# R-Java: A Reflective Java Extension

**Elisa Tomioka, José de Oliveira Guimarães, Antonio Francisco do Prado**

Departamento de Computação, UFSCar, São Carlos, SP, 13565-905, Brazil

email: {elisa, jose, prado}@dc.ufscar.br

*Abstract. Java has been largely used for Internet and distributed programming. Java is object-oriented, reasonably simple, and portable. However, an important concept is missing in this language: metaobjects. A metaobject intercepts messages sent to the object to which it is attached allowing a programmer to modify the behavior of existent code with a few changes in the source code. This article presents a Java extension that supports a kind of metaobjects called shells which are simple, statically typed, and efficient. They fit nicely in the Java paradigm of simplicity and safety. Shells are type safe and demand few changes in the syntax and in the compiler. In order to implement shells, it is necessary either to use a native method or to add an instruction to the Java Virtual Machine.*

## 1 Introduction

Java [9] has been widely used as a programming language in the World Wide Web. It is reasonably simple, object-oriented, portable, and offers support for distributed applications. The last two features make Java ideal to be used in the WWW which is composed by different machines and operating systems spread throughout the planet. The language portability allows a single program to run in different platforms without changing its behavior. The distributed support allows programs in different machines on the Web to cooperate with each other.

Although Java is object-oriented and has all the flexibility of this paradigm, there is one concept missing in this language: behavioral reflection. Reflection is the ability of a program to examine its own structure (structural or introspective reflection) or to change its own computation (behavioral reflection). A language that supports introspective reflection allows a program to discover the class of an object, to examine the methods of this class, the parameter types of each method, and so on. Introspective reflection is already supported by Java through Java Core Reflection [16]. Behavioral reflection takes place when a program changes its own behavior. The program may insert (remove) instance variables and methods into classes, change the inheritance hierarchy, and modify the method look-up algorithm for a single object or for the entire program.

A metaobject is an object that intercepts messages sent to another object thus controlling its behavior. When a message is sent to an object ***Q***, the metaobject attached to it can execute its own code, redirect the message to another object, or send the message to object ***Q***. Since metaobjects do change the method look-up for a single object, they implement behavioral reflection.

Metaobjects compose a software layer called the meta-level which controls the program behavior. The meta-level does not deal directly with the program requirements. It just helps the program to reach its goals. The separation of domains between program and meta-level produces programs easier to modify and mantain. Deep changes in the program behavior can be made by small changes in the meta-level.

Metaobjects can be used to monitor classes and objects, debug a single object at run-time, check the parameters passed to object methods, make the implementation of design patterns [5] easier [11], implement fault tolerance [19], object distribution, and parallelism transparently.

When a metaobject intercepts a message sent to the object it controls, it can redirect the message to another object in another machine. That makes it easy to distribute objects through different platforms. The object that sends the message may not know that the method will be executed in another machine: the distribution is transparent.

Dynamic shells [12] are an efficient, statically typed, and simple kind of metaobjects initially designed for the Green Language [13]. Because of these features we added dynamic shells to Java creating an extension called R-Java (from Reflective Java). The philosophy of simplicity and safety of Java is preserved in this language extension.

## 2 Dynamic Shells

Dynamic shells are a simple kind of metaobjects. A shell can be attached to a normal object to intercept messages sent to the object.

Figure 1 shows a normal object ***Q***, represented by a circle, that was initially referenced by variable `s`. This figure shows also a shell ***F***, represented by a rounded rectangle, attached to object ***Q***. After the attachment, variable `s`, like any other reference to object ***Q***, will refer to shell ***F***. This figure only presents the concepts of shells: it is not intended to explain how shells are implemented.

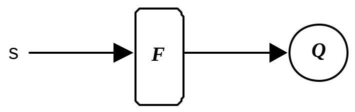

**Figura 1: Shell F attached to object Q**

The class of shell ***F*** can only define methods with the same interface as the methods of ***Q*** class. If ***F*** class defines method `m`, this method will be executed when a message `m` is sent to object ***Q***. But if ***F*** class does not define a method `m`, method `m` of ***Q*** will be executed. If object ***Q*** knows how to respond to a message `m`, so will object ***Q*** attached to shell ***F***. This means an object with an attached shell knows how to respond to the same set of messages as the object alone. Then the object type is not modified by shell attachment and no type error is introduced by shells.

Messages sent to `self` in ***Q*** methods will be intercepted by the shell. So shells are unlike wrapper classes [5] which compose a layer that just forwards the messages to the object. In wrapper classes the self reference is not maintained. In shells, it is.

Message sends to `super` inside ***Q*** methods are not intercepted. A message send to `super` is a message send to `self` in which the method to be executed at run time is found at compile time in a search that begins at the superclass. Since the method is found at compile time, no interception at run time by the shell is possible.

A shell may have instance variables to keep information about the object to which it is attached. The shell instance variables can only be manipulated by shell methods. The access to these variables will be faster if the class of the object to which the shell is attached is reflective. A reflective class is declared in R-Java by putting the class modifier `reflective` before the class name.

An example of reflective and shell classes is shown in Figure 2. Class `Window` has a method `draw` which draws a window in the screen. Note this class was declared as a reflective class.

```
reflective class Window {
  ...
  public void draw() { ... }
  }

shell class Border (Window) {
  private void drawBorder() { ... }
  public void draw() {
    /* draws a border */
    this.drawBorder();
    /* draws the window */
    super.draw();
    }
  }
```

**Figura 2: A dynamic shell class declaration**

Shell class `Border` was declared using the class modifier `shell`. After the shell class name there should appear the base class name between parentheses (`Window` in this example). So a `Border` shell can be attached to objects of class `Window` or its subclasses. The set of `Border` methods must be a subset of the set of class-`Window` methods. Let `w` be an object of class `Window` or subclass of `Window`. The command

```
Reflect.attachShell (w, new Border());
```

attaches dynamically a `Border` shell to `w`. Only `Window` object `w` is affected. Now when a message `draw` is sent to `w` the shell method is executed which draws a border by calling `drawBorder` and then calls the object method `draw` through `super`. Of course, any message sent to this object through *any* variable (not only `w`) will be intercepted by the shell.

The method `attachShell` will throw an exception if the class of object `w` does not belong to a set of classes defined at compile time. If the programmer wants to attach `Border` shells to objects of a class `X`, she must specify this at compile time. This requirement could be removed if the program created classes at run time.

A shell class may inherit from other shell class. Although there is no semantic or implementation problems related to this feature, it has not been implemented.

Shells are an efficient kind of metaobjects. A message send to an object with an attached shell is as fast as a message send to an object without a shell. This is true when the method to be executed belongs either to the shell or to the object. Performance degradation only occurs in methods that access shell instance variables. It is necessary to set a pointer in the beginning of each shell method that accesses shell instance variables. If the object class is reflective this pointer is set to an object instance variable called `sv`. Otherwise this pointer is set to an address found in a hash table look-up using the object address as key.

Shells can be used to change the behavior of objects of a class even when the source code of this class is not available: the original class need not to be modified.

**Method interceptAll**

In other languages, when a message is sent to an object with a metaobject, the metaobject method `methodCall` is invoked regardless of the message. So one can change the behavior of all object methods by defining only one method `methodCall` in the metaobject class. The shell features seen till now only allow one to modify one method at a time, thus making shells a restricted kind of metaobjects. To change the behavior of all object methods one should define each object method in the shell class.

The `interceptAll` feature allows shells to have the same functionality as metaobjects. One can declare a method

```
public Object interceptAll (Method met, Object[] args)
```

in the shell class. When a message `m` is sent to an object ***Q*** with a shell, the shell method `m` will be executed. If the shell does not have a method `m` but has a method `interceptAll`, the message parameters are packed in an array `args` passed as parameter to a call to shell method `interceptAll`. The first real parameter is an object of class `Method`[1] that describes the ***Q*** method that would be executed if there were no shell. The `interceptAll` method can call method `m` of ***Q*** using the method `invoke` of class `Method`:

```
met.invoke (this, args);
```

[1] Class from Java Core Reflection.

Using this feature, one can send a message through a network to another machine where the message can be unpacked and sent to another object. This mechanism makes it easy to implement distributed programs as made in the Open C++ language [2] [3].

In Java every class is a subclass of `Object`. This allows the elements of `args` to be of any type except the basic types like `int` and `double`. To allow `args` to store also values of basic types there are some classes whose purpose is to pack basic values. For example an object of class `Integer` stores an integer value and has methods to get and set the value. In a call "`a.m(1)`", number `1` will be wrapped in an `Integer` object before being inserted into array `args`.

Several papers describe language constructs similar to shell without `interceptAll`: the trap mechanism of KSL [14], the metaobject construct of Foote and Johnson [4], predicate classes [1], environmental acquisition [6], and contexts [22]. Shells without `interceptAll` have also been used to make it easy to implement some patterns like Decorator and Strategy [5]. In pattern Decorator, a class is used to add functionality to objects of some other class. For example, class decorator `Border` is used to add a border to objects of class `Window`. To add a border to a `Window` object `Q`, one should create a `Border` object and make it refer to `Q`. The `Border` object will forward all messages but `draw` to `Q`. Method `draw` of the `Border` object will draw a border and then call method `draw` of `Q` to draw a window. This pattern is easily implemented using shells as shown in Figure 2.

## 3 Dynamic Shell Implementation

This section describes how language Java was extended to support dynamic shells. The implementation of dynamic shells for R-Java was based on the implementation made for the Green language [12] [13].

### 3.1 Representation of Objects, Shells, and Shell Classes

In Java all variables whose types are classes are pointers to objects. And each object has a pointer to its class. Figure 3 (a) shows the internal representation of an object of class `A`. Variable `a` is a pointer to an object which has a pointer `classInfo` to an object of class `Class` representing class `A`. This `Class` object has a method table for class `A` and other information about this class. The object instance variables are put after pointer `classInfo`.

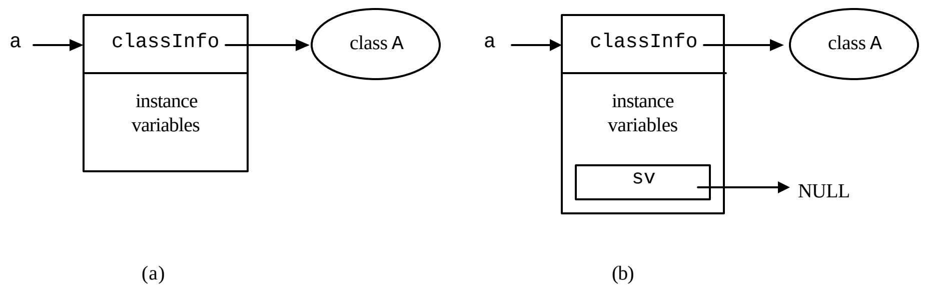

**Figura 3: Internal representation of a (a) non-reflective and (b) reflective object of class A**

Figure 3 (b) shows the representation of an object of a reflective class `A`. An object of a reflective class will be identical to an object of a normal class except by an extra instance variable called `sv`. This variable has type `Object` and points to `null` if the object is not attached to a shell. If it is attached, this variable points to an object with the shell instance variables as shown in Figure 4 (a). Variable `sv` is defined in class `ReflectiveObject` which is inherited directly or indirectly by all reflective classes. A reflective class should inherit from `ReflectiveObject` (which inherits from `Object`) or from another reflective class. If the reflective class does not explicitly inherits from other class (as class `Window` of Figure 2), the compiler makes it inherit from `ReflectiveObject`.

The programmer should tell the compiler a shell class `B` will be used to create metaobjects that will be attached to objects of a class A. With this knowledge, the compiler splits class `B` into two classes, `B_A_m` and `B_ivc`. Class `B_A_m` inherits from `A` and has all class-`B` methods. Class `B_ivc` has all class-`B` instance variables and constructors. Class `B_A_m` has no instance variables and class `B_ivc` has no methods. To attach a shell of class `B` to an object of a reflective class `A` (as that of Figure 3 (b)) is to change the object class to `B_A_m` and make the object instance variable `sv` point to an object of class `B_ivc`. The object layout after the attachement is shown in Figure 4 (a).

To attach a shell to an object is to change its class. Since `B_A_m` inherits from `A` and does not define any new method, no method signature is added to or removed from the object. Class `B_A_m` does not declare instance variables. If it did, `B_A_m` and `A` objects would have different layouts. This would prevent the changing of the object class from `A` to `B_A_m` when a `B` shell is attached to it.

The `B` instance variables are declared in `B_ivc`. The object variable `sv` points to a `B_ivc` object as in Figure 4 (a). The methods declared in `B_A_m` that use `B` (the shell class) instance variables are compiled in such a way they use these variables through pointer `sv` of the object.

Now we explain how the class `A` of an object, a run-time information, can be related to the creation of class `B_A_m` (which inherits `A`) at compile time.

At compile time the programmer should associate to each shell class `B` a set of classes called the "allowed set" of `B`. A shell of `B` can only be attached at run time to an object of a class specified in the allowed set of B. This requirement would be unnecessary if we created classes B_A_m at run time.

For each class A that belongs to the allowed set of B, the R-Java compiler:

1. creates a class called B_A_m with the B methods;
2. makes B_A_m inherit from A;
3. includes a static variable prev in B_A_m which will refer to the Class object describing class A;
4. if class A is not reflective, includes a static variable ht of class Hashtable in B_A_m;
5. creates a class B_ivc with the variables and constructors of shell class B. All its instance variables are declared public. This class has no methods;
6. inserts at the beginning of each B_A_m method that accesses shell instance variables code to assigns to an auxiliary pointer shellV the address of a class-B_ivc object with the shell instance variables. This address will be got:
   ? from the object variable sv if class A is reflective[2] or;
   ? through a hash table ht using the object address as key if class A is not reflective.

The access to a shell instance variable inside B_A_m methods is made using the auxiliary pointer shellV and not through the variable sv (when A is reflective) or using the hash table ht (when A is not reflective). This is necessary because if the shell is removed from the object by a shell method, sv will point to null (if A is reflective) or the reference to the shell memory will be deleted from the hash table ht (if A is not reflective) causing an error if the shell tries to access its instance variables through sv or ht. Even after the shell is removed from the object, the auxiliary variable shellV will continue to point to the class-B_ivc object with the shell instance variables. This object will be collected by the garbage collector as any other object.

## 3.2 R-Java Library Classes Definition

We are going to better define the library classes ReflectiveObject and Reflect of R-Java. Class ReflectiveObject has only instance variable sv:

```
class ReflectiveObject {
  Object sv;
}
```

Variable sv will refer to the shell instance variables. All reflective classes must inherit from another reflective class or from ReflectiveObject.

Class Reflect has methods attachShell and removeShell to attach and remove shells from objects. The attachShell method attaches a shell to an object as in:

[2] This variable is inherited by all reflective classes from class ReflectiveObject.

```
try {
  Reflect.attachShell (a, new B());
} catch (ShellException e) { ... }
```

Method `attachShell` will:

1. test if the class of `a` belongs to the "allowed set" of shell class `B`. If not, the method will throw an exception `ShellException`. Note that:
   ? the "allowed set" of a class is defined by the programmer at compile time;
   ? the class of `a` will only be known at run time;
   ? this test would not be necessary if class `B_A_m` were dynamically created;
2. if the class of `a` is reflective, `attachShell` will make pointer `sv` of the object point to the shell instance variables which are stored in an object of class `B_ivc` created and passed as parameter to `attachShell` — see Figure 4 (a). If the class of `a` is not reflective, the pair (address of object `a`, address of object created by "`new B_ivc()`") is inserted in the hash table `ht` in which `ht` is a static variable of class `B_A_m`. The address of object `a` is used as key. This hash table will be used inside the shell methods to get the address of the shell instance variables (which are in a `B_ivc` object);
3. make the object pointer `classInfo` point to class `B_A_m` created at compile time. So the object will use the methods of class `B_A_m`. See Figure 4 (a).

In the example above, there is a call to the constructor of shell class `B_ivc` in

```
Reflect.attachShell (a, new B_ivc());
```

This call is made before the attachment of object `a` to the shell. Then the `B_ivc` constructor cannot:

? access the variables of the object to which the shell is attached or call its methods or constructors;
? call the methods of the shell class.

These restrictions are not serious since in most cases the shell constructors only initiate their own variables and, in case they need data from the object they are attached to, the data can be passed to them by parameters of a constructor.

To remove the last shell attached to the object, the programmer should call the method `removeShell` of class `Reflect` as follows:

```
try {
    Reflect.removeShell (a);
  } catch (ShellException e) { ... }
```

If there is no shell attached to object `a`, method `removeShell` will throw an exception `ShellException`. Otherwise, the method `removeShell` will

1. make the object pointer `classInfo` point to class `A` which is the object class before it was attached to the shell. Note that this class is referred by variable `prev` of class `B_A_m`. If class `B_A_m` does not inherit from any other shell class, it will inherit directly from `A` and `prev` will just refer to `B_A_m` superclass. However, if `B_A_m` inherits from a shell class, `prev` will refer to the first normal class found in a search beginning in `B_A_m` and continuing up in the superclass chain;
2. if class `A` is reflective, assign `null` to object pointer `sv`. If `A` is not reflective, the address of the class-`B_ivc` object with the shell instance variables is removed from the hash table `ht` of class `B_A_m`. In both cases, the class-`B_ivc` object with the shell instance variables will only be deallocated by the garbage collector.

Figure 4 (a) shows a class-`B_ivc` shell attached to an object of reflective class `A` and Figure 4 (b) shows the configuration after the shell is removed from the object. There may be one or more local variables `shellV` referring to the shell instance variables. Each of these variables belongs to a shell method that has not finished its execution — it is in the stack of called methods.

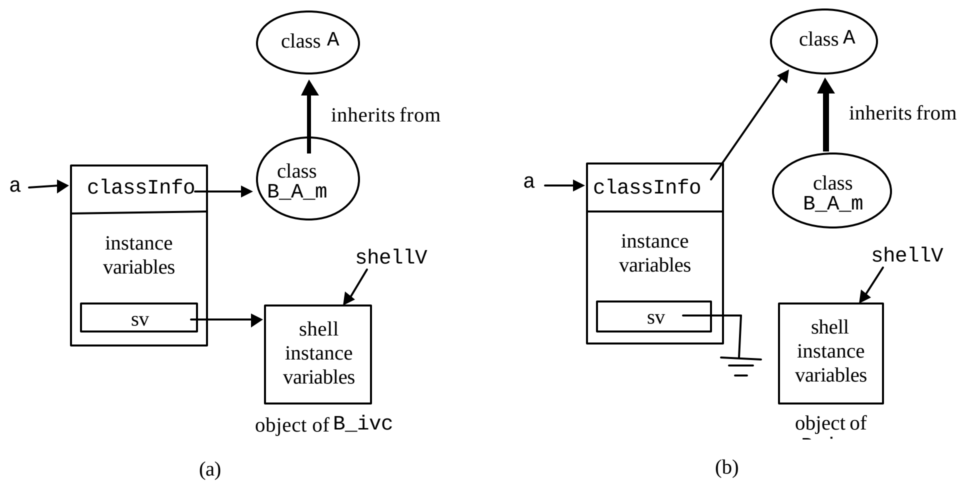

**Figura 4: Reflective object with a shell (a) attached and (b) removed from it**

## 3.3 Implementation of *interceptAll*

Suppose a shell of class `B` is attached to an object of class `A` and a message `m` is sent to the object. If `B` defines a method `m`, this method will be executed. If `B` does not define a method `m` but defines a method `interceptAll`, then this method is executed and receives as parameters the method `m` and its arguments both packed in objects.

To make this possible, the compiler creates and inserts a method `m` in class `B_A_m` for each method `m` defined in `A` but not in shell class `B`. This method has the same signature as method `m` of `A`. That means class `B_A_m` has a method for each method defined in `A`. Either the method was created by the compiler or defined by the programmer in shell class `B`.

Method `m` of `B_A_m` is implemented as shown in the example of Figure 5.

Variable `method_A_m` was initialized with an object of class `Method` that describes class-`A` method `m`. This object is got through method `getMethod` of class `Class` passing as parameters the name of the method (`m`) and the types of the formal parameters of the method (`int`). This initialization is made in the static initializer of class `B_A_m`:

```
static {
   ...
   method_A_m = A.class.getMethod ("m", parameterTypes);
   ...
}
```

This code is executed when the class is loaded. The parameters received by method `m` of `B_A_m` are packed into objects of class `Object` and inserted into array `args`. After this method `interceptAll` of `B_A_m` is called passing the variables `method_A_m` and `args` as parameters.

```
public void m (int n) {

  // create an object of class Integer that packs
  // the parameter n and inserts it into the array
  Object []args = { new Integer(n) };

  // call method interceptAll
  this.interceptAll (method_A_m, args);
  }
```

**Figura 5: Example of implementation of method m of class B_A_m**

So the message `m` sent to the object of class `A` will cause the execution of method `m` of `B_A_m` which calls the method `interceptAll` defined by the programmer in the shell class `B`. Method `m` of `B_A_m` returns the same value returned by method `interceptAll`. In general there will be a *cast* to the return value type of the method. For example, if `m` returned `Window`, the last command of method `m` of Figure 5 would be

```
return (Window ) this.interceptAll( method_A_m, args );
```

### 3.4 The Java Virtual Machine

The Java interpreter was changed to recognize a new instruction called `chclass`. This instruction is necessary to change the class of an object at run time. The `chclass` instruction is only used inside the `attachShell` and `removeShell` methods of class `Reflect`. Figure 6 shows the specification of this instruction.

`chclass` is safe because:

1. the new object class should be subclass of the old class or vice-versa;
2. the set of public method signatures of both classes should be equal;
3. the subclass, either the old or new object class, should not declare any instance variable;

If these requirements are not fullfilled, `chclass` throws exception `ShellException`. If they are, the layout of the objects of the new and old classes are equal. Then objects of both classes are equivalent and can replace one another.

The Java interpreter scans a program bytecodes before executing them to discover if there is any security violation or type error. This approach cannot be used to check the correctness of `chclass` instructions. This correctness depends on the class of `objectref` (see Figure 6) which will only be known at run time. Therefore our R-Java implementation did not demand any changes in the bytecode verifier.

The just-in-time compiler was modified to recognize and work with the `chclass` instruction.

The Java Virtual Machine need not to be modified in order to change the class of an object at run time. This can be made by a native method. We have made this implementation by creating a native method `chclass`. To use this method is apparently better than to change the Java Virtual Machine (JVM), an essentially non-portable modification. However, a close inspection reveals that the use of a native method `chclass` is also non-portable. Each JVM may define its own object layout. Therefore, the native method `chclass` is made to work with a particular JVM since it depends on the object layout to change the object class. A native method `chclass` for a JVM probably will not work when used with another JVM implementation.

**Instruction:** chclass

**Operation:** Changes the class of an object

**Stack:**

..., *objectref*, *classref* ? ...

**Description:**

The value of the top of the stack (*classref*) must be a reference to an object of class Class and the value immediately under it (*objectref*) must be a reference to an object.

The two values are poped off the stack and the class of the object referred by *objectref* is changed to the class represented by *classref*. The class of *objectref* should be subclass of the class represented by *classref* or vice-versa. The set of public method signatures of both classes should be equal and the subclass should not declare any instance variable. Otherwise exception **ShellException** is thrown.

**Figura 6: The new chclass instruction for the Java Virtual Machine**

Then we are faced with two options:

1. to change the JVM by adding instruction `chclass`;
2. to use a native method `chclass`;

Option 1 makes R-Java programs non-portable. Option 2 implies the safety of a program is not guaranteed: it is not possible to check security violations or type errors in native methods. Then whenever one uses a native method there is no guarantee the program is safe. There may even be, after the call to the native method, the sending of a message to an object that does not have the corresponding method.

## 4 Examples

Shells can trace messages sent to an object as shown in Figure 7. A shell of class `TracePerson` can be attached to objects of class `Person` to print a message in the screen every time the object receives a message `set`. Instance variable `num` of `TracePerson` is initiated in the constructor of the shell class and is incremented each time the object receives a message `set`. Typically `num` would be initiated with `0`.

Figure 8 shows the classes in Java created by the compiler using the classes of Figure 7. The class `TracePerson_ivc` of Figure 8 has the variables and constructors of the shell class `TracePerson` of Figure 7 and the class `TracePerson_Person_m` has its methods.

If class `Person` of Figure 7 were not declared as reflective, the classes in Java created by the compiler would be those shown in Figure 9.

Reflective classes can inherit from other reflective classes. Class `Person` could inherit from a class `Creature`. In this case the compiler would make `Creature` inherit from `ReflectiveObject` and `Person` inherit from `Creature`.

```
reflective class Person {
  String name;
  int    age;
  public void set (String name, int age) {
    this.name = name;
    this.age  = age;
  }
}

shell class TracePerson (Person) {
  private int num;
  public TracePerson (int x) {
    num = x;
  }
  public void set (String name, int age) {
    num++;
    System.out.println ("message No. ");
    System.out.println (num);
    System.out.println (" sent to " + super.name);
    super.set (name, age);
  }
}
```

**Figura 7: Tracing methods of an object**

## 5 Related Work

The Java language supports introspective reflection. Information about objects and classes can be accessed at run time. For example method `getClass()` of class `Object` (inherited by every class) returns an object describing the object class. This object belongs to class `Class` and stores the class name and information about its superclass, methods, and so on. However objects of class `Class` only *describe* classes — they cannot change them. That means the objects of `Class` do not implement behavioral reflection.

Recently some reflective architectures for behavioral reflection have been proposed as extensions to the Java language. These are presented next.

### 5.1 Reflective Java

This protocol [26][27] makes message sends reflective. A message sent to an object of a reflective class is redirected to a metaobject. That is made without any change in the Java language, its compiler, or in the Virtual Machine. A pre-processor is used like in Open C++

[2] [3] to create a reflective subclass from the normal class. This subclass has a pointer to a metaobject and forwards the messages to it.

## 5.2 MetaXa

Golm proposed an introspective and behavioral reflection protocol called MetaXa [8], formely known as MetaJava [7]. The introspective reflection is implemented by a set of classes that describe the structure of the program (classes, methods, variables) similar to Java Core Reflection [16]. Behavioral reflection in MetaXa is implemented as a set of classes and demanded changes in the Java Virtual Machine.

```
class Person extends ReflectiveObject {
  String name;
  int    age;
  public void set (String name, int age) {
    this.name = name;
    this.age  = age;
  }
}

class TracePerson_ivc {
  public int num;
  public TracePerson_ivc (int x) {
    num = x;
  }
}

class TracePerson_Person_m extends Person {
  public static Class prev = Person.class;

  public void set (String name, int age) {
    TracePerson_ivc shellV;
    shellV = (TracePerson_ivc )sv;
    shellV.num ++;
    System.out.println ("message No. ");
    System.out.println (shellV.num);
    System.out.println ( " sent to " +
                         super.name);
    super.set (name, age);
  }
}
```

**Figura 8: Java classes created from the example of Figure 7**

MetaXa allows one to:

- ? intercept messages that are sent and received;
- ? control the access to instance variables and object creation;

? control the locking of objects and the loading of classes to memory.

In this language it is possible to associate a metaobject to an object, to a reference (variable), or to a class. In this last case the metaobject intercepts all the message sends to all the objects of the class.

```
class Person {
  String name;
  int    age;
  public void set (String name, int age) {
    this.name = name;
    this.age  = age;
  }
}

class TracePerson_ivc {
  public int num;
  public TracePerson_ivc (int x) {
    num = x;
  }
}

class TracePerson_Person_m extends Person {
  public static Class prev = Person.class;
  public static Hashtable ht = new Hashtable();

  public void set (String name, int age) {
    TracePerson_ivc shellV;
    shellV = (TracePerson_ivc ) ht.get (this);

    shellV.num ++;
    System.out.println ("message No. ");
    System.out.println (shellV.num);
    System.out.println (" sent to " + super.name);
    super.set (name, age);
  }
}
```

**Figura 9: Example in Java for non-reflective classes**

## 5.3 Guaraná

Guaraná [20] [21] is a reflective architecture that allows the program to intercept message sends and accesses (read and write) to instance variables. Each of these operations is transformed into an object delivered to the metaobject that controls the object. There is an elaborate system to compose metaobjects which is the main feature of Guaraná. A composer metaobject keeps a list of other metaobjects and delegates messages to them.

To implement Guaraná it was not necessary to change Java language although some instructions of the Virtual Machine were redefined. The instruction that call a method and the ones which access instance and static variables (read and write) were changed to test for the presence of a metaobject.

## 5.4 Dalang

This Java extension demanded no changes in the Java Virtual Machine or access to the program source code [24][25]. It creates wrappers for classes by handling the bytecodes of the classes. To make a class reflective at compile or run time, Dalang builds a wrapper class with the same interface as the original class. This wrapper class is much like the class created when one uses method `interceptAll` in a shell class. Each method will be similar to the method of Figure 5. The wrapper class inherits from a metaobject class, not from the original class.

All objects of a reflective class have an associate metaobject, which cannot be removed or changed.

Language OpenJava [23] supports metaobjects but it will not be discussed in this paper. OpenJava metaobjects exist at compile time and those of R-Java exist at run time. They are not equivalent and it would not be reasonable to compare them.

## 6 Discussion

The existing reflective architectures for Java are complex when compared to shells. They are also inefficient if just a subset of the object methods should be intercepted. To understand these points, it is necessary to study how metaobjects work in a typical architecture.

There is a class `MetaObject` that must be inherited by any other metaobject class. The method

```
public Object interceptMethodCall (MethodCall aCall)
```

of `MetaObject` must be redefined in subclasses. When an object attached to a metaobject receives message `m` as in:

```
x.m (1, b);
```

the message will be packed into an object of `MethodCall` used as an argument to a call to method `interceptMethodCall` of the metaobject. The `MethodCall` object contains all the message data, which includes the name and parameters.

Method `interceptMethodCall` can call the method of object `x` that would be called if there were no metaobject. This is done by a special method of the metaobject or by a method of object `aCall`.

This approach requires the understanding of a metaobject protocol used for object-metaobject interaction. This results in a model more complex than shells. Besides that, every message send is packed into an object delegated to the metaobject. This operation is very inefficient since it requires the creation of many objects and their manipulation.

This metaobject model is a simplified version of metaXa [7] [8] , Reflective Java [26] [27], Guaraná [20] [21], and Dalang [24]. In fact, there are some specific points of each of these Java extensions that need to be considered.

MetaXa [7] [8] has a `MetaObject` class from which every metaobject class must inherit. This class defines a lot of methods among which:

? `attachObject` to attach a metaobject to an object;
? `continueExecutionObject`, which is equivalent to a call to `super` in a shell class — calls the original method of the object;
? `doExecuteObject`, which is equivalent to a message send to `this` inside a shell class.

In Reflective Java metaobjects [26] [27] can only be attached to objects of classes declared as reflective. This prevents a metaobject to be attached to an object of a non-reflective class. All objects of a reflective class will be attached to metaobjects. This means poor performance since in most of the cases just part of the class objects will need metaobjects. One cannot unattach a metaobject from an object. At most we can replace it.

The limitations of Reflective Java are due to a design choice: the designers did not want to change the compiler or the Java virtual machine.

Guaraná [20][21] supports a powerful system of metaobject composition which, in our opinion is rather complex.

Dalang designers have chosen not to change the Java Virtual Machine bringing some limitations to this Java extension. Some of these are solved just by changing the current implementation [24] leaving only two non-wanted characteristics, in our opinion:

1. all objects of a reflective class are attached to metaobjects of the same metaobject class;
2. one cannot change or remove the metaobject of an object.

Both Dalang and Reflective Java did not demand changes in the JVM making them work with existing systems to a large extend. Even with some limitations, these reflective Java extensions may be all someone needs to implement her program.

As said before, the use of `interceptAll` in a shell class brings the good and the bad of metaobjects to shells. However, the declaration and use of `interceptAll` is much simpler than the use of a `MetaObject` class. And the programmer need not to learn a set of classes that compose the metaobject protocol. She should only know about method `invoke` of class `Method`. The syntax and semantics of shells are very close to those of normal classes, making the concept easy to understand. The problem with shells is that they require the adding of a new instruction `chclass` to the Java Virtual Machine thus making reflective program with shells non-portable. However, this problem seems to be inherent to metaobject implementation: to intercept the methods of a single object one needs

? to change its class through an instruction like `chclass` or;

? to intercept all message sends or;

? to test at the beginning of all methods of a class for the presence of a metaobject. This would slow down all objects of the class, even those not attached to metaobjects. The rule "Don't use, don't pay" is broken. Besides that, the compiler needs to know if a class is reflective[?] when compiling it or the class should be loaded at run time and its bytecodes changed by a special class loader, as in Dalang.

The two first solutions, which we believe are the viable ones, require changes in the Java Virtual Machine. Note the second solution slows down all the program.

Below is a summary of some restrictions and particularities of shells.

? A shell class may have a superclass.

? The superclass of a reflective class must be reflective.

? In order to attach a shell to an object of a class, the class need not to be reflective. However, accesses to shell instance variables will be faster if it is.

? The allowed set of a shell class should be defined at compile time. In order to attach a shell of class `S` to an object of class `A`, the programmer should add `A` to the allowed set of `S`.

**Performance**

To study the performance of R-Java metaobjects we will use a class A with methods

```
void m() { }
int m1( int x ) { return 0; }
A m( A a ) { return this; }
```

and a shell with a method `interceptAll`. The `interceptAll` method just calls, with `invoke`, the object method that would be called if there were no shell:

```
met.invoke(this, args);
```

That is, the shell attachment does not modify the object behavior. After attaching this shell to an `A` object there is a decrease in performance which is shown in Figure 11. The figures are the ratios "shelltime/normalTime" in which "shellTime"is the time it takes to execute the method if the shell is attached to the `A` object. "normalTime" is the time taken by a message send to the `A` object without a shell.

The first column of the table shows the figures when an optimized version of `interceptAll` is used. the second column refers to an implementation without any optimization. In the optimized version, it takes 3.55 times as much to call `m` when the `A` object has a shell than to call `m` when there is no shell.

[?] That is, objects of the class may be attached to metaobjects.

Our compiler does the following optimizations:

✍ it replaces a call "`met.invoke(this, args)`" by a switch with one case label for each method that can be called. Therefore there is no need for a method search at run time;

✍ it does not allocate memory for an array `args` with length zero.

In MetaXa and Guaraná, when a message is sent to an object with an attached metaobject, the message is packed in an object describing the message send. The creation of this object is not made in R-Java making it faster than those languages.

To compare the performances, let us use the number 7.86 of the cell in line "a.m()" and column "NON-OPTIMIZ."of the table of Figure 10. The corresponding figures in MetaXa and Guaraná are 215 and 150, at least.

The figure for Reflective Java is not available [28] and for Dalang is about 7 [24]. Dalang uses a method wrapping mechanism that has some similarities to R-Java.

It is worth noting the figures presented in Figure 11 are greater but not radically different from the ones for the Green language [12]. For example, the date for "`a.m()`" in Green are 3.3 (optimized) and 4.6 (non-optimized) which are not far from 3.55 and 7.86 in R-Java.

The similar performance was expected since the R-Java implementation was based on the Green one. The data of Figure 10 were got using Sun stations and the Just-In-Time JDK compiler for the R-Java code.

It was necessary to use a little trick in order to use the JDK compiler since this compiler does not recognize R-Java code. We prepared a class like `TracePerson_Person_m` of Figure 8 for `A` and the shell class. Then we used an object of this class to measure the times corresponding to an `A` object attached to a shell.

| | *OPTIMIZ.* | *NON-OPTIMIZ* |
|---|---|---|
| *`a.m()`* | *3.55* | *7.86* |
| *`a.m1(1)`* | *12.99* | *19.20* |
| *`a.m2(a)`* | *6.97* | *10.11* |

**Figura 10: A table with interceptAll performance**

## 7 Conclusion

R-Java was based in the language Green [12] [13] in which shells were first introduced. However, R-Java shells are not just a copy of Green shells. Although the concept is very similar, the implementation is completely different since Green code is translated to C and Java is translated to bytecodes. It was necessary to add a new instruction to the Java virtual machine, create a new class `ReflectiveObject`, and change all low level aspects of Green implementation to make it work with bytecodes.

In order to attach a metaobject of a shell class to an object of a class, the programmer must tell the compiler this class belongs to the allowed set of the shell class. This restriction may be lifted in a future implementation of R-Java by dynamically creating classes as made in Dalang.

The implementation described in this paper is simple because it generates some Java classes instead of generating bytecodes directly. The implementation of shell classes and reflective classes did not demand changes in the Java Virtual Machine. The only modification needed in the Virtual Machine was the implementation of the `chclass` instruction. This instruction is only used inside the library class `Reflect`. To implement class `Reflect` we modified the Java assembler Jasmin [15] and the Java Virtual Machine [18] interpreter Kaffe [17] making them recognize the new instruction `chclass`. The Java compiler changed to recognize the shell classes was Guavac [10].

R-Java shells are faster and simpler than normal metaobjects yet powerful. R-Java shells are much faster than metaobjects because they do not need to create an object representing the intercepted message send. If only a few methods are to be intercepted, the programmer can define them in the shell class as in Figure 2. There will be no overhead at run time if the base class is reflective. If it is not, there will be only a hash table look-up in the beginning of each shell method that accesses shell instance variables. This mechanism of message intercepting, which does not reify message send, has been added to several languages [1] [4] [6] [14] [22] and is very useful in implementing design patterns.

R-Java requires the addition of one instruction to the Java Virtual Machine or the use of a native method which depends on a specific implementation of the JVM. These requirements are undesirable but it seems impossible to lift them. There should be some mechanism to change an object class at run time in order to attach or remove a metaobject. It should be noted the new instruction added to the JVM checks its parameters to prevent a run-time type error.

Finally, R-Java is simple. Shells are very similar to subclasses making them very easy to learn. They do not really need a complex metaobject protocol with new commands and definitions.

**Acknowledgements.** This work was financed by FAPESP under process number 97/12209-7.

## 8 References

1. Chambers, C. Predicate Classes. *European Conference on Object-Oriented Programming - ECOOP'93*, LNCS 707, 1993.
2. Chiba, S. Open C++ Programmer's Guide. Technical Report 93-3, Department of Information Science, University of Tokyo, Tokyo, Japan, 1993.
3. Chiba, S. and Masuda, T. Designing an Extensible Distributed Language with a Meta-Level Architecture. *Proceeding of ECOOP'93. Lecture Notes in Computer Science* No. 707, 1993.
4. Foote, B. and Johnson, R. Reflective Facilities in Smalltalk-80. *SIGPLAN Notices*, vol. 24, no. 10, October 1989. OOPSLA 89.
5. Gamma, E; Helm, R; Johnson, R and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series, Addison-Wesley, Reading, MA, 1994.
6. Gil, J. and Lorenz, D. Environmnetal Acquisition: a New Inheritance-Like Abstraction Mechanism. *SIGPLAN Notices*, vol. 31, no. 10, October 1996. OOPSLA 96.
7. Golm, M. Design and Implementation of a Meta Architecture for Java. Diplomarbeit im Fach Informatik, Friedrich-Alexander Universität, Erlangen-Nürnberg, Jan. 1997.
8. Golm, M. MetaXa and the Future of Reflection. *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.
9. Gosling J; Joy, B; Steele, Guy. *The Java Language Specification*. Sun Microsystems Computer Corporation, Version 1.0, August 1996.
10. *Guavac: A free compiler for the Java language*, Version 1.0. Available at: ftp://ftp.de.uu.net/pub/programming/languages/java/guavac
11. Guimarães, J.O.; Tomioka E. and Prado, A.F. Usando Metaobjetos para Implementar Padrões. *II Simpósio Brasileiro de Linguagens de Programação*, Campinas, SP, Setembro 1997.
12. Guimarães, J.O. Reflection for Statically Typed Languages. *European Conference on Object-Oriented Programming - ECOOP'98*, LNCS 1445, Eric Jul (Ed.). Also available at http://www.dc.ufscar.br/~jose/green/shell.zip, 1998.
13. Guimarães, J.O. The Green Language: Definition and Comments. Available at http://www.dc.ufscar.br/~jose/green/green.htm, 1998.
14. Ibrahim, M.; Bejcek, W. and Cummins, F. Instance Specialization without Delegation. *Journal of Object-Oriented Programming*, June 1991.
15. *Jasmin: A Java Assembler Interface*, Version 1.0. Available at: http://found.cs.nyu.edu/meyer/jasmin/
16. *JAVA Core Reflection: API and Specification*. JavaSoft, Mountain View, CA, USA, October 1996.
17. *Kaffe: A free virtual machine to run Java code*, Version 1.0.b2. Available at: http://www.kaffe.org/

18. Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification*, Java Series, Addison-Wesley, September 1996.

19. Lisboa, M.L.; Rubira, C.M.F. Técnicas de Programação para Tolerância a Falhas. *I Simpósio Brasileiro de Linguagens de Programação*, Belo Horizonte, MG, Setembro 1996.

20. Oliva, A; Buzato, L.E.; Garcia, I.C.; The Reflexive Architecture of Guaraná.. Available at: http://www.dcc.unicamp.br/~oliva.

21. Oliva, A. and Buzato, L.E. Composition of Meta-Objects in Guaraná. *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java.*

22. Seiter, L.; Palsberg, J and Lieberherr, K. Evolution of Object Behavior using Context Relations. *IEEE Transactions on Software Engineering*, vol. 24, no. 1, January 1998.

23. Tatsubori, M. *An Extension Mechanism for the Java Language*. Master Thesis, University of Tsukuba, 1999.

24. Welch, I. and Stroud, R. Dalang – A Reflective Java Extension. *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

25. Welch, I. Personal Communication, 1999.

26. Wu, Z. and Schwiderski, S. Reflective Java: Making Java even More Flexible. FTP: Architecture Projects Management Limited (apm@ansa.co.uk), Cambridge, UK, 1997.

27. Wu, Z. Reflective Java and a Reflective Component-Based Transaction Architecture. *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

28. Wu, Z. Personal Communication, 1999.