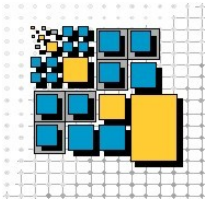

Otto-Friedrich-Universität Bamberg

Lehrstuhl für Praktische Informatik

Referat

Im Rahmen des Seminars

Programmierung

Zum Thema:

Reflexive Programmiersprachen

—

Konzepte und Umsetzung in Java

Vorgelegt von:

Jan Petendi

Betreuer: Prof. Dr. Guido Wirtz

Bamberg, WS 07/08

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Entwicklung neuer Konzepte der Programmierung reicht von Assembler, über prozeduraler bis hin zu objektorientierter Programmierung; Ziel dieser Entwicklung ist dabei nie eine erweiterte Funktionalität zu schaffen, da letztendlich alles wieder abgebildet wird auf Maschinencode. Vielmehr sollen einem Softwareentwickler Möglichkeiten geschaffen werden, Software effizienter entwickeln zu können. Innerhalb dieser Entwicklung entstanden auch die Konzepte reflexiver Programmiersprachen, die einen Bruch zum traditionellen Programmieren bedeutete, in welchem ein Programmierer eine Applikation im Quellcode so schreiben musste, dass ihre Struktur und das Verhalten nach dem Kompilieren statisch war und es keine Möglichkeiten gab, überhaupt eine externe Sichtweise auf ein laufendes Programm zu erhalten oder vielmehr noch darauf aktiv Einfluss zu nehmen. Reflexive Programmiersprachen ermöglichen es, dass eine laufende Applikation eine externe Sichtweise auf sich selbst erhält und auf Grund dieser externen Sicht ganz andere Handlungsfreiheiten besitzt.
Vorliegende Arbeit erklärt zunächst die Konzepte reflexiver Programmiersprachen, indem theoretische Grundlagen beschrieben und reflexive Operationen definiert werden. Danach wird am Beispiel der Programmiersprache *Java* aufgezeigt, wie das Konzept praktisch umgesetzt wird. Dazu werden zunächst die bereits vorhandenen reflexiven Möglichkeiten von *Java* - die *java reflection API* betrachtet und bewertet; schließlich werden Möglichkeiten gezeigt, wie sich zusätzliche reflexive Operationen in *Java* realisieren lassen, als mächtigste Lösung hierfür wird *Javassist* vorgestellt werden. Die Arbeit schließt mit einer Bewertung der Umsetzung nach sicherheitstechnischen und zeitlichen Gesichtspunkten.

2 Konzepte reflexiver Programmiersprachen

Der Begriff Reflexion hat im allgemeinen Sprachgebrauch je nach Kontext 2 unterschiedliche Bedeutungen.
Reflexion als:

- die rein passive Eigenschaft aufgrund einer Einwirkung von außen etwas zurückzugeben
- die Fähigkeit den eigenen Zustand und die Umgebung wahrzunehmen und dementsprechend zu handeln

Reflexion als Eigenschaft einer Programmiersprache beinhaltet beide Bedeutungen, was im nun folgenden Abschnitt näher erläutert wird.

2.1 Grundlagen

Die erste formale Definition, welche Eigenschaften für eine reflexive Programmiersprache gelten müssen, lieferte B.C. Smith in seiner Doktorarbeit 1982 für die Programmiersprache `LISP` [2]. Trotz der Erarbeitung dieser für eine prozedurale Sprache, wurden die Aussagen aus systemtheoretischer Sicht formuliert und können so unverändert auch auf jede objektorientierte Sprache übertragen werden (vgl. [1, S.242-252]):

1. es muss eine Repräsentation dieses Systems vorhanden sein, auf welche das System selbst Zugriff hat

2. zwischen der Repräsentation des Systems und dem System selbst muss eine kausale Verbindung bestehen

3. diese kausale Verbindung muss robust sein

Die erste Anforderung, welche Smith an ein reflexives System stellt, besagt zum einen, dass eine *geeignete* und *vollständige* Repräsentation für dieses System vorliegt und dass zum anderen das System selbst zu jedem Zeitpunkt Zugriff auf diese haben muss. Zieht man den Quelltext eines Programms für diese Repräsentation in Betracht, welcher durch wiederkehrendes Parsen die Struktur des Programms extrahieren lässt, so muss man erkennen, dass sowohl die Forderung nach Eignung (Parsen ist viel zu zeitaufwändig für eine effiziente Anwendung) als auch die nach Vollständigkeit (der globale Zustand wie Variablenbelegungen etc. ist aus dem Quelltext unersichtlich) nicht erfüllt werden.
Anforderung 2 insistiert eine kausale Verbindung zwischen Repräsentation und System. Änderungen an der Repräsentation verändern das System und umgekehrt. Auch hier kann man wiederum den Quelltext als ungeeignet ausschließen, Veränderungen am Quelltext lassen das laufende System unverändert.
Da Veränderungen an laufenden Programmen einiges an Gefahren bergen, fordert Smith als drittes, dass alle reflexiven Operationen so umgesetzt werden müssen, dass zu keinem Zeitpunkt Schäden am laufenden Programm auftreten können.

Abbildung 1 illustriert dabei ein Modell, welches obigen Anforderungen gerecht wird und von Friedman und Wand vorgeschlagen wurde [3]. Die Operation *reify* ("verdinglichen") wandelt das laufende Programm in ein Objekt um, welches das Programm manipulieren kann, die Operation *reflect* integriert dieses Objekt als Komponente des Programms (vgl. [3, S.350]). Im Gegensatz zu Smith wird hier das zu manipulierende Programm der Instanzebene (*"base level"*) vom manipulierenden Programm der Metaebene (*"meta level"*) streng getrennt.

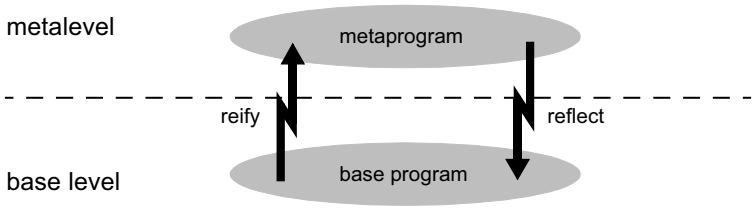

Abbildung 1: Ausführung eines reflexiven Systems nach Friedman und Wand aus [1, S.243,Abbildung A.1]

Der Schritt zwischen diesem Modell, welches ebenfalls für prozedurale Sprachen entwickelt wurde, zu einem für objektorientierte Sprachen ist klein. Reflektive objektorientierte Programmiersprachen benötigen neben den *normalen* Objekten zusätzlich eine Menge von sogenannten *Metaobjekten*, welche Struktur und Verhalten des Programms repräsentieren. Eine explizite *reify*-Operation ist im Allgemeinen nicht vorhanden, vielmehr werden die Metaobjekte mit Beginn der Programmausführung vom Laufzeitsystem generiert und bleiben während der gesamten Ausführungszeit bestehen.
Alle modernen objektorientierten Sprachen gehören zur Menge der *class-based languages* (vgl. dazu besonders [4]), welche die Eigenschaft haben, dass jedes Objekt Instanz von *genau einer* Klasse ist. Ein Objekt reagiert damit auf alle Methoden, die von dessen

Klasse unterstützt werden. Eine Klasse unterstützt eine bestimmte Methode, wenn sie in dieser Klasse definiert ist bzw. wenn diese Methode der Klasse vererbt wurde. Es ist also naheliegend, dass man zum Unterstützen von Reflexion den Klassenbegriff *reifiziert*, und man somit zusätzlich eine Menge von Objekten erhält, sogenannte *Klassenobjekte*.
In [1, S.245] werden zusammenfassend folgende Anforderungen an eine reflektive *class-based language* gestellt:

1. es existiert eine nichtleere, endliche Menge von Objekten, die jeweils eindeutig durch eine *Objektreferenz* bestimmt werden
2. jedes Objekt besitzt eine eindeutig zuordbare Klasse
3. jede Klasse ist reifiziert durch ein Objekt

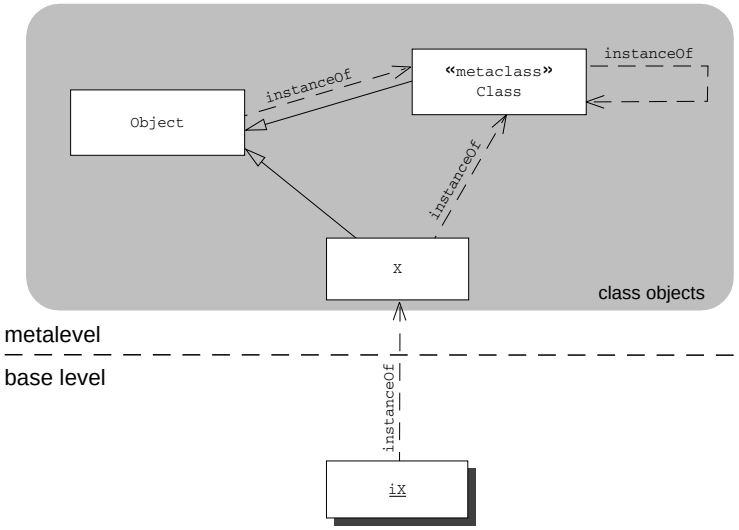

Abbildung 2: Hierarchie in einer reflexiven objektorientierten Programmiersprache aus [1, S.248,Abbildung A.4]

Wie das nun konkret umgesetzt wird, zeigt Abbildung 2. Demnach stehen alle Klassen eines Systems in einer Vererbungsrelation zur Klasse *Object*, eine Instanz einer Klasse *X* (hier iX) unterstützt damit auch alle Methoden von *Object*. Nach Anforderung 3 muss für jede Klasse ein Klassenobjekt existieren, welches zwangsläufig ebenfalls erbt von *Object*. Anforderung 2 verlangt wiederum, dass jedem Objekt eine Klasse zugeordnet werden muss - auch den Klassenobjekten. Diese sind Instanzen der Klasse *Class*, welche ihrer Art nach eine sog. *Metaklasse* ist, dies wird weiter unten noch näher erläutert werden. Man merkt, dass die Anforderungen 2 und 3 gemeinsam einen Zyklus bilden [1]. Da nach Anforderung 1 die Menge der Objekte endlich sein muss (es steht auch immer nur ein endlicher Speicher zur Verfügung), referenziert sich *Class* selbst in der *instanceOf*-Relation.

Die wichtigsten Begriffe sollen abschließend nochmal näher definiert werden:

Metaobjekt kapselt Struktur und Verhalten einer laufenden Applikation und bietet Methoden darauf Einfluss zu nehmen

[1] so müsste dem Objekt von *Class* wieder eine Klasse zugeordnet werden, welche reifiziert werden würde durch ein Objekt usw.

Klassenobjekt stellt die Reifizierung einer Klasse dar

Metaklasse ist eine Klasse deren Instanzen Klassen sind (deren Reifizierung nennt sich damit *Metaklassenobjekt*)

Auf Basis dieser Definition lässt sich dann noch weiter Folgendes anführen: Klassenobjekte sind Metaobjekte; nicht alle Klassen sind Metaklassen (weil sie *normale* Objekte instanziieren) und nicht alle Metaobjekte sind Metaklassenobjekte - weiter unten werden auch noch die Metaobjekte `Constructor`, `Field` und `Method` (Reifizierungen der gleichnamigen Konstrukte) vorgestellt, welche keine Klassen instanziieren können.

2.2 reflexive Operationen

Hier wird nun die Menge reflexiver Operationen genauer betrachtet und nach ihrer Art eingeteilt werden, in der Literatur werden diese auch "metaobject protocol" [5] genannt. Genauer gesagt liefert bei objektorientierten Programmiersprachen die Funktionalität der Methoden von Metaobjekten dieses "metaobject protocol".
Grob untergliedert man sie in *Introspection* und *Intercession*.

Introspection (von lat. *introspicere,-spexi: hineinschauen*) beschreibt den Vorgang die vorhandene Struktur und den Zustand einer laufenden Applikation zu untersuchen.
Eine Auswahl möglicher Operationen hierfür ist:

- Auflisten aller Klassen einer Applikation
- Auflisten aller Methoden, Felder und Konstruktoren einer Klasse (mit Name, Parametertypen, Modifikatoren)
- Auflisten aller implementierten `Interface`s einer Klasse
- Anzeigen der Superklasse
- Auslesen des Wertes eines Feldes

Intercession (von lat. *intercedere,-cessi: dazwischengehen,eingreifen*) beschreibt die Fähigkeit, aktiv Einfluss auf eine laufende Applikation zu nehmen. Je nach Art des Eingriffs unterteilt man hier noch weiter in *behavioural Reflection* und *structural Reflection*.

behavioural Reflection nimmt Einfluss auf das Verhalten einer Applikation unter Berücksichtigung der bestehenden Struktur, mögliche Operationen hierfür sind beispielsweise:

- Setzen des Wertes einer Variablen
- Aufrufen einer Methode auf einem Objekt
- Abfangen und Umleiten eines Methodenaufrufs

structural Reflection kann die Struktur einer Applikation zur Laufzeit ändern und ist somit die mächtigste Art reflexiver Operationen, hierfür kann man folgende Operationen nennen:

- Hinzufügen neuer Klassen
- Hinzufügen und Ändern von Methoden, Feldern und Konstruktoren zu einer vorhandenen Klasse
- Hinzufügen und Ändern der implementierten `Interfaces` einer Klasse
- Ändern der Superklasse einer Klasse

Bewertet man diese Operationen nach der Möglichkeit Schäden an laufenden Applikationen zu erzeugen, erkennt man, dass mit steigender Mächtigkeit der Operationen auch deren "Gefährlichkeit" zunimmt; das Setzen von Variablenwerten von außerhalb oder das Umdefinieren der Superklasse eines Klasse sollte mit Bedacht durchgeführt werden, da dadurch schwerwiegende Inkonsistenzen und Abstürze erzeugt werden können.

2.3 Anwendungsgebiete reflexiver Operationen

Die Anwendungsgebiete reflexiver Operationen sind vielfältig und einige Anwendungen lassen sich erst mit diesen Operationen richtig bewerkstelligen. Im Folgenden sollen kurz einige Anwendungsgebiete angerissen werden.

Debugging Mit Operationen der *introspection* lässt sich der Zustand einer Applikation aus externer Sicht ständig überwachen und etwaige Fehler finden.

lernende Systeme durch Lernalgorithmen gelernte Fakten werden dynamisch in neue oder komplexere Klassen übersetzt und direkt in die laufende Applikation eingebaut.

verteilte Systeme hier kann man mit Hilfe reflexiver Operationen sehr viel bewerkstelligen; angefangen von der Anpassung inkompatibler `Interfaces` von Komponenten, kann man hier sogar sich selbstverwaltende Systeme erzeugen, deren fehlerhafte Komponenten per Reflexion automatisch ausgetauscht oder sogar repariert werden können.

Softwaretechnik die durch Reflexion geschaffenen Möglichkeiten sind so mächtig, dass ganz neue Arten eine Applikation zu bauen geschaffen wurden. In diesem Zusammenhang lässt sich der Begriff des *metaprogramming* anführen. Metaprogramme sind Programme, welche Programme erzeugen (ein Compiler ist das typische Metaprogramm). War es vorher immer so, dass der Compilerbau von Experten bewerkstelligt wurde, so lassen sich nun die Operationen ein Programm auf dieser feingranularen Ebene zu betrachten und manipulieren viel komfortabler ausführen.

3 Umsetzung reflexiver Operationen in Java

Nachdem im vorherigen Abschnitt das Konzept reflexiver Programmiersprachen beleuchtet wurde, werden nun exemplarisch die reflexiven Fähigkeiten der Programmiersprache `Java` [2] betrachtet. Dazu wird zunächst anhand von kurzen Codebeispielen ein Überblick über die Verwendung der *java reflection API* und deren Implementierung gegeben. Daraufhin werden die vorhandenen Fähigkeiten der *java reflection API* eingeordnet in Bezug auf die weiter oben definierten reflexiven Operationen und in diesem Zusammenhang Möglichkeiten aufgezeigt diese zu erweitern. Als eine dieser Möglichkeiten wird `Javassist`[3] [6] näher betrachtet. Am Ende wird diese Umsetzung noch nach sicherheitstechnischen und zeitlichen Gesichtspunkten bewertet.

3.1 java reflection API

Im Folgenden soll ein sehr grober Überblick über die in Java vorhandene Umsetzung reflexiver Operationen gegeben. Um den Rahmen dieser Seminararbeit nicht zu sprengen, werden Beschreibungen zur genauen Verwendung der *java reflection API* auf ein Minimum beschränkt bleiben. Vielmehr soll dieser Abschnitt als Verständnisgrundlage dienen zum Vergleich zwischen dem Konzept der Reflexion und deren Umsetzung.
`Java` wurde propagiert unter dem Slogan *write once, run anywhere*, was neben der Fähigkeit zur Portabilität auf unterschiedlichsten System auch die Fähigkeit verspricht in dynamischen Umgebungen ausgeführt werden zu können. Somit gab es schon mit Release von `Java 1.0` die Möglichkeiten über `ClassLoader` zur Laufzeit neue Klassen nachzuladen und damit ggf. auch bereits vorhandene Klassendefinitionen zu ersetzen. Mit `Java 1.1` wurde dann das Paket `java.lang.reflect` definiert, welches meist auch gleichgesetzt wird mit dem Begriff *java reflection API*. Neben der Metaklasse `Class` wurden da die Klassen `Constructor` , `Field` und `Method` eingeführt. Mit `Java 1.5` und der Einführung von parametrisierbaren Klassen (sog. *generics*) wurde auch die *java reflection API* dahingehend überarbeitet.

3.1.1 Ausgewählte Konstrukte

Hier werden Ausschnitte der wichtigsten Konstrukte der *java reflection API* vorgestellt. Die zu Illustration dieser verwendeten Codebeispiele benutzen die in Anhang A.1 zu findende `BeispielKlasse`.

Class repräsentiert die Reifikation einer *Javaclass* und bildet somit den Ausgangspunkt für alle reflexiven Operationen.
Zugriff auf ein Klassenobjekt erhält man z.B. über:

- `beispielKlassenObjekt.getClass()` wenn ein Objekt des betreffenden Typs vorhanden ist

[2] Basis ist die aktuell verfügbare Spezifikation - JSE 1.6
[3] `http://labs.jboss.com/javassist/`, zuletzt besucht am 29.12.2007

- `de.petendi.seminar.BeispielKlasse.class` wenn zur Compilezeit bereits der vollqualifizierte Klassenname bekannt ist
- `Class.forName(String klassenName)` wenn sich der Klassenname erst zur Laufzeit ergibt

Mit dem so erhaltenen Klassenobjekt lassen sich z.B. folgende Operationen durchführen:

- `getModifiers()` liefert Informationen über die Klassenmodifikatoren (wie z.B. `public`, `abstract`, `final`, etc.)
- `getSuperclass()` liefert das Klassenobjekt der Superklasse (falls vorhanden)
- `getInterfaces()` liefert die Klassenobjekte aller implementierten `Interface`s (falls vorhanden)
- `getPackage()` liefert Informationen über das Paket, welches diese Klasse definiert (Name, Versionsnummer u.a.)
- `newInstance()` liefert eine Instanz dieser Klasse zurück, indem der parameterlose Konstruktor aufgerufen wird (ist dieser nicht vorhanden, ober nicht sichtbar, lässt sich eine Instanz nur über die Metaklasse `Constructor` erzeugen)

Constructor Über die Methode `getConstructors()` im Klassenobjekt lassen sich alle Konstruktoren dieses Klassenobjekts abrufen, übergibt man der Methode `getConstructor(Class<T> ...)` eine ParameterListe, wird ein auf diese Liste passender zurückgegeben (falls vorhanden). Neben Methoden zum Abfragen von Name, Anzahl und Typ zu übergebender Parameter, ist die wohl wichtigste Operation das Erzeugen von Objekten. Listing 1 zeigt den Unterschied zwischen dem *normalen* Erzeugen von Objekten und dem mittels Reflexion.

Listing 1: Erzeugen von Objekten mit und ohne Reflexion

```
1 BeispielKlasse normalErzeugtesObjekt = new BeispielKlasse("
    Test");
2 try {
3 Constructor constructor;
4 //Möglichkeit 1 - vorausgesetzt dies ist der einzig vorhandene
     Konstruktor
5 constructor = BeispielKlasse.class.getConstructors()[0];
6 //Möglichkeit 2 sucht einen Konstruktor, welcher als einzigen
    Parameter ein "String" verlangt
7 constructor = BeispielKlasse.class.getConstructor(java.lang.
    String.class);
8 BeispielKlasse reflexivErzeugtesObjekt = (BeispielKlasse)
    constructor.newInstance("Test");
9 } catch (Exception e) {//Fehlerbehandlung}
```

Field Die Methode `getFields()` liefert analog alle Felder eines Klassenobjekts, hier besteht durch Aufruf der Methode `getField(String feldName)` zudem die Möglichkeit direkt das Feld mit dem übergebenen Namen zu erhalten. Damit lassen sich nun der Feldname und dessen Typ (über `getType()`) ermitteln. Wichtig sind hier zudem natürlich die Methoden zum Auslesen und Verändern des Feldwertes. Da Java neben Referenztypen auch acht primitive Datentypen [4] besitzt, existieren neben `Object get(Object zuManipulierendesObjekt)`, `set(Object zuManipulierendesObjekt, Object neuerWert)` auch jeweils ein Methodenpaar pro primitivem Datentyp (`getInt`,`setInt`; ...). Seit `Java 1.5` wird die *autoboxing* genannte Fähigkeit unterstützt, die bei Bedarf primitive Datentypen in ihre Referenztypen (sog. *Wrapperklassen*) umwandelt und umgekehrt [5]; alle Methodenpaare für primitive Datentypen sind deshalb nicht mehr zwingend notwendig, aber weiterhin in der Klassendefinition von `Field` vorhanden. Listing 2 zeigt, wie man Zugriff auf ein privates Feldes via Reflexion erhält.

Listing 2: Setzen des Wertes eines privaten Feldes via Reflexion

```
1 try {
2 BeispielKlasse beispielKlassenObjekt = new BeispielKlasse("
    Test");
3 Field beispielStringFeld = Class.forName("de.petendi.seminar.
    BeispielKlasse").getField("beispielString");
4 //erlaube Zugriff auf das Feld (siehe "AccessibleObject")
5 beispielStringFeld.setAccessible(true);
6 beispielStringFeld.set(beispielKlassenObjekt, "Reflexion");
7 } catch (Exception e) {}
```

Method Über `getMethods()` bzw. `getMethod(String feldName,Class<T> ...)` kann man Zugriff auf die Methoden eines Klassenobjekts bekommen. Nun lassen sich u.a. Informationen über den Namen, Rückgabetyp, Anzahl und Typ der Parameter und Modifikatoren aufrufen. Über die Methode `invoke(Object zuManipulierendesObjekt, Object ...)`lässt sich auf dem zu manipulierendem Objekt unter Angabe der erforderlichen Parameter dessen Methode aufrufen, Listing 3 zeigt ein kurzes Beispiel dafür auf.

Listing 3: Aufruf einer Methode mit und ohne Reflexion

```
1 BeispielKlasse beispielKlassenObjekt = new BeispielKlasse("
    Test");
2 String value;
3 //normaler Methodenaufruf
4 value = beispielKlassenObjekt.getBeispielString();
5 try {
6 //suche die Methode mit angegebenem Namen und keinen
    Parametern
7 Method method =  beispielKlassenObjekt.getClass().getMethod("
    getBeispielString", null);
8 value = (String) method.invoke(beispielKlassenObjekt,null);
9 }
10 catch(Exception e){}
```

AccessibleObject In obigen Beispielen wurde bereits Gebrauch gemacht von der Methode `setAccessible(boolean zugriffErlaubt)`. In der Javaklassenhierarchie ist `Ac-`

[4] boolean, byte, char, short, int, long, float, double

[5] die Zuweisungen *Integer wert = 1* bzw. *int wert = new Integer(1)* waren davor nicht möglich

cessibleObject die Superklasse von Constructor, Field und Method. Im Normalfall ist auch der Zugriff via Reflexion auf geschützte Elemente nicht erlaubt. Über die Methoden in AccessibleObject kann man allerdings explizit den Zugriff auf solche gewähren. Da solche Operationen eine laufende Applikation stark beeinflussen und auch schädigen können, bietet *Java* einige Schutzmechanismen dafür an, dies wird in Abschnitt 3.3.1 noch näher erläutert werden.

Proxy Die bisher erläuterten Möglichkeiten beschränkten sich darauf auf vorhandenen Klassenobjekten (welche entweder bereits zur Compilezeit vorhanden sind oder über den ClassLoader nachgeladen werden) Operationen durchzuführen. Mit Java 1.3 wurde die Möglichkeit eingeführt, in begrenztem Maße dynamisch *neue* Klassenobjekte zur Laufzeit generieren zu können. Die Klasse Proxy bietet die Möglichkeit auf Basis von Interfaces (welche wie oben beschrieben bereits vorhanden sein müssen) eine sog. *Proxyklasse* zu erzeugen. Ein *Proxy* bietet die Möglichkeit transparent Anfragen weiterzuleiten [6], hier werden die Methodenaufrufe der von der Proxyklasse unterstützten Interfaces weitergeleitet an die Klasse InvocationHandler.
Die Klasse Proxy bietet u.a. folgende statische Methoden an:

- Class<?> getProxyClass(ClassLoader loader, Class<?>[] interfaces) erzeugt unter Angabe des ClassLoaders und einer Menge von Interfaces das gewünschte Klassenobjekt
- Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h) um sich den Weg zu ersparen, den Konstruktor der Proxyklasse manuell via Reflexion aufzurufen, kann man unter zusätzlicher Angabe des InvocationHandlers sofort ein Objekt dieses Klassenobjektes erzeugen.

Listing 4 zeigt, wie man mit Hilfe von Proxy die Funktionalität implementiert, dass Methodenaufrufe beliebiger Interfaces geloggt werden (inspiriert von [1, S.81 ff.], das komplette Beispiel (einschließlich Änderungen an BeispielKlasse) ist zu finden in Anhang A.5. Hierbei ist anzumerken, dass das in LoggingInvocationHandler übergebene Objekt alle Interfaces der Proxyklasse implementieren muss, um keine Fehlermeldung zu erzeugen (dies wird hier nicht abgefangen, sollte aber unter normalen Umständen immer geprüft werden).

Listing 4: Logging von Methodenaufrufen mit Hilfe einer Proxyklasse

```
1 //Ausschnitt aus LoggingInvocationHandler
2 public Object invoke(Object proxy, Method method, Object[]
    args)throws Throwable {
3         System.out.println(method.getName() +" von "+object.
              getClass().getSimpleName()+" wurde aufgerufen");
4         return method.invoke(object, args);}
5 //(...)
6 IBeispielKlasse normalesObjekt = new BeispielKlasse("normales
    Objekt");
7 LoggingInvocationHandler invocationHandler = new
    LoggingInvocationHandler(normalesObjekt);
8 Class<?> klassenObjekt =  normalesObjekt.getClass();
```

[6] für die genaue Definition eines Proxys sei an dieser Stelle z.B. verwiesen auf [7]

```
9  Proxy.getProxyClass(klassenObjekt.getClassLoader(),
       klassenObjekt.getInterfaces());
10 IBeispielKlasse proxyBeispielKlasse = (IBeispielKlasse) Proxy.
       newProxyInstance(klassenObjekt.getClassLoader(),
       klassenObjekt.getInterfaces(),invocationHandler);
```

3.1.2 Bewertung der vorhandenen Konstrukte

Ordnet man die Möglichkeiten der *java reflection API* ein in Bezug auf die Art reflexiver Operationen aus Abschnitt 2.2, so muss man erkennen, dass bei weitem nicht alles unterstützt wird. Vollständig umgesetzt ist die Fähigkeit der *introspection*; angefangen von der Menge der geladenen Klassenobjekten, über deren genaue Struktur (Konstruktoren, Methoden, Felder) bis hin zum Zustand der Applikation (Werte von Instanzvariablen) lassen sich alle notwendigen Operationen dafür ausführen. Sehr viel eingeschränkter umgesetzt wurde die Möglichkeit zur Beeinflussung einer Applikation (*intercession*), man hat hier erstmal nur die Möglichkeit neue Instanzen vorhandener Klassenobjekte zu erzeugen, Methoden auf Objekten auszuführen und die Werte von Feldern zu ändern. Zusammen mit dem `Proxy` Konstrukt lassen sich so bis zu einem bestimmten Grad diese Operationen nachbilden. Exemplarisch soll nun kurz eine mögliche Vorgehensweise zur Nachbildung dieser Operationen beschrieben, sowie ihre Schwächen aufgezeigt werden:
`direkt nach Starten der Applikation werden alle vorhandenen Klassenobjekte durch Proxyklassen ersetzt und alle auftretenden Instanziierungen der ursprünglichen Klasse durch eine der Proxyklasse ersetzt, so dass jeder Methodenaufruf überwacht werden kann. Der eigentlich ausgeführte Code wird dynamisch über den ClassLoader-Mechanismus hinzugefügt.`
Dieses Vorgehen für die generische Umsetzung von *intercession* scheitert u.a. daran, dass das `Proxy` Konstrukt nur Proxyklassen definieren kann, um `Interface`s zu implementieren, kopieren eines beliebigen Klassenobjekts mit Feldern und allen Methoden (auch z.B. die mit dem Modifikator `private`) ist nicht möglich. Darüberhinaus ist ein Ersetzen von Objekten über Reflexion auch nur sehr begrenzt möglich, alle Objekte, die innerhalb von Methodenrümpfen erzeugt werden, sind mit den vorhandenen Möglichkeiten nicht erreichgeschweige denn manipulierbar. Das Nachladen von Code über den `ClassLoader` ist auch viel zu unflexibel, da zum einen immer ein gesamtes Klassenobjekt nachgeladen werden muss und zudem diese Klassendefinition bereits als kompilierter *JavaBytecode* vorliegen muss.

3.2 Erweiterung der java reflection API

Dieser Abschnitt zeigt zu Beginn kurz auf, welche Möglichkeiten allgemein bestehen, um die bestehenden reflexiven Fähigkeiten von *Java* zu erweitern und bewertet diese. Danach wird *Javassist* als mächtigste dieser Lösungen näher betrachtet.

3.2.1 Anforderungen an die Umsetzung

Die Möglichkeiten eine Programmiersprache um zusätzliche Operationen zu erweitern sind vielfältig, aber nicht alle eignen sich gleich gut, einerseits, um Reflexion zu unterstützen

und andererseits um sich gut in die bestehende Architektur der Programmiersprache zu integrieren. Bei *Java* kann man zusätzliche Funktionalität auf verschiedenen Ebenen einbauen, welche auch mit dem Zyklus einer *Java* Applikation zusammenfallen (vgl. [6, S.18 ff.]:

beim Kompilieren Änderungen der Struktur (*structural reflection*) oder das Abfangen von Methodenaufrufen zum dynamischen Ausführen von Code (*behavioural reflection*) werden durch Modifizieren des Quellcodes vollzogen und erst durch Kompilieren von diesem zugänglich gemacht. Neben dem Vorteil, dass es auf dieser Ebene ein Leichtes ist eventuelle Änderungen an der Syntax von *Java* zu unterstützen, hat diese Umsetzung auch einige gravierende Nachteile. Zum einen muss der Quellcode immer vorhanden sein (der Quellcode von externen Bibliotheken muss nicht zwingend offenliegen) und andererseits ist es so immer notwendig für eine *Reifikation* den Quellcode zu parsen und für die *Reflektion* der Modifikationen diesen zu kompilieren, was äußerst zeitintensiv ist (vgl. dazu auch Abschnitt 2.1)

bevor der Code der Laufzeitumgebung zugänglich gemacht wird *Java* kompiliert den Quellcode nicht in nativen Maschinencode, sondern in sog. *Bytecode*, der von der *Java Virtual Machine (JVM)* erst zur Laufzeit in Maschinencode übersetzt wird, dieser binäre *Bytecode* kann ohne den Aufwand des Neukompilierens direkt modifiziert werden, bevor er der *JVM* zugänglich gemacht wird. Dieses Vorgehen wird von unterschiedlichen Projekten angewandt. *Kava* [8] fügt im *Bytecode* sog. *hooks* ein (Haken, die aktiviert werden, wenn diese Codestelle ausgeführt wird), um in das Laufzeitverhalten einer Applikation einzugreifen; dadurch ist man nicht mehr auf Proxyobjekte angewiesen, um Methodenaufrufen weiterzuleiten, aber *Kava* unterstützt dadurch nur *behavioural reflection*, da es die Struktur der Applikation unverändert lässt. Dieser Mangel wird von *Javassist* behoben, was die Möglichkeit bietet vollständig neuen *Bytecode* zu generieren, während eine Applikation läuft und diesen die *JVM* dynamisch nachladen zu lassen. Mit *Javassist* lässt sich die Funktionalität von *Kava* nachbilden und zudem unterstützt es auch noch *structural reflection*, weshalb diese Lösung im nachfolgenden Abschnitt noch detaillierter betrachtet werden wird.

während der Code von der Laufzeitumgebung ausgeführt wird Lösungen wie *MetaXa* [9] bieten eine eigene Implementierung der *JVM* an. Vorteile hier sind, dass man auf sehr feingranularer Ebene Zugriff auf eine Applikation erhält, um reflexive Operationen durchzuführen und dass durch eine direkte Modifikation der *JVM* auch eine große Optimierung der Geschwindigkeit bei der Ausführung dieser Operationen umgesetzt werden kann. Der große Nachteil dabei ist, dass durch die Bindung an eine proprietäre *JVM* keine Portabilität mehr gegeben ist.

3.2.2 Beispielerweiterung Javassist

Wie bereits beschrieben basiert *Javassist* darauf *ByteCode* zur Laufzeit umzuschreiben und über den `ClassLoader`-Mechanismus bereitzustellen. Somit ist einerseits die Portabilität weiterhin gesichert und zum anderen wird hiermit ein mächtiges Werkzeug geschaffen, um alle in Abschnitt 2.2 definierten Operationen zu unterstützen. *Javassist* liefert dafür

u.a. die Klassen `CtClass`, `CtConstructor` , `CtField` und `CtMethod`, welche neben den oben beschriebenen Methodensignaturen der *java reflection API* einiges an zusätzlicher reflexiver Funktionalität definieren. Das grobe Vorgehen beim Arbeiten mit *Javassist* ist dabei immer folgendes:

1. Erzeugung eines `CtClass` Objekts (welches den *ByteCode* kapselt)
2. Durchführung beliebiger reflexiver Operationen mit diesem Objekt
3. Umwandlung des `CtClass` Objekts in ein *normales* `Class` Objekt

Im Folgenden wird kurz ein Ausschnitt aus den Möglichkeiten von *Javassist* gegeben; zur Illustration dieser wird die bereits in obigen Beispielen verwendete `BeispielKlasse` (Anhang A.1) in einem durchgängigen Beispiel mit Hilfe von *Javassist* zusammengebaut und in ein `Class` Objekt umgewandelt werden, das komplette Beispiel ist zu finden in Anhang A.6.

CtClass Bietet zusätzlich zu der in `Class` definierten Funktionalität Methoden, um Konstruktoren, Felder, Methoden hinzuzufügen oder zu entfernen, weiterhin ist es möglich den Klassennamen und die Superklasse zu ändern. Man kann bestimmen, welche `Interface`s implementiert werden oder welche Klassenmodifikatoren gelten. Über `toClass()` (mit möglichen zusätzlichen Parametern) kann man es in ein `Class` Objekt umwandeln; per `writeFile(String verzeichnis)` wird der *ByteCode* dieser Klasse ins übergebene Verzeichnis geschrieben.
Folgender Codeausschnitt erzeugt die Definition von `BeispielKlasse` und wandelt das vorhandene `Class` Objekt eines `String` in ein `CtClass` Objekt um:

```
1 //erzeugt analog zu ClassLoader CtClass Objekte
2 ClassPool pool = ClassPool.getDefault();
3 CtClass ctBeispielKlasse = pool.makeClass("de.petendi.seminar.
    BeispielKlasse");
4 CtClass ctString = pool.get("java.lang.String");
```

CtField Hier besteht die Möglichkeit bei vorhandenen Objekten Name, Typ und Modifikatoren zu verändern; daneben kann man über die Konstruktoren von `CtField` uninitializierte Felder generieren, für komplexere Operationen existiert auch die Methode `make(...)`, welche ein `CtField` direkt aus Quellcode erstellt. Die innere Klasse `Initializer` definiert, wie und womit das Feld initialisiert werden soll, wenn es einem `CtClass` Objekt hinzugefügt wird.
Im Beispiel wird das `String` Feld *beispielString* mit dem Modifikator `private` hinzugefügt:

```
5 CtField beispielString = new CtField(ctString,"beispielString"
    , ctBeispielKlasse);
6 beispielString.setModifiers(Modifier.PRIVATE);
7 //erlaubt trotz private-Modifikator globalen Zugriff (bei
    Bedarf über die java Reflection API)
8 beispielString.getFieldInfo().setAccessFlags(AccessFlag.PUBLIC
    );
9 ctBeispielKlasse.addField(beispielString);
```

CtConstructor Bereits vorhandenen `CtConstructor` Objekten kann man zusätzlich lediglich einen neuen Rumpf zuweisen; das geschieht per `setBody(...)`, wo entweder von einem anderen `CtConstructor` Objekt der Rumpf kopiert oder er direkt über einen `String` als Quellcode übergeben wird.
`CtConstructor` Objekte erzeugen kann man über Factorymethoden in `CtNewConstructor`, auch hier besteht die Möglichkeit ein vorhandenes zu kopieren oder per `make(...)` ein neues zu generieren.
Das Beispiel wird fortgeführt, indem ein Konstruktor erzeugt wird, welcher einen `String` erwartet und das Feld *beispielString* initialisiert:

```
10 //erzeugt ein CtConstructor-Objekt mit angegebenem Quellcode
       und Referenz auf ctBeispielKlasse
11 CtConstructor  stringConstructor = CtNewConstructor.make
12 ("public BeispielKlasse(String beispielString) {this.
       beispielString = beispielString;}", ctBeispielKlasse);
13 //fügt diesen hinzu
14 ctBeispielKlasse.addConstructor(stringConstructor);
```

CtMethod Hier ist es ähnlich wie zuvor, vorhandenen `CtMethod` Objekten kann man einen neuen Rumpf zuweisen oder den Namen ändern, neue Objekte erzeugt man via Methoden in `CtNewMethod` entweder als Kopie einer vorhandenen Methode oder per `make(...)`, zudem kann man mit `getter(...)` und `setter(...)` unter Angabe eines `CtField` die zugehörige getter- bzw. setter-Methode generieren. Eben dies wird in nachfolgendem Codebeispiel mit dem Feld `beispielString` gemacht:

```
15 //erzeugt eine getter-Methode für beispielString
16 CtMethod getterMethod = CtNewMethod.getter("getBeispielString"
       , beispielString);
17 //fügt diese hinzu
18 ctBeispielKlasse.addMethod(getterMethod);
19 //analog setter-Methode
20 CtMethod setterMethod = CtNewMethod.setter("setBeispielString"
       , beispielString);
21 ctBeispielKlasse.addMethod(setterMethod);
```

Wenn das `CtClass` Objekt komplett ist, kann es umgewandelt werden in ein `Class` Objekt. Nach dieser Umwandlung lassen sich keine Änderung mehr daran vornehmen:

```
22 Class<?> beispielKlasse = ctBeispielKlasse.toClass();
```

3.3 weitere Gesichtspunkte

Im vorhergehenden Abschnitt wurde die Umsetzung reflexiver Operationen in *Java* nur insoweit vorgestellt, dass die Möglichkeiten beschrieben und in Bezug auf das allgemeine Konzept bewertet wurden. Hier soll nun noch kurz auf Aspekte der Sicherheit beim Durchführen reflexiver Operationen sowie deren Zeitaufwand eingegangen werden.

3.3.1 Sicherheit

Reflexion ist ein mächtiges Instrument und schon Smith (vgl. Abschnitt 2.1) forderte deshalb, dass das Durchführen reflexiver Operationen entsprechend sicher gemacht werden muss. Wie bereits angesprochen ist *Java* explizit dafür gebaut worden, um dynamisch neuen Code nachladen zu können und damit verbundene Sicherheitsrisiken zu minimieren. Das Sicherheitsmodell - *Java Sandbox* genannt - besteht aus drei Teilen: *Verifier*, *ClassLoader* und *SecurityManager* (vgl. [10, ch. 2.5-2.8]).

Verifier In Abschnitt 3.2 wurde die Möglichkeit angeführt direkt den ByteCode umzuschreiben, um reflexive Operationen durchzuführen. Der *Verifier* ist das erste Sicherheitskonstrukt,welches der ByteCode bestehen muss, bevor er in einer *JVM* ausgeführt werden kann; neben einigen anderen Konsistenzchecks wird hier auch die Typsicherheit gewährleistet, so kann z.B. garantiert werden, dass ein `Integer` Feld nur genau diesen Typ annimmt oder dass weiterhin eine Methode genau die Parameter ihrer Signatur entgegennimmt und den richtigen Rückgabetyp hat. Ein mit *Javassist* gebautes Feld vom Typ `Integer`, welches aber mit einem `String` initialisiert wird, könnte nicht in die *JVM* gelangen.

ClassLoader Der *ClassLoader* sorgt dafür, dass ByteCode in die *JVM* geladen wird, die Quelle des ByteCodes ist hierbei transparent, so kann dieser auf dem lokalen Dateisystem liegen, über ein Netzwerk geladen werden oder direkt dynamisch erstellt werden (wie es bei *Javassist* der Fall ist). Hierbei sind folgende Sachen zu beachten: ein *SecurityManager* kann das Nachladen von ByteCode verbieten; nach dem Laden muss der Code die Verifikation (s.o.) bestehen; bereits vorhandene Klassendefinitionen, die über einen anderen *ClassLoader* erstellt wurden, können nicht ersetzt werden. Aus genau diesem Grund können bei *Javassist* nach dem Umwandeln in ein `Class` Objekt keine Änderungen mehr vorgenommen werden.

SecurityManager Bevor in *Java* sicherheitskritische Operationen ausgeführt werden können, wird immer der systemeigene *SecurityManager* um Erlaubnis gefragt; ist dieser vorhanden können Operationen wie das Lesen oder Schreiben auf das Dateisystem, Nachladen von Bytecode oder eben auch alle reflexiven Operationen unterbunden werden.

3.3.2 Zeitaufwand

Der Quellcode wird beim Vorgang des Kompilierens in Bytecode immer noch optimiert, so dass der Zugriff auf Felder oder der Aufruf von Variablen sehr schnell durchgeführt werden kann. Es ist ganz klar, dass ein reflexives Durchführen dieser Operationen ein Vielfaches länger dauert; über den Weg eines Metaobjekts muss einiges an Verwaltungsaufgaben durchgeführt werden und zudem lassen sich solche Operationen auch nicht schon im Vorfeld durch einen Compiler optimieren. Hat man also die Möglichkeit reflexive Operationen zu umgehen, sollte dies auf jeden Fall genutzt werden. Genaue Aussagen zum Zeitaufwand sind von vielen Komponenten abhängig (Art und Geschwindigkeit von CPU und Speicher,

konkrete Implementierung der *JVM*, etc.), deshalb lassen sich pauschale Aussagen dazu nur begrenzt treffen. In *Java programming dynamics, Part 2: Introducing reflection* [11] wird ein Benchmark dafür durchgeführt. Die dort getroffenen Ergebnisse wurden allerdings nicht mit der aktuellen *Javaversion* 1.6 und auf bereits veralteter Hardware durchgeführt, daher wurde der Benchmark in folgender Umgebung erneut ausgeführt:

- Prozessor: Intel Core 2 Quad Q6600 64bit; 2.4Ghz
- RAM: 2048MB DDR2; 800Mhz
- OS: Windows Vista Business Edition 64bit
- JavaVersion: 1.6 Update 3; Sun Microsystems Inc.

	Durchschnittszeit in ms		
	reflexiv	**normal**	**Quotient**
Feldzugriff:	12585	57	220,79
Methodenaufruf:	901	41	21,98
Objekterzeugung:	212	121	1,75

Tabelle 1: Ergebnisse des Benchmarks reflexiver Operationen

Der frei verfügbare Quellcode [7] ist auch in Anhang A.7 zu finden, die Zusammenfassung des Benchmarks findet sich in Tabelle 1.
Man kann erkennen, dass, während die Objekterzeugung über Reflexion noch recht performant ist, ein Methodenaufruf bereits gut 20 Mal länger dauert. Reflexiver Feldzugriff dauert über 200 Mal länger und ein extensiver Gebrauch dieser Operationen würde sich damit merklich auf die Performanz einer Applikation auswirken, die hier erhaltenen Ergebnisse kann man allerdings nicht repräsentativ nennen.

4 Zusammenfassung und Ausblick

Vorliegende Arbeit gab einen Überblick der Konzepte von reflexiven objektorientierten Programmiersprachen, es wurde aufgezeigt, welche Anforderungen an diese konzeptuell gestellt werden und wie sich diese Anforderungen praktisch umsetzen lassen. Man kann sagen, dass die Anforderungen von Smith in *Java* und der Erweiterung *Javassist* sauber umgesetzt wurden, so dass auch sicherheitstechnische Aspekte beachtet werden. In Abschnitt 2.3 wurden schon sehr kurz Möglichkeiten angesprochen, die Reflexion in der Softwaretechnik erst neu ermöglicht.
Erst auf Basis dieser neuen Möglichkeiten entwickelte sich ein ganz neues Programmierparadigma - die *aspektorientierte Programmierung*, welche einige Schwächen der objektorientierten Programmierung behebt; der Artikel *Aspect-Oriented Programming using Reflection and Metaobject Protocols* [12] baut auf das hier Beschriebenene auf und gibt eine Einführung in die aspektorientierte Programmierung.

[7] `http://download.boulder.ibm.com/ibmdl/pub/software/dw/library/j-dyn0603.zip`, zuletzt besucht am 29.12.2007

Literatur

[1] I. R. Forman, N. Forman, and P. D. Ira R. Forman, *Java Reflection in Action (In Action series)*. Greenwich, CT, USA: Manning Publications Co., 2004.

[2] B. C. Smith, "Reflection and semantics in a procedural language," Ph.D. dissertation, Massachusetts Institute of Technology, January 1982. [Online]. Available: http://repository.readscheme.org/ftp/papers/bcsmith-thesis.pdf, zuletzt besucht am 29.12.2007

[3] D. P. Friedman and M. Wand, "Reification: Reflection without metaphysics," in *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*. New York, NY, USA: ACM, 1984, pp. 348–355.

[4] P. Wegner, "Dimensions of object-based language design," in *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*. New York, NY, USA: ACM, 1987, pp. 168–182.

[5] G. Kiczales and J. D. Rivieres, *The Art of the Metaobject Protocol*. Cambridge, MA, USA: MIT Press, 1991.

[6] S. Chiba, "Load-time structural reflection in java," in *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2000, pp. 313–336. [Online]. Available: http://www.csg.is.titech.ac.jp/~chiba/pub/chiba-ecoop00.pdf, zuletzt besucht am 29.12.2007

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[8] I. Welch and R. J. Stroud, "From dalang to kava - the evolution of a reflective java extension," in *Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*. London, UK: Springer-Verlag, 1999, pp. 2–21.

[9] J. Kleinoder and M. Golm, "Metajava: an efficient run-time meta architecture for java/sup tm/," in *IWOOOS '96: Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*. Washington, DC, USA: IEEE Computer Society, 1996, p. 54.

[10] G. McGraw and E. W. Felten, *Securing Java: getting down to business with mobile code*. New York, NY, USA: John Wiley & Sons, Inc., 1999. [Online]. Available: http://www.securingjava.com/, zuletzt besucht am 29.12.2007

[11] D. Sosnoski, "Java programming dynamics, part 2: Introducing reflection," *IBM developersworks*, 2003. [Online]. Available: http://www.ibm.com/developerworks/library/j-dyn0603/, zuletzt besucht am 29.12.2007

[12] G. T. Sullivan, "Aspect-oriented programming using reflection and metaobject protocols," *Commun. ACM*, vol. 44, no. 10, pp. 95–97, 2001. [Online]. Available: http://people.csail.mit.edu/gregs/cacm-sidebar.pdf, zuletzt besucht am 29.12.2007

A Quellcode aller Beispiele

A.1 BeispielKlasse

```
1 package de.petendi.seminar;
2 class BeispielKlasse {
3         private String beispielString;
4 
5         public BeispielKlasse(String beispielString) {
6                 this.beispielString = beispielString;
7         }
8 
9         public String getBeispielString() {
10                return beispielString;
11        }
12 
13 }
```

A.2 Constructor

```
1 BeispielKlasse normalErzeugtesObjekt = new BeispielKlasse("
    Test");
2 try {
3 Constructor constructor;
4 //Möglichkeit 1 - vorausgesetzt dies ist der einzig vorhandene
     Konstruktor
5 constructor = BeispielKlasse.class.getConstructors()[0];
6 //Möglichkeit 2 sucht einen Konstruktor, welcher als einzigen
    Parameter ein "String" verlangt
7 constructor = BeispielKlasse.class.getConstructor(java.lang.
    String.class);
8 BeispielKlasse reflexivErzeugtesObjekt = (BeispielKlasse)
    constructor.newInstance("Test");
9 } catch (Exception e) {//Fehlerbehandlung}
```

A.3 Field

```
1 try {
2 BeispielKlasse beispielKlassenObjekt = new BeispielKlasse("
    Test");
3 Field beispielStringFeld = Class.forName("de.petendi.seminar.
    BeispielKlasse").getField("beispielString");
4 //erlaube Zugriff auf das Feld (siehe "AccessibleObject")
5 beispielStringFeld.setAccessible(true);
6 beispielStringFeld.set(beispielKlassenObjekt, "Reflexion");
7 } catch (Exception e) {}
```

A.4 Method

```
1 BeispielKlasse beispielKlassenObjekt = new BeispielKlasse("
    Test");
2 String value;
3 //normaler Methodenaufruf
```

```
value = beispielKlassenObjekt.getBeispielString();
try {
//suche die Methode mit angegebenem Namen und keinen Parametern
Method method =  beispielKlassenObjekt.getClass().getMethod("getBeispielString", null);
value = (String) method.invoke(beispielKlassenObjekt,null);
}
catch(Exception e){}
```

A.5 Proxy

```
interface IBeispielKlasse {

        public abstract String getBeispielString();

        public abstract void setBeispielString(String beispielString);

}

class BeispielKlasse implements IBeispielKlasse {
        private String beispielString;

        public BeispielKlasse(String beispielString) {
                this.beispielString = beispielString;}

        public String getBeispielString() {
                return beispielString;}

}

public class LoggingInvocationHandler implements InvocationHandler {
        //das Objekt, auf welchem der weitergeleitete Methodenaufruf ausgeführt wird
        private Object object;
        public LoggingInvocationHandler(Object object) {
                this.object = object;}
        public Object invoke(Object proxy, Method method, Object[] args)throws Throwable {
        System.out.println(method.getName() +" von "+object.getClass().getSimpleName()+" wurde aufgerufen");
        //das Objekt muss alle Interfaces der Proxyklasse implementieren (dies sollte normalerweise aus Sicherheitsgründen schon VOR Generierung der ProxyKlasse abgeprüft werden)
        return method.invoke(object, args);}
}

public static void main(String[] args) {
        IBeispielKlasse normalesObjekt = new BeispielKlasse("normales Objekt");
        LoggingInvocationHandler invocationHandler = new LoggingInvocationHandler(normalesObjekt);
        Class<?> klassenObjekt =  normalesObjekt.getClass();
```

```
35          Proxy.getProxyClass(klassenObjekt.getClassLoader(),
36                              klassenObjekt.getInterfaces());
37          IBeispielKlasse proxyBeispielKlasse = (IBeispielKlasse
               ) Proxy.newProxyInstance(klassenObjekt.
               getClassLoader(),klassenObjekt.getInterfaces(),
               invocationHandler);
38          //Methodenaufruf über die Proxyklasse (hier wird vor
               Augabe des Wertes auf der Konsole "
               getBeispielString() von BeispielKlasse wurde
               aufgerufen" ausgegeben)
39                System.out.println(proxyBeispielKlasse.
                     getBeispielString());
40          //direkter Methodenaufruf (ohne Ausgabe von
               LoggingInvoationHandler)
41          System.out.println(proxyBeispielKlasse.
               getBeispielString());}
```

A.6 Javassist

```
1  //erzeugt analog zu ClassLoader CtClass Objekte
2  ClassPool pool = ClassPool.getDefault();
3  //erzeugt ein CtClass-Objekt mit angebenem Namen (im Moment
       noch ein Stub ohne Funktionalität)
4  CtClass ctBeispielKlasse = pool.makeClass("de.petendi.seminar.
       BeispielKlasse");
5  //erzeugt ein CtClass-Objekt für das bereits vorhandene Class-
       Object eines Strings
6  CtClass ctString = pool.get("java.lang.String");
7  //erzeugt ein Feld des Typs String, mit angebenem Namen und
       Referenz auf ctBeispielKlasse
8  CtField beispielString = new CtField(ctString,"beispielString"
       , ctBeispielKlasse);
9  //Modifikator "private"
10 beispielString.setModifiers(Modifier.PRIVATE);
11 //erlaubt trotz private-Modifikator globalen Zugriff (bei
       Bedarf über die java Reflection API)
12 beispielString.getFieldInfo().setAccessFlags(AccessFlag.PUBLIC
       );
13 //fügt dieses hinzu
14 ctBeispielKlasse.addField(beispielString);
15 //erzeugt ein CtConstructor-Objekt mit angegebenem Quellcode
       und Referenz auf ctBeispielKlasse
16 CtConstructor  stringConstructor = CtNewConstructor.make
17 ("public BeispielKlasse(String beispielString) {this.
       beispielString = beispielString;}", ctBeispielKlasse);
18 //fügt diesen hinzu
19 ctBeispielKlasse.addConstructor(stringConstructor);
20 //erzeugt eine getter-Methode für beispielString
21 CtMethod getterMethod = CtNewMethod.getter("getBeispielString"
       , beispielString);
22 //fügt diese hinzu
23 ctBeispielKlasse.addMethod(getterMethod);
24 //analog setter-Methode
25 CtMethod setterMethod = CtNewMethod.setter("setBeispielString"
       , beispielString);
26 ctBeispielKlasse.addMethod(setterMethod);
```

```
27 //wandelt das CTClass-Objekt in ein "normales" Class-Objekt um
       , welches ab diesem Zeitpunkt auch ohne Einschränkungen so
       verwendet werden kann
28 Class<?> beispielKlasse = ctBeispielKlasse.toClass();
```

A.7 Benchmark

A.7.1 Feldzugriff:

```
1
2  package timing;
3
4  import java.io.*;
5  import java.lang.reflect.*;
6
7  public class TimeAccesses
8  {
9          public static final int MULTIPLIER_VALUE = 53;
10         public static final int ADDITIVE_VALUE = 4;
11         public static final long PAUSE_TIME = 100;
12
13         protected long m_start;
14         protected boolean m_initialized;
15         protected int m_match;
16         protected int m_value;
17         protected long m_totalTime;
18         protected int m_passCount;
19
20         public class TimingClass
21         {
22                 protected int m_value;
23         }
24
25         protected static void printJVM() {
26                 System.out.println("Java version " +
27                         System.getProperty("java.version"));
28                 String text = System.getProperty("java.vm.name
                       ");
29                 if (text != null) {
30                         System.out.println(text);
31                 }
32                 text = System.getProperty("java.vm.version");
33                 if (text != null) {
34                         System.out.println(text);
35                 }
36                 text = System.getProperty("java.vm.vendor");
37                 if (text == null) {
38                         text = System.getProperty("java.vendor
                               ");
39                 }
40                 System.out.println(text);
41         }
42
43         protected void initTime() {
44                 m_start = System.currentTimeMillis();
```

```
45          }
46
47          protected void reportTime(int pass, int value) {
48                  long time = System.currentTimeMillis() -
                        m_start;
49                  System.out.print(" " + time);
50                  if (m_initialized) {
51                          if (m_match != value) {
52                                  System.out.println("\nError -
                                        result value mismatch");
53                                  System.exit(1);
54                          }
55                  } else {
56                          m_match = value;
57                          m_initialized = true;
58                  }
59                  if (pass == 0) {
60                          m_totalTime = 0;
61                  } else {
62                          m_totalTime += time;
63                          m_passCount = pass;
64                  }
65          }
66
67          protected void reportAverage() {
68                  int avg = (int)((m_totalTime + m_passCount /
                        2) / m_passCount);
69                  System.out.println("\n average time = " + avg
                        + " ms.");
70          }
71
72          protected void pause() {
73                  for (int i = 0; i < 3; i++) {
74                          System.gc();
75                          try {
76                                  Thread.sleep(PAUSE_TIME);
77                          } catch (InterruptedException ex) {}
78                  }
79          }
80
81          public int accessDirect(int loops) {
82                  m_value = 0;
83                  for (int index = 0; index < loops; index++) {
84                          m_value = (m_value + ADDITIVE_VALUE) *
                                MULTIPLIER_VALUE;
85                  }
86                  return m_value;
87          }
88
89          public int accessReference(int loops) {
90                  TimingClass timing = new TimingClass();
91                  for (int index = 0; index < loops; index++) {
92                          timing.m_value = (timing.m_value +
                                ADDITIVE_VALUE) *
93                                      MULTIPLIER_VALUE;
94                  }
```

```
95              return timing.m_value;
96          }
97
98          public int accessReflection(int loops) throws
                Exception {
99              TimingClass timing = new TimingClass();
100             try {
101                 Field field = TimingClass.class.
                        getDeclaredField("m_value");
102                 for (int index = 0; index < loops;
                        index++) {
103                     int value = (field.getInt(
                            timing) + ADDITIVE_VALUE) *
104                             MULTIPLIER_VALUE;
105                     field.setInt(timing, value);
106                 }
107                 return timing.m_value;
108             } catch (Exception ex) {
109                 System.out.println("Error using
                        reflection");
110                 throw ex;
111             }
112         }
113
114         public void runTest(int passes, int loops) throws
                Exception {
115             System.out.println("\nDirect access using
                    member field:");
116             for (int i = 0; i < passes; i++) {
117                 initTime();
118                 int result = accessDirect(loops);
119                 reportTime(i, result);
120                 pause();
121             }
122             reportAverage();
123             System.out.println("Reference access to member
                    field:");
124             for (int i = 0; i < passes; i++) {
125                 initTime();
126                 int result = accessReference(loops);
127                 reportTime(i, result);
128                 pause();
129             }
130             reportAverage();
131             System.out.println("Reflection access to
                    member field:");
132             for (int i = 0; i < passes; i++) {
133                 initTime();
134                 int result = accessReflection(loops);
135                 reportTime(i, result);
136                 pause();
137             }
138             reportAverage();
139         }
140
```

```
141         public static void main(String[] args) throws
                Exception {
142                 printJVM();
143                 TimeAccesses inst = new TimeAccesses();
144                 for (int i = 0; i < 2; i++) {
145                         inst.runTest(5, 10000000);
146                 }
147         }
148 }
```

A.7.2 Methodenaufruf:

```
1
2 package timing;
3
4 import java.io.*;
5 import java.lang.reflect.*;
6
7 public class TimeCalls
8 {
9         public static final int MULTIPLIER_VALUE = 53;
10        public static final int ADDITIVE_VALUE = 4;
11        public static final long PAUSE_TIME = 100;
12
13        protected long m_start;
14        protected boolean m_initialized;
15        protected int m_match;
16        protected int m_value;
17        protected long m_totalTime;
18        protected int m_passCount;
19
20        public class TimingClass
21        {
22                protected int m_value;
23
24                public void step() {
25                        m_value = (m_value + ADDITIVE_VALUE) *
                              MULTIPLIER_VALUE;
26                }
27
28                public int step(int value) {
29                        return (value + ADDITIVE_VALUE) *
                              MULTIPLIER_VALUE;
30                }
31        }
32
33        protected static void printJVM() {
34                System.out.println("Java version " +
35                        System.getProperty("java.version"));
36                String text = System.getProperty("java.vm.name
                      ");
37                if (text != null) {
38                        System.out.println(text);
39                }
40                text = System.getProperty("java.vm.version");
41                if (text != null) {
```

```
42                      System.out.println(text);
43              }
44              text = System.getProperty("java.vm.vendor");
45              if (text == null) {
46                      text = System.getProperty("java.vendor
                           ");
47              }
48              System.out.println(text);
49      }
50
51      protected void initTime() {
52              m_start = System.currentTimeMillis();
53      }
54
55      protected void reportTime(int pass, int value) {
56              long time = System.currentTimeMillis() -
                   m_start;
57              System.out.print(" " + time);
58              if (m_initialized) {
59                      if (m_match != value) {
60                              System.out.println("\nError -
                                   result value mismatch");
61                              System.exit(1);
62                      }
63              } else {
64                      m_match = value;
65                      m_initialized = true;
66              }
67              if (pass == 0) {
68                      m_totalTime = 0;
69              } else {
70                      m_totalTime += time;
71                      m_passCount = pass;
72              }
73      }
74
75      protected void reportAverage() {
76              int avg = (int)((m_totalTime + m_passCount /
                   2) / m_passCount);
77              System.out.println("\n average time = " + avg
                   + " ms.");
78      }
79
80      protected void pause() {
81              for (int i = 0; i < 3; i++) {
82                      System.gc();
83                      try {
84                              Thread.sleep(PAUSE_TIME);
85                      } catch (InterruptedException ex) {}
86              }
87      }
88
89      private void step() {
90              m_value = (m_value + ADDITIVE_VALUE) *
                   MULTIPLIER_VALUE;
91      }
```

```
92
93          public int callDirectNoArgs(int loops) {
94                  m_value = 0;
95                  for (int index = 0; index < loops; index++) {
96                          step();
97                  }
98                  return m_value;
99          }
100
101         private int step(int value) {
102                 return (value + ADDITIVE_VALUE) *
                        MULTIPLIER_VALUE;
103         }
104
105         public int callDirectArgs(int loops) {
106                 int value = 0;
107                 for (int index = 0; index < loops; index++) {
108                         value = step(value);
109                 }
110                 return value;
111         }
112
113         public int callReferenceNoArgs(int loops) {
114                 TimingClass timing = new TimingClass();
115                 for (int index = 0; index < loops; index++) {
116                         timing.step();
117                 }
118                 return timing.m_value;
119         }
120
121         public int callReferenceArgs(int loops) {
122                 TimingClass timing = new TimingClass();
123                 int value = 0;
124                 for (int index = 0; index < loops; index++) {
125                         value = timing.step(value);
126                 }
127                 return value;
128         }
129
130         public int callReflectNoArgs(int loops) throws
                Exception {
131                 TimingClass timing = new TimingClass();
132                 try {
133                         Method method = TimingClass.class.
                                getMethod("step", new Class [0]);
134                         Object[] args = new Object[0];
135                         for (int index = 0; index < loops;
                                index++) {
136                                 method.invoke(timing, args);
137                         }
138                         return timing.m_value;
139                 } catch (Exception ex) {
140                         System.out.println("Error using
                                reflection");
141                         throw ex;
142                 }
```

```
143         }
144
145         public int callReflectArgs(int loops) throws Exception
                {
146             TimingClass timing = new TimingClass();
147             try {
148                     Method method = TimingClass.class.
                            getMethod
149                             ("step", new Class [] { int.
                                class });
150                     Object[] args = new Object[1];
151                     Object value = new Integer(0);
152                     for (int index = 0; index < loops;
                            index++) {
153                             args[0] = value;
154                             value = method.invoke(timing,
                                args);
155                     }
156                     return ((Integer)value).intValue();
157             } catch (Exception ex) {
158                     System.out.println("Error using
                            reflection");
159                     throw ex;
160             }
161         }
162
163         public void runTest(int passes, int loops) throws
                Exception {
164             System.out.println("\nDirect call using member
                    field:");
165             for (int i = 0; i < passes; i++) {
166                     initTime();
167                     int result = callDirectNoArgs(loops);
168                     reportTime(i, result);
169                     pause();
170             }
171             reportAverage();
172             System.out.println("Direct call using passed
                    value:");
173             for (int i = 0; i < passes; i++) {
174                     initTime();
175                     int result = callDirectArgs(loops);
176                     reportTime(i, result);
177                     pause();
178             }
179             reportAverage();
180             System.out.println("Call to object using
                    member field:");
181             for (int i = 0; i < passes; i++) {
182                     initTime();
183                     int result = callReferenceNoArgs(loops
                            );
184                     reportTime(i, result);
185                     pause();
186             }
187             reportAverage();
```

```
188                 System.out.println("Call to object using
                        passed value:");
189                 for (int i = 0; i < passes; i++) {
190                         initTime();
191                         int result = callReferenceArgs(loops);
192                         reportTime(i, result);
193                         pause();
194                 }
195                 reportAverage();
196                 System.out.println("Reflection call using
                        member field:");
197                 for (int i = 0; i < passes; i++) {
198                         initTime();
199                         int result = callReflectNoArgs(loops);
200                         reportTime(i, result);
201                         pause();
202                 }
203                 reportAverage();
204                 System.out.println("Reflection call using
                        passed value:");
205                 for (int i = 0; i < passes; i++) {
206                         initTime();
207                         int result = callReflectArgs(loops);
208                         reportTime(i, result);
209                         pause();
210                 }
211                 reportAverage();
212         }
213
214         public static void main(String[] args) throws
                Exception {
215                 printJVM();
216                 TimeCalls inst = new TimeCalls();
217                 for (int i = 0; i < 2; i++) {
218                         inst.runTest(5, 10000000);
219                 }
220         }
221 }
```

A.7.3 Objekterzeugung:

```
1
2 package timing;
3
4 import java.io.*;
5 import java.lang.reflect.*;
6 import java.lang.reflect.Array;
7
8 public class TimeCreates
9 {
10         public static final long PAUSE_TIME = 100;
11
12         protected long m_start;
13         protected boolean m_initialized;
14         protected int m_match;
15         protected int m_value;
```

```
16          protected long m_totalTime;
17          protected int m_passCount;
18
19          public class TimingClass
20          {
21                  protected int m_value;
22          }
23
24          protected static void printJVM() {
25                  System.out.println("Java version " +
26                          System.getProperty("java.version"));
27                  String text = System.getProperty("java.vm.name
                        ");
28                  if (text != null) {
29                          System.out.println(text);
30                  }
31                  text = System.getProperty("java.vm.version");
32                  if (text != null) {
33                          System.out.println(text);
34                  }
35                  text = System.getProperty("java.vm.vendor");
36                  if (text == null) {
37                          text = System.getProperty("java.vendor
                                ");
38                  }
39                  System.out.println(text);
40          }
41
42          protected void initTime() {
43                  m_start = System.currentTimeMillis();
44          }
45
46          protected void reportTime(int pass) {
47                  long time = System.currentTimeMillis() -
                        m_start;
48                  System.out.print(" " + time);
49                  if (pass == 0) {
50                          m_totalTime = 0;
51                  } else {
52                          m_totalTime += time;
53                          m_passCount = pass;
54                  }
55          }
56
57          protected void reportAverage() {
58                  int avg = (int)((m_totalTime + m_passCount /
                        2) / m_passCount);
59                  System.out.println("\n average time = " + avg
                        + " ms.");
60          }
61
62          protected void pause() {
63                  for (int i = 0; i < 3; i++) {
64                          System.gc();
65                          try {
66                                  Thread.sleep(PAUSE_TIME);
```

```
67                      } catch (InterruptedException ex) {}
68              }
69      }
70
71      public void createObjectDirect(Object[] objs) {
72              for (int i = 0; i < objs.length; i++) {
73                      objs[i] = new Object();
74              }
75      }
76
77      public void createObjectReflect(Object[] objs) {
78              try {
79                      Class oclas = Object.class;
80                      for (int i = 0; i < objs.length; i++)
                            {
81                              objs[i] = oclas.newInstance();
82                      }
83              } catch (Exception ex) {
84                      ex.printStackTrace();
85                      System.exit(1);
86              }
87      }
88
89      public void createArrayDirect(int size, Object[] objs)
            {
90              for (int i = 0; i < objs.length; i++) {
91                      objs[i] = new byte[size];
92              }
93      }
94
95      public void createArrayReflect(int size, Object[] objs
            ) {
96              for (int i = 0; i < objs.length; i++) {
97                      objs[i] = Array.newInstance(byte.class
                            , size);
98              }
99      }
100
101     public void runTest(int passes, int loops) throws
            Exception {
102             Object[] objs = new Object[loops];
103             System.out.println("\nDirect Object creation:"
                    );
104             for (int i = 0; i < passes; i++) {
105                     initTime();
106                     createObjectDirect(objs);
107                     reportTime(i);
108                     pause();
109             }
110             reportAverage();
111             System.out.println("Reflection Object creation
                    :");
112             for (int i = 0; i < passes; i++) {
113                     initTime();
114                     createObjectReflect(objs);
115                     reportTime(i);
```

```
116                     pause();
117                 }
118                 reportAverage();
119                 System.out.println("Direct byte[8] creation:")
                        ;
120                 for (int i = 0; i < passes; i++) {
121                     initTime();
122                     createArrayDirect(8, objs);
123                     reportTime(i);
124                     pause();
125                 }
126                 reportAverage();
127                 System.out.println("Reflection byte[8]
                        creation:");
128                 for (int i = 0; i < passes; i++) {
129                     initTime();
130                     createArrayReflect(8, objs);
131                     reportTime(i);
132                     pause();
133                 }
134                 reportAverage();
135                 System.out.println("Direct byte[64] creation:"
                        );
136                 for (int i = 0; i < passes; i++) {
137                     initTime();
138                     createArrayDirect(64, objs);
139                     reportTime(i);
140                     pause();
141                 }
142                 reportAverage();
143                 System.out.println("Reflection byte[64]
                        creation:");
144                 for (int i = 0; i < passes; i++) {
145                     initTime();
146                     createArrayReflect(64, objs);
147                     reportTime(i);
148                     pause();
149                 }
150                 reportAverage();
151         }
152
153         public static void main(String[] args) throws
                Exception {
154             printJVM();
155             TimeCreates inst = new TimeCreates();
156             for (int i = 0; i < 2; i++) {
157                 inst.runTest(5, 500000);
158             }
159         }
160 }
```