Guaraná: A Tutorial

Alexandre Oliva
Luiz Eduardo Buzato

Relatório Técnico IC–98-31

Setembro de 1998

Guaraná: A Tutorial

Alexandre Oliva
oliva@dcc.unicamp.br

Luiz Eduardo Buzato
buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação
Universidade Estadual de Campinas

September 1998

Abstract

This text is a tutorial for people interested in using our Java^{TM}-based implementation of **Guaraná**, a reflective architecture that aims at flexibility, security and reuse of meta-level code. It shows what kind of operations can be intercepted with **Guaraná** and how meta-objects can monitor and modify base-level behavior. It also introduces composition of meta-objects, and discusses dynamic reconfiguration and management of meta-configurations. Several tricks and internal details of the implementation are exposed, through the use of numerous examples and detailed explanations.

1 Introduction

Guaraná [4] is a reflective architecture whose meta-object protocol allows reuse of meta-level code through composition of meta-objects, in a simple, flexible and secure manner. It has been implemented by modifying an open-source Java[1] Virtual Machine [3], but **Guaraná** does not require any change to the Java^{TM} Programming Language [1, 2].

This text assumes some familiarity with the Java^{TM} Programming Language and the reflective architecture of **Guaraná**, as it contains several examples coded in Java that demonstrate how to use the Application Programming Interface (API) of **Guaraná**.

In Section 2, we describe **what kind** of base-level interactions can be intercepted by **Guaraná**. Section 3 considers the implementation of meta-objects, showing **how** to enable and use these intercepted interactions. In Section 4, we discuss some details of the implementation and present some advanced programming tips. Finally, in Section 5, we summarize the topics presented.

2 Basics

Just like several other reflective architectures, **Guaraná**'s reflective capabilities are based on meta-objects intercepting interactions between objects. However, unlike other architectures,

[1]Java is a trademark of Sun Microsystems, Inc.

we have not modified the original programming language. Thus, there is no compile-time association of classes with meta-classes: the meta-level behavior of an object is orthogonal to its class. Meta-objects can be dynamically associated with objects.

Instead of introducing the reflective capabilities of **Guaraná** through the examination of a complex application, we have decided to use a very simple meta-object, called MetaLogger, to explain the use of the reflective mechanisms implemented by **Guaraná**. MetaLogger writes onto the console a descriptive log every time it is activated, due to interception or to other kinds of meta-level interaction. The format of these logs is going to be explained as they appear along the tutorial.

We provide several example programs, that are included in the latest releases of **Guaraná**. Line-numbered listings of the source files and outputs of their executions are provided. In the text, we are going to refer to source lines with numbers between curly braces, such as {1} or {3–5}, and to output lines with numbers between angle brackets, such as <4> or <7–10>.

2.1 Starting up

Let us take a look at a very basic example, that demonstrates how to associate an object with a meta-object. Method main of Program 1, that is invoked when the program is started as a Java application, creates an instance of class ConfBasic {7} and a MetaObject of class MetaLogger {8}. Then, it requests the kernel of **Guaraná**, represented by the class Guarana in package `BR.unicamp.Guarana`, to replace the `null` meta-object with the newly created one, in the meta-configuration of object o {9}. Note, in the output of this program, Output 1, that the reconfiguration has succeeded, since the MetaLogger was initialized <1>: method initialize is invoked just before a meta-object is associated with a base-level object.

The ugly string printed after the colon is the Java-standard String representation of the base-level object the meta-object was just associated with. The name of the class of the object is separated with an "`@`" sign from the memory address of the object, printed in hexadecimal notation. We are going to refer to this String with the term *object-id*.

After the reconfiguration, method meth is invoked {10}. This invocation is intercepted by the kernel of **Guaraná** and presented to the MetaLogger, that prints the object-id, the method-id of the method to be invoked and the argument list <2>. A *method-id* is defined as the name of the class where the method is defined, followed by a dot and the method name. The name of the class is included to avoid ambiguities that might arise if a superclass defined a method with the same name. The type of each argument is also printed, to remove potential ambiguities due to overloaded methods.

Since this method is `synchronized` {3}, just before it starts running, a <monitor enter> operation is intercepted <3> and performed. A MetaLogger is programmed to request the result of any operation it receives, and, even though monitor-related operations produce no useful result, a `null` result is presented anyway <4>, as a notice that the monitor operation was successful.

Since method meth adds the contents of field i with the value of the argument j {4}, it must read the value of field i. The MetaLogger receives this operation too, and prints the object-id and the field-id <5>. Analogously to the method-id, a field-id is defined as the

Program 1 `ConfBasic.java`

```
 1  public class ConfBasic {
 2      int i = 3;
 3      synchronized int meth(int j) {
 4          return i + j;
 5      }
 6      public static void main(String[ ] argv) {
 7          ConfBasic o = new ConfBasic();
 8          BR.unicamp.Guarana.MetaObject mo = new MetaLogger();
 9          BR.unicamp.Guarana.Guarana.reconfigure(o, null, mo);
10          o.i = o.meth(1);
11      }
12  }
```

```
 1  Initialize: ConfBasic@10de80
 2  Operation: ConfBasic@10de80.ConfBasic.meth(int 1)
 3  Operation: ConfBasic@10de80.<monitor enter>
 4  Result: return null
 5  Operation: ConfBasic@10de80.ConfBasic.i
 6  Result: return 3
 7  Operation: ConfBasic@10de80.<monitor exit>
 8  Result: return null
 9  Result: return 4
10  Operation: ConfBasic@10de80.ConfBasic.i=4
11  Result: return null
```

Output 1: `guarana ConfBasic`

name of the class where the field is declared, followed by a dot and the field name. The MetaLogger also prints the result of the operation <6>: the value with which field i was initialized {2}.

When the execution of method meth terminates, the lock associated with object o is implicitly released, so a <monitor exit> operation is performed <7>, its result (`null`) is printed <8>, and so is the result of the execution of method meth <9>.

Finally, this result of the method is assigned to field i of object o {10}. The MetaLogger represents the non-`static` field assignment operation as the object-id, the field-id, an assignment operator and the assigned value <10>. Since assignments produce no useful result, the result of any such operation is `null` <11>.

2.2 Intercepting array operations

In addition to method invocations, field accesses and monitor operations, **Guaraná** can intercept operations on arrays, such as `length`, element read and element write. Program 2 presents a simple example of such interceptions; the example is accompanied by Output 2.

Program 2 `ArrayBasic.java`

```
 1  public class ArrayBasic {
 2      public static void main(String[ ] argv) {
 3          Object[ ] array = new Object[3];
 4          BR.unicamp.Guarana.MetaObject mo = new MetaLogger();
 5          BR.unicamp.Guarana.Guarana.reconfigure(array, null, mo);
 6          array[0] = new Integer(array.length);
 7          array[1] = array[0].toString();
 8          array[2] = array.getClass().getClass();
 9      }
10  }
```

```
 1  Initialize: [Ljava.lang.Object;@115ec8
 2  Operation: [Ljava.lang.Object;@115ec8.length
 3  Result: return 3
 4  Operation: [Ljava.lang.Object;@115ec8[0]=java.lang.Integer@1b9b68
 5  Result: return null
 6  Operation: [Ljava.lang.Object;@115ec8[0]
 7  Result: return java.lang.Integer@1b9b68
 8  Operation: [Ljava.lang.Object;@115ec8[1]=java.lang.String@1ec450
 9  Result: return null
10  Operation: [Ljava.lang.Object;@115ec8.java.lang.Object.getClass()
11  Result: return java.lang.Class@1ae258
12  Operation: [Ljava.lang.Object;@115ec8[2]=java.lang.Class@31a58
13  Result: return null
```

Output 2: `guarana ArrayBasic`

First, method main creates an array of 3 references to Objects {3}, then a MetaLogger {4}. Since any Java array is also an Object, it can be given a meta-configuration. The program instructs the kernel of **Guaraná** to associate the MetaLogger with the array of Objects {5}. The association is successful <1>. Note that the Java-standard String representation of an array of a class type is composed by a left square bracket, a capital "L", the name of the class and a semicolon.

Next, the program obtains the `length` of the array, to compute the first argument to be passed to the constructor of the `new` Integer {6}; this operation <2> and its result <3> are intercepted (by the kernel of **Guaraná**) and printed (by the MetaLogger). The Integer object is assigned to element 0 of the array {6}. This assignment is intercepted and printed <4>, as is its `null` result <5>. The representation of an array element-id is intuitive, albeit visually unpleasant: the array object-id is followed by the index of the array element between square brackets. For an array assignment operation, the MetaLogger will print, after the element-id, an assignment operator and the assigned value; in this case, an object-id.

Afterwards, method main reads the value just stored in array[0] <6–7>, invokes method toString of the returned Integer {7} (not intercepted, because the Integer was not associated with a meta-object), and assigns the returned String to element 1 of the array <8–9>.

Finally, the program invokes method getClass of the array object <10–11>, then invokes method getClass of the returned Class object {8}. Note that the second invocation of getClass is not intercepted, because the Class object that represents the class array of Objects has a `null` meta-configuration. The result of this second invocation is stored in element 2 of the array <12–13>. Observe that the Class reference returned as the result of the first invocation of getClass <11> is different from the one stored in array[2] <12>; while the former is the Object that reprents the class Object[ ], the latter is the Object that reprents the class Class itself.

2.3 Intercepting class operations

Class (`static`) operations will be intercepted by the meta-configuration of the Class object that represents the class, as in Program 3. First, the program instantiates ClassBasic {5}, and this instance will remain with a `null` meta-configuration. Then, it creates a MetaLogger {6} and obtains a reference to the Class object that represents the class ClassBasic {7}, using the class pseudo-field notation introduced in Java 1.1. In line {8}, it installs the MetaLogger as the primary meta-object of class ClassBasic, i.e., of the Class object that represents it. Line <1> of Output 3 shows the association was successful.

Program 3 `ClassBasic.java`

```
 1  public class ClassBasic {
 2      static void doNothing() {}
 3      static synchronized void doNothing(boolean b) {}
 4      public static void main(String[ ] argv) {
 5          ClassBasic o = new ClassBasic();
 6          MetaLogger mo = new MetaLogger();
 7          Class c = ClassBasic.class;
 8          BR.unicamp.Guarana.Guarana.reconfigure(c, null, mo);
 9          c.hashCode();
10          o.hashCode();
11          doNothing();
12          o.doNothing(true);       // calls static method
13      }
14  }
```

Invocations of methods of the Class object {9} are intercepted and presented to the MetaLogger <2–3>. Note, however, that the meta-configuration of the class does not affect the interception of operations on its instances: their meta-configurations are unrelated. Therefore, a method invocation of an instance of a reflective class (that is, a class whose meta-configuration is not empty) would only be intercepted if the instance itself were reflective. In the example, Object o is not reflective, so the second invocation of hashCode is not intercepted {10}.

Invocations of `static` methods, operations on `static` fields, and synchronization operations on a class (for instance, invoking a `static synchronized` method) are also presented

```
 1  Initialize: java.lang.Class@dda58
 2  Operation: java.lang.Class@dda58.java.lang.Object.hashCode()
 3  Result: return 907864
 4  Operation: ClassBasic.doNothing()
 5  Result: return null
 6  Operation: ClassBasic.doNothing(boolean true)
 7  Operation: java.lang.Class@dda58.<monitor enter>
 8  Result: return null
 9  Operation: java.lang.Class@dda58.<monitor exit>
10  Result: return null
11  Result: return null
```

Output 3: `guarana ClassBasic`

to the meta-object associated with the class represented by the Class object. For example, the invocation of a `static` method {11} is intercepted and presented to the class meta-configuration <4–5>. Even if the invocation expression appears to refer to a non-`static` method, as in line {12}, if compile-time overload resolution selects a `static` method, the operation is intercepted and presented to the meta-object of the class <6>, as is its result <11>. In this case, because the selected method is `synchronized`, monitor enter and exit operations are also intercepted <7–10>.

2.4 Meta-configuration propagation

In the previous examples, meta- and base-level code are intertwined, that is, code that modifies meta-configurations and base-level code appear together, in the same class. This clearly goes against the separation of concerns that can be attained through reflection. Fortunately, **Guaraná** provides mechanisms that allow a complete separation of the base and the meta application.

The meta application starts first, and sets up meta-configurations for classes and objects of the base application it is programmed to control. Then, it starts the base application. From then on, the meta application will only regain control by intercepting operations addressed to base-level classes or objects associated with meta-objects it has installed.

The MetaLogger class, for example, can be used as a meta application, because it provides a method main that creates a MetaLogger and associates it with a class specified as its first command-line argument. Then, it invokes method main of that class, passing to it the remaining command-line arguments.

Program 4, for example, is a simple non-reflective application, that can be made reflective by starting it as specified in the caption of Output 4. Method main of class MetaLogger, the meta application, associates a MetaLogger with class PropagBasic <1>, then invokes method main of that class <2>, just like the Java interpreter would do if it had been started as "`java PropagBasic`".

The meta application can configure the base application so as to determine the meta-configurations of dynamically created objects, due to the ability to intercept object creation provided by **Guaraná**. Whenever a class is instantiated, the primary meta-object of the creator is requested to provide a primary meta-object for the new object, before the object

Program 4 `PropagBasic.java`

```
1  public class PropagBasic {
2      public static void main(String[ ] argv) {
3          new PropagBasic(true);
4      }
5      PropagBasic(boolean another) {
6          if (another)
7              new PropagBasic(false);
8      }
9  }
```

```
 1 Initialize: java.lang.Class@1b46d8
 2 Operation: PropagBasic.main([Ljava.lang.String; [Ljava.lang.String;@1b9fa0)
 3 Configure PropagBasic@134c28 based on java.lang.Class@1b46d8: propagated
 4 Initialize: PropagBasic@134c28
 5 Message: BR.unicamp.Guarana.NewObject@1f8410 for java.lang.Class@1b46d8
 6 Operation: PropagBasic@134c28.PropagBasic(boolean true)
 7 Operation: PropagBasic@134c28.java.lang.Object()
 8 Result: return null
 9 Configure PropagBasic@134c78 based on PropagBasic@134c28: propagated
10 Initialize: PropagBasic@134c78
11 Message: BR.unicamp.Guarana.NewObject@1f8590 for java.lang.Class@1b46d8
12 Operation: PropagBasic@134c78.PropagBasic(boolean false)
13 Operation: PropagBasic@134c78.java.lang.Object()
14 Result: return null
15 Result: return null
16 Result: return null
17 Result: return null
```

Output 4: `guarana MetaLogger PropagBasic`

is constructed. If the object is created in a static method, its creator is defined as the class that contains the static method; otherwise, the creator is the object whose non-`static` method was being executed.

In line {3}, a `static` method of class PropagBasic creates a new object, so the meta-object of this class is requested to configure it <3>. Since MetaLoggers are programmed to *propagate* into meta-configurations of new objects, the new object is associated with a MetaLogger too <4>. Although it cannot be deduced from the output, no additional MetaLogger has to be created in this case. Since a single instance of MetaLogger can handle multiple base-level objects, the MetaLogger just installs itself in the meta-configuration of the new object.

After the meta-configuration of a new object is established, the meta-configuration of the class of the new object is informed of the instantiation, so that it can try to affect the meta-configuration of the object. The mechanism used to inform the meta-configuration of the class is one example of use of a general inter-meta-object communication facility that is part of the meta-object protocol of **Guaraná**: instances of classes that implement `interface` Message can be broadcast by the components of the meta-configuration of any given object. An instance of the class NewObject is thus broadcast to the meta-configuration of the class PropagBasic, to notify the creation of an instance thereof <5>. When a MetaLogger is presented a Message, it will always print the the String representation of the Message (by default, its object-id), the word "`for`" and the base-level object to whose meta-configuration the Message was broadcast.

Only after the broadcast terminates, the constructor of the new object is invoked. It is intercepted by the object's meta-configuration <6>, as is the implicit invocation of the constructor of the base class <7> that, like any other constructor invocation, returns `null` <8>.

Since the constructor is invoked with a `true` argument {3}, the test in line {6} results `true`, so a new PropagBasic object will be created {7}. Once again, the meta-object of the creator (the first PropagBasic object) is implicitly requested to provide a meta-configuration for the new object <9>, so the MetaLogger propagates itself <10>. Then, a NewObject message is broadcast to the meta-configuration of the class of the new object <11>, and its constructors are invoked <12–13>.

Finally, the outstanding invocations return: first, the two constructors of the second object <14–15>, then the pending constructor of the first object <16>, called in <6>, and finally the method main <17>, called in <2>.

3 Intermediate

Up to this point, we have covered almost every kind of interaction between the kernel of **Guaraná** and meta-objects associated with base objects. From now on, we are going to cover more complex interaction patterns, such as dynamic reconfiguration and meta-object composition. Then, we are going to present real implementations of meta-objects.

3.1 Dynamic reconfiguration

We have already introduced the method reconfigure, provided by the kernel of **Guaraná**. However, there are some details that have yet to be presented. Program 5 and Output 5 uncover such details.

In this example, three MetaLoggers are created {6–8}. In order to identify the output produced by each MetaLogger, each one is given a prefix, that will be inserted in the beginning of every line it produces. The first of these MetaLoggers, named cl, is associated with class ReconfigureBase {9}<1>, a base class of Reconfigure {3}.

When the program instantiates class Reconfigure {10}, unlike in the previous example, neither a configure request is issued nor a NewObject message is broadcast. These meta-level operations do not take place because class Reconfigure, that is both the creator of the new object and its class, has an empty meta-configuration, in spite of its superclass having a non-empty one.

When the program reconfigures object o so that its meta-object becomes a {11}, a new kind of message is broadcast <2>. Whenever an object whose meta-configuration is `null`

Program 5 `Reconfigure.java`

```
import BR.unicamp.Guarana.Guarana;
class ReconfigureBase {}
public class Reconfigure extends ReconfigureBase {
    public static void main(String[ ] argv) {
        final MetaLogger
            cl = new MetaLogger().setPrefix("cl:  "),
            a = new MetaLogger().setPrefix("a:  "),
            b = new MetaLogger().setPrefix("b:  ");
        Guarana.reconfigure(ReconfigureBase.class, null, cl);
        final Reconfigure o = new Reconfigure();
        Guarana.reconfigure(o, null, a);      // cl lets a become primary
        Guarana.reconfigure(o, null, b);      // a replaced with b
        Guarana.reconfigure(o, b, a);      // b replaced with a
        Guarana.reconfigure(o, b, a);      // ignored by a
        Guarana.reconfigure(o, null, null);      // b replaced with null
        Guarana.reconfigure(o, a, b);      // ignored by the kernel
    }
}
```

```
cl: Initialize: java.lang.Class@ddab8
cl: Message: BR.unicamp.Guarana.InstanceReconfigure@1b9cb8 for java.lang.Cla
ss@ddab8
cl: Operation: java.lang.Class@ddab8.java.lang.Class.getSuperclass()
cl: Result: return java.lang.Class@30438
a: Initialize: Reconfigure@134c08
a: Reconfigure Reconfigure@134c08: MetaLogger@1b88f0 -> MetaLogger@1b8938
b: Initialize: Reconfigure@134c08
a: Release: Reconfigure@134c08
b: Reconfigure Reconfigure@134c08: MetaLogger@1b8938 -> MetaLogger@1b88f0
a: Initialize: Reconfigure@134c08
b: Release: Reconfigure@134c08
a: Reconfigure Reconfigure@134c08: MetaLogger@1b88f0 -> MetaLogger@1b88f0
a: Reconfigure Reconfigure@134c08: MetaLogger@1b88f0 -> null
a: Release: Reconfigure@134c08
```

Output 5: `guarana Reconfigure`

is reconfigured, an InstanceReconfigure message is broadcast to the meta-configuration of the class of the object, then to the meta-configuration of its base class, and so on, up to the root of the inheritance hierarchy. Thus, meta-objects that belong to meta-configurations of classes may modify reconfiguration requests issued to its non-reflective instances. Note that method getSuperclass is invoked on class ReconfigureBase <3–4>: this is the kernel of **Guaraná** moving up in the inheritance hierarchy.

Since, in this example, no meta-object modified the reconfiguration request, meta-object a becomes the primary meta-object of o <5>. But another reconfiguration is soon requested {12}. Since the base object is reflective already, the meta-configurations of its classes do not participate in the reconfiguration process, only its primary meta-object does.

The kernel of **Guaraná** replaces the `null` argument in the reconfiguration request with a reference to the current primary meta-object before delegating the request to it, so the MetaLogger a prints a reference to itself on the left of the arrow in <6>, and a reference to b on the right of the arrow. Although it might have ignored the request or modified it, we can see it has accepted to be replaced, because b is initialized <7>, then a is requested to release the object <8>. A release invocation takes place just after a meta-object is removed from the meta-configuration of an object.

Instead of specifying `null`, we may name any particular meta-object as the second argument of a reconfigure request. In line {13}, a meta-object known to be the primary one is specified <9>. However, in **Guaraná**, a meta-object (called *composer*) may delegate operations to others, so the second argument in the reconfiguration request can be used to specify other meta-objects in the composition hierarchy.

When presented a reconfiguration request, a meta-object is supposed to return a meta-object to replace it. In this example, b finds itself in the second argument <9>, and accepts to be replaced with a, the third argument <10–11>. If it did not intend to be replaced, it should have returned a reference to itself, even if the second argument of the reconfiguration request were not a reference to itself—the kernel of **Guaraná** does not care whether the references compare equal, it will just replace a meta-object with the value it returns. Furthermore, although the third argument is supposed to be the meta-object that intends to replace the one passed as the second argument, it may be completely disregarded by the existing meta-objects, or used just as a hint to create the components of the new meta-configuration.

The program continues in line {14}, by repeating the reconfiguration request, but now it is a that listens to it <12>. Since it does not match the second argument, it ignores the reconfiguration request, returning itself. No initialization nor release takes place.

In line {15}, once again, the second argument to reconfigure is `null`, so the kernel of **Guaraná** passes the primary meta-object as the second argument. Thus, MetaLogger a matches the request <13>, and accepts to be replaced with a `null` meta-object. No initialization takes place—it is pointless to `initialize` a `null` meta-object—, but the previous meta-object is released <14>, and object o has become non-reflective again.

The final invocation of reconfigure {16} is ignored by the kernel of **Guaraná**, because a `null` meta-configuration would only match a request that had `null` as the second argument.

3.2 Composing meta-objects

The meta-object protocol of **Guaraná** was designed so as to make it possible to create a meta-object that interacts with other meta-objects just like the kernel of **Guaraná** would, providing them with operations, results, messages, initialization and release requests in a way that these meta-objects may believe they are directly called by the kernel of **Guaraná**.

This special kind of meta-object is called a composer. The implementation of **Guaraná** includes a useful implementation of composer: the SequentialComposer. Essentially, it delegates operations to an array of meta-objects, and delegates results to these meta-objects in reverse order.

Program 6 creates an instance of class Composer {4}, two MetaLoggers {6–7}, an array containing these MetaLoggers {8} and a SequentialComposer that delegates to them {9}. As you may notice in Output 6, one of the MetaLoggers prefixes all lines it outputs with "`a: `", and the other, with "`b: `". Lines <1–2>, for example, are printed when the composer is associated with the Compose object {10}. When method initialize of the composer is invoked, it delegates the initialization request to the meta-objects contained in the array passed to its constructor.

Program 6 `Compose.java`

```
 1  import BR.unicamp.Guarana.*;
 2  public class Compose {
 3      public static void main(String[ ] argv) {
 4          Compose o = new Compose();
 5          MetaObject
 6              a = new MetaLogger().setPrefix("a:   "),
 7              b = new MetaLogger().setPrefix("b:   "),
 8              mos[ ] = { a, b },
 9              mo = new SequentialComposer(mos);
10          Guarana.reconfigure(o, null, mo);
11          o.another();
12      }
13      Compose another() {
14          return new Compose();
15      }
16  }
```

When method another is invoked on the Compose object {11}, it is intercepted and presented to the composer, that delegates first to meta-object a <3>, then to meta-object b <4>, and finally tells the kernel of **Guaraná** to perform the Operation.

Method another just creates a new Compose object {14}, but this involves propagation of meta-configuration. The SequentialComposer requests its meta-objects to configure the new object <5–6>, then it creates a new SequentialComposer that delegates to the meta-objects selected for the new object <7–8>.

```
 1  a: Initialize: Compose@134608
 2  b: Initialize: Compose@134608
 3  a: Operation: Compose@134608.Compose.another()
 4  b: Operation: Compose@134608.Compose.another()
 5  a: Configure Compose@134cc8 based on Compose@134608: propagated
 6  b: Configure Compose@134cc8 based on Compose@134608: propagated
 7  a: Initialize: Compose@134cc8
 8  b: Initialize: Compose@134cc8
 9  a: Operation: Compose@134cc8.Compose()
10  b: Operation: Compose@134cc8.Compose()
11  a: Operation: Compose@134cc8.java.lang.Object()
12  b: Operation: Compose@134cc8.java.lang.Object()
13  b: Result: return null
14  a: Result: return null
15  b: Result: return null
16  a: Result: return null
17  b: Result: return Compose@134cc8
18  a: Result: return Compose@134cc8
```

Output 6: `guarana Compose`

Note that the invocations of the constructors of the new object are intercepted and delegated to both meta-objects <9–12>, then the results of the constructor invocations (`null`) are presented to them, but first to b <13,15>, then to a <14,16>. This inversion is a particular characteristic of SequentialComposer; other composers might do it differently.

Finally, the result of the invocation of method another is presented to the original composer, that delegates it to the MetaLoggers <17–18>.

3.3 Modifying results

In the next example, we are going to show an actual implementation of MetaObject. Program 7 shows how a meta-object is supposed to handle intercepted operations and results, and how it can modify the results of intercepted operations, after its execution, or even prevent their execution.

First, let us take an overview of the presented code. Two static variables are defined, namely, hashCode and toString {3}. Variable hashCode is initialized {5–7} by searching class Object for a method named "`hashCode`", that does not take any argument—so the array of classes that specify its argument list has length zero. Method getDeclaredMethod of class Class returns an instance of class Method that represents this searched method. Variable toString is initialized similarly {8–10}. These variables will ease the verification of whether operations correspond to invocations of the methods they represent.

The first method handle {12–19} is invoked by the kernel of **Guaraná** (or by a composer) before an operation (the first argument) is delivered to its target object (the second one). Our implementation will check whether the operation is an invocation of any of the two selected methods. Although it would be syntactically correct to compare the methods using operator ==, the program would be semantically wrong, because there may be

Program 7 `ModifyResult.java`

```
import BR.unicamp.Guarana.*;
public class ModifyResult extends MetaObject {
    final static java.lang.reflect.Method hashCode, toString;
    static {
        try { hashCode =
                    Object.class.getDeclaredMethod("hashCode", new Class[0]);
        } catch (NoSuchMethodException e) { hashCode = null; }
        try { toString =
                    Object.class.getDeclaredMethod("toString", new Class[0]);
        } catch (NoSuchMethodException e) { toString = null; }
    }
    public Result handle(final Operation op, final Object ob) {
        final java.lang.reflect.Method meth = op.getMethod();
        if (hashCode.equals(meth))
            return Result.returnInt(0, op);      // make it return 0
        if (toString.equals(meth))
            return Result.modifyResult;      // asks for permission to modify it
        return Result.noResult;      // not interested in the result
    }
    public Result handle(final Result res, final Object ob) {
        Operation op = res.getOperation();
        if (toString.equals(op.getMethod()))
            return Result.returnObject("modified "      // replace result
                                    + res.getObjectValue(), op);
        return null;
    }
    public static void main(String[ ] argv) {
        Object o = new Object();
        MetaObject[ ] mos = {
            new MetaLogger().setPrefix("a:   "),
            new ModifyResult(),
            new MetaLogger().setPrefix("b:   ") };
        Guarana.reconfigure(o, null, new SequentialComposer(mos));
        System.out.println(o);
    }
}
```

multiple Method objects associated with a single method, and this operator only compares *references*. Thus, the comparison must be performed using method equals, that compares the actual *values* of the objects, that is, it yields `true` if, and only if, two Method objects correspond to the same method of the same class. Note that our implementation does not test whether the operation is a method invocation. Nevertheless, it is correct, because method getMethod of class Operation returns `null` if the operation is not a method invocation, causing the comparison performed by method equals to yield `false`. However, if the program invoked method equals of object meth, passing hashCode or toString as arguments, it would be incorrect: a NullPointerException would be raised for operations other than method invocations, because meth would be `null`.

If the intercepted operation is an invocation of method hashCode {14}, the MetaObject will produce a result for the operation {15}, so that the operation will never be delivered to the target object for execution. Otherwise, if it is an invocation of method toString {16}, the MetaObject will request to be presented, and to possibly modify, the result of the operation {17}, after it is executed. For any other operation, the meta-object indicates it is not interested in the result {18}.

The second method handle {20–26} is invoked after a result is produced for an operation, either by its execution or by another meta-object. Our implementation just checks whether the Operation the Result refers to {21} is an invocation of method toString {22}, in which case it will return a new Result object, containing a modified String {23–24}. Otherwise, it returns `null`, what is equivalent to returning the Result it was presented.

The execution of method main {27–35} produces Output 7. First, it creates an Object {28} and associate it with a SequentialComposer {33} that delegates to a MetaLogger {30}, an instance of our implementation of meta-object {31} and another MetaLogger {32}. We may notice the configuration was successful from the initialization messages printed by the MetaLoggers <1–2>.

```
 1  a: Initialize: java.lang.Object@134898
 2  b: Initialize: java.lang.Object@134898
 3  a: Operation: java.lang.Object@134898.java.lang.Object.toString()
 4  b: Operation: java.lang.Object@134898.java.lang.Object.toString()
 5  a: Configure java.lang.StringBuffer@1f0f30 based on java.lang.Object@134898:
     not propagated
 6  b: Configure java.lang.StringBuffer@1f0f30 based on java.lang.Object@134898:
     not propagated
 7  a: Operation: java.lang.Object@134898.java.lang.Object.getClass()
 8  b: Operation: java.lang.Object@134898.java.lang.Object.getClass()
 9  b: Result: return java.lang.Class@30438
10  a: Result: return java.lang.Class@30438
11  a: Operation: java.lang.Object@134898.java.lang.Object.hashCode()
12  a: Result: return 0
13  b: Result: return java.lang.String@1f1d10
14  a: Result: return java.lang.String@1f1f50
15  modified java.lang.Object@0
```

Output 7: `guarana ModifyResult`

Last thing method main does is to print the Object it has created to the standard output. This requires representing the Object as a String, so method println invokes toString, as logged by the MetaLoggers <3–4>. Between <3> and <4>, the ModifyResult meta-object was prrequested to handle the operation, but it produces no visible output.

The standard implementation of method toString concatenates, in a StringBuffer, the name of the class of the object with an "@" sign and an hexadecimal representation of the hash code of the object. MetaLoggers do not propagate into meta-configurations of StringBuffers to avoid excessive noise <5–6>, and the standard implementation of method configure in class MetaObject does not propagate: it just returns `null`. The SequentialComposer notices that none of its meta-objects propagated into the meta-configuration of the new object, and neither does it, so the StringBuffer gains an empty meta-configuration.

The method toString invokes getClass to find out what class the object belongs to <7–10>, then invokes getName on this class, but does not produce any output, because the class of the object, java.lang.Object, has an empty meta-configuration.

Then, the method toString obtains the hash code of the target object, by invoking method hashCode <11>. When the ModifyResult meta-object is presented this operation {14}, it provides a zero result for it {15}. Thus, meta-object b never sees the operation; the result is presented to a <12> and returned as if the operation had yielded it.

Finally, method toString returns the String representation of the StringBuffer that was used to concatenate the class name, the "@" sign and the hash code of the object. MetaLogger b is presented this String as the result of the operation first <13>, then the ModifyResult meta-object is. Instead of letting the result reach a unmodified, the ModifyResult meta-object notices the original operation was an invocation of method toString, and concatenates the string "`modified `" to the original result. Note that the String object presented as result for a <14> is not the same that was presented to b <13>.

Since both a and the SequentialComposer return the modified result, the modified String is printed by method println: it contains the "`modified `" prefix, and the hash code after the "@" sign is zero <15>.

3.4 Modifying operations

In addition to modifying results of operations, meta-objects can also create operations from the meta level, as does Program 8. The subclass of MetaObject implemented in this example assumes its instances will be associated with instances of the class defined in Program 9.

Like in the previous example, a few `static` variables are defined in ModifyOperation: toString {3} denotes method toString of class Object {6–8}, whereas callSuper {4} denotes the field callSuper of class ModifyOperationBase {9–11}.

This example introduces the class OperationFactory: it is used to create Operations addressed to base-level objects from the meta level. If a meta-object intends to create arbitrary operations to objects it reflects upon, it must save the OperationFactory it is initialized with, as does method initialize {14–15}.

Operation factories allow meta-objects to violate access control, giving them privileged access even to private fields and methods of base-level objects. Furthermore, using operation factories, it is possible to create and perform synchronization operations such as entering

Program 8 `ModifyOperation.java`

```
import BR.unicamp.Guarana.*;
public class ModifyOperation extends MetaObject {
      static final java.lang.reflect.Method toString;
      static final java.lang.reflect.Field callSuper;
      static {
            try { toString =
                        Object.class.getDeclaredMethod("toString", new Class[0]);
            } catch (NoSuchMethodException e) { toString = null; }
            try { callSuper =
                        ModifyOperationBase.class.getDeclaredField("callSuper");
            } catch (NoSuchFieldException e) { callSuper = null; }
      }
      OperationFactory opf = null;
      public void initialize(final OperationFactory opf, final Object o)
      { this.opf = opf; }
      public Result handle(final Operation op, final Object ob) {
            System.out.println("Operation:  " + op);
            if (op.isMethodInvocation()) try {
                  final java.lang.reflect.Method m = op.getMethod();
                  if (!toString.equals(m) && m.getName().equals("toString")
                        && opf.read(callSuper).perform().getBooleanValue()) {
                        Operation newOp = opf.invoke(toString, new Object[0], op);
                        System.out.println("Replaced with:  " + newOp);
                        return Result.operation(newOp, Result.inspectResultMode);
                  }
            } catch (IllegalAccessException e) { }
            return Result.inspectResult;
      }
      public Result handle(final Result res, final Object ob)
      { System.out.println("Result:  " + res); return null; }
      public static void main(String[ ] argv) {
            final Object oFalse = new ModifyOperationBase(false);
            Guarana.reconfigure(oFalse, null, new ModifyOperation());
            System.out.println("oFalse:  " + oFalse);
            final Object oTrue = new ModifyOperationBase(true);
            Guarana.reconfigure(oTrue, null, new ModifyOperation());
            System.out.println("oTrue:  " + oTrue);
      }
}
```

Program 9 `ModifyOperationBase.java`

```
import BR.unicamp.Guarana.*;
class ModifyOperationBase {
    final private boolean callSuper;
    public ModifyOperationBase(final boolean callSuper) {
        this.callSuper = callSuper;
    }
    public String toString() { return "derived method was called"; }
}
```

and leaving an object's monitor. Another ability provided by operation factories, explored in our example, is to modify overload resolution and dynamic dispatching mechanisms.

Method handle {16–28} prints a description of every operation it is requested to handle, just like the corresponding method of class MetaLogger does. Note that method toString of class Operation does not invoke any method of the base-level object. If it did, these interactions would have to be intercepted, possibly leading to infinite recursion. Instead of calling method toString of the base-level Object, it calls an alternate method of the kernel of **Guaraná**, that emulates the execution of the former, without any interceptable interaction with the Object.

After printing a description of the operation, method handle checks whether the operation is a method invocation {18} of a method {19} named toString, but not the one implemented in class Object {20}, and finally tests whether field callSuper of the base-level object is `true` or `false` {21}.

This last condition illustrates one of the possible uses of operation factories: to create operations from the meta level and perform them. In this case, the meta-object uses its OperationFactory to create an Operation that reads the value of field callSuper, then performs it. Note that, although field callSuper is `private`, the operation can be created and performed, due to the fact that an operation factory gives privileged access to the object it refers to. Also note that the target object of the operation is not specified: an operation factory is associated with a base-level object when it is instantiated, and it can only create operations addressed to that object.

After the field read operation is performed, its result is returned in a Result object, so method getBooleanValue must be used to obtain its value. If it yields `false`, so does the whole condition, and method handle terminates returning Result.inspectResult {27}. Otherwise, we may observe the other use for an operation factory: to replace an operation that is being handled.

In line {22}, the operation factory is requested to create an operation that will invoke method toString of class Object, without any arguments—this is why the array of Objects is created with length zero. But note that a third argument is passed to invoke: the operation currently being handled.

When an operation is passed as the last argument to a method of an operation factory, the new operation will be a replacement operation, that is, it may be used to replace the original operation. But it will only effectively replace the current operation if the method handle returns a Result object containing the new Operation {24}. Such a Result object may also contain a request to inspect (or modify) the result of the operation, after it is performed.

The other method handle {29–30} just prints every result it is presented, almost exactly like the corresponding method of MetaLogger does.

```
 1  Operation: ModifyOperationBase@1b9328.ModifyOperationBase.toString()
 2  Operation: ModifyOperationBase@1b9328.ModifyOperationBase.callSuper
 3  Result: return false
 4  Result: return java.lang.String@1be830
 5  oFalse: derived method was called
 6  Operation: ModifyOperationBase@1e0350.ModifyOperationBase.toString()
 7  Operation: ModifyOperationBase@1e0350.ModifyOperationBase.callSuper
 8  Result: return true
 9  Replaced with: ModifyOperationBase@1e0350.java.lang.Object.toString()
10  Operation: ModifyOperationBase@1e0350.java.lang.Object.getClass()
11  Result: return java.lang.Class@1b2498
12  Operation: ModifyOperationBase@1e0350.java.lang.Object.hashCode()
13  Result: return 1966928
14  Result: return java.lang.String@1eb510
15  oTrue: ModifyOperationBase@1e0350
```

Output 8: `guarana ModifyOperation`

Output 8 presents the output produced by running method main {31–38}. First, it creates an instance of ModifyOperationBase {32} with a `false` callSuper {3} (in Program 9), associates it with an instance of a ModifyOperation meta-object {33}, then prints a String representation of it preceded by "`oFalse: `" {34}.

In order to obtain the String representation of the Object, method toString is invoked, and dynamic dispatching selects the implementation {7} in class ModifyOperationBase <1>. The meta-object notices that a method named toString was invoked {20}, and creates an operation to read field callSuper {21}. This operation is intercepted <2>, despite the fact that it was created in the meta level. The result is presented to the meta-object too <3>. Since callSuper is `false`, the test performed by the meta-object {20–21} fails, so it lets the operation pass unchanged {27}, and it returns the string hard-coded in method toString {7} of class ModifyOperationBase <4>, as shows the line printed {34} by method main <5>.

When an instance of ModifyOperationBase is created with a `true` callSuper {35}, however, the meta-object behaves differently. Dynamic dispatching selects the derived implementation <6>, but now, the meta-object finds callSuper to be true <7–8>, so it replaces the invocation of method toString of class ModifyOperationBase with an invocation of method toString of class Object {22–24}<9>. No further dynamic dispatching takes place, so the standard implementation of method toString is really executed, as we can notice by the methods this execution invokes <10–13>. Finally, method toString returns <14>, and method main prints {37} the standard String representation of object oTrue <15>.

As a final note, we should observe that the meta-object of this example should never be associated with more than one base-level object, because it can only store one operation factory {13}. If it were ever associated with two or more objects, it would always read field callSuper from the most recently initialized object, that might not be the target of the operation being handled. If it tried to create a replacement operation for a different target object, an exception would be thrown, but creating non-replacement operations has no such restriction, so mistakes might have gone unnoticed.

3.5 Using messages

Guaraná provides a mechanism that allows any object whose class `implements interface` Message to be broadcast to the components of the meta-configuration of an object. This allows communication between meta-level components without requiring them to be explicitly named or even known in advance. In Program 10, for example, defines a Message that can be used to look for a meta-object in the meta-configuration of an object. It could be easily modified to carry any kind of information to be given to a selected meta-object, or to look for meta-objects that present a certain property, such as being an instance of a selected class, instead of just comparing references.

When an instance of class AreYouThere is created, a meta-object to be looked for {4} must be specified {5–7}, and wasThere is initialized to `false` {3}. A meta-object that understands this kind of Message should invoke method IamHere {8–11} to flag its existence; if the given reference is the desired one {9}, wasThere is set to `true` {10}. One could check whether the meta-object was found by invoking method wasThere {12}, but method lookFor {13–17} provides a more convenient interface for using this mechanism. It creates the AreYouThere message {14}, broadcasts it to meta-configuration of the specified object {15}, then tests whether the specified meta-object was found {16}.

Method main {18–31} instantiates a meta-object that can handle AreYouThere messages {19–24}, using the anonymous inner class notation introduced in Java 1.1. Method handle {20–23} is invoked by broadcast of the kernel of **Guaraná**; if the broadcast Message is an instance of class AreYouThere {21}, it invokes method IamHere {22}.

Then, the program creates an object {25} and looks for the created meta-object in its meta-configuration {26}. As expected, Output 9 shows it is not there <1>. Then, method main reconfigures the object so that mo becomes its primary meta-object {27}. After that, the program looks for it again {28}, and now it is found <2>. Finally, the object is reconfigured again, so that its primary meta-object becomes `null` {29}, and, once again, the meta-object cannot be found any more {30}<3>.

3.6 Creating Proxies

Proxies are useful for at least two purposes: they can be used to represent objects from different address spaces and as object shells in which real objects are going to be created or reincarnated from persistent storage. In this example, we show how to turn a proxy, created using a meta-level interface, into a real object.

Program 11 defines class MakeProxy, a specialization of MetaObject, and two inner classes, namely, Base {3} and Init {4}. Base is just an arbitrary base-level class that will be instantiated from the meta-level, and Init is an implementation of Message we are going to use to communicate with the MakeProxy meta-object.

Program 10 `AreYouThere.java`

```
import BR.unicamp.Guarana.*;
public class AreYouThere implements Message {
    private boolean wasThere = false;
    private final MetaObject mo;
    public AreYouThere(final MetaObject mo) {
        this.mo = mo;
    }
    public void IamHere(final MetaObject mo) {
        if (this.mo == mo)
            wasThere = true;
    }
    public boolean wasThere() { return wasThere; }
    public static boolean lookFor(final MetaObject mo, final Object o) {
        AreYouThere m = new AreYouThere(mo);
        Guarana.broadcast(m, o);
        return m.wasThere();
    }
    public static void main(String[ ] argv) {
        final MetaObject mo = new MetaObject() {
            public void handle(final Message m, final Object o) {
                if (m instanceof AreYouThere)
                    ((AreYouThere)m).IamHere(this);
            }
        };
        Object o = new Object();
        System.out.println(lookFor(mo, o));
        Guarana.reconfigure(o, null, mo);
        System.out.println(lookFor(mo, o));
        Guarana.reconfigure(o, mo, null);
        System.out.println(lookFor(mo, o));
    }
}
```

```
false
true
false
```

Output 9: `guarana AreYouThere`

Program 11 `MakeProxy.java`

```
import BR.unicamp.Guarana.*;
public class MakeProxy extends MetaObject {
    public static class Base {}
    public static class Init implements Message {}
    public static void main(String [ ] argv) {
        Guarana.reconfigure(Base.class, null,
                                        new MetaLogger().setPrefix("c:   "));
        MetaObject mo = new SequentialComposer(new MetaObject [ ] {
            new MetaLogger().setPrefix("o:   "), new MakeProxy()
        }) {
            public MetaObject reconfigure
            (final Object o, final MetaObject pre, final MetaObject pos) {
                if (pre == this) return pos;
                else return super.reconfigure(o, pre, pos);
            }
        };
        Base o = (Base)Guarana.makeProxy(Base.class, mo);
        Guarana.reconfigure(Base.class, null, null);
        Guarana.broadcast(new Init(), o);
        Guarana.reconfigure(o, null, null);
        System.out.println(o);
    }
    private OperationFactory opf;
    public void initialize(final OperationFactory opf, final Object ob)
    { this.opf = opf; }
    public void handle(final Message m, final Object ob) {
        if (m instanceof Init) try {
            final java.lang.reflect.Constructor
                c = Base.class.getDeclaredConstructor(new Class[0]);
            opf.construct(c, new Object[0]).perform();
        } catch (NoSuchMethodException e) {
        } catch (IllegalAccessException e) {}
    }
}
```

Method main associates a MetaLogger with prefix "`c: `" with class Base {6–7}, as you may see in Output 10 <1>, then it creates an instance of an anonymous subclass of SequentialComposer {8–16} that delegates to a MetaLogger with prefix "`o: `" and a MakeProxy meta-object {9}. The sequential composer is specialized so that its method reconfigure {11–15} accepts to be replaced {13} (the default implementation {14} will only accept reconfiguration requests for meta-objects it delegates to).

```
 1  c: Initialize: java.lang.Class@1b04f8
 2  c: Message: BR.unicamp.Guarana.NewProxy@1e4728 for java.lang.Class@1b04f8
 3  c: Operation: java.lang.Class@1b04f8.<monitor enter>
 4  c: Result: return null
 5  c: Message: BR.unicamp.Guarana.InstanceReconfigure@1e4770 for java.lang.Clas
    s@1b04f8
 6  c: Operation: java.lang.Class@1b04f8.java.lang.Class.getSuperclass()
 7  c: Result: return java.lang.Class@30438
 8  o: Initialize: MakeProxy$Base@134ce8
 9  c: Operation: java.lang.Class@1b04f8.<monitor exit>
10  c: Result: return null
11  c: Reconfigure java.lang.Class@1b04f8: MetaLogger@1b9298 -> null
12  c: Release: java.lang.Class@1b04f8
13  o: Message: MakeProxy$Init@134d78 for MakeProxy$Base@134ce8
14  o: Operation: MakeProxy$Base@134ce8.MakeProxy$Base()
15  o: Operation: MakeProxy$Base@134ce8.java.lang.Object()
16  o: Result: return null
17  o: Result: return null
18  o: Release: MakeProxy$Base@134ce8
19  MakeProxy$Base@134ce8
```

Output 10: `guarana MakeProxy`

A proxy of class Base is created by invoking method makeProxy of class Guarana {17}. Whenever a class is instantiated, the meta-configuration of the class is notified with a NewObject message. However, when a proxy instance of a class is created, the broadcast message is an instance of class NewProxy <2>, a subclass of NewObject, so that meta-objects of the class can behave differently for proxy objects.

The makeProxy request also includes a MetaObject specification; after the NewProxy message is broadcast, a reconfigure request is issued to install the specified MetaObject as the primary meta-object of the proxy. If the meta-configuration of the class has already configured the proxy with a meta-object, the reconfiguration request would be presented to this meta-object. However, in our example, the proxy was not made reflective yet, so the meta-configurations of the class of the proxy and of its superclasses will be given the opportunity to modify the reconfigure request.

First, method reconfigure synchronizes on the Class object that represents the class of the proxy <3–4>, to ensure that the reconfiguration is atomic (if the object were reflective already, the synchronization would be performed on its primary meta-object). Then, an InstanceReconfigure message is broadcast to class of the proxy <5>, then to its superclass, and so on, as implied in <6,7>. Since no class meta-configuration modifies the reconfigure request, the SequentialComposer is associated with the proxy and initialized. It propagates the initialization to its component meta-objects <8>. Method initialize of our meta-object saves the operation factory it is presented in opf {23–25}. When reconfiguration terminates, so does the synchronization on the class of the proxy <9–10>.

In order to avoid excessive noise in the output, the MetaLogger is removed from the meta-configuration of class Base {18}<11–12>, then we broadcast an Init message to the meta-configuration of the proxy {19}<13>. When method handle {26–33} is invoked, it notices the Message is an Init message {27}. So it looks up a constructor taking no arguments in class Base {28–29}, creates an operation that invokes that constructor and performs it {30}. This invocation is intercepted <14>, and so is the implicit invocation of the constructor of the superclass {15}, as well as their `null` results <16–17>.

Finally, method main removes the composer from the meta-configuration of the object {20}<18>, and the object becomes a regular non-reflective object, indistinguishable from any object created with a `new` expression. Just to probe the behavior of the object, we print its String representation {21}<19>.

It is worth noting that invoking a constructor on a proxy is not strictly necessary. When the object is going to be used only as a proxy, no operation would ever reach it, so there is no reason to construct a real object. But even if this is not the case, and the object will actually execute operations, constructor invocation may be skipped, and fields may be initialized, if necessary, by creating and performing operations from the meta level.

4 Advanced

In this section, we are no longer going to present full executable examples: they would be too long and complex. Instead, we are going to discuss some advanced issues regarding meta-configuration management and security, using small code snippets for illustrative purposes only.

4.1 Meta-objects with restricted access

When a meta-object is initialized, it is given an operation factory that may allow it to create operations for the base level object from the meta-level. However, composers may initialize meta-objects they delegate to with operation factories that restrict the kind of operations they can create. The primary meta-object is always initialized with an operation factory that provides full access to the base-level object (unless the meta-object is reflective itself, and its own meta-object modifies the invocation of initialize).

In Program 12, for example, we present an implementation of OperationFactory that throws IllegalAccessExceptions {8,15} when requested the creation of any operation that accesses `private` fields {7,14}. The constructor of this class takes another OperationFactory as argument {4}, and the default implementation of methods from class OperationFactoryFilter delegate requests to the operation factory given as the constructor argument {9,16}.

This allows the creation of a hierarchy of operation factories that resembles the hierarchy of composers and meta-objects, except that composers refer to their children, whereas

Program 12 `OpFactNoPrivate.java`

```
 1  import java.lang.reflect.*;
 2  import BR.unicamp.Guarana.*;
 3  public class OpFactNoPrivate extends OperationFactoryFilter {
 4      public OpFactNoPrivate(final OperationFactory opf) { super(opf); }
 5      public Operation read(final Field field, final Operation op)
 6      throws IllegalAccessException, IllegalArgumentException {
 7          if (Modifier.isPrivate(field.getModifiers()))
 8              throw new IllegalAccessException("access denied");
 9          else return super.read(field, op);
10      }
11      public Operation write(final Field field, final Object value,
12                             final Operation op)
13      throws IllegalAccessException, IllegalArgumentException {
14          if (Modifier.isPrivate(field.getModifiers()))
15              throw new IllegalAccessException("access denied");
16          else return super.write(field, value, op);
17      }
18      static {
19          Guarana.reconfigure(OpFactNoPrivate.class, null, new MetaObject() {
20              public void handle(final Message m, final Object o) {
21                  if (m instanceof InstanceReconfigure)
22                      ((InstanceReconfigure)m).setMetaObject(null);
23              }
24              public MetaObject reconfigure
25                  (final Object o, final MetaObject m, final MetaObject n)
26                  { return this; }
27          });
28      }
29  }
```

operation factories refer to their parents. Thus, the identity of meta-objects higher in the composition hierarchy is protected from lower ones.

However, OperationFactoryFilters carry a potential security hole: if a child meta-object is able to associate a meta-object with an OperationFactoryFilter, it may be able to obtain a reference to the OperationFactory the filter delegates to, as this reference is stored in a private field of it. Thus, we have associated a meta-object with class OpFactNoPrivate {19–27} that prevents its non-reflective instances from being reconfigured {20–23} and rejects any reconfiguration that might remove itself from the meta-configuration of its class {24–26}. This is accomplished by handling messages of type InstanceReconfigure {21} and removing any meta-object they might carry into the meta-configuration of its instances {22}, and by always returning `this` for any reconfiguration request {26}.

This meta-object does not take care of meta-configurations introduced by meta-configuration propagation, though: if the meta-object that creates this operation factory is reflective, its meta-configuration may freely propagate into the meta-configuration of the operation factory, spoiling the offerred protection mechanism.

4.2 Multi-object meta-objects

Meta-objects sometimes have to interact with multiple base-level objects. Some are stateless, in the sense that they do not need to store any information about base-level objects they interact with, so they may safely disregard methods initialize and release.

Other meta-objects, like the one presented in Program 13, must maintain information regarding several objects, for example, in order to create operations for all those objects. This could be redesigned so as to have single-object meta-objects only, but this stricter requirement may lead to complicated solutions, so we have decided to support stateful multi-object meta-objects.

The easiest way to maintain information about multiple objects is to create a Hashtable that maps an object to the information stored about the object {4}. However, the implementation of hash table compares objects by invoking method equals, and obtains their hash codes by invoking method hashCode. When these methods are invoked, they are likely to be intercepted, and this may lead to infinite recursion.

In order to avoid this kind of problem, **Guaraná** provides the class HashWrapper, that should be used to wrap references to base-level objects used as keys in hash tables. HashWrappers only compare references to objects, instead of invoking method equals, and use the hash code returned by method hashCode defined in class Guarana, that emulates the behavior of method hashCode of class Object without interacting with the object.

Another problem that multi-object meta-objects must be aware of is garbage collection: storing references to base-level objects in meta-objects may prevent objects from being garbage collected. Assume, for example, that meta-object m belongs to the meta-configurations of objects o1 and o2, so m stores explicit references to them. Assume o1 is referred by other active objects, but o2 is not. Under normal conditions, o2 would be candidate for garbage collection. However, since o1 holds an implicit (possibly indirect) reference to m, and m holds a reference to o2, o2 will not be considered unreachable.

The only way to solve this problem would be to use weak references, but this concept is not supported as of release 1.1 of the Java API [5]. Since it will be available in the next release of the Java API, in the future, we may decide to modify HashWrapper so that it only stores weak references to objects.

The implementation in Program 13 handles multiple invocations of initialize and release, that may take place at reconfiguration time. For each base-level object, the most recently provided operation factory is maintained {6}, as well as a count indicating the difference between the number of invocations of initialize and the number of invocations of release {7}. When the meta-object is initialized {13–19}, the base-level object is wrapped in a HashWrapper {15} and looked up in the objDict Hashtable {16}. If the object is listed in the objDict already, method initialize of the object data {9} is executed, so as to save the new operation factory and increment the initialization count {10}. If the object was not listed in the objDict yet, a new ObjData object is created and registered in the objDict {18}.

Program 13 `MultiMeta.java`

```
 1  import BR.unicamp.Guarana.*;
 2  import java.util.Hashtable;
 3  public abstract class MultiMeta extends MetaObject {
 4      protected Hashtable objDict = new Hashtable();
 5      protected static class ObjData {
 6          public OperationFactory opf;
 7          public int initcount = 1;
 8          public ObjData(OperationFactory opf) { this.opf = opf; }
 9          public synchronized void initialize(OperationFactory opf)
10          { this.opf = opf; ++initcount; }
11          public synchronized boolean release() { return --initcount == 0; }
12      }
13      public synchronized void
14          initialize(final OperationFactory opf, final Object ob) {
15          final HashWrapper obw = new HashWrapper(ob);
16          final ObjData od = (ObjData)objDict.get(obw);
17          if (od ≠ null) od.initialize(opf);
18          else objDict.put(obw, new ObjData(opf));
19      }
20      public synchronized void release(final Object ob) {
21          final HashWrapper obw = new HashWrapper(ob);
22          final ObjData od = (ObjData)objDict.get(obw);
23          if (od ≠ null && od.release())
24              objDict.remove(ob);
25      }
26  }
```

When method release {20–25} is invoked, the object is also wrapped {21} and looked up in the objDict {22}. If it is not found {22}, something must be wrong, but the error condition is ignored. If it is found, method release of the ObjData {11} is performed to decrement the initcount. When this counter reaches zero, the meta-object was told to release the object as many times as it was initialized, so it can remove the object from the objDict {24}.

4.3 Coping with replaced operations

There are two issues to care about, regarding replaced operations. First, there is the composer issue: a composer must only accept replacement operations that actually replace the operation it requested a component meta-object to handle. Similarly, replacement results should refer to the same operation the replaced result referred to.

The other issue has to do with stateful meta-objects. A meta-object must be aware that it may be presented results of operations it was never requested to handle or whose results it was not interested in. This may be caused by reconfiguration, composer laziness (not remembering whether the meta-object requested for a result or not) or operation replacement.

The reconfiguration problem might be avoided if composers refused or delayed reconfigurations while there were pending operations, but this deadlocks if the thread that is handling an operation requests a reconfiguration. Another solution, that fixes the second problem too, is to implement composers that remember what meta-objects should be presented the results for each operation.

But this leads to the third problem: the result handed to the composer may refer to an operation that replaced the operation the composer was originally requested to handle. Therefore, composers, and meta-objects in general, that intend to match results with operations, should store a reference to the original operation, instead of any replacement thereof. For this reason, every Operation provides methods to find out whether it is a replacement (isReplacement), the replaced operation (getReplaced) and the original operation (getOriginal), that iterates through the replaced operations until it reaches the original operation.

A replacement operation must have result types compatible with operations it replaces, i.e., the result type of the replacement must be implicitly convertible to the result type of the replaced operation, and the exception types that may propagate out of the replacement operation must be convertible to exception types that might propagate out of the replaced operation.

However, there is one exception to this rule: it is possible to create untyped placeholder operations using method nop of class OperationFactory. These operations are intended to be replaced with actual operations by some meta-object, usually the one that created it. This is useful to simulate operations intended to be intercepted only by meta-objects located after its creator in the operation handling sequence. When the meta-object receives the placeholder, it returns the actual intended operation as a replacement of the placeholder. With regard to the return value, it may either let it remain unchanged or replace it with, say, an exceptional value.

It should also be noted that it is possible to create other invalid operations from the meta-level. For instance, it is possible to create a method invocation operation with arguments that cannot be converted to the expected parameter types, or an assignment operation to a field that does not exist in the target object. However, before any operation is delivered to the base object, it is validated, and an exceptional result is produced if validation fails.

One possible use of this feature is to create *pseudo* methods and fields, that are introduced by meta-objects, and do not exist in the base level. Pseudo-objects of type Method and Field can be created (makeProxy) and used to identify such inexistent operations.

A more interesting application of this feature is to extend arrays. For example, meta-objects can arrange for arrays to seem to contain more (or less) elements, or even to grow or shrink on demand. Elements of a persistent array, for example, can be reincarnated on demand, instead of all at once.

4.4 Reconfiguration details

Guaraná goes to a great length to ensure that meta-level reconfigurations are atomic, and that operations are only delivered to the base-level object after its current primary meta-object has been requested to handle the operation. Two methods of class Guarana participate in this effort: reconfigure and perform.

Method reconfigure tries to ensure that reconfigurations are atomic. This is attained by synchronizing the reconfiguration operation either on the object's primary meta-object, if it is not `null`, or on the class the object belongs to. In either case, after entering the `synchronized` block, if the primary meta-object has changed already, the block is left and entered again, until it holds a lock on the current primary meta-object or class object.

The reconfiguration will take place while the lock is held, so that no other thread can start a reconfiguration on the same object. First, if the primary meta-object is currently null, the class meta-configuration and its superclasses are presented an InstanceReconfigure message; otherwise, the current meta-object is presented a reconfigure request.

Whatever the method, it will end up returning the candidate primary meta-object. Before establishing this candidate as the new primary meta-object, **Guaraná** will test whether the primary meta-object changed. This is possible because the InstanceReconfigure message or the reconfigure request may have caused some meta-object to decide to reconfigure the same object. If the meta-object is found to have changed, the candidate meta-object is simply discarded. Otherwise, it will be initialized with a still invalid operation factory.

If, during the initialization, the primary meta-object reference changes, the candidate meta-object is just requested to release the object and the reconfiguration terminates. Otherwise, the candidate meta-object is finally installed as the new primary meta-object, and the previous meta-object is told to release the object.

At the moment the primary meta-object is modified, the operation factory presented to the previous meta-object and any other operation factories based on it are immediately invalidated. This ensures that meta-objects removed from a meta-configuration cannot create new operations.

However, an evil meta-object might have already created an operation for later (ab)use before it was removed from a meta-configuration. It will be able to request **Guaraná** to perform it. However, even operations created from the meta-level are subject to interception, so the new primary meta-object will be requested to handle the operation, and it may refuse to let the operation reach the object.

But the previous meta-object might have foreseen the new meta-object's refusal, and it could have already requested **Guaraná** to perform the operation, and it could be delaying the execution of the operation by not returning from method handle.

In order to prevent this kind of misbehavior, before delivering any operation to a base-level object, method perform will check whether the primary meta-object is still the same that was requested to handle the operation. If it has changed, and the original meta-object requested the result of the operation, it is presented a `null thrown` result. After that, the new primary meta-object is requested to handle the operation. This process repeats until the meta-configuration becomes stable, that is, the primary meta-object remains unchanged while it handles the operation.

After the operation is delivered to the base-level object, the result is presented (if requested) to the meta-object that was requested to handle it, even if the primary meta-object has already changed, and the old one will not be able to do much with the result.

5 Conclusion

By studying this tutorial, you should have learned how to code meta-objects so as to monitor and extend the behavior of base-level objects. You should have understood the protocol for handling operations and results, the protocol for dynamic reconfiguration. You may also have seen some utility for the message broadcasting mechanism, as well as the meta-configuration propagation protocol.

You may have gained some insight on how composers are supposed to behave, and how they can prevent untrustworthy meta-objects from entering the meta-configuration of objects they control, or how to let them in, albeit preventing them from creating unwanted operations or ignoring their results.

Despite its length, this tutorial does not present a single whole picture; instead, it shows several small examples that we hope are enough for one to be able to start using **Guaraná**, gaining experience with it and possibly mastering it with the advanced discussions.

A Obtaining Guaraná

Additional information about **Guaraná** can be obtained in the Home Page of **Guaraná**, at the URL `http://www.dcc.unicamp.br/~oliva/guarana/`. The source code of its implementation atop of the *Kaffe OpenVM*, on-line documentation and full papers are available for download. **Guaraná** is *Free Software*, released under the GNU General Public License, but its specifications are open, so non-free clean-room implementations are possible.

B Acknowledgments

This work is partially supported by FAPESP (*Fundação de Amparo à Pesquisa do Estado de São Paulo*), grants 95/2091-3 for Alexandre Oliva and 96/1532-9 for LSD-IC-UNICAMP (*Laboratório de Sistemas Distribuídos, Instituto de Computação, Universidade Estadual de Campinas*).

References

[1] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Weslley, 1996.

[2] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Java Series. Addison–Wesley, September 1996. Version 1.0.

[3] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Java Series. Addison–Wesley, January 1997.

[4] Alexandre Oliva, Islene Calciolari Garcia, and Luiz Eduardo Buzato. The reflexive architecture of Guaraná. Technical Report IC-98-14, Instituto de Computação, Universidade Estadual de Campinas, April 1998.

[5] Sun Microsystems Computer Corporation, Mountain View, CA, USA. *Java API Documentation*, December 1996. Version 1.1.