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

Abstract

Reflection-based libraries could sometimes 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. Then, we show how these principles have been applied to implement an efficient and general double-dispatch solution for Java.

1 Introduction

During software development, 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 [2, 8]. This feature, that allows a method implementation to be chosen with respect to the dynamic type (late-binding) of both the receiver and a single argument, can be achieved programmatically through the visitor design pattern [7]. However, providing it at language level improves genericity and reusability, easing the design, the maintenance and the evolution of applications [10].

Several techniques could be used to implement such a Java extension. The language could be modified, producing standard Java bytecode, for instance introducing new keywords [1, 4]. The Java Virtual Machine (JVM) could also be modified, either to accept an extended bytecode or to modify its standard semantics [5]. An alternative solution consists in providing 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 [6, 8]. This problematic could be generalized to other Java extensions where reflection-based approaches are applicable [12, 9, 11].

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.

2 Design choices

The most classical approach to implement Java extensions consists in modifying the language syntax and in providing the corresponding translator or compiler. This approach has the advantage of being static and thus, allows many computations to be performed before execution. It has the corresponding drawback: it can only use information available at compile time. However, in the context of Java's dynamic class loading, some essential information may only be available at runtime. Furthermore, the resulting dedicated language is often difficult to promote (it requires specific knowledges) and to maintain (when the underlying language evolves), even if it produces standard bytecode.

A second approach, which is sometimes complementary when runtime information is required, consists in modifying the JVM or its semantics. It provides precise runtime information on the executing application with a minimum overhead, but its implementation requires tricky knowledge of the internal of a specific virtual machine. Furthermore, applications implemented using such a modified virtual machine require a special execution environment. It yields dedicated solutions that depend on specific execution environments and this seriously constraints their portability.

An alternative approach, chosen to provide Java with runtime type information [12], multi-dispatch [6, 8] or aspect-oriented features [11], consists in using reflection to access JVM's runtime internal structures. Like the second approach, it can benefit from dynamic information and furthermore, since it is a pure standard Java solution, it offers an optimal portability. It has the advantage of being simple and directly accessible through the Java Core Reflection API. Contrarily to previous approaches, this also allows several specific features to be used or combined,

on demand, inside a single application. Moreover, being provided as a simple Java extension library, its deployment only consists in adding a single JAR file in the "classpath" of a standard Java environment. Unfortunately, reflection-based implementations are usually considered to be inefficient in space and time.

3 Reflection-based implementation issues

In this section, we point out general problems that are intrinsic to reflection-based implementations and we give guidelines to make them practicable.

First, from the essence of the Java Core Reflection API, any reflection-based implementation implies a significant cost 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 200 times slower[1] than the corresponding classical call and still 8 times slower if the JVM is started is server mode.

Next, the genericity of reflective methods implies the construction of several intermediate objects. For instance, a single call to the `invoke()` method usually requires the creation of an `Object` array, to be filled with arguments. Moreover, dealing with primitive argument types imposes supplementary costs. Indeed, for any primitive-type argument or return value (`int`, `double`...), its reflective manipulation must be done through the corresponding wrapper type (`Integer`, `Double`...). Then, to abstract a primitive argument value onto the reflection level, its wrapper object has to be constructed (boxing). For return values, in addition to the potential need of unwrapping primitive types (unboxing), any return value has to be cast to conform to the return-type of the method. These creations, manipulations and type casts obviously affect the effectiveness in time and space.

On the other hand, since Java does not allow modifications of its internal structures through reflection, some data structures have to be duplicated outside the JVM, in the application. For instance, if you want to associate information with classes, you cannot add a field to `Class`

[1] Tests performed using j2sdk1.5.0_beta3 for Linux on a Pentium 2.4 Ghz with 512 Mb.

objects but you have 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 [12, 6, 8], allow reflection-based implementations to be improved.

The first guideline to follow consists in clearly identifying 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 their task. 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 and class verification), thus they are not 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, in the worst case, it produces large data structures. Thus, programmers have to find the right balance between invocation time, creation time and space. The choice of the algorithm is essential to make reflective implementation practicable.

The second guideline consists in trying to minimize the use of expensive reflective mechanisms for the invocation-time part of the implementation. The first objective is to eliminate as many reified objects as possible in order to save space and to minimize indirections. Another goal is to minimize generic reflexive calls, by replacing them with dedicated calls in order to avoid intermediate object creations, casts and boxing/unboxing. This could be achieved by generating dedicated bytecode at creation time [8].

The third guideline consists in reducing space overhead, sharing as many 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 insure that creation time information is still usable when some new information is discovered at invocation time. Again, it is primordial to drive algorithmic choices by space concerns. Developers must also take into account multi-threading, but this worry is usually con-

tradictory with time and space performances. Indeed, to maintain data coherence, synchronization is usually required and induces extra duration for method calls. More precisely, a method call is about 10 times slower if synchronized, even without mutual exclusion delay. One way to avoid synchronization consists in relaxing coherence and in duplicating data but this leads to space overhead.

4 About visitors: walk, run and sprint

The rest of this paper deals with some solutions that have been 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 framework providing improvements in this direction. Then, we detail the Sprintabout, a new framework 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.

4.1 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 to modify the code.

Nevertheless, adding a new functionality for each class of an existing hierarchy is less straightforward. For instance, as proposed in [8], 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)`[2] such that, for instance, `sum(new A0(), new A1(), new A2())` returns `3 (0+1+2)`.

A first solution consists in testing, 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, in particular in presence of 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 consist in adding 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 to modify all classes, but also 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). When their sources are available, they usually should not be modified. Thus, programmers cannot simply add methods to such classes.

Two classical techniques allow this difficulty to be worked around: class inheritance and delegation. However, on the one hand, inheritance is not always applicable. Indeed, the obvious case of final classes apart, libraries sometimes use factory methods to create objects, preventing access to their constructors. 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 and hard-to-maintain code.

A better solution consists in specifying 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 in providing 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.

The programmer could design its code to face this lack programmatically, following the visitor design pattern [7]. It simulates late-binding on the argument of a visitor method `visit(a)`

[2] We use the *varargs* language syntax of JDK1.5.

using the classical late-binding on an `a.accept(Visitor v)` method provided by data classes. If this technically solves the problem of adding a new functionality to a given class without modifying it, this design pattern has several limitations. First, it can only be used with class engineered to accept visitors, i.e., implementing 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.

4.2 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 [10], Grothoff proposed the Runabout [8] 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 specify the 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 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, that is known to be time consuming.

At the same time these works on double-dispatch were performed, Forax *et al.* developed a reflective framework to provide full multi-polymorphism in Java [6], 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 parameter 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();
    MultiMethod mm = MultiMethod.create(SumJMMF.class, "value", 1);
    int sum = 0;
    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 impose to inherit 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. 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 Java Core Reflection API.

4.3 The Sprintabout

Inspired by previous related works, we have designed and implement the Sprintabout, following the guidelines we draw in section 3. The code below illustrates how it could 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) {
    VisitorGenerator vg = new VisitorGenerator();
    SumSprintabout v = vg.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 type of the argument provided to `valueAppropriate()` calls.

The implementation of the Sprintabout respects the guidelines we outlined in section 3. To do so, at creation time, an hashtable is created to be used at invocation-time. This hashtable associates types with integers that index the method to call for this type of argument. At creation time, it is initially filled by the method `createVisitor()`. 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[3]. Then,

[3]If the type is not in the hashtable, the method index for the closest type is returned and their association is cached into the hashtable.

the dispatch is performed using this index in a simple `switch/case` statement to call the right method. The appendix A gives the code obtained by decompiling the bytecode generated for the method `valueAppropriate()` of our example. Uniqueness of this bytecode generation is insured by the classloader caching mechanism.

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 profile 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 some static type-checking on argument types, e.g. it is possible to impose arguments to implement a particular interface. Moreover, the "appropriate" method signature may return values of any type, including primitive types, without requiring cast or boxing/unboxing.

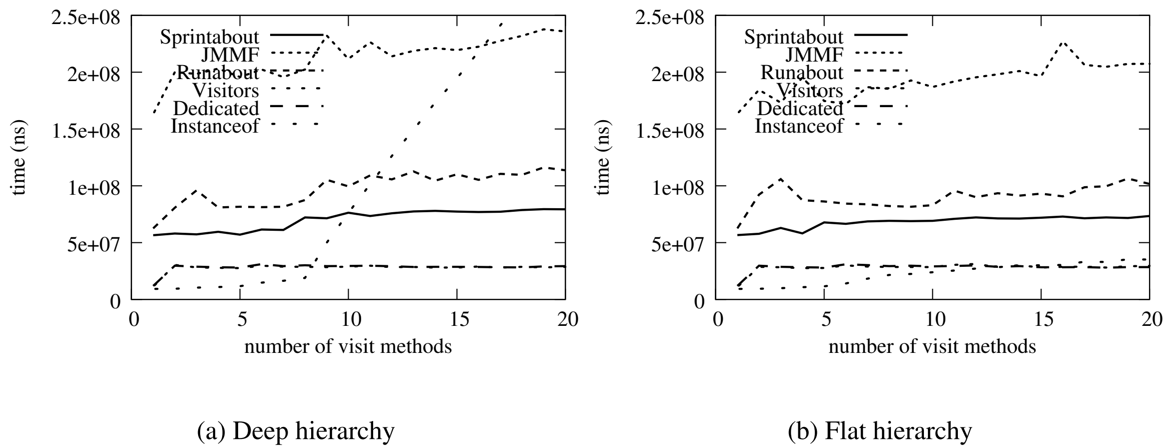
As JMMF and contrarily to the Runabout, the Sprintabout does not require the implementation class to inherit from some special class (i.e. Runabout) and thus allows any other class extension.

Current implementation of Sprintabout only supports classes as argument types and imposes the visit class to have a default constructor without argument. We plan to leverage these limitations in future implementations of Sprintabout and to perform full multi-dispatch on multiple arguments. Sprintabout implementation relies on JDK1.5 generics, on JSR155 for concurrency and uses ASM [3] for code generation. It is freely available on the Web.

4.4 Experimental results

In this section, we present an evaluation of invocation performances of Sprintabout compared with other double-dispatch solutions. We compare six techniques: *Instanceof*, implementing filtering using `instanceof` tests, *Dedicated*, where a dedicated method is added inside each classes, *Visitors*, that implements the visitor design pattern, *Runabout*, *JMMF* and *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 1 presents results of these tests with a standard “client” execution environment whereas Figure 2 presents the same results 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 j2sdk1.5.0_beta3 under Linux.



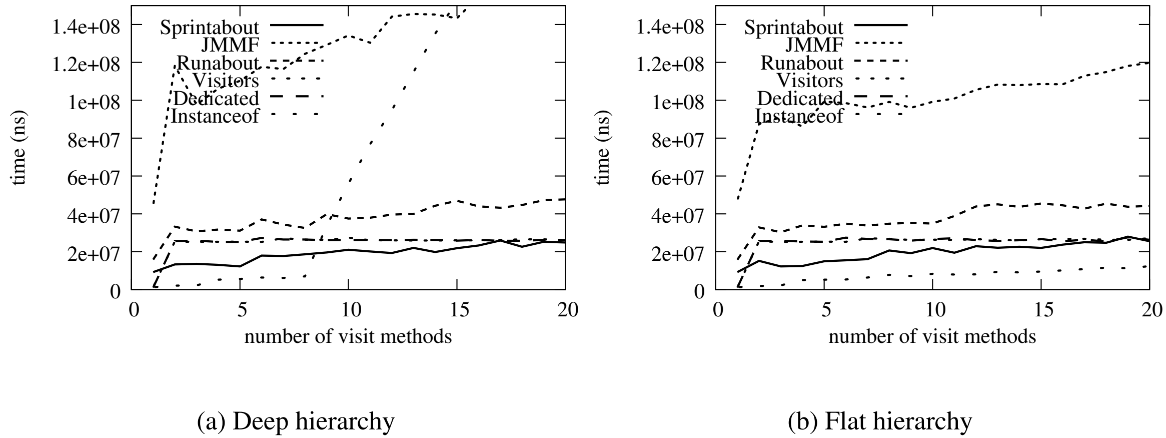
(a) Deep hierarchy

(b) Flat hierarchy

Figure 1: Dispatch time

Tests illustrated by Figure 1 show that Visitors and Dedicated techniques (which overlap in the figure) have the best performance on average. Instanceof technique has the best performance for the flat hierarchy. However, in presence of the deep one, its performances strongly decrease when the number of visit increases. Instanceof performances strongly depends on the hierarchy and seems to rely on some caching techniques. Considering the reflection-based implementations, we see that the Spintabout is about three times faster than the generic JMMF and one third 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 perfomances to be improved. This also proves that the Sprintabout, which eliminates object reification and genericity of method calls, has better performance than Runabout.

Tests performed in server mode, illustrated by Figure 2, give comparable results to previous



(a) Deep hierarchy

(b) Flat hierarchy

Figure 2: Dispatch time for JVM in server mode

ones. Nevertheless, Sprintabout has now better performances than Visitor and Dedicated techniques. It shows that the simple dispatch implementation of Sprintabout eases the generation of optimized code.

5 Conclusion

This paper promotes the use of reflection based implementations to extends the Java environment. It proposes several guidelines to make these kinds of implementation practicable:

- perform pre-computations at creation time to reduce invocation time overhead;
- perform code generation to minimize the invocation-time use of reified objects and of the generic methods of the Java Core Reflection API;
- share as many data as possible to reduce space overhead.

As an illustration, it details the implementation of the Sprintabout, a reflection-based implementation of double-dispatch for Java that conforms to these guidelines. Expermimental results compare it with other techniques and validate the usability of our approach.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] 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.

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

[11] 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.

[12] 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.

A Generated code

The following Java source code has been obtained by decompiling the bytecode generated by Sprintabout for the example of section 4.3.

```
import fr.umlv.sprintabout.ConcurrentIntMap;
import fr.umlv.sprintabout.VisitorGenerator;
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 AssertionError("bad␣index,␣oups");
    }
  }
  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;
  }
}
```