Das Java-Reflection-API

Spieglein, Spieglein

Das Reflection-API informiert zur Laufzeit über Klassen, ihre Methoden und Felder. So bildet es die Basis für die Komponentensysteme Beans und J2EE. Der Coffee-Shop zeigt, wie sich mit Reflection dynamisch Klassen laden und ausführen lassen. Bernhard Bablok

Wer bin ich? Das fragt sich nicht nur mancher Programmierer nach durchhackter Nacht, auch die Software selbst gerät bisweilen in eine Identitätskrise. Kompiliertem Code fällt es meist schwer, die ursprünglichen High-Level-Informationen zu rekonstruieren. Objektorientierte Sprachen besitzen dafür oft ein nützliches Gegenmittel: Introspection, Metaprogrammierung und Reflection sind Mechanismen, die es Code ermöglichen, Kenntnis über die zugrunde liegenden Klassen zu erhalten. Das ermöglicht das Laden dynamischer Codeteile, die Entwicklung komponentenbasierter Systeme und klügerer Tools für objektorientiertes Software-Engineering.

Re-Engineering

So genannte Case-Tools (Computer-aided Software Engineering) konnten sich bisher nicht auf breiter Basis durchsetzen. Eine der Ursachen ist, dass zwar der Weg vom Case-Tool zum Code funktioniert, der Rückweg aber oft nicht. Und wer will schon den eigenen Code innerhalb eines Case-Tools pflegen, wo es doch hervorragende Editoren gibt? Und wenn der Code nur in kompilierter Form vorliegt, ist in den meisten Sprachen sowieso Schluss.

In Teilbereichen erweisen sich solche Tools aber durchaus als hilfreich, beispielsweise bei der Verwaltung von Klassen in der objektorientierten Programmierung. Java unterstützt das seit Version 1.1 durch die neuen Klassen des Reflection-API. Der Name geht darauf zurück, dass das API es einem laufenden Programm erlaubt, sich selbst zu betrachten, genauer: seine Klassen, Felder und Methoden. Denn Javas Sprachkern bietet hier anders als andere Sprachen keine wirkliche Unterstützung (es gibt nur den »instanceof«-Operator).

Die folgenden Abschnitte erläutern das Prinzip und wichtige Klassen des Reflection-API, zeigen Einsatzgebiete und demonstrieren an einem Beispiel, wie man das API nutzt, auch wenn es nicht um Entwicklungstools oder Komponentensysteme geht.

Die Class-Datei

Die Datei beginnt mit der hexadezimalen Magic-Nummer »cafebabe«, danach folgen Informationen zur Klasse (Abbildung 1). Schon vor dem JDK 1.1 konnte man diese Information auslesen und sinnvoll nutzen, was allerdings etwas mehr eigene Low-Level-Programmierung erforderte (siehe [2]).

Mit dem Reflection-API ist es nicht mehr notwendig, die Datei selbst zu parsen. Das übernehmen die Klasse »java.lang.Class« sowie die Klassen im Package »java.lang.reflect« (Constructor, Field, Method …). Sie parsen selbst die Class-Datei und liefern alle Informationen zurück, die darin enthalten sind.

Die Aufteilung des Reflection-API auf mehr als ein Package ist historisch bedingt und kommt auch an anderen Stellen vor, zum Beispiel bei den Font-Klassen, vergleiche [1]. Die Klasse »Class« existiert schon seit der Version 1.0 des JDK, wurde aber mit 1.1 deutlich erweitert. Die neuen Klassen sind alle im Package »java.lang.reflect« zu finden.

Klassen und Methoden

Abbildung 2 (erzeugt mit ArgoUML, [3]) zeigt wichtige Klassen des Reflection-API mit den relevanten Methoden. Ausgangspunkt ist die Klasse »Class«, die Informationen über Klassen, Felder,

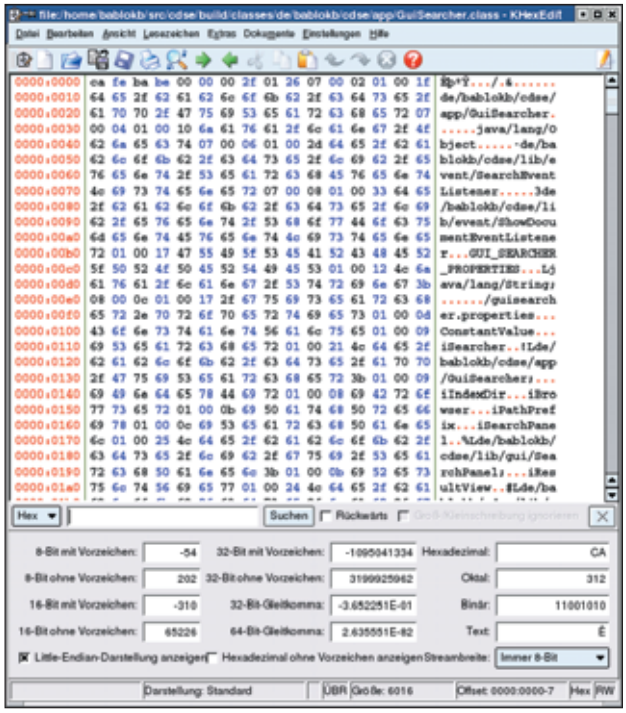

Abbildung 1: Der Hex-Editor zeigt den Anfang des kompilierten Java-File, also der Bytecode-Datei zur Klasse »GuiSearcher«.

Konstruktoren, Methoden, Modifier (wie »public« und »private«) mittels diverser »get«-Methoden bereitstellt.

Viele der Methoden sind in zwei Varianten vorhanden, für Felder beispielsweise »getFields()« und »getDeclaredFields()«. Die »getDeclared«-Varianten geben alle Informationen zurück, die innerhalb der eigentlichen Klassendefinition vorkommen, auch die Member, die nicht öffentlich (»public«) sind. Die Varianten ohne »Declared« beschränken sich auf die Public-Schnittstelle, geben aber Member von Superklassen oder implementierten Interfaces aus.

Zusätzlich gibt es für Konstruktoren und Methoden jeweils eine Variante, die alle Member zurückgibt, sowie eine Variante, die gezielt nach einem Member mit gegebener Argumentsignatur sucht.

Das API erlaubt es aber nicht nur, Informationen über die Member-Elemente einer Klasse auszulesen. Mit ihm lassen sich zudem einzelne Felder lesen und schreiben – mit den »set«- und »get«-Methoden in »Field« – sowie Konstruktoren und Methoden ausführen. Felder direkt manipulieren ist nur selten eine gute Idee, aber neue Objekte erzeugen und Methoden dynamisch aufrufen, das ist je nach Kontext durchaus sinnvoll, wie das Beispiel unten zeigen wird.

Konstruktoren finden

Schon in der JDK-Version 1.0 konnte der Programmierer Objekte dynamisch erzeugen, indem er »Class.newInstance()« verwendete. Allerdings nur in jenen Fällen, bei denen ein Konstruktor ohne Argumente vorhanden war. Seit Java 1.1 ist diese Einschränkung durch die neue Constructor-Klasse gefallen. Die Methode »Class.getConstructor()« liefert ein Objekt der Klasse »Constructor«, die eine Methode »newInstance()« besitzt, die Argumente akzeptiert.

Analog zu den Konstruktoren funktionieren dynamische Methodenaufrufe. Über das Klassenobjekt holt man sich ein Objekt der Klasse »Method«, dessen »invoke()«-Methode ruft dann die andere, aufzurufende Methode auf. Die Details zeigt das Beispiel unten: Die ganze Magie besteht darin, bei der Methodensuche die gewünschten Argumenttypen zu übergeben und beim Aufruf dafür zu sorgen, dass die Argumente den Typen entsprechen.

Mit dem Reflection-API lässt sich der Zugriffsschutz der durch »private«- oder »protected«-Modifier geschützten Member aushebeln. In Umgebungen, die nicht vertrauenswürdigen Code laden, kann man das über einen Security Manager verhindern. Aber generell entspricht es nicht der objektorientierten Lehre, mittels Reflection auf nicht öffentliche Methoden oder Felder zuzugreifen – Entwicklungstools ausgenommen.

Typische Einsatzgebiete

Das Reflection-API ist immer dann nützlich, wenn maximale Flexibilität bei minimaler Information gefordert ist, also Objekte verarbeitet werden müssen, auf deren Interface man keinen Einfluss hat. Die erwähnten Case-Tools sind ein Einsatzgebiet. Im Gegensatz zu anderen Sprachen, bei denen vorhandener Quellcode zwingend erforderlich ist, können Java-Tools einfach Jar-Dateien laden und die entsprechenden Details dem Nutzer anzeigen.

So lassen sich einzelne Objekte erzeugen, grafisch manipulieren und mittels Serialisierung für eine spätere Wieder-

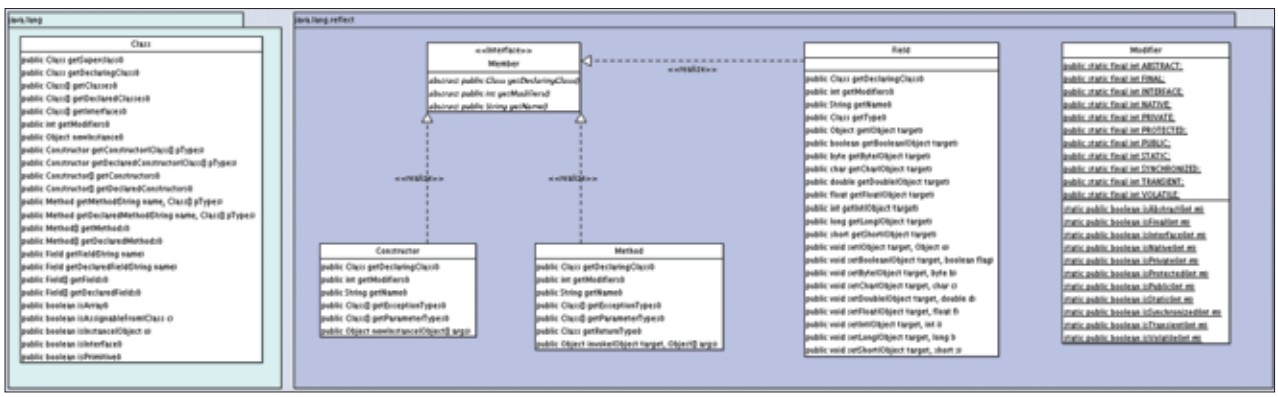

Abbildung 2: Die wichtigsten Reflection-Klassen und -Methoden: »Field«, »Modifier«, »Constructor« und »Method« sind Realisierungen (Is-A-Relation) des Member-Interface.

verwendung abspeichern. Weitere wichtige Anwendungsgebiete dafür sind Servlet/JSP- und J2EE-Container (allgemeiner: Komponentenmodelle). Nicht ohne Grund wurden das Beans-Modell und die entsprechenden Klassen zur selben Zeit in das JDK aufgenommen.

Ganz konventionell

Namenskonventionen legen dabei fest, wie Methoden heißen müssen. Diese Konventionen vereinfachen es, zu einem gegebenen Wert oder Member die passende Methode im API zu finden. Auch Debugger profitieren davon, zur Laufzeit Objekte zu manipulieren. Wie man sich denken kann, führt diese Flexibilität aber zu Performancenachteilen. Die Suche nach Methoden ist noch eine schnelle Operation, die dynamische Ausführung dagegen nicht mehr.

Genug der Theorie: Das folgende, eher atypische, doch nicht sinnlose Beispiel demonstriert den Einsatz des Reflection-API. Es handelt sich um ein Programm, das Werte mathematischer Funktionen berechnet. Das Beispiel beschränkt sich auf die Funktionen der Klasse »java .lang.Math«. Im Prinzip wäre der Rechner sogar während der Laufzeit dynamisch erweiterbar: Der Anwender müsste entsprechende Klassen mit neuen Funktionen nur im »CLASSPATH« entpacken und dann über eine Ladefunktion dem Rechner bekannt machen.

Das Beispiel ist bewusst einfach gehalten. Die »main()«-Methode besteht aus einer Schleife, die Benutzereingaben von der Standardeingabe liest (siehe Zeilen 43 bis 78 im **Listing 1**). Die Methode »parse()« (Zeilen 7 bis 13) liest die eingegebene Zeile, parst sie mit einem einfachen »StringTokenizer« und gibt eine Liste von Strings zurück. Die Funktion ist das erste Element dieser Liste, danach folgen die Argumente.

Nach Mustern suchen

Der nächste Schritt ist die Suche nach der passenden Java-Methode. Hierzu sucht das Programm nach einer Methode der »Math«-Klasse, die die richtige

Listing 1: »MathCalc.java«

```
01 import java.lang.reflect.*;
02 import java.io.*;
03 import java.util.*;
04
05 public class MathCalc {
06
07   private LinkedList parse(String line) {
08     StringTokenizer tokenizer = new StringTokenizer(line," \t\n\r\f,()");
09     LinkedList list = new LinkedList();
10     while (tokenizer.hasMoreTokens())
11       list.add(tokenizer.nextToken());
12     return list;
13   }
14
15   ////////////////////////////////////////////////
16
17   private Method searchMethod(LinkedList list) throws
       NoSuchMethodException {
18     if (list.size() < 1)
19       throw new IllegalArgumentException("empty line!");
20     String name = (String) list.removeFirst();
21
22     Class[] args = new Class[list.size()];
23     for (int i=0;i<args.length;++i)
24       args[i] = Double.TYPE;
25     return Math.class.getDeclaredMethod(name,args);
26   }
27
28   ////////////////////////////////////////////////
29
30   public Object execute(Method method, LinkedList arguments)
31     throws IllegalAccessException, IllegalArgumentException,
32                                InvocationTargetException {
33     Double[] args = new Double[arguments.size()];
34     Iterator iter = arguments.iterator();
35     int i = 0;
36     while (iter.hasNext())
37       args[i++] = new Double((String) iter.next());
38     return method.invoke(null,args);
39   }
40
41   ////////////////////////////////////////////////
42
43   public static void main(String[] args) {
44     try {
45       MathCalc mc = new MathCalc();
46
47       BufferedReader input =
48         new BufferedReader(new InputStreamReader(System.in));
49       while (true) {
50         System.out.print("> ");
51         String line = input.readLine();
52         if (line == null)
53           break;
54         else if (line.length() == 0)
55           continue;
56         else {
57           try {
58             long t1 = System.currentTimeMillis();
59             LinkedList list = mc.parse(line);
60             long t2 = System.currentTimeMillis();
61             Method m = mc.searchMethod(list);
62             long t3 = System.currentTimeMillis();
63             Object result = mc.execute(m,list);
64             long t4 = System.currentTimeMillis();
65             System.out.println(result.toString() +
66                        "\n\tparse:   " + (t2-t1) + "msec" +
67                        "\n\tsearch:  " + (t3-t2) + "msec" +
68                        "\n\texecute: " + (t4-t3) + "msec\n");
69           } catch (Exception e2) {
70             System.out.println("error: " + e2.toString());
71           }
72         }
73       }
74       input.close();
75     } catch (Exception e) {
76       e.printStackTrace();
77     }
78   }
79 }
```

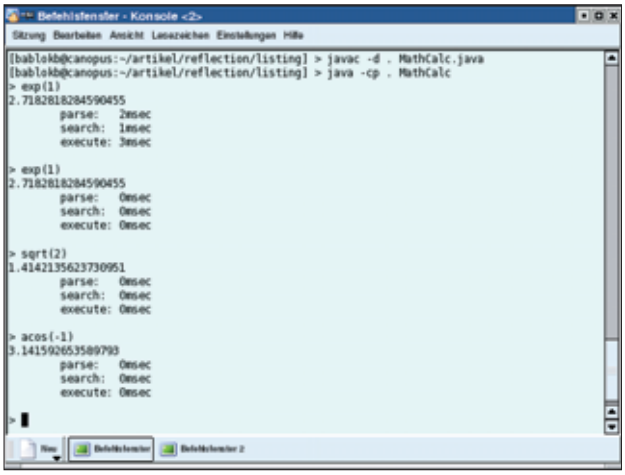

Abbildung 3: »MathCalc« lädt durch die Benutzereingabe ausgewählte mathematische Funktionen aus dem »Math«-API abhängig von ihrer Signatur, führt sie aus und misst die Laufzeit.

Anzahl an Parametern vom Typ »double« besitzt (Zeilen 17 bis 26). Allerdings ist das Beispielprogramm insofern vereinfacht, als es nur Methoden mit »double«-Argumenten verwendet. Eine Verallgemeinerung des Beispiels müsste viel mehr Aufwand treiben, insbesondere bei der Umwandlung der Argumente, die als Strings vorliegen, in die richtigen Objekttypen.

Wenn das Programm die Methode gefunden hat, führt es sie aus. Das besorgt ihrerseits die Methode »execute()« in den Zeilen 30 bis 39. Die Hauptarbeit ist es dabei, die Argumente in ein »Object«-Array umzuwandeln. Wegen der beschriebenen Vereinfachung muss es nur entsprechende »Double«-Wrapperobjekte erzeugen, das »Double«-Objekt besitzt praktischerweise einen Konstruktor, der einen String annimmt (Zeile 37). Annahmen über den Ergebnistyp muss das »MathCalc«-Programm nicht machen, da es das Ergebnis über die Methode »toString()« ausgibt.

Die Abbildung 3 zeigt einen Beispiellauf des Programms. Neben dem reinen Ergebnis gibt das Programm auch die Laufzeit aus: Sie ist auf einem modernen Rechner zu vernachlässigen.

Dynamisch, aber langsam

Etwas genauere Messungen sind mit dem Programm in Listing 2 möglich. Hier ist die Laufzeit der dynamischen Version ungefähr um den Faktor 20 langsamer. Ersetzt man die Funktion »max(x,y)« (Zeilen 11 und 17) durch »log(x)«, fällt dieser Faktor auf 4 bis 5, die Argumentumwandlung kostet also einen großen Teil der Performance. Schaltet man den JIT-Compiler ab (mit der Kommandozeilenoption »-Xint« des Java-Interpreters), verbessert sich das Verhältnis ebenfalls zugunsten der dynamischen Aufrufe. Das ist verständlich, denn ein JIT-Compiler kann diese im Gegensatz zu statischen Aufrufen nicht optimieren.

finally{}

Die Beispiele zeigen: Dynamische Aufrufe sind teuer – aber auch nicht so sehr, dass man auf ihren Einsatz generell verzichten müsste. Letztlich hängt es vom Gesamtsystem ab, wie stark dynamische Aufrufe ins Gewicht fallen. Auch die Wartbarkeit von Programmen spielt eine Rolle. Listing 1 zeigt, dass der Code selbst bei einem Beispielprogramm mit vereinfachenden Annahmen nicht gerade simpel ist. Trotzdem ist er eleganter und durchsichtiger als endlose Switch-Statements. Die kompletten Listings der Beispiele sind auf dem Server [4] des Linux-Magazins zu finden.

Das Reflection-API macht es den Tool-entwicklern leicht, kompilierte Klassen zu verwenden. Ist der Code nicht durch einen Obfuscator entstellt, lassen sich mit Reflection fremde Klassen re-engineeren. Auch der Erfolg komponentenbasierter Systeme wie JSP und J2EE wurde erst durch Reflection möglich. Wenn sein Einsatz auch auf Spezialgebiete beschränkt ist, zeigt es doch die Mächtigkeit der Sprache Java: Hier wurden grundlegende Fähigkeiten nachgerüstet, ohne dafür den Sprachumfang zu verändern. *(ofr)* ■

Infos

[1] „Gesetzte Typen: Fonts unter Java“: Linux-Magazin 02/04, S. 111
[2] „Java für Profis: Die virtuelle Java-Maschine“, Teil 1: Linux-Magazin 05/97: [http://www.linux-magazin.de/Artikel/ausgabe/1997/05/JVM/jvm1.html]
[3] Homepage von ArgoUML: [http://argouml.tigris.org]
[4] Listings zu diesem Artikel: [http://www.linux-magazin.de/Service/Listings/2004/03/Coffeeshop]

Der Autor

Bernhard Bablok arbeitet bei der Allianz Versicherungs AG im Bereich Data-Warehouse-Systeme. Wenn er nicht Musik hört, mit dem Radl oder zu Fuß unterwegs ist, beschäftigt er sich mit Themen rund um Objektorientierung. Er ist unter [coffee-shop@bablokb.de] zu erreichen.

Listing 2: »DynPerf.java«

```
01 import java.lang.reflect.*;
02
03 public class DynPerf {
04    public static void main(String[] args) {
05       try {
06          DynPerf mc = new DynPerf();
07          double i;
08
09          long t1 = System.currentTimeMillis();
10          for ( i = 1; i < 100000; ++i)
11            Math.max(i, i+1);
12          long t2 = System.currentTimeMillis();
13
14          Class[] types = new Class[2];
15          types[0] = Double.TYPE;
16          types[1] = Double.TYPE;
17          Method m = Math.class.
              getDeclaredMethod("max", types);
18          Double[] arguments = new Double[2];
19
20          long t3 = System.currentTimeMillis();
21          for (i = 1; i < 100000; ++i) {
22            arguments[0] = new Double(i);
23            arguments[1] = new Double(i+1);
24            m.invoke(null,arguments);
25          }
26          long t4 = System.currentTimeMillis();
27          System.out.println("direct:  " +
              (t2 - t1) + "msec"
28            + "\ndynamic: " + (t4 - t3) +
              "msec\n");
29       } catch(Exception e) {
30         e.printStackTrace();
31       }
32    }
33 }
```