Reflection-based implementation of Java extensions: the double-dispatch use-case

Rémi Forax, Étienne Duris, Gilles Roussel

JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 4, No. 10, 2005

Reflection-based implementation of Java extensions: the double-dispatch use-case

Rémi Forax, Institut Gaspard–Monge, Université de Marne-la-Vallée, France
Etienne Duris, Institut Gaspard–Monge, Université de Marne-la-Vallée, France
Gilles Roussel, Institut Gaspard–Monge, Université de Marne-la-Vallée, France

Reflection-based libraries may be used to extend the expressive power of Java without modifying the language nor the virtual machine. In this paper, we present the advantages of this approach together with general guidelines allowing such implementations to be practicable. We show how these principles have been applied to develop an efficient and general double-dispatch solution for Java, and we give the details of our implementation.

1 INTRODUCTION

Java programmers sometimes realize that their programming language lacks some feature. For instance, they could wish to have a double-dispatch mechanism to ease implementations dealing with tree-like structures or binary methods [4, 17]. This feature, which is a special case of multi-methods [11, 21, 8, 3, 22, 23, 10, 12, 1, 27, 2, 14], allows a method implementation to be chosen with respect to the dynamic type (late-binding) of both the receiver and a single argument. It can be achieved programmatically through the visitor design pattern [19, 15]. However, providing it at language level improves genericity and reusability, easing the design, maintenance and evolution of applications [25].

Several techniques could be used to implement such a Java extension. The language could be modified, producing standard Java bytecode, e.g., introducing new keywords [3, 10]. The Java Virtual Machine (JVM) could also be modified, either to accept an extended bytecode or to modify its standard semantics [12]. An alternative solution is to provide such an extension as a pure Java library, using the Java Core Reflection mechanism and bytecode generation, leaving the language and the virtual machine unchanged [13, 17]. This remark could be generalized to other Java extensions where reflection-based approaches are applicable [28, 20, 26].

This paper aims to show the worth of pure Java solutions which are the most flexible ones, although the least employed for efficiency reasons. First, section 2 argues their design advantages compared to environment-intrusive ones. In section 3, general implementation-design guidelines are given to make them practicable. Based

on related works, section 4 shows how we apply these principles to provide an efficient double-dispatch implementation, the Sprintabout. Its implementation details are discussed in section 5.

2 DESIGN CHOICES

The most classical approach to implement Java extensions is to modify the language syntax and to provide the corresponding translator, pre-processor or compiler. This approach has the advantage of being performed at compile time, allowing many costly computations to be performed before execution. In particular, it allows the compiler to perform some static type-checking that is known to enhance software safety. It has the corresponding drawback: it can only use compile-time information. Indeed, in the context of Java's dynamic class loading, some essential information such as type hierarchy may only be available at runtime. Furthermore, the resulting dedicated language is difficult to maintain, particularly when the underlying language evolves.

A second approach, that may be complementary to the extension of the syntax when runtime information is required, is to modify the Java virtual machine or its semantics. It has the advantage of providing precise runtime information on the executing application with a minimum overhead. However, it requires tricky knowledge of the internal of a specific virtual machine. Furthermore, applications implemented using such a modified virtual machine require a specific execution environment which is difficult to maintain and that may not be compatible with existing applications. Thus, this approach yields dedicated solutions whose portability is seriously restrained.

An alternative approach, already chosen to provide Java with runtime type information [28], multi-dispatch [13, 17] or aspect-oriented features [26], is to use reflection to access JVM's runtime internal structures. Like the second approach, it benefits from dynamic information and, as a pure standard Java solution, it offers optimal portability. Even if it could led to runtime type exception (just as type casts), it has the advantage of being simple and directly accessible through the Java Core Reflection API. Contrarily to previous approaches, it allows several specific features to be used or combined, on demand, inside a single application. Moreover, being provided as a simple Java extension library, a new feature deployment only consists in adding a single JAR file to the "classpath" of a standard Java environment. Unfortunately, reflection-based implementations are usually considered to be inefficient in space and time.

All of these approaches present advantages and limitations, but this paper deliberately focuses on enhancements of reflection-based techniques.

3 USING REFLECTION

This section points out problems of reflection-based implementations and gives guidelines to make them practicable.

First, from the essence of the Java Core Reflection API, any reflection-based implementation implies significant costs in space and time. Indeed, the reification of JVM internal objects induces some performance overhead compared with direct accesses available inside the JVM. Beyond object creations, this induces additional indirections. For instance, the reflective invocation of a simple method without argument is known to be slower than the corresponding classical call [6, 7]. Tests performed on a 2.4GHz Pentium 4 with 512Gb of RAM using SUN jdk1.5.04 under Linux (two firsts raws of Figure 1) show that reflective invocation is about 50 times slower than classical one, and even worse in server mode[1].

Next, the genericity of reflective methods implies the construction of several intermediate objects that impacts on performance. In particular, a single call to the `invoke()` method usually requires the creation of an `Object` array to be filled with arguments and, when dealing with primitive types, boxing/unboxing is also necessary. Indeed, for any primitive-type argument or return value (`int`, `double`...), its reflective manipulation must be done through the corresponding wrapper type (`Integer`, `Double`...). Thus, to abstract a primitive value onto the reflection level, its wrapper object has to be constructed (boxing). Moreover, return values have to be cast to conform to the return-type of the method and sometimes, unwrapping primitive types (unboxing) is also necessary. These remarks still hold with jdk 1.5 even if boxing/unboxing and array creation are now hidden to the programmer. Figure 1 shows that performance of method calls with no argument and with one `int` argument are comparable, even for reflective invocations. However, this is only true if object and array creations are not taken into account[2].

Finally, since Java does not allow modifications (extensions) to its internal structures through reflection, some data structures have to be duplicated outside the JVM, in the application. For instance, in order to associate information with classes, it is impossible to directly add a field to the class `Class` and it requires to rely on external structures such as hashtables. Thus, using reflection may lead to important performance penalties in terms of time and space if implementation is carried out without caution.

We argue that general strategies, already used by other frameworks [28, 13, 17], allow reflection-based implementations of language extensions to be improved. Our goal is not to improve Java reflective calls [6, 7] in general but rather to give some guidelines that allow to enhance the design and implementation of reflection-based implementation in special contexts such as language extensions.

[1] Server mode corresponds to a special version of the Sun Hotspot™ virtual machine that performs aggressive optimizations [24].

[2] Figure 1 details subcases of reflective call with one argument, that measure distinct invocation times taking or not taking into account array and `Integer` object creations.

# int args	Invocation type	Object creations	Client mode	Server mode
Zero	normal		3373 ms	120 ms
	reflective		165810 ms	23295 ms
	synchronized		18079 ms	13602 ms
	RFX		21089 ms	118 ms
One	classical		3421 ms	120 ms
	reflective	no	171274 ms	24135 ms
		array	207749 ms	49670 ms
		Integer	405753 ms	61753 ms
		Integer and array	435332 ms	83736 ms
	RFX		18319 ms	118 ms

Figure 1: Performance of one hundred million method calls

The first guideline to follow is to clearly identify two stages in the implementation: the creation time, when objects with reflective purpose are created, and the invocation time, when such objects are actually used to perform multiple tasks. To reduce time overhead, as many computations as possible have to be transfered from invocation time (just in time) to creation time. Indeed, in Java, durations of many computations remain negligible compared to class loading (disk access, class verification) and thus may not be perceived by users. Nevertheless, these pre-computations imply some states to be stored in order to be available just in time. This strategy transfers most computations from invocation time to creation time but may produce large data structures. Thus, programmers have to find the right balance between invocation time, creation time and space. The choice of algorithms and data structures used during these two phases is essential to make reflective implementation practicable.

The second guideline is to reduce space overhead, sharing as much data as possible. This can be achieved through specific algorithm implementations. However, this sharing should also conform to classical object-oriented design principles. Indeed, it is usually possible to share data between objects of the same class (e.g. through static fields), between classes if they are related by inheritance, or between threads. These data structures should support incremental modifications to ensure that creation-time information are reusable when some new information is discovered at invocation time. Developers must also take into account multi-threading since this worry is usually contradictory with time and space performance. Indeed, to maintain data coherence, synchronization is usually required and induces extra duration for method calls. More precisely, a method call is slower if synchronized (see Figure 1), even without delay induced by mutual exclusion. One way to avoid synchronization is to relax coherence and duplicate data but this leads to space overhead.

The third guideline is to try to minimize the use of expensive reflective mechanisms for the invocation-time part[3] of the implementation. The first objective is to eliminate as many reified JVM's internal objects as possible, in order to save space and to minimize indirections. Another goal is to minimize generic reflective calls, replacing them with dedicated calls in order to avoid intermediate object creations, casts and boxing/unboxing. This can be achieved by generating, at creation time, dedicated bytecode [17] according to a precise method signature specified by the developer instead of relying on the generic signature of the reflective API. More precisely, many JVM's internal reified objects like `Method` or `Class` do not necessarily require to refer to the internal state of the JVM. Thus they may be partially simulated by user-defined objects which can be more efficient than reflective API ones. In particular, a `Method` object that refer to a public[4] method may be efficiently simulated by generating and instantiating a class implementing the corresponding method call. Moreover, in order to reduce intermediate object and array creations required by the generic `invoke()` method call, we propose to let the developer specify precisely the method signature as an abstract method in an abstract class. Code generation is then used at creation time to implement this abstract method by a call to the method with a corresponding signature on a specific receiver.

In order to illustrate this approach and its worth, we developed a class `RFX`. This class provides reflective-like method invocations that are more efficient than those of the generic reflective API. Suppose that you have to call a method with name `m` and two `int` parameters on a receiver whose class is unknown at compile time. With `RFX`, instead of using the generic class `Method`, you declare an abstract class[5] containing at least an abstract method `m()` with a first parameter of type `Object`, corresponding to the receiver, and two other `int` parameters, as follows:

```
abstract class MethodM {
  public abstract int m(Object receiver, int arg1, int arg2);
}
```

According to the receiver's class `c` (dynamically provided in the following example by `retreiveClass()`), a call to `RFX.bind()` generates[6] the bytecode of a concrete subclass of `MethodM` and returns an instance of it, `mm`. Then, given an object `o` of class `c`, invocation of `o.m(1,2)` is performed through the call `mm.m(o,1,2)`.

```
Class<?> c = retreiveClass();
MethodM mm = RFX.bind(MethodM.class,c);
```

[3] They may also be minimized for creation-time part but this is less crucial.

[4] This is not the case for other method kinds, like private ones, that require access to the JVM's internal in order to pass security checks.

[5] An interface may also be used but with some performance penalty.

[6] The class is generated only once thanks to the classloader's cache.

```
Object o = c.newInstance();
int i = mm.m(o,1,2);
```

The generated subclass implements the abstract method `m()` of `MethodM` by a direct call to the method `m()` on the receiver (see appendix A). Thanks to the object `mm`, the invocation of the method `m` is efficient, avoiding object or array creation. Figure 1 shows that the `RFX` method call is only 7 times slower than a classical method and 8 times faster than a standard reflective method call. Performance is even better in server mode where a `RFX` call is as fast as a classical one.

4 VISITORS: WALK, RUN AND SPRINT

We now deal with several solutions proposed to provide Java with double-dispatch. We first recall the interest of this mechanism and we argue the worth of offering it through a simple specific feature, freeing the programmer from its implementation details. Next, we present several existing frameworks providing improvements in this direction. Then, we detail the Sprintabout, a new framework allowing double-dispatch we have designed and implemented with respect to previous guidelines for reflection-based extensions. Finally, we present experimental results that show its efficiency compared with other approaches, including ad-hoc hard-to-maintain ones.

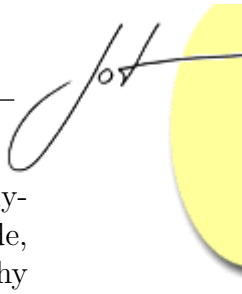
Motivation

The classical late-binding mechanism provided by Java allows a method implementation to be chosen dynamically, according to the type of the receiving object. This simplifies code reusability since, when new classes implementing the method are introduced, programmers do not have to implement complex selection mechanism nor modify the code.

Nevertheless, adding a new functionality for each class of an existing hierarchy is less straightforward. For instance, as proposed in [17], consider the following type hierarchy.

```
interface A { }
class A0 implements A { }
class A1 implements A { }
class A2 implements A { }
```

Imagine that, given an `array` of type `A[]`, we want to compute $\sum_{a \in \texttt{array}} value(a)$ where $value(a) = i$ if a is of type `A`i. In other words, we want a method `int sum(A... array)` such that, for instance, `sum(new A0(),new A1(),new A2())` returns `3` (`0+1+2`).



A first solution is to test, with the Java `instanceof` statement, the actual dynamic type of each array element that is statically typed `A`. The corresponding code, based on successive and nested tests could become very intricate if the type hierarchy concerns interfaces with multiple inheritance. Anyway, it is tedious to implement, error prone and hard to maintain if the hierarchy or the semantics evolves.

A more object-oriented, but naive and dedicated solution, would be to add in each class a method `int value()` that returns the right number. To be simply used with an array of type `A[]`, this not only requires modification of all classes, but also of the interface. Nevertheless, nowadays, most applications are built using classes or objects provided by third-party libraries (Java standard API, or off-the-shelf libraries). Even if their sources are available, they usually should not be modified. Thus, programmers cannot simply add methods to such classes[7].

Two classical object-oriented techniques allow this difficulty to be worked around: class inheritance and delegation. However, on the one hand, inheritance is not always applicable: the case of final classes apart, some libraries use factory methods to create objects and prevent constructor access. On the other hand, if delegation could always be used by defining new wrapping classes for each data type, this leads to data structure duplications and to burdensome hard-to-maintain code.

A better design is to specify a set of methods `int value (T t)` in a separate class, one for each "interesting" type (e.g. `A0`, `A1` and `A2` in our example), and to provide a general mechanism allowing the right one to be chosen with respect to the dynamic type of `t`. This late-binding mechanism on the argument of a method call, known as double-dispatch, would actually allow behaviors (methods) to be specified outside classes, separating algorithms from data. Unfortunately, this mechanism is not available in Java.

To face this lack programmatically, the programmer could design his code following the visitor design pattern [19, 15]. It simulates late-binding on the argument of a visitor method `visit(a)` using the classical late-binding on an `a.accept (Visitor v)` method provided by data classes. If this technically solves the problem of adding new functionality to a given class without modifying it, this design pattern has several limitations. First, it requires data classes to be engineered to accept visitors, i.e., to implement an `accept()` method. Second, the behavior is strongly tied to the existing type hierarchy: it must implement a special `Visitor` interface which, for full generality, must include one `visit()` method for each accessible type. This has two drawbacks. The programmer must implement every `visit()` method even if some of them could have been captured by a widening reference conversion of the argument type. Moreover, any extension of the type hierarchy requires the `Visitor` interface to be extended; then, all existing visitor implementations have to be modified since they must implement the new visitor interface to be accepted by existing classes.

[7] This may be done through Aspect Oriented Programming features without class modification [18].

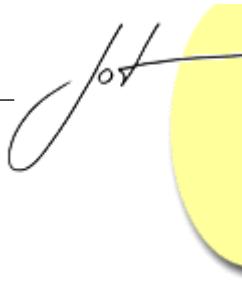
Double-dispatch implementations

Several research works proposed to implement the double-dispatch as a reflection-based extension. Based on the Walkabout of Palsberg and Barry Jay [25], Grothoff proposed the Runabout [17] that allows a new functionality to be specified over data without modifying their class nor requiring them to implement `accept(Visitor v)` methods. The programmer specifies behaviors through `visit()` methods (one for each interesting parameter type) in a class that extends the provided general class `Runabout`. This class contains a method `visitAppropriate(Object o)` that is able to dispatch the invocation to the right method `visit()` with respect to its (dynamic) argument type.

```
public class SumRunabout extends Runabout {
  int sum=0;
  public void visit(A0 a) { sum+=0; }
  public void visit(A1 a) { sum+=1; }
  public void visit(A2 a) { sum+=2; }
  public static int sum(A... array) {
    SumRunabout v=new SumRunabout();
    for(A a:array)
      v.visitAppropriate(a);
    return v.sum;
  }
}
```

Compared with the Walkabout, the Runabout minimizes the use of the Java Core Reflection API to improve its performance. When an instance of the `Runabout` is created, reflection is used to find all available `visit()` methods. For each parameter type, a class is generated and loaded on-the-fly. It contains a method dedicated to invoke the `visit()` method corresponding to this parameter type. An instance of each of these classes is created and stored in a *dynamic code map*. Thus, at invocation time, the `visitAppropriate()` method simply uses its argument class and the dynamic code map to dispatch the invocation to the appropriate code. This technique avoids the use of the generic `invoke()` method, known to be time consuming.

At the same time this work was performed, Forax *et al.* developed a reflective framework to provide full multi-polymorphism in Java [13], i.e. late-binding on all method arguments. With the Java Multi-Method Framework (JMMF), the programmer constructs a `MultiMethod` object that represents all methods with a given name and a given number of parameters that are accessible in a given class. To perform a multi-polymorphic invocation with an array of arguments, it suffices to call the `invoke()` method of the multi-method object. JMMF dispatches the call to the most specific method, according to dynamic types of arguments.



```
public class SumJMMF {
  public int value(A0 a) { return 0; }
  public int value(A1 a) { return 1; }
  public int value(A2 a) { return 2; }
  public static int sum(A... array) throws Exception {
    SumJMMF v=new SumJMMF();
    int sum=0;
    MultiMethod mm=MultiMethod.create(SumJMMF.class, "value", 1);
    for(A a:array)
      sum+=mm.invoke(v,a);
    return sum;
  }
}
```

The aim of JMMF is more general than the simple double-dispatch and more generic than the Runabout. It does not require inheritance from any class, allowing another inheritance. Furthermore, it does not constraint the name of the method to be `visit` and allows several distinct multi-methods to be defined in a same class. Nevertheless, even if it computes data-structure at creation time in order to minimize invocation-time overhead (and even allows them to be shared between multi-methods), its implementation efficiency suffers from relying on the Java Core Reflection API.

The Sprintabout

Inspired by previous related works, we have designed and implemented the Sprintabout, following the guidelines we drew in section 3. The code below illustrates how it can be used on our running example.

```
public abstract class SumSprintabout {
  public int value(A0 a) { return 0; }
  public int value(A1 a) { return 1; }
  public int value(A2 a) { return 2; }
  public abstract int valueAppropriate(A a);
  public static int sum(A... array) {
    SumSprintabout v=
      VisitorGenerator.createVisitor(SumSprintabout.class);
    int sum=0;
    for(A a:array) sum+=v.valueAppropriate(a);
    return sum;
  }
}
```

In this example, the abstract method whose name ends with `Appropriate` is used to identify the methods to be considered for multi-dispatch, i.e. `value()`. The `createVisitor()` method collects these methods through reflection on its argument class. It returns a visitor object, standing for all methods `value()` belonging to the class and in charge of dispatching invocations according to the dynamic type of the argument provided to `valueAppropriate()` calls.

The Sprintabout implementation respects guidelines outlined in section 3. At creation time, a hashtable is created to be used at invocation time. Initially filled by the method `createVisitor()`, it associates types with integers that index the method to call for this type of argument. It could be completed dynamically, when new argument types are discovered, in order to cache the method identifier resolved for this type. As a static field, this hashtable is shared between all instances of the same class.

The implementation bytecode for the abstract "appropriate" method is generated at creation time. At invocation time, when this concrete generated method is called, it performs a lookup in the hashtable to retrieve the method index associated with the type of its argument; if the type is not found, the method's index for the closest type must be determined and their association is cached into the hashtable. Then, the dispatch is performed using this index in a simple `switch/case` statement to call the right method. The source code obtained by decompiling the bytecode generated for the method `valueAppropriate()` of our example is given in Appendix B. The classloader caching mechanism ensures that the bytecode generation is only performed once. This implementation technique allows Sprintabout to become completely independent of reflective method call at invocation time. Compared with the Runabout that reifies one dedicated code object (generated class) for each `visit()` method, the Sprintabout only builds a single visitor class for all methods. This minimizes memory footprint and delegation overhead.

Moreover, compared to other approaches, the specification of the abstract "appropriate" method allows the user to give a signature as precise as possible. Indeed, this method does not have to be generic since it is specifically generated for each particular visitor class. This feature allows the compiler to perform some static type-checking on argument types, e.g. it is possible to constraint argument type. For instance, in previous example, the declaration of `valueAppropriate()` imposes the argument to implement the interface `A`. Finally, the "appropriate" method signature may return values of any type, including primitive types, without requiring cast or boxing/unboxing. As JMMF and contrarily to the Runabout, the Sprintabout does not require the implementation class to inherit from some special class (i.e. Runabout) and thus allows any other class extension.

Sprintabout implementation relies on jdk 1.5 generics, on JSR133 for concurrency and uses ASM [5] for code generation. It is freely available on the Web [8] and its implementation is detailed in section 5.

[8] http://igm.univ-mlv.fr/~forax/works/sprintabout/

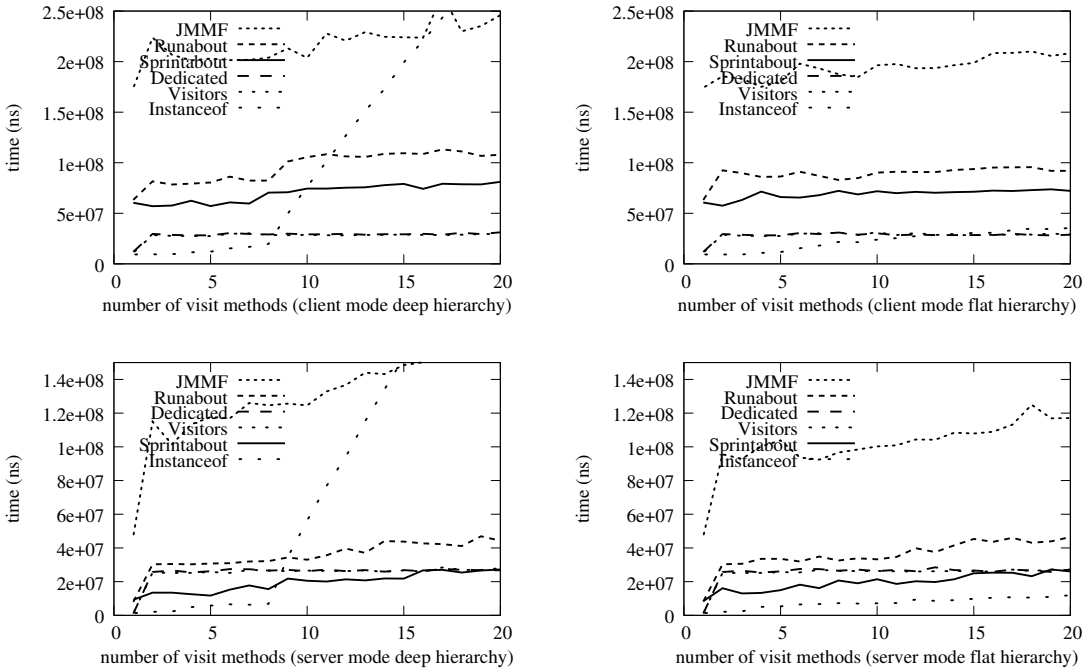


Figure 2: Dispatch time in client and server mode, with deep and flat type hierarchy

Experimental results

In this section, we present an evaluation of invocation performance of Sprintabout compared with other double-dispatch solutions. We compare it with six techniques: *Instanceof*, implementing filtering using `instanceof` tests, *Dedicated*, where a dedicated method is added inside each data class, *Visitors*, that implements the visitor design pattern, *Runabout*, implementing double-dispatch using inheritance of the class `Runabout`, *JMMF*, implementing double-dispatch as a special case of multi-dispatch provided by the library JMMF and finally *Sprintabout*.

Our tests present the evolution of the time required by one million method invocations when the number of methods increases. In these tests, we use two distinct type hierarchies to provide the argument values of method calls. In the *deep* hierarchy, all types are related by inheritance whereas in the *flat* one, they are unrelated. Figure 2 presents results of these tests in both standard "client" execution environment and with a virtual machine started in server mode. All the tests of this section have been performed on a 2.4GHz Pentium 4 with 512Gb of RAM using SUN jdk1.5.04 under Linux, but results on the same architecture using Windows XP are comparable.

These tests show that Visitors and Dedicated techniques (whose graphs overlap) have the best performance on average. Instanceof technique has the best performance for the flat hierarchy. However, for the deep one, its performance strongly decreases when the number of visit increases; it seems that Instanceof implementation relies on some caching techniques that fails when hierarchy becomes to large.

Considering the reflection-based implementations, we see that the Spintabout is about three times faster than the generic JMMF and one fourth faster than the Runabout, which also uses a code generation technique. This shows that avoiding the use of Java Core Reflection mechanism allows Runabout and Sprintabout invocation time perfomance to be improved. This also proves that the Sprintabout, which eliminates JVM's internal object reification and genericity of method calls, has better performance than Runabout.

Tests performed in server mode give comparable results. Nevertheless, Sprintabout has now better performance than Visitor and Dedicated techniques. This behavior comes from the execution performance of optimized code [9] that is better for switch statements produced by Sprintabout than for virtual calls used by other techniques.

5 SPRINTABOUT IMPLEMENTATION

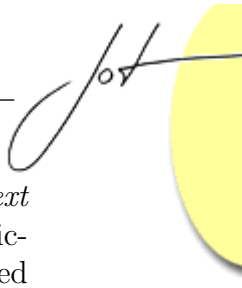
This section describes more precisely some Sprintabout implementation details. Our aim is not to give burdensome technical details, but rather to point out some problems and the way they are addressed. First, we briefly describe the process overview, and its two main steps: creation time and invocation time. Next, we detail particular contexts, determined at creation time, where specific algorithms may be used at invocation time. These algorithms, given an argument type at invocation time, select the right method to invoke. Finally, we describe our specific implementation of hashtable that, given our particular context, deals efficiently with concurrency.

Overview and general principle

The Sprintabout implementation respects guidelines outlined in section 3. At creation time, all declared parameter types of concerned methods are considered in order to prepare the invocation time process.

First of all, at creation time, the method `createVisitor()` of the `VisitorGenerator` is called with a class as argument. In this class, given a method *`method`*`Appropriate()`, a reflective retrieval of all methods named *`method`*`()` with one parameter is performed. Parameter types of these methods are collected and associated with indexes[9] (non-negative integers). If `n` is the number of parameter types, a *precision matrix* `PM` of size `n` $\times$ `n` is constructed. In this binary matrix, `PM[i][j] = 1` if and only if `i` is the index of a type `Ti` that is *more precise* than the type `Tj` indexed by `j`. The precision relation defined by this matrix, noted `Ti` $\leq$ `Tj`, intuitively means that a method declaring a parameter of type `Tj` accepts arguments of type `Ti`. This precision relation is usually provided by the method `isAssignableFrom()` of `java.lang.Class` or, for special cases such as primitive types or arrays, by specific treatments according to the Java Language Specification, chapter 5 [16]. This

[9] An index identifies both a type and the method that declares a parameter of this type.



matrix is widely used at creation time, in order to determine the *definition context* of the set of methods considered. The definition context depends on the particularities of the parameter types hierarchy and induces the choice of a dedicated algorithm that selects the right method to call at invocation-time. In addition to the choice of the algorithm, the creation-time step builds the main data-structure used at invocation time: a hashtable whose keys are types and values are indexes. This hashtable is initially filled at creation time with the declared parameter types and their indexes.

At invocation time, given the type `Arg` of an actual argument `arg` of a call to *`method`*`Appropriate()`, the invocation-time algorithm uses the hashtable to find the index identifying the method to invoke. If the type `Arg` is found in the hashtable, this directly yields the index. Otherwise, the algorithm has to inspect all less precise types (usually supertypes) of `Arg` in order to find the *closest type* registered in the hashtable; this process may require the use of the precision matrix. Sometimes, several less precise types are not comparable and it is not possible to determine a closest type. In this case, a runtime exception is thrown. The result of this process is cached, by inserting in the hashtable the association between the argument type `Arg` and the index found.

The case of `null` argument

Since this process uses the type of the argument, the case where the argument is `null` must be treated separately. In such a situation, the method to be called is the *most precise* one, if it exists, i.e., the method whose parameter type is more precise than every other declared parameter types. The existence of such a most precise method is determined at creation time using the precision matrix. Indeed, if a type of index i is the most precise then $\Sigma_{j \in 1..n} \texttt{PM}[i][j] = n$. If a most precise method exists, its index is associated in the hashtable with an unused type (e.g. `void.class`). At invocation time, when the value `null` is encountered, the algorithm artificially looks for this type in the hashtable. If a most precise method does not exist, then a predefined index (e.g., `-1`) is associated in the hashtable with `void.class` and, at invocation time, when this index is found, a runtime exception is thrown.

Definition contexts and algorithms

The particular treatment of a `null` argument does not depend on the definition context. In the same way, whatever is the definition context, if the argument is not `null`, its type `Arg` is looked for in the hashtable. If an index is found, the corresponding method is invoked, otherwise, invocation-time process depends on the definition context determined at creation time. This section describes these different contexts, how they are determined at creation time, and their corresponding invocation-time algorithms.

Only classes in parameter types

The simplest definition context is when all declared parameter types are classes like in Figure 3 (a). Since Java does not allow multiple inheritance of classes, looking step by step in the hashtable for direct superclass of any argument type yields a single index, if one exists. At creation time, in order to determine this definition context, we simply use the method `isInterface()` of `java.lang.Class` over each declared parameter type. This particular context allows us to discard the precision matrix. The corresponding invocation-time algorithm simply retrieves the superclass of `Arg`, using the method `getSuperclass()` of `java.lang.Class`, and looks for it in the hashtable. This is done step by step until either it finds an index (the method to invoke) or reaches the root class `Object` (in this case a runtime exception is thrown). The association between `Arg` and its index is cached in the hashtable for further use.

Totally ordered parameter types

Another definition context allows the precision matrix to be discarded. It occurs when all declared parameter types are totally ordered by the precision relation. In other words, the hierarchy is linear: each declared parameter type is either more precise or less precise than any other declared parameter types like in Figure 3 (b). This context is determined from the precision matrix verifying that, for all $i \in 1..n$:

$$\Sigma_{j\in 1..n}\texttt{PM}[i][j] + \Sigma_{j\in 1..n}\texttt{PM}[j][i] = n + 1$$

In this case, several declared parameter types less precise than an argument type may exist but, since they are totally ordered, one of them is the most precise. To ease to selection of the most precise method and to avoid the use of the precision matrix at invocation time, the indexes of declared parameter types are reordered in the precision order[10]. Then, to retrieve the most precise type it suffices to retain the type with the greatest index. This renumbering is realized at creation time, and the algorithm used at invocation time only uses the hashtable. More precisely, it traverses recursively all types less precise than `Arg` (supertypes) in order to find, if it exists, the type with greatest index. During this traversal through the type hierarchy, for each type found in the cache the branch leading to less precise types is pruned since they will never yield a greater index. If a greatest index is found, the result is cached in the hashtable and the corresponding method is invoked; otherwise, a runtime exception is thrown.

Precision relation and *bad fork*

The difference between the last two definition contexts relies on the notion of *fork* between parameter types. We say that there is a *fork* in the hierarchy of parameter

[10] In this particular case, a well-ordered numbering should respect, $\forall i, k \in 1..n,\ i \leq k \Leftrightarrow \Sigma_{j\in 1..n}\texttt{PM}[i][j] \leq \Sigma_{j\in 1..n}\texttt{PM}[k][j]$

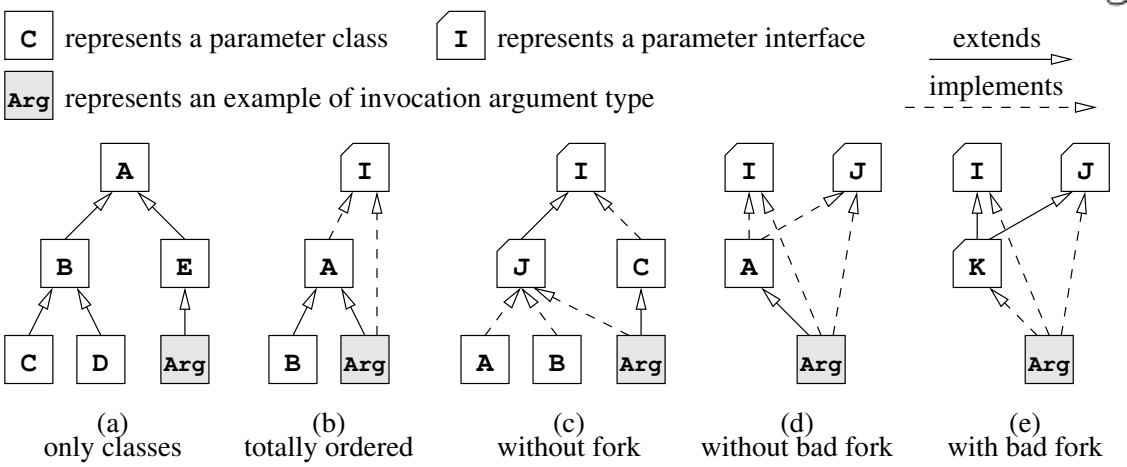


Figure 3: Examples of parameter type hierarchies for different definition contexts

types if a given type is more precise than at least two other parameter types that are not comparable, as in examples (d) and (e) of Figure 3.

Generally, looking for less precise types of a given argument type leads to several parameter types and the most precise one must be determined, if it exists. Suppose that two incomparable parameter types less precise than the argument type are found. If there is no fork, then it is possible to conclude that no most precise method exists, like in Figure 3 (c). Otherwise, if a fork exists, a most precise method may exist. Indeed, a third parameter type may exist that is more precise than the two incomparable ones, and thus, further comparisons are required.

In order to minimize comparisons, we propose to retrieve types the most precise first. This ensures that when two types are found in this order then either one is more precise than the other, or a most precise does not exist. This property can sometimes be ensured by the traversal of the type hierarchy. More precisely, if one of the three types involved in a fork is a class, then the most precise type is necessarily a class, as in Figure 3 (d); a traversal that performs `getSuperclass()` before `getInterfaces()` will encounter this most precise class first. Otherwise, the fork, called "bad fork", only involves interfaces, as in Figure 3 (e), and the hierarchy traversal cannot guarantee retrieval of types in order. In this case, parameter type indexes need to be renumbered at creation time, like in the totally ordered context, to allow the invocation-time algorithm to test them in order.

To be able to characterize the definition context at creation time, after collecting all declared parameter types, we look for an interface definition that explicitly extends two other parameter interfaces. If such a bad fork is found, the definition context is said to be "with bad fork", else "without bad fork".

In a definition context without bad fork, at invocation time, given an argument type `Arg` not found in the cache, the algorithm begins like for totally ordered types. It traverses the hierarchy of less precise types (supertypes) of `Arg`, looking for classes prior to interfaces, and if none or only one index is found, it respectively throws an

exception or invoke the corresponding method. Otherwise, as soon as two distinct indexes are found, their types must be compared. If they are comparable, the most precise is retained and the process continues. If they are not comparable, the context allows us to stop the traversal and to conclude that there is no most precise method and an exception is thrown.

In a context definition with a bad fork, at invocation time, the algorithm starts similarly, but must collect all indexes of types less precise than `Arg` before performing any comparison. Then, while at least two indexes remain, the two largest `i` and `j` are considered. If the corresponding types are not comparable, the type renumbering performed at creation time ensures that there is no more precise type, and thus no most precise method: a runtime exception is thrown. Otherwise, if one of the corresponding types is more precise than the other, its index is retained and the comparison process continues with remaining indexes. When the process succeeds with a single index, the corresponding method is invoked.

In both contexts, successful index retrievals are cached.

Then, in better cases, complexity of method retrieval at invocation time has the same complexity as a dynamically-sized hashtable access (roughly $O(1)$). The worst case requires to traverse the hierarchy of the argument type in order to retrieve all its supertypes and then to identify the most precise one. In the worst case, this process is linear ($O(n)$) in the number, n, of types in the hierarchy. Linearity is ensured by the creation-time pre-processing which requires d^2 calls to the method `isAssignableFrom()`, where d is the number of declared parameter types. Complexity of `isAssignableFrom()` is, in the worst case, linear in the size of the type hierarchy. Since complexity of other pre-processing steps are less than $O(d^2)$, the general worst-case complexity of creation-time process is $O(d^2 \times n)$.

Data-structure implementation and concurrency

The cache used in our implementation allows to retrieve, given a class object, the index (integer) corresponding to a visit method. This cache is implemented by a dedicated hashtable. Indeed, compared to the generic implementation available in the Java API, our implementation can avoid boxing/unboxing of the index value into `Integer`. It is also possible to use reference equality `==` instead of `equals()` on `Class` objects since uniqueness is ensured by the virtual machine. Another optimization concerns concurrency. Compared with `java.util.concurrent.ConcurrentHashMap`, our implementation may be simplified since we never need to remove an entry from the cache. The cache only supports `insert()` and `get()` operations. To improve performance of the cache, we have chosen to completely relax the synchronization on the `get()` operation. This is possible since all hashtable modifications are atomic. Freshness of values is ensured by `volatile` variable accesses according to the causality of the Java Memory Model (JSR 133). We use mutual exclusion on the `insert()` operation for data coherence, however, we allow multiple instances of a same entry to be present in the hashtable to improve time performance.

6 CONCLUSION

This paper promotes the use of reflection based implementations to extends the Java environment. It proposes several guidelines to make such implementations practicable. First, as many computations as possible have to be performed at creation time to reduce invocation-time overhead. Next, as many data as possible have to be shared to reduce space overhead, taking into account concurrency issues. Finally, code generation may be used to minimize the invocation-time use of reified objects and of the generic methods of the Java Core Reflection API. This latter point is illustrated by a simple class `RFX` that improves reflective method invocations.

To illustrate this approach, we describe a reflection-based implementation of double-dispatch for Java that conforms to these guidelines. Experimental results compare this implementation with other techniques and validate the usability of our approach.

A RFX GENERATED CODE

The following Java class corresponds to the bytecode generated by the `RFX.bind()` method if, in the example of section 3, the class `c` provided by the method `retreiveClass()` has name `Test`.

```
public class MethodM$+$Test$ extends MethodM {
  public MethodM$+$Test$() {
    super();
  }
  public int m(Object target,int arg1,int arg2) {
    return ((Test)target).m(arg1,arg2);
  }
}
```

B SPRINTABOUT GENERATED CODE

The following Java source code has been obtained by decompiling the bytecode generated by Sprintabout for the example of section 4.

```
import fr.umlv.sprintabout.*;
public class SumSprintabout$+$Visitor extends SumSprintabout {
  public int valueAppropriate(A a) {
    return $dispatch$value$1$(VisitorGenerator
```

```
        .getFromMap($map$value$1$0, a), a);
  }
  private int $dispatch$value$1$(int i, A a) {
    switch(i) {
      case 1: return value((A0)a);
      case 2: return value((A1)a);
      case 3: return value((A2)a);
      default:
        throw new RuntimeException("No such method");
    }
  }
  private static final ConcurrentIntMap $map$value$1$0;
  static {
    ConcurrentIntMap concurrentintmap=new ConcurrentIntMap(6);
    concurrentintmap.insert(A0.class, 1);
    concurrentintmap.insert(A1.class, 2);
    concurrentintmap.insert(A2.class, 3);
    $map$value$1$0=concurrentintmap;
  }
}
```

REFERENCES

[1] G. Baumgartner, M. Jansche, and K. Lufer. Half & half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Dept. of Computer and Information Science, Ohio State University, Mar. 2002.

[2] L. Bettini, S. Capecchi, and B. Venneri. Translating double-dispatch into single-dispatch. In *Proc. of WOOD'04*, ENTCS. Elsevier, 2004.

[3] J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods for Java. In *OOPSLA'97*, number 32–10 in SIGPLAN Notices, pages 66–76, Atlanta, Georgia, Oct. 1997. ACM Press.

[4] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[5] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, Grenoble, France, Nov. 2002.

[6] W. Cazzola. SmartMethod: an Efficient Replacement for Method. In *SAC'04*, pages 1305–1309, Nicosia, Cyprus, Mar. 2004. ACM Press.

[7] W. Cazzola. Smartreflection: Efficient introspection in java. *Journal of Object Technology*, 3(11):117–132, Dec. 2004.

[8] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP'92 proceedings*, LNCS, Utrecht, The Netherlands, July 1992. Springer.

[9] C. Click. Java technology performance myths exposed, 2005. http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TS-3268.pdf.

[10] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch. In *OOPSLA'00 proceedings*, ACM SIGPLAN Notices, Minneapolis, USA, Oct. 2000.

[11] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In *ECOOP'87 Proceedings*, LNCS, pages 151–170, Paris, France, June 1987. Springer.

[12] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi-dispatch in the Java Virtual Machine design and implementation. In *COOTS'01 proceedings*, San Antonio, USA, Jan. 2001.

[13] R. Forax, E. Duris, and G. Roussel. Java multi-method framework. In *TOOLS Pacific'00 Proceedings*, Sidney, Australia, Nov. 2000. IEEE Computer.

[14] R. Forax, E. Duris, and G. Roussel. A reflective implementation of java multi-methods. *IEEE Transactions on Software Engineering (TSE)*, 30(12):1055–1071, 2004.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification – Second Edition*. Addison-Wesley, 2000.

[17] C. Grothoff. Walkabout revisited: The runabout. In *ECOOP'03 proceedings*, LNCS, pages 103–125. Springer, 2003.

[18] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proc. of OOPSLA'02*, pages 161–173, Seattle, USA, Nov. 2002. ACM SIGPLAN.

[19] D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings of OOPSLA'86*, pages 347–349, Portland, Oregon, Nov. 1986.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

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

[22] M. Kizub. Kiev language specification. An extension of Java language that inherits Pizza features and provides multi-methods (`http://forestro.com/kiev/`), July 1998.

[23] T. Millstein and C. Chambers. Modular statically typed multimethods. In *ECOOP'99 proceedings*, number 1628 in LNCS, pages 279–303, Lisbon, Portugal, June 1999.

[24] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} server compiler. In *Proc. of JVM-01*, pages 1–12, Monterey, California, USA, Apr. 2001. USENIX Association.

[25] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *COMPSAC'98 proceedings*, pages 9–15. IEEE Computer Society, 1998.

[26] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in java. In *Proceedings of Reflection'01*, number 2192 in LNCS, Kyoto, Japan, Sept. 2001. Springer-Verlag.

[27] J. Smith. Draft proposal for adding multimethods to c++, Sept. 2003. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1529.html`.

[28] M. Viroli and A. Natali. Parametric polymorphism in java: an approach to translation based on reflective features. In *Proceedings of OOPSLA'00*, pages 146 – 165, Minneapolis, Minnesota, United States, 2000.

ABOUT THE AUTHORS

Rémi Forax is Maître de Conférences at University of Marne-la-Vallée since 2003. His main research areas concern the use of reflection and of bytecode generation to enhance the Java programming and executing environment. He can be reached at remi.forax@univ-mlv.fr. See also http://igm.univ-mlv.fr/~forax.

Etienne Duris is Maître de Conférences at University of Marne-la-Vallée since 2000. His research focuses on program transformations and on the use of reflection to extend the expressive power of programming languages. He can be reached at etienne.duris@univ-mlv.fr. See also http://igm.univ-mlv.fr/~duris.

Gilles Roussel is Professor at University of Marne-la-Vallée since 2004. His research works cover program transformations, language processing enhancements, genericity and routing. He can be reached at gilles.roussel@univ-mlv.fr. See also http://igm.univ-mlv.fr/~roussel.