Pointcut pc = Pcd.call("test.Account#transfer(..")
        .and.within("test.AccountTest.testTransfer(..)");
}
```

Wird nun irgendwo in unserem Code die Methode transfer() der Klasse test.Account aufgerufen, wird unser Advice ausgeführt. Oder?

Nicht ganz, denn durch and.within() stellen wir noch eine zusätzliche Bedingung: die Methode muss aus unserem Testcase ausgeführt werden, damit der Advice tatsächlich ausgeführt wird.

Dadurch, dass ein Aspekt via Annotations konfiguriert wird, bleibt der Quelltext m.E. sehr leserlich. Die Aspekte können so tatsächlich als Teil eines Projektes realisiert werden und sind nicht nur „Zugemüse“.

Eine der interessanteren Anwendungsgebiete für Annotations, welche von vielen Entwicklern als zu kompliziert verschrien werden.

Fazit und persönliche Meinung

Javassist ist ein sehr mächtiges Hilfsmittel. Allerdings sind gute Java Kenntnisse unabdingbar, um zu verstehen, was im Hintergrund abläuft. Ein grosser Vorteil im Vergleich zu anderen Werkzeugen ist die Möglichkeit, Klassen zur Laufzeit zu verändern. Wie bereits mehrmals erwähnt ist Javassist aber keinesfalls ein AOP System im klassischen Sinne. Es kann aber durchaus die Basis für ein AOP Framework bilden (siehe JBossAOP und GluonJ).

GluonJ ist auf Basis von Javassist entstanden und sieht sehr viel versprechend aus. Besonders gefällt, dass das System auf Annotations basiert und der Quelltext somit besonders leserlich bleibt.

Ein grosser Nachteil von AOP Systemen ist die Übersichtlichkeit. Ohne gute Entwicklungsumgebung (siehe AspectJ und Eclipse) ist es schwierig, den Überblick zu behalten. Ausserdem muss man auch aufpassen, nicht die gesamte Business-Logik in Aspekte auszulagern. Schliesslich finden AOP Systeme vor allem in der OO-Welt und nicht nur Aspekte sondern allen voran Objekte haben Verantwortlichkeiten.

Weiterführende Informationen

http://www.csg.is.titech.ac.jp/projects/gluonj/	GluonJ
http://www.csg.is.titech.ac.jp/~chiba/javassist/	Javassist
http://labs.jboss.com/portal/jbossaop	JBossAOP
http://www.eclipse.org/aspectj/	AspectJ
http://de.wikipedia.org/wiki/Aspektorientierte_Programmierung	AOP Generell