Non Standard MOPs for Java. OpenJava and Javassist.

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano

Outline

1. Non Standard MOPs
   - Non Standard MOPs Overview

2. OpenJava
   - OpenJava Overview
   - Example: Verbose Execution

3. Javassist
   - Javassist Overview
   - Example of Binary Component Adaptation
   - Example: How to Simulate the Meta-Object Approach with Javassist

Non Standard MOPs Overview

Why We Need Non Standard MOPs

We have provided some hints about how to use Java's MOP
- a set of functional extensions (MOP class here);
- a set of reusable class-to-class transformations;
- examples of large grain use for adaptable applications.

Should be clear that reflection in Java is not as powerful as it could be
- it is missing the appropriate intercessional features

Some restrictions could be got around by code generation and dynamic compilation, but it is tricky and error-prone.

Different Kinds of MOPs for Java

Compile Time
- reflect language constructs;
- based on source to source transformations;
- per class basis changes.

Some MOPs: OpenJava (Tatsubori and Chiba 1999), Reflective Java.

Load Time
- reflect on the bytecode;
- based on the specialization of the class loader;
- the source is not necessary;
- per class basis changes.

Some MOPs: BCEL (Markus Dahm 1999), Javassist (Shigeru Chiba 2000).
Non Standard MOPs.

Different Kinds of MOPs for Java (Cont’d).

Run-Time (VM-based)
- modification of the VM to intercept events (e.g., message sends);
- lose of portability;
- can be per object basis.

Some MOPs: MetaXa (Michael Gölm 1999) and Iguana/J (B. Redmond and V. Cahill 2000).

Run-Time (Proxy-Based)
- transparent generation of interceptors (simple delegation);
- can be based on source or bytecode generation/modification (can make use of compile/load-time MOPs for implementation);
- can be per object basis

OpenJava.
Characteristics.

OpenJava works on a per-class basis associating to every class another class (the meta-object) that govern the translation process.

Meta-objects and objects interact through a well-defined MOP.

The OpenJava's work is based on a Java to Java translation.

The meta-objects represent the program in execution in the meta-level.

How to Program with OpenJava.

The reflective programming in OpenJava mainly takes three steps:

- we have to decide how the base-level program has to look after the translation; then
- we have to detect which part of the base-level code will be involved in the translation and to decide which ancillary code is necessary during the translation; and
- we have to write a meta-object that will carry out the designed translation.

Notwithstanding that the translation is carried out at compile-time, the reflective API is very simple and looks like the ones provided by java.lang.reflect.

Example: Verbose Execution.

```
public class Hello {
    public static void main(String[] args) {
        hello();
    }
    static void hello() {
        System.out.println("Hello, world.");
    }
}
```

The base-level code, excluding the `instantiates` statement, is pure Java code.

The statement:

```
    instantiates VerboseClass
```

says to the compiler that the class Hello has a meta-object instance of the class VerboseClass.

The class VerboseClass describes how the class Hello has to be translated.

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("main is called.");
        hello();
    }
    static void hello() {
        System.out.println("hello is called.");
        System.out.println("Hello, world.");
    }
}
```

Figured out how the code should look we have to write a meta-object to do that.
The class `VerboseClass` generates the code to put into the class `Hello`.

```java
import openjava.mop.*;
import openjava.ptree.*;
public class VerboseClass extends OJClass {
    public void translateDefinition() throws MOPException {
        OJMethod[] methods = getDeclaredMethods();
        for (int i = 0; i < methods.length; ++i) {
            Statement printer = makeStatement("System.out.println( "Nome del metodo: " + methods[i] + ");");
            methods[i].getBody().insertElementAt(printer, 0);
        }
    }
}
```

To implement such a translation we had written the class `VerboseClass` associated to the class `Hello`:
- it extends the class `openjava.mop.OJClass`;
- it overrides the method `translateDefinition()` that translates the body of the methods;
- the methods declared in the base-level classes are retrieved thanks to a call to the `getDeclaredMethods()`;
- the call to the method `makeStatement()` allows to build a statement from a string on-the-fly;
- the method `getBody()` returns a list of statements representing the body of the method; statements can be added to or removed from that representation.

Javassist is a class library for editing Java bytecode and allowing to:
- modify a class at load time (load-time MOP);
- create a new class at run time.

Implemented as a high-level API, does not require knowledge of bytecode specifications
- 100% Java, portable (JVM <1.2 - based on class loading model)

Javassist is largely used software (e.g., it is part of the jBoss architecture), it has been developed by Shigeru Chiba (Tokyo Institute of Technology) and is freely available at:
http://www.is.titech.ac.jp/~chiba/javassist
Non Standard MOPs.

Javassist Overview.

Example of Binary Component Adaptation.

Example: How to Simulate the Meta-Object Approach with Javassist.

Javassist.

Javassist Core API.

High-level API: source code abstractions.

Improvement of Java structural reflection by another reification of Java classes and primitive types:

- CtClass, CtMethod, CtConstructor, CtField, FieldInitializer, CtPrimitiveType, Modifier ...

Introduction of new classes describing class loading and modifying:

- Loader, ClassPool, ClassPath
- Translator.

Javassist.

ClassPool and Translator.

ClassPool
- get(String): returns a reference to a CtClass;
- write(*): translates a CtClass into a file and write it to output stream;
- insertClassPath, appendClassPath: manage class lookup process;
- makeClass, makeInterface: create class from scratch;
- static services to get the class pool with or without Translator.

Translator: the entry point for transformations
- start(ClassPool): translator initialization;
- onWrite(ClassPool, String): before the class is given to the loader - time for modifications.

Javassist.

CtClass vs Class.

Provides the protocols for operating on a class (before it is loaded into the JVM) or for dynamic creation of new classes.

Introspection
- same as java.lang.reflect;
- isModified, isFrozen (already loaded, cannot be modified).

Intercession
- setName, setSuperclass, setModifiers, ...;
- addInterface, addField, addMethod, ...;
- replaceClassName: replace classname occurrences within a class (using a map).

Transformation
- getClassFile: return a ClassFile object (javassist.bytecode), for low-level bytecode manipulation;
- instrument(CodeConverter): apply converter on methods and constructors.

Clean but limited
- restrictions on type, why? it doesn't break the encapsulation.
Javassist.
Example: Binary Component Adaptation.

Update classes so that they fit into a framework, transforming their bytecode automatically.

- We have a class `Calendar` implementing a third-party interface `Writable`:

```java
class Calendar implements Writable {
    public void write(ostream s) { ... }
}
```

- In a new version of the class library, `Writable` has been replaced by `Printable` (it also defines `print()`): we have to update `Calendar`.

```java
class Calendar implements Printable {
    public void write(ostream s) {...}
    public void print() { this.write(System.out); }
}
```

Javassist.
Example: Binary Component Adaptation (Cont'd).

```java
class Exemplar implements Printable {
    public void write(ostream s) { ... }
    public void print() { write(System.out); }
}

public class RunAdaptation {
    public static void main(String[] args) {
        Adapter adapt = new Adapter();
        ClassPool pool = ClassPool.getDefault(adapt);
        Loader cl = new Loader(pool);
        cl.run("MyApplication", args);
    }
}

class Adapter implements Translator {
    CtMethod printM; CtClass printable;
    public void start(ClassPool pool) {
        this.printable = pool.get("Printable");
        CtClass ex = pool.get("Exemplar");
        this.printM = ex.getDeclaredMethod("print", []);
    }
    public void onWrite(String cn, ClassPool p) {
        CtClass c = p.get(cn);
        CtClass[] interfaces = c.getInterfaces();
        for (int i=0; i<interfaces.length; i++)
            if (interfaces[i].getName().equals("Writable")) {
                interfaces[i] = printable;
                c.setInterfaces(interfaces);
                c.addMethod(CtNewMethod.copy(printM, c, null));
                return;
            }
    }
}
```

Javassist.
Javassist Bytecode-Level API.

`javassist.bytecode.*`;
- low-level API: bytecode abstractions;
- core API services are implemented with it;
- can be used for finer-grained transformations:
  - `ClassFile` is the base entity for bytecode editing;
    - `ClassFile = <ConstPool, Methods, Fields, Attributes>;`
    - obtained from `CtClass`, stream or from scratch;
    - write to output stream;
  - `OpCode` defines all bytecode instructions as constants
    - `ALOAD, ASTORE, INVOKEVIRTUAL, ...`

Javassist.
ClassFile.

**Intercession**
- `addAttribute(AttributeInfo)``
- `addMethod(MethodInfo)``
- `addField(FieldName)``
- `renameClass, setName``
- `setAccessFlag``
- `setInterfaces, setSuperclass`.  

**Introspection**
- `getConstPool, getAttribute, getMethod, ...;`
- `isFinal, isAbstract, isInterface`
Javaassist.

XXXInfo (MethodInfo, FieldInfo, ...).

All XXXInfo classes have methods for introspection and interception:
- e.g., MethodInfo has getCodeAttribute().

CodeAttribute is a structure with raw bytecode:
- (byte[]) + info (length, max stack, exceptions, ...)

CodeIterator is used to parse and manipulate raw code:
- insert code, replace code, ...;
- go to next instruction ...

Walter Cazzola
OpenJava and Javassist
Slide 25 of 31

Javaassist.
Example: How to Simulate the Meta-Object Approach.

Example: How to Simulate the Meta-Object Approach (Cont'd).

package sample.reflect;
import javassist.reflect.*;
import javassist.reflect.Metaobject;

public class Person {
  public String name;
  public static int birth = 3;

  public Person(String name, int birthYear) {
    name = name;
    birth = birthYear;
  }

  public String getName() {
    return name;
  }

  public int getAge(int year) {
    return year - birth;
  }

  public static void main(String[] args) {
    Person p = new Person(args[0], 1960);
    System.out.println("name: " + p.getName());
    if (p instanceof Metalevel) {
      ((Metalevel)p)._setMetaobject(new Metaobject(p, null));
      System.out.println("«the metaobject has been turned off.»");
    }
    System.out.println("age: " + p.getAge(1999));
  }
}

Walter Cazzola
OpenJava and Javassist
Slide 26 of 31

Javaassist.
Example: How to Simulate the Meta-Object Approach (Cont'd).

package sample.reflect;
import javassist.reflect.*;
import javassist.reflect.ClassMetaobject;
import javassist.reflect.Loader;

public class Main {
  public static void main(String[] args) throws Throwable {
    Loader cl = (Loader) Main.class.getClassLoader();
    cl.makeReflective("sample.reflect.Person", "sample.reflect.VerboseMetaobj",
                    "javassist.reflect.ClassMetaobject");
    cl.run("sample.reflect.Person", args);
  }
}

Walter Cazzola
OpenJava and Javassist
Slide 27 of 31

Javaassist.
Example: How to Simulate the Meta-Object Approach (Cont'd).

package sample.reflect;
import javassist.reflect.*;
import javassist.reflect.ClassMetaobject;
import javassist.reflect.Loader;

public class Main {
  public static void main(String[] args) throws Throwable {
    Loader cl = (Loader) Main.class.getClassLoader();
    cl.makeReflective("sample.reflect.Person", "sample.reflect.VerboseMetaobj",
                    "javassist.reflect.ClassMetaobject");
    cl.run("sample.reflect.Person", args);
  }
}

[16:18]cazzola@ulik>java javassist.reflect.Loader sample.reflect.Main Gastone
** constructed: sample.reflect.Person
** field write: name
** trap: getName() in sample.reflect.Person
** file read: name
name: Gastone
*the metaobject has been turned off.*
age: 25
MOPs vs Object-Oriented Inheritance.

Conclusions.

Pros:
- reifying the elements and the state of programs as instances of classes;
- using these classes to define protocols describing: object creation/destruction, message sending, error handling, class inheritance, …;
- using class inheritance to specialize these classes and the associated protocols,

Cons:
- a potential confusion between a class (meta-class) and a meta-object;
- introduce complex meta-level architectures (cf. Smalltalk, CLOS);
- or too simple (cf. Java).

Adaptability/flexibility/transparency vs efficiency/security?
- partial evaluation (genericity vs specialization);
- configurable reflective systems.

Complicated semantics
- no universal and well-defined theory;
- methodology to define crosscutting meta-objects?
- composition rules for meta-objects are still missing.

Reflection seems a promising software technology to deal with adaptability and separation of concerns when the application domain is complex.

Referents

- Shigeru Chiba.
  Load-Time Structural Reflection in Java.

- Shigeru Chiba, Michiaki Tatsubori, Marc-Olivier Killijian, and Kozo Itano.
  OpenJava: A Class-based Macro System for Java.

- Michiaki Tatsubori.
  An Extension Mechanism for the Java Language.
  Master of engineering dissertation, Graduate School of Engineering, University of Tsukuba, University of Tsukuba, Ibaraki, Japan, February 1999.