SmartMethod: an Efficient Replacement for Method

Walter Cazzola
DICo - Department of Informatics and Communication,
Università degli Studi di Milano
cazzola@dico.unimi.it

ABSTRACT

In the last few years the interest in reflection has grown and many modern programming languages/architectures have provided the programmer with reflective mechanisms. As well as any other novelty also reflection has detractors. They rightly or wrongly accuse reflection to be too inefficient to be used with real profit. In this work, we have investigated about the performance of Java reflection library (especially of the class `Method` and of its method `invoke`) and realized a mechanism which improves its performances. Our mechanism consists of a class, named `SmartMethod` and of a parser contributing to transform reflective invocations into direct call carried out by the standard invocation mechanism of Java. The `SmartMethod` class is compliant — that is, it provides exactly the same services —, with the class `Method` of the standard Java core reflection library but it provides a more efficient reflective method invocation.

Keywords: Reflection, Optimization, Java, Java Core Reflection Library.

1. INTRODUCTION

In the last few years many researchers (see for example [3, 15]) stressed the relevance of reflection, reflective behavior and meta-level architectures. This growing interest in reflection is also testified by the fact that both Java [1] and .NET [8] architectures, — that is, two of the most used programming architectures — are intrinsically reflective [12] or provide the programmer with many reflective features (see the Java core library [13] and [6] for an overview of the reflective features of Java).

Reflection is defined as the activity performed by an agent when doing computations about itself [10]. This activity involves several aspects: the most used are *introspection* and *intercession*. They are defined as the ability for a program to respectively observe and modify its own structure, state and execution.

Modern programming languages/environments such as Java and .NET provide the programmer with limited reflective capabilities as a mix of introspection and intercession. Generally, the programmer can dynamically reify some structural aspect of his program as methods, and classes (i.e., he can observe the program structure); then he can discretionary use such reifications for invoking methods and creating new objects (i.e., he can modify the program behavior). Unfortunately, coming into the limelight both merits and flaws of reflection are more evident. The most raised issue against the use of a reflective solution is related to its performance. Obviously, introspection and intercession are expensive tasks when carried out during program execution. Many attempts have been done to improve this situation, most of them are related to move reflection from run-time to compile-time [4, 11] or load-time [5]. But, there are still many situations where the introspection and intercession must take place at run-time. e.g., in remote communications where we are looking for unknown services.

In this work, we have focused our attention on the Java language, and on the efficiency of its reflection library [13]. In particular we have investigated how to improve the performance of method invocation through reifications (that is, we have improved the efficiency of the method `invoke` of the `Method` class) and implemented a class, named `SmartMethod` with all the functionalities of the corresponding class `Method` of the Java reflection library but improving the efficiency of method invocation.

The rest of the paper is organized as follows. Section 2 shows the basic idea for optimizing the Java method `invoke`, whereas section 3 goes deepen in the realization. Section 4 gives a glance at the performance improvements. Finally in section 5 we draw our conclusions and propose some future works.

2. UNFOLDING METHOD LOOKUP

The Java core reflection library [13] provides the programmer with classes (`Field`, `Method`, and so on) whose instances reify specific aspects (that is, fields, methods and so on) of a class. Such aspects are reified by invoking specific methods (e.g., `getFields` for reifying all the fields of a class) on a reification of a Java class (that is, an instance of the class `Class`).

In this work, we focus our attention on the `Method` class. Each of its instances reifies all data related to a given method (i.e., its name, its argument types, its return type and so on). The reflective invocation of the method:

```
public void testMethod(float f);
```

defined by the class `TestClass` is carried out by the following snippet of code:

```
Class c = Class.forName("TestClass");
Method m =
   c.getMethod("testMethod", new Class[]{Float.TYPE});
m.invoke(c.newInstance(), new Object[]{new Float(7)});
```

The Method class invests the method invoke with the ability of really invoking the reified method.

As explained in the API documentation, the invoke invokes the underlying method represented by **this** (an instance of Method), on the specified object with the specified parameters. Individual parameters are automatically unwrapped to match primitive formal parameters, and both primitive and reference parameters are subjected to method invocation conversions as necessary. The underlying method is invoked using dynamic method lookup (cf. [7] section 15.12.4.4); in particular, dynamic dispatching based on the run-time type of the target object will occur.

The described approach implies that a lot of execution time is spent for simulating the method lookup, for checking the method compatibility and, above all, for dispatching the method call to the referring object in accordance with the fact that the method is virtual, inherited or invoked through an interface. This fact is fairly evident comparing the time spent in invoking a method through a class with the time spent by using an interface (see table 1).

Our idea for speeding up method invocation is quite simple and consists of letting the compiler resolve method overloading for us and delegating the real invocation to the standard invocation mechanism (not to the reflective one).

To realize this mechanism, we have stolen the *stub* idea from the Java RMI [14]. Each class is associated with a stub, whose name is after the class name, implementing an *ad hoc* invoke method tailored on the class and implementing also some ancillary routines. This ancillary invoke, called smartInvoke, constructs a per-class structures that renders the method dispatch more efficient providing a binding for each method defined by the class to its direct invocation. For example, the stub associated to our TestClass provides the following mapping for the method testMethod:

```
public class TestClass_InvokeStub
                          extends SmartInvokeStub {
  public Object smartInvoke(Object o, int h, Object[] a){
    switch (h) {
      case -1822788606:
        ((TestClass)o).testMethod(
                        ((Float)a[0]).floatValue());
        return null;

          /* similarly, the removed code would show the invoca-
             tion of the other methods defined or inherited by the
             TestClass class. */

    } /* end switch */
  }
}
```

The ancillary smartInvoke, basically, is a **switch** indexed on the hashcode of the method that in somehow can be called by an instance of the class. This means that it is indexed on all **public**, **protected**, default-access and **private** methods both defined and inherited by the class. This fact could sound weird but the Java core reflection library permits, thanks to the setAccessible method, a metaprogrammer to circumvent the visibility modifier declared by the programmer.

3. METHOD OPTIMIZATION

The described idea is embedded in a more pretentious project consisting in optimizing the whole Java core reflection library. The optimization of method invocation has been realized by: (i) writing a class, named SmartMethod, which uses the described mechanism and (ii) a parser which unfolds the method invocation and builds the described stubs. The class SmartMethod is described in section 3.1 whereas in section 3.2 we describe how the parser works.

3.1 Method vs SmartMethod

We have built a class, named SmartMethod which provides the programmer with exactly all the functionalities provided by the Method class. This means, as shown in the code below, that the public interface of the class SmartMethod defines the same methods, with the same signature as the class Method.

```
package cazzolaw.lang.reflect;

import java.lang.reflect.Member;
import java.lang.reflect.AccessibleObject;

public class SmartMethod
            extends AccessibleObject implements Member {
    /* SmartMethod constructor */
  public SmartMethod(String m, Class[] c, Class d) {...}

    /* Methods inherited from AccessibleObject */
  public static void setAccessible(AccessibleObject[] a,
        boolean flag) throws SecurityException {...}
  public void setAccessible(boolean flag)
        throws SecurityException {...}
  public boolean isAccessible() {return _accessible;}

    /* Methods in the Member interface */
  public Class getDeclaringClass(){return declaringClass;}
  public int getModifiers() {return _modifiers;}
  public String getName() {return methodName;}

     /* Methods defined in Method */
  public boolean equals(Object obj) {...}
  public Class[] getExceptionTypes() {return excpTypes;}
  public Class[] getParameterTypes() {return argsTypes;}
  public Class getReturnType() {return returnClass;}
  public int hashCode() {return code;}
  public toString() {...}

  public Object invoke(Object obj, Object[] args)
    throws InvocationTargetException,
      IllegalArgumentException,IllegalAccessException {...}
}
```

In the standard Java core reflection library, method reification is the result of class inspection — that is, the class Class provides some methods (getMethod, getMethods, getDeclaredMethod, and getDeclaredMethods) which look at a class for declared and inherited methods —, hence no explicit Method creation is neither necessary nor allowed. Since we want to confine the modifications to the java.lang.reflect library and we do not want to provide the class with extra functionality, we cannot adopt class inspection as a way for reifying methods but we had to provide the programmer with a public constructor for the SmartMethod class rather than a constructor with default access right as for the Method class.

The SmartMethod class as well as the Method class must provide a general approach to method reification, therefore it cannot be statically bound to the stubs or cannot directly embed them. Therefore, the SmartMethod class is bound to the stub by a dynamic clientship that is perfected when a method is reified (that is, when an instance of the SmartMethod class is created).

The constructor knows which is the declaring class of the method to be reified (information passed to the constructor of the Method class as well) from this information it is able to determine which is the stub tailored on such a class and to get an instance of such a stub.

```
public SmartMethod(String m, Class[] c, Class d) {
 stub = (SmartInvokeStub)((Class.forName(
    d.getName()+"_InvokeStub")).newInstance());
 code = stub.hashCode();
 returnClass = stub.smartGetReturnType(code);
 _modifiers = stub.smartGetModifiers(code);
 _exceptions = stub.smartGetExceptionsType(code);
 _accessible = true;
}
```

As visible in the code of the constructor above, the name of the class of the stub is built from the name of its referent class by appending the string `_InvokeStub`. At the moment, the management of the possible name clashing associated with this solution is out of our scope.

The stub provides the method reification with a connection to the low-level method invocation unfolding. This separation grants the flexibility of the approach because free the implementation of the `SmartMethod` class from the knowledge of the static type of the caller but it is also one of its flaws, because we invoke stub's methods by exploiting late binding and therefore by resolving method dispatch to the stub at run-time.

```
public Object invoke(Object obj, Object[] args) {
  return stub.smartInvoke(obj, code, args);
}
```

The `invoke` method (code reported above) is just an interface to the `smartInvoke` defined in the stub and linked at construction-time. Similarly, all information about the method (return type, exception types, and so on) are encapsulated in the stub (thanks to the ancillary routines cited before) and can be retrieved, when the method is reified, efficiently as well.

At the moment, the constructor declaration is the only difference in the interface and therefore in the use of the `SmartMethod` class with respect to the `Method` class. In the future, we are going to extend the `Class` class with the necessary mechanism for reifying the methods as `SmartMethod` instances as well. This fact apart, the `SmartMethod` class provides the programmer with the same functionalities as the `Method` class and it is subjected to the same restrictions. This means that we can reify the same category of methods both using `Method` and `SmartMethod` and the reflective invocation can be inhibited by revoking the `ReflectPermission` through a security manager as well.

3.2 Stub Generation

We provide a tool, named `SmartInvokeC`, for the automatic generation of the stub of a class from its bytecode. Therefore, our approach to method reification is independent of the availability of the source code of the declaring class, but it only needs the bytecode of the class[1].

The `SmartInvokeC` inspects the bytecode of a class looking for information about the methods that are invocable, that is, as said before, all the methods, independently of their right accesses, declared by one of the classes in the class hierarchy of the inspected class. Inspection takes place recursively on each class *c* in the class hierarchy by collecting all its declared methods; this raking of methods is carried out by exploiting the standard reflective mechanism (i.e., the `getDeclaredMethods` method) retrieving all the methods (both **`public`**, **`protected`**, default-access and **`private`**) declared in the class (see the code of the `getMethodsList` method, reported below).

The Java method invocation mechanism establishes that a method matching the signature of the invoked method whose visibility is not hidden by the definition of another method with the same signature is activated. Therefore, methods can be called only when the dynamic type of the calling object corresponds to a class whose visibility is not obscured by another class which defines a method with the same signature as the invoked method, in this case the overriding method is invoked. This behavior is also granted with our reflective invocation by removing the overridden methods from the list of the invocable methods and delegating their activation to the dynamic lookup provided by the direct invocation of Java.

Methods collection and classification is realized by the following snippet of code:

```
protected Method[] getMethodsList() {
  Class _class = Class.forName(ClassName);
  Package _package = _class.getPackage();
  Package _spackage = _package;
  Method[] _ms;
  Vector _vm = new Vector(), _vmp = new Vector();
  while (_class != null) {
    _ms = _class.getDeclaredMethods();
    for(int i=0;i<_ms.length; i++)
      if !(_vm.contains(_ms[i])||_vmp.contains(_ms[i]))
       if Modifier.isPrivate(_ms[i].getModifiers()) ||
         (Modifier.isProtected(_ms[i].getModifiers()) &&
           (!isTheSamePackage(_package, _spackage)))
              _vmp.add(_ms[i]);
        else _vm.add(_ms[i]);
    _class = _class.getSuperclass();
    if (_class != null) _spackage=_class.getPackage();
  }
  Method[] _ml = new Method[_vm.size()+_vmp.size()];
  _publicMembers = _vm.size();
  for(int j=0; j<_vm.size(); j++)
     _ml[j] = (Method)_vm.elementAt(j);
  for(int j=0; j<_vmp.size();j++)
    _ml[j+_publicMembers] = (Method)_vmp.elementAt(j);
  return _ml;
}
```

Collected methods are classified in two groups after their access rights. The former group collects the **`public`** and default-access methods and the methods defined as **`protected`** in the same package of the examined class; whereas the second collects the remaining methods. We distinguish the methods in these two categories because the methods in the first category can be invoked without restrictions. On the contrary, the invocation of the methods in the second category must obey to some restrictions due to their access qualification. We have gone round such restrictions by delegating their invocation to the Java Native Interface (JNI) [9]. C/C++ programs can invoke Java methods without undergoing to the access restrictions defined by the Java class. Notwithstanding that to find an application for the reflective invocation of **`private`** methods is not trivial, we have implemented it for compatibility with the standard reflective invocation.

The whole mechanism for fast retrieving and invoking a method is based on hashcoding the information necessary to discriminate such a method from the others. The method signature (that is, its

[1]At the moment, we are investigating a mechanism for generating the stub when the class is loaded by the JVM without affecting the efficiency. With such a mechanism we can benefit of the `SmartMethod` improvements also invoking methods whose bytecode is available only at run-time.

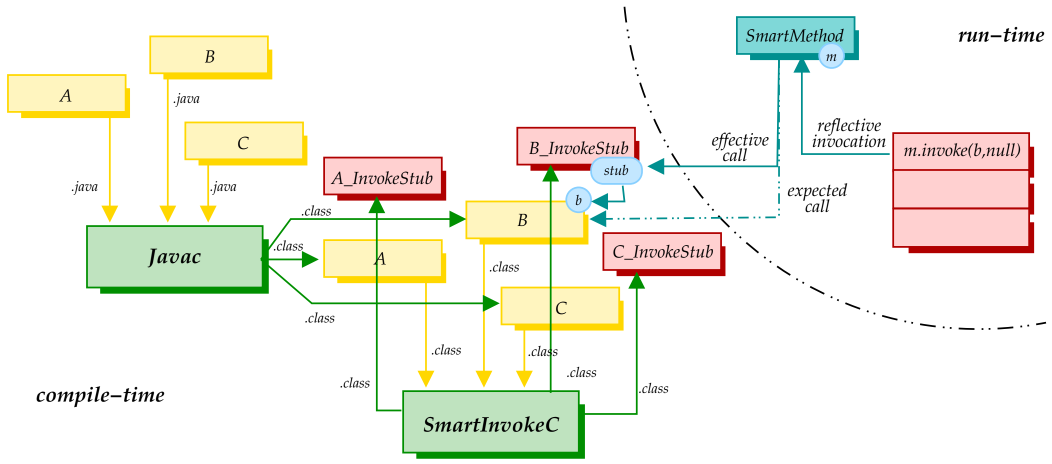

Figure 1: Compiling the stubs and reflective method invocation through the stubs

name and the classes of its arguments) contains all the necessary information for discriminating methods. We do not consider the method return type because it does not contribute to the overloading resolution (that is Java does not allow the definition of two methods with the same name and arguments but different return type). The hashcodes associated with each method are calculated by the `SmartInvokeC` tool during the bytecode analysis. All the hashcodes are calculated from the method signature and embedded in the stub (in particular, in the method `hashCode` of the stub). Conflicts are managed by associating a manually incremented sub-hashcode with the calculated one. At method reification the corresponding hashcode is retrieved from the stub. This mechanism guarantees the unicity of the hashcode, its fast retrieving and therefore a perfect and fast mechanism for discriminating the invocable methods in the stub.

Figure 1 summarizes how compilation takes place and how the reflective method invocation exploits the stub. As usual, bytecodes are generated from the classes by the Java compiler (e.g., `javac`). From the bytecode of the classes, the `SmartInvokeC` creates and compiles the corresponding stub classes. At run-time, method reification associate the reified method with the stub of the class declaring the reified method. The reflective invocation of a method through the `invoke` method of the `SmartMethod` class is hijacked to the stub and after to the class declaring such a method rather than directly to declaring class. The intermediated step permits to transform the reflective call in a direct call as explained in the previous sections.

4. PERFORMANCE EVALUATION

We have quantified the performances of the `invoke` of the class `SmartMethod` with respect to standard Java method invocation and the method `invoke` of the Java core reflection library. The aim of our experiments consists of measuring how long standard Java method invocation and both kind of reflective invocations take to invoke a method. All the experiments were performed on an Intel® P4@2.2 GHz with 512Mb RAM running Linux (kernel version 2.4.21), and `jdk v1.4.2`.

The scenario of our experiments is composed of a class which implements a simple interface. This class defines some dummy methods; these methods are distinguishable for their access right (**`public`**, **`protected`**, default-access or **`private`**). Methods taken in consideration for the experiments — following the hints given by the Sun's FAQ on Java HotSpot VM benchmarking available at `java.sun.com/docs/hotspot/PerformanceFAQ.html` —, have an argument, compute some values by using such an argument and return the computed value. In this way, we avoid the optimizations carried out by HotSpot as short method inlining and the removal of dead code that will not render germane the comparison with the direct method call. Basically, the benchmarking has been carried out by repeatedly invoking on a class (or on an interface) these methods and then by calculating the average of the achieved time.

Table 1 summarizes the results of our experiments. The second column represents how long the standard method invocation takes to invoke a **`public`** method. Similarly, the third and the forth columns show how long takes the same invocation carried out by using, respectively, the standard `invoke` and our `invoke`. The experiments have been done by invoking the methods through a class (second and third rows) and through an interface (fifth and sixth rows) and both enabling and disabling the HotSpot just-in-time compiler.

Calling a method on a class results nearly 1.3 times faster by using our approach with respect to the standard `invoke`. Notwithstanding this improvement, as expected (see section 2), we got the best by invoking methods on an interface. In this case, our approach is about forteen times faster than standard `invoke`. However, we are still far from getting the same performance as by using direct invocation.

The table, for sake of space, summarizes the results only related to the invocation of **`public`** methods. We have gotten quite similar figures invoking **`protected`** and default-access methods. On the contrary, less good results have been gotten by invoking **`private`** methods. This fact can be ascribed to the use of JNI for working around the access protection, a further improvement should be gotten by writing pure Java code which directly accesses to the JVM

call via a class	*direct call*	*invoke*	*smart-invoke*
HotSpot	0.0000867	0.000204	0.000154
HotSpot (disabled)	0.0006602	0.0017125	0.0012946
call via an interface	*direct call*	*invoke*	*smart-invoke*
HotSpot	0.0001213	0.0027938	0.0002116
HotSpot (disabled)	0.000606	0.0039545	0.0012208
* *all the reported time are expressed in milliseconds.*			

Table 1: Direct call, `invoke` and smart invoke in comparison.

for invoking the **private** methods.

5. CONCLUSIONS

In this paper we have exposed our idea for optimizing the performances of the reflective method invocation in Java. Basically the idea consists in delegating the method lookup and the late binding to the standard invocation mechanism provided by Java. We have also proved the effectiveness of our solution by implementing it as a class — named `SmartMethod` — which provides the same functionalities (especially, method reification and invocation) of the `Method` class and a parser which unfolds the method lookup mechanism allowing the direct call of each method through the instances of `SmartMethod`.

A different approach to render more efficient the `invoke` method consists of having a pure object-oriented implementation of the `Method` class. Basically, the class `Method` maintains the same interface but it is an abstract class and its subclasses will embed, in the implementation of their `invoke` method, the direct call to the method they are reifying. Therefore, instances of `Method` are never created, instances of its subclasses are created and used instead.

Surely this approach could be more elegant and flexible than the one proposed in this paper and probably it gives the same benefits or better in terms of performance but it has two major problems that have pressed us to not consider this approach. First, this approach presupposes to directly instantiate the class reifying the method to invoke rather than instantiate the class `Method` with the right parameters (it does not provide the programmer with a uniform approach to method reification). Second, surely the first in importance, we have to pay what we gain in terms of performance with an elevate proliferation of classes (a new class for each method). Besides, we still need a parser which examines the classes and generates all the concrete sub-classes of the `Method` class.

Our approach is still far from being perfect. Most of the time which separates a direct invocation from our reflective invocation are lost in type conversions (we must cast the arguments from `Object` to the type expected by the method), in binding the `SmartMethod` with the stub of the right class (late binding) and in mangling Java code with C++ code through the JNI interface (C++ permits to work around invocation restrictions, e.g., it permits to invoke **private** methods). We are studying a way to overcome these flaws and render more and more efficient our reflective invocation. We are also extending the approach to the `Constructor`, and `Field` classes of the Java core reflection library.

We are also investigating the impact of our improvements in existing reflective applications. We think that the research area on reflective and adaptive middleware should greatly benefit from performance improvements in the reflective invocation and we would like to prove that experimenting on the mChaRM [2] reflective middleware.

Acknowledgments

The author wishes to thank Angelika Langer because the basic idea of delegating method lookup at the standard mechanism rose arguing with her about reflection performances at rOOts'02 in Bergen. The author wishes also to thank Giovanni Lagorio because the spare-time spent with him has solved many technical and theoretical problems tied to Java and its specification.

6. REFERENCES

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachusetts, second edition, Dec. 1997.

[2] W. Cazzola. Remote Method Invocation as a First-Class Citizen. *Distributed Computing*, 2003. To Appear.

[3] W. Cazzola, R. J. Stroud, and F. Tisato, editors. *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, June 2000.

[4] S. Chiba. A Meta-Object Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *Sigplan Notices*, pages 285–299, Austin, Texas, USA, Oct. 1995. ACM.

[5] S. Chiba. Load-Time Structural Reflection in Java. In E. Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, LNCS 1850, pages 313–336, Cannes, France, June 2000. Springer-Verlag.

[6] I. R. Forman and N. B. Forman. *Java Reflection*. Manning Publications, 2004.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachusetts, second edition, 2000.

[8] K. Hoffman, J. Gabriel, D. Gosnell, J. Hasan, C. Holm, E. Musters, J. Narkiewickz, J. Schenken, T. Thangarathinam, S. Wylie, and J. Ortiz. *Professional .NET Framework*. Wrox Press., 2001.

[9] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachusetts, June 1999.

[10] P. Maes. Concepts and Experiments in Computational Reflection. In N. K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, Oct. 1987. ACM.

[11] H. Masuhara and A. Yonezawa. Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In E. Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, LNCS 1445, pages 418–439. Springer-Verlag, July 1998.

[12] Microsoft Corporation. .NET Framework Developer's Guide: Reflection Overview. Technical report, Microsoft Developer Network (MSDN), 2003. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguid%e/html/cpconreflectionoverview.asp`.

[13] SUN Microsystems. Java™ Core Reflection API and Specification. Technical report, SUN Microsystems, Feb. 1997.

[14] SUN Microsystems. Java™ Remote Method Invocation - Distributed Computing for Java. White paper, SUN Microsystems, 1998. Internet Publication - `http://www.sun.com`.

[15] A. Yonezawa and S. Matsuoka, editors. *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, volume 2192 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, Sept. 2001.