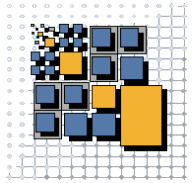


Reflexive Programmiersprachen

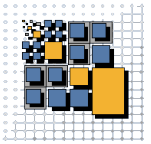
Konzepte und Umsetzung in Java

Jan Petendi

01.02.08

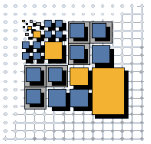
Lehrstuhl für Praktische Informatik
Fakultät WIAI
Otto-Friedrich-Universität Bamberg

Einführung



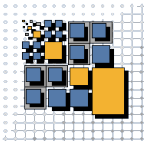
- Entwicklung neuer Programmierkonzepte:
  - Nicht: Schaffung neuer Funktionalität
    - Letztendlich immer Umwandlung in Maschinen-Sprache
  - Sondern - Erleichterung für den Softwareentwickler:
    - Konzentration auf die inhaltliche Umsetzung
    - Grundlagen für robusten Code *Typsicherheit, Garbage Collector ...*
    - Softwarequalität *Wiederverwendbarkeit, Portabilität, Wartbarkeit*
- Anwendungsgebiete reflexiver Operationen:
  - Debugging *externe Überwachung, Fehlerfindung*
  - lernende Systeme *automatische Umwandlung von Fakten in Code*
  - verteilte Systeme Anpassen *inkomp. Interfaces , Austausch von defekten Komponenten*
  - Softwaretechnik *Metaprogramming => Compiler*
  - .....

Reflexion



- Allgemeiner Sprachgebrauch:
  - die rein passive Eigenschaft aufgrund einer Einwirkung von außen etwas zurückzugeben
  - die Fähigkeit den eigenen Zustand und die Umgebung wahrzunehmen und dementsprechend zu handeln
- reflexive Programmiersprache:
  - Laufendes Programm bietet Möglichkeiten Kenntnis über die globale Struktur und den Zustand zu erhalten
  - Laufendes Programm hat selbst Kenntnis davon und kann darauf Einfluss nehmen

Forschung

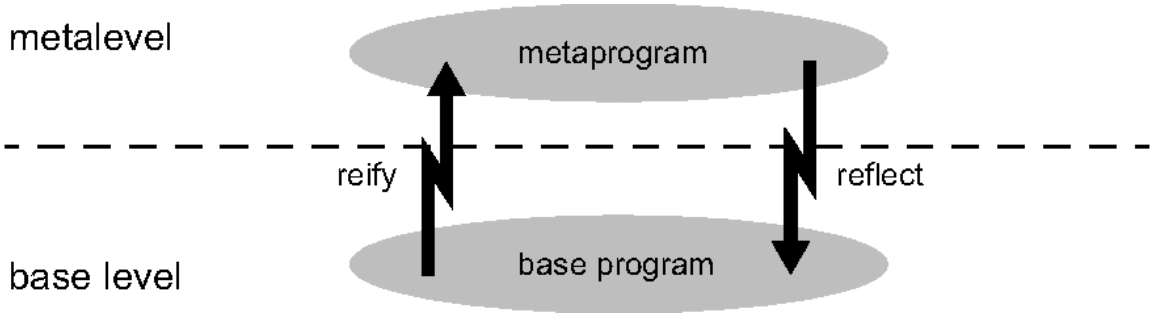


- B.C. Smith „Reflection and semantics in a procedural language“ Ph.D.Dissertation, M.I.T. ,1982
  - Laufendes Programm als System:
    1. es muss eine Repräsentation dieses Systems vorhanden sein, auf welche das System selbst Zugriff hat
    2. zwischen der Repräsentation des Systems und dem System selbst muss eine kausale Verbindung bestehen
    3. diese kausale Verbindung muss robust sein

    vorhandener Quelltext ausreichend?

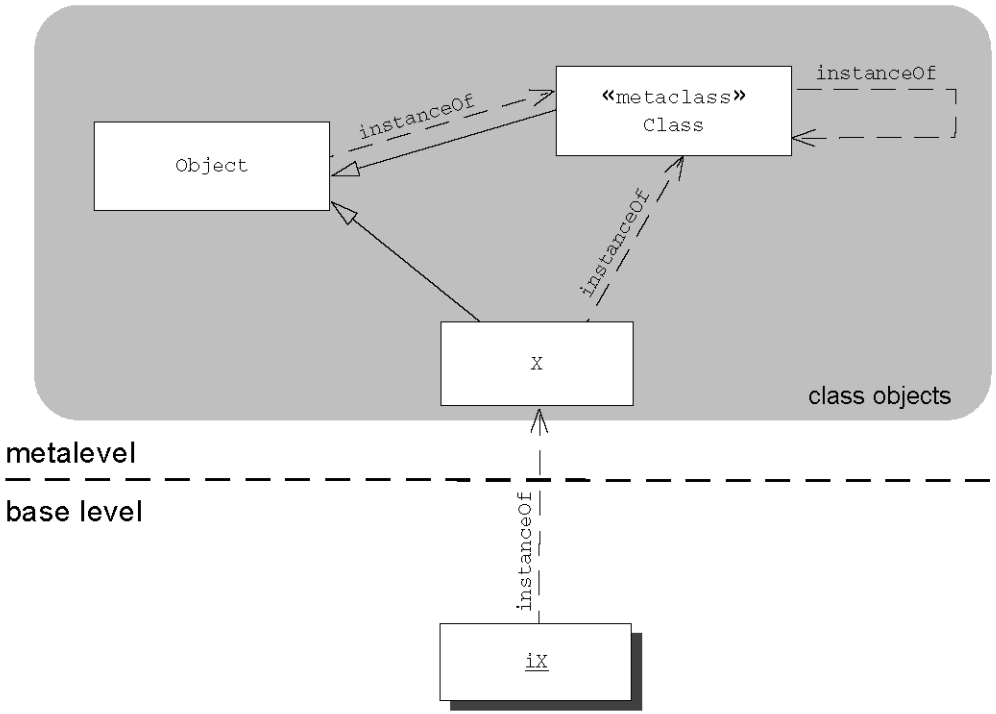
  - Ungeeignet:
    - Ständiges Parsen viel zu zeitaufwändig
    - Zustand (Variablenbelegung etc.) unersichtlich
    - Keine kausale Verbindung

Forschung (2)



- Friedman und Wand "Reification: Reflection without metaphysics," 1984
  - „Verdinglichung" des laufenden Programms => **Metaobjekt**
  - Manipulation dieses Objekts
  - Reintegration des manipulierten Objekts in das laufende Programm

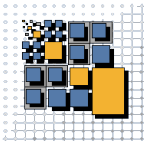
Reflexive OO-Sprachen



Anforderungen:

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

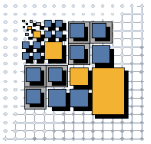
Begriffsdefinitionen



- Metaobjekt *kapselt Struktur und Verhalten einer laufenden Applikation und bietet Methoden darauf Einfluss zu nehmen*
- Klassenobjekt *stellt die Reifizierung einer Klasse dar*
- Metaklasse *ist eine Klasse deren Instanzen Klassen sind*
- Metaklassenobjekt *stellt die Reifizierung einer Metaklasse dar*

- Anmerkungen:
  - Klassenobjekte sind Metaobjekte
  - nicht alle Klassen sind Metaklassen(sie instanziieren *normale* Objekte)
  - nicht alle Metaobjekte sind Metaklassenobjekte (*Constructor, Field, Method können keine Klassen instanziieren)*

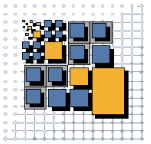
Reflexive Operationen



- Introspection *untersucht die vorhandene Struktur und den Zustand*
  - Auflisten aller Klassen
  - Auflisten aller Methoden, Felder, Konstruktoren einer Klasse
  - Auflisten aller implementierten Interfaces einer Klasse
  - Auslesen des Wertes eines Feldes
- Intercession *nimmt aktiv Einfluss auf eine laufende Applikation*
  - Behavioural Reflection *beeinflusst Verhalten ohne Strukturänderungen*
    - Setzen des Wertes einer Variablen
    - Aufrufen einer Methoden auf einem Objekt
    - Abfangen und Umleiten eines Methodenaufrufs
  - Structural Reflection *nimmt Strukturänderungen vor*
    - Hinzufügen neuer Klassen
    - Hinzufügen/Ändern von Methoden,Felder,Konstruktoren zu Klassen
    - Ändern der Superklasse

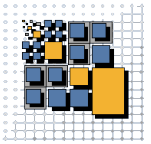
Bewertung – sinnvoll, gefährlich?

Unterstützung reflexiver Operationen

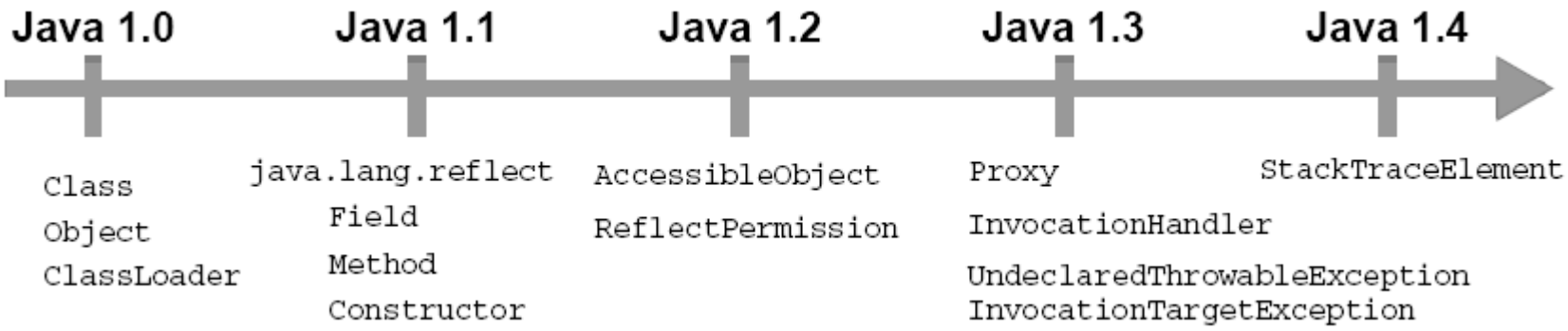


- Beim Kompilieren des Codes
  - Modifikation des Quellcodes und kompilieren mit den Änderungen
  - Nachteil: Quellcode muss vorhanden sein, sehr zeitaufwändig bei vielen Änderungen zur Laufzeit
- Vor Zugriff der Laufzeitumgebung auf den Code
  - Modifikation des kompilierten Codes:
    - keine Änderung vorhandenen Codes, sondern Umleitung der Programmlogik („hooks“)
    - Generierung von neuem Code und Umschreiben des vorhandenen
  - Nachteil: in Ausführung befindlicher Code nicht (bzw. nur begrenzt) veränderbar
- Während der Codeausführung
  - Laufzeitumgebung stellt selbst die erforderliche Funktionalität bereit
  - Nachteil: Portabilität nicht mehr gewährleistet

Java Reflection API

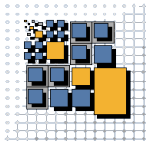


- **write once, run anywhere**
- konzipiert für dynamische Umgebungen
- JVM stellt die reflexive Funktionalität zur Verfügung



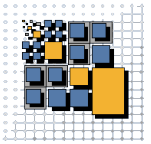
- Java 1.5:
  - *generics : Class<T>*
  - *autoboxing: int x = new Integer(1); Integer y = 1;*
  - *varargs: compute(String ... args); compute(String[] args);*

Beispielklasse



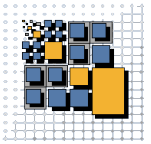
```
4 class BeispielKlasse implements IBeispielKlasse{
5     private String beispielString;
6
7     public BeispielKlasse(String beispielString) {
8         this.beispielString = beispielString;
9     }
10
11    /**
12     * @see de.petendi.seminar.IBeispielKlasse#getBeispielString()
13     */
14    public String getBeispielString() {
15        return beispielString;
16    }
17
18    /**
19     * @see de.petendi.seminar.IBeispielKlasse#setBeispielString(java.lang.String)
20     */
21    public void setBeispielString(String beispielString) {
22
23        this.beispielString = beispielString;
24
25    }
26 }
```

Class



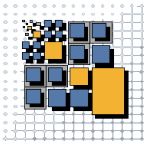
- Zugriff auf das Klassenobjekt über:
  - *beispielKlassenObjekt.getClass()*
  - *de.petendi.seminar.BeispielKlasse.class*
  - *Class.forName(String klassenName) throws ClassNotFoundException*
- Methodenauswahl von Class:
  - *getModifiers()*
  - *getSuperclass()*
  - *getInterfaces()*
  - *getPackage()*
  - *getDeclaredConstructors()*
  - *getDeclaredFields()*
  - *getDeclaredMethods()*

Constructor



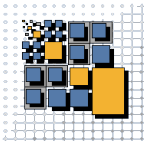
```
BeispielKlasse normalErzeugtesObjekt = new BeispielKlasse("Test");
try {
    Constructor constructor;
    // Möglichkeit 1 - vorausgesetzt dies ist der einzig vorhandene
    // Konstruktor
    constructor = BeispielKlasse.class.getConstructors()[0];
    // Möglichkeit 2 sucht einen Konstruktor, welcher als einzigen
    // Parameter
    // ein "String" verlangt
    constructor = BeispielKlasse.class
            .getConstructor(java.lang.String.class);
    BeispielKlasse reflexivErzeugtesObjekt = (BeispielKlasse) constructor
            .newInstance("Test");
    System.out.println("reflexiv erstelltes Objekt: "
            + reflexivErzeugtesObjekt);
} catch (Exception e) {
    // zu behandelnde Ausnahmen: SecurityException,
    // NoSuchMethodException, IllegalArgumentException,
    // InstantiationException, InvocationTargetException
}
```

Field



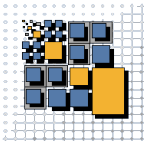
```
try {
    BeispielKlasse beispielKlassenObjekt = new BeispielKlasse("Test");
    Field beispielStringFeld = Class.forName(
            "de.petendi.seminar.BeispielKlasse").getDeclaredField(
            "beispielString");
    System.out
            .println("getBeispielString-Methodenaufruf vor dem Setzen: "
                    + beispielKlassenObjekt.getBeispielString());
    // erlaube Zugriff auf das Feld (siehe "AccessibleObject")
    beispielStringFeld.setAccessible(true);
    beispielStringFeld.set(beispielKlassenObjekt, "Reflexion");
    System.out
            .println("getBeispielString-Methodenaufruf nach dem Setzen: "
                    + beispielKlassenObjekt.getBeispielString());
} catch (Exception e) {
    // zu behandelnde Ausnahmen: SecurityException,
    // IllegalArgumentException, IllegalAccessException,
    // NoSuchFieldException(, ClassNotFoundException)
}
```

Method



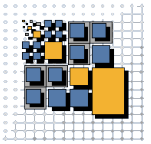
```
BeispielKlasse beispielKlassenObjekt = new BeispielKlasse("Test");
String value;
// normaler Methodenaufruf
value = beispielKlassenObjekt.getBeispielString();
try {
    // suche die Methode mit angegebenem Namen und keinen Parametern
    Method method = beispielKlassenObjekt.getClass().getMethod(
            "getBeispielString",(Class) null);
    value = (String) method.invoke(beispielKlassenObjekt, (Class) null);

    System.out.println("Reflexiver getBeispielString-Methodenaufruf: "
            + value);
}

catch (Exception e) {
    // zu behandelnde Ausnahmen: SecurityException,
    // IllegalArgumentException, IllegalAccessException,
    // NoSuchMethodException, InvocationTargetException

}
```

Proxy



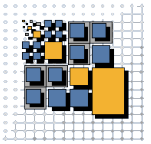
- Proxy
  - Class<?> getProxyClass(ClassLoader loader, Class[] interfaces)
  - Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)
- interface InvocationHandler
  - Object invoke(Object proxy, Method method, Object[] args)

Proxy Beispiel



```
public class LoggingInvocationHandler implements InvocationHandler {
    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");
        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();
        Proxy.getProxyClass(klassenObjekt.getClassLoader(),
                klassenObjekt.getInterfaces());
        IBeispielKlasse proxyBeispielKlasse = (IBeispielKlasse) Proxy
                .newProxyInstance(klassenObjekt.getClassLoader(),
                        klassenObjekt.getInterfaces(),
                        invocationHandler);
        System.out.println(proxyBeispielKlasse.getBeispielString());}

    }
```

Exkurs: RMI Stubs seit Java 1.5

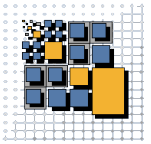


```
IExtendsRemote remote = (IExtendsRemote)
  LocateRegistry.getRegistry.lookup(„IExtendsRemote“);
```

- Bis 1.5:
  - Benutzung von *rmic*, um Stubs zu generieren
  - *StubNotFoundException*, falls ein Stub nicht vorhanden
- Heute:
  - Dynamische Erzeugung eines Proxy, falls Stub nicht vorhanden
    - RemoteObjectInvocationHandler(RemoteRef ref)
  - Vorhandene Stubs werden ignoriert mit:

```
System.setProperty(„java.rmi.server.ignoreStubClasses“,“true“);
```

Exkurs: RMI Stubs seit Java 1.5 (2)

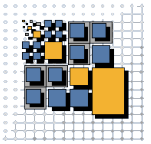


```
public static Remote createProxy(Class implClass, RemoteRef clientRef,
        boolean forceStubUse) throws StubNotFoundException {
    Class remoteClass;

    try {
        remoteClass = getRemoteClass(implClass);
    } catch (ClassNotFoundException ex) {
        throw new StubNotFoundException(
                "object does not implement a remote interface: "
                        + implClass.getName());
    }

    if (forceStubUse
            || !(ignoreStubClasses || !stubClassExists(remoteClass))) {
        return createStub(remoteClass, clientRef);
    }

    ClassLoader loader = implClass.getClassLoader();
    Class[] interfaces = getRemoteInterfaces(implClass);
    InvocationHandler handler = new RemoteObjectInvocationHandler(clientRef);

    /* REMIND: private remote interfaces? */

    try {
        return (Remote) Proxy.newProxyInstance(loader, interfaces, handler);
    } catch (IllegalArgumentException e) {
        throw new StubNotFoundException("unable to create proxy", e);
    }
```

Quelltextausschnitt von sun.rmi.server.Util

Quelle: http://download.java.net/jdk6/6u3/promoted/b05/index.html

Bewertung der java reflection API

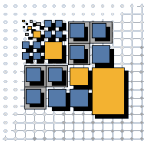


- Introspection vollständig unterstützt
- Intercession:
  - Erzeugung von Instanzen vorhandener Klassen
  - Methodenaufruf vorhandener Objekte
  - Ändern von Feldwerten
  - Proxy
- ausreichend zur Umsetzung vollständiger Intercession?

„Direkt nach Starten der Applikation werden alle vorhandenen Klassenobjekte durch Proxyklassen ersetzt und alle auftretenden Instanzen 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.“

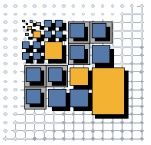
- *Nein:*
  - Proxy unterstützt nur interface-Methoden
  - Objekte innerhalb von Methodenrümpfen nicht ersetzbar

Javassist



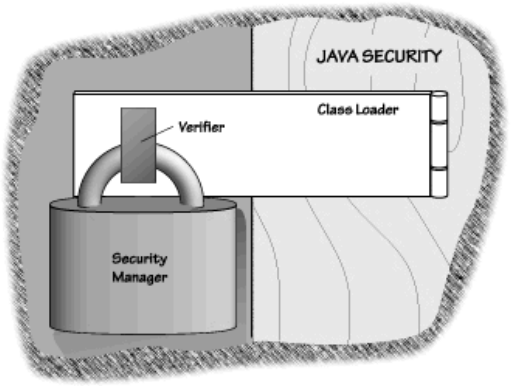
- http://labs.jboss.com/javassist/
- **Loadtime** structural Reflection
  - Kapselung des Bytecodes in sog. CompileTime Konstrukte
    - *CtClass, CtConstructor, CtField, CtMethod*
    - Methodensignaturen der standard Reflection API
    - Zusätzlich Methoden für *Intercession*
- Allgemeines Vorgehen:
  1. Erzeugung eines CTClass-Objekts
  2. Durchführung beliebiger reflexiver Operationen mit diesem Objekt
  3. Umwandlung des CtClass Objekts in ein *normales* Class Objekt

Javassist Beispiel



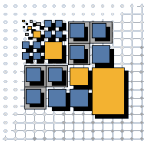
```
ClassPool pool = ClassPool.getDefault();
//erzeugt ein CtClass-Objekt mit angebenem Namen (im Moment noch ein Stub ohne Funktionalität)
CtClass ctBeispielKlasse = pool.makeClass("de.petendi.seminar.BeispielKlasse");
//erzeugt ein CtClass-Objekt für das bereits vorhandene Class-Object eines Strings
CtClass ctString = pool.get("java.lang.String");
//erzeugt ein Feld des Typs String, mit angebenem Namen und Referenz ctBeispielKlasse
CtField beispielString = new CtField(ctString,"beispielString", ctBeispielKlasse);
//Modifikator "private"
beispielString.setModifiers(Modifier.PRIVATE);
//erlaubt trotz private-Modifikator globalen Zugriff (bei Bedarf über die java Reflection API)
beispielString.getFieldInfo().setAccessFlags(AccessFlag.PUBLIC);
//fügt dieses hinzu
ctBeispielKlasse.addField(beispielString);
//erzeugt ein CtConstructor-Objekt mit angegebenem Quellcode und Referenz auf ctBeispielKlasse
CtConstructor stringConstructor = CtNewConstructor
        .make(
                "public BeispielKlasse(String beispielString) {this.beispielString = beispielString;}",
                ctBeispielKlasse);
//fügt diesen hinzu
ctBeispielKlasse.addConstructor(stringConstructor);
//erzeugt eine getter-Methode für beispielString
CtMethod getterMethod = CtNewMethod.getter("getBeispielString", beispielString);
//fügt diese hinzu
ctBeispielKlasse.addMethod(getterMethod);
//analog setter-Methode
CtMethod setterMethod = CtNewMethod.setter("setBeispielString", beispielString);
ctBeispielKlasse.addMethod(setterMethod);
// wandelt das CTClass-Objekt in ein "normales" Class-Objekt um, welches
// ab diesem Zeitpunkt auch ohne Einschränkungen verwendet werden kann
Class<?> beispielKlasse = ctBeispielKlasse.toClass();
```

Sicherheit



- „Java Sandbox“
  - Verifier *garantiert u.a. Typsicherheit*
    - IntegerFeld.set(refObjekt,“Test“) nicht möglich
  - ClassLoader *verwaltet das transparente Laden von Bytecode*
    - Vorhandener ByteCode lässt sich nicht über anderen ClassLoader ersetzen
    - Javassist --> **Loadtime** structural Reflection
  - Securitymanager *kann sicherheitskritische Operationen unterbinden*
    - ReflectionPermission

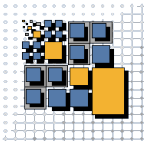
Zeitaufwand



- Allgemein
  - hoher Verwaltungsaufwand (Metaobjekt, Sicherheit, ...)
  - keine Optimierung (Compiler) möglich
  - konkrete Angaben schwierig (JVM Implementierung, Prozessor,RAM...)
- Benchmark
  - http://www.ibm.com/developerworks/library/j-dyn0603/
  - Core 2Quad 2.4Ghz,2048MB RAM,Vista Business 64bit
  - Sun Java 1.6 Update 3

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

Fazit



- Grundlage für viele Anwendungsgebiete
- Basis für „flexibles Programmieren“
- Zusammenspiel mit *Annotations*
- Metaprogramming

- ***Aspektorientierung***

Danke für die Aufmerksamkeit