JavaCloak — Considering the Limitations of Proxies for Facilitating Java Reflection

Karen Renaud[1] & Huw Evans[2]

Department of Computer Science and Information Systems,

University of South Africa

P O Box 392, UNISA, 0003, SOUTH AFRICA

E-mail: `renaukv@unisa.ac.za`

Abstract

This paper discusses issues pertaining to mechanisms which can be used to change the behaviour of Java classes at runtime. The proxy mechanism will be compared to and contrasted with other standard approaches to this problem. Some of the problems the proxy mechanism is subject to will be expanded upon. The question of whether statically-developed proxies can ever be a viable alternative to bytecode rewriting was investigated by means of the JavaCloak system, which uses statically-generated proxies to alter the run-time behaviour of externally-developed code. The issues addressed in this paper include ensuring type safety, dealing with the self problem, object encapsulation, and issues of object identity and equality. Some performance figures are provided which demonstrate the load the proxy mechanism places on the system. The paper concludes that the proxy mechanism, as utilised for runtime reflection, cannot be viable.

1 Introduction

There is often a need to specialise the runtime behaviour of classes. There are a number of non-functional requirements which could require such specialisation such as, for example, security, system instrumentation and distribution. Runtime specialisation can be achieved by:

1. tailoring source code [20],
2. providing a customised Java Virtual Machine (JVM) [8, 10, 15].
3. providing a custom classloader which integrates a meta-object protocol with the original bytecode (Javassist & Kava) [2, 19], or
4. using wrapper/proxy objects (Dalang) [18].

Each of the above approaches has both advantages and disadvantages. Section 2 will discuss the pros and cons of each mechanism. In an attempt to quantify the effect that statically-generated proxies have on system performance the JavaCloak system was developed, using Java. JavaCloak uses statically-developed proxies to alter system runtime behaviour. The system is described in Section 3. Section 4 discusses the problems JavaCloak encountered — related to the use of proxies for reflection. Section 5 presents the results of performance measurements with JavaCloak proxies. Section 6 concludes.

2 Evaluation of Reflection Mechanisms

2.1 Source Code Tailoring

Programmers often do not have access to the source code, and so this option will not be universally available. If the programmer *does* have the source code, and does not impinge on a licence agreement by altering it, it still does not make source-code tailoring a good option. Source-code tailoring will always be an inelegant and untenable solution to the problem because programmers may introduce new errors into the program, change the behaviour of the program, or omit to insert code consistently throughout the program.

2.2 Customised JVMs

Customised JVMs are a better solution than source-code tailoring but are often not a viable option due to their being non-standard, and often tightly linked to one specific platform.

2.3 Bytecode Engineering

Some researchers have investigated tools and techniques that allow Java bytecode to be changed without needing access to the source code. These tools can be divided into two categories, those that support generic bytecode rewriting [6, 4] and those that provide a meta-object protocol [18] and dynamically adapt code at run-time [11].

The bytecode-rewriting tools in the first category are very useful as they allow programmers to change a class definition to meet a local need. For example, the bytecode of a class can be changed to ensure that it is compatible with the Java Object-Serialization mechanism so that instances of the modified class can be passed across a network or written to disk. However, bytecode rewriting has a number of disadvantages. Firstly, the changes must be applied every time a class requiring it is recompiled and there is the extra effort of determining whether a new version requires amending or not. In addition, systems that require bytecode to be rewritten, such as ObjectStore's PSE Pro [5], may burden the programmer with the management of two sets of classes. For example, one class (`A`) may depend on another class (class `B`) that has to be post-processed. Class `B` should therefore be compiled first and post-processed before class `A` is compiled. Tracking these kinds of relationships for significant bodies of code is a non-trivial problem. This kind of bytecode rewriting is simply too low level and powerful for the average programmer. A higher level of abstraction is required and thus the meta-object protocol approach has been developed [19].

Meta-object protocols [12] are a powerful programming paradigm for associating new behaviour with a program. The authors of [18] have defined a meta-object protocol for Java that rewrites bytecode at load time. The Kava approach allows either the addition of new behaviour to the class or the ability to selectively override invocations on a method-by-method basis [18]. Kava also allows exception handling to be intercepted so that exceptions can be overridden or reported in a more effective manner.

Javassist [2] is also a bytecode rewriting tool based on structural reflection. It also makes use of a meta-object protocol. However, it does not support reflection on methods inherited from superclasses [19] — a problem that has been solved by Kava.

The system described by Keller and Hölzle in [11] is targeted at integrating Java classes with other, non-compatible, classes. The classes are made compatible by means of the programmer identifying which class should have its bytecode modified at run-time to ensure that the classes can interoperate. Their system ensures that incompatible classes can be used together by adding, renaming or removing methods, or by changing the class hierarchy.

Dynamic bytecode rewriting imposes a runtime overhead when loading classes. This overhead could be reduced by performing the post-processing once, statically, before the program is run. However, all the classes that the program could possibly use would have to be identified and it would have to be re-applied every time the related meta-object is changed.

2.4 Wrappers/Proxies

The proxy approach is a well-known software engineering technique [9]. Proxies can be set up at different times:

- compile time — such proxies are generated statically, and compiled and loaded by the JVM instead of the original classes.
- load time — such proxies are generated on-the-fly by providing the JVM with a customised class loader. This class loader generates the wrapper and provides it in place of the original class [17].
- runtime — this can only be done by means of a reflective JVM [14], which allows the loaded bytecode to be altered so that a proxy can be substituted for the original class.

The Java 2 platform (version 1.3) defines dynamic `Proxy` classes [1] — classes that implement a list of interfaces specified at runtime when the class is created. A `Proxy` class, however, has a number of undesirable properties (for the purposes of runtime specialisation):

1. dynamic proxy classes are `public`, `final` and cannot be abstract;
2. the proxy class must extend `java.lang.reflect.Proxy`;
3. a programmer-defined invocation handler must be used to dispatch the method invocation to the wrapped instance at run-time; and

4. the Java security domain of the proxy is the same as that of system classes loaded by the bootstrap classloader, such as `java.lang.Object`, because the code for the proxy class is generated by trusted system code. This system domain is typically granted the Java security permission `java.security.AllPermission`, and so these proxy classes are effectively not restricted by the Java 2 security mechanism.

Given these limitations of Java-provided proxies, an approach that exploits statically generated proxy classes to enable post-implementation customisation of class behaviour — named JavaCloak [16] — was investigated to determine the performance penalty associated with static proxies.

3 The JavaCloak System

The problems with load-time generated proxies are well known [17]. However, statically-generated proxies have not been tried and tested, due to the fact that the JVM will not load proxies in place of the original classes unless they have the same names as the original classes, and if they have the same names proxy objects cannot access instances of the original classes and so proxies are unable to delegate method invocations. In order to test statically-generated proxies two problems have to be overcome:

1. how does one get the JVM to load the proxies instead of the original classes if both sets of classes are to be made available to the JVM?
2. how does one then access the original classes from within the proxies — in the light of the fact that the proxies and the original classes have the same name (which must be the case for the JVM to have loaded them)?

The JavaCloak system has solved these problems by using a customised classloader and by manipulating the `CLASSPATH` at runtime. The proxy objects are statically and automatically generated, by using the Java introspection mechanism (`java.lang.reflect`), to mediate access to the original objects — and are produced in the form of Java code. The required behavioural changes can be incorporated into this code by the programmer — a process that requires no additional programmer skills[1]. The programmer is free either to augment the classes with reporting facilities in the very simplest case, or to change the behaviour of the methods completely by invoking a method on an instance of another class altogether, or on a remote object. After the customisation and compilation of proxies they are inserted into the system, at runtime, and these proxies then delegate all calls on the proxy methods to instances of the original object. The process becomes more complex when parameters and return values are involved — hence the inclusion of the JavaCloak runtime manager.
The key enabling features of JavaCloak, shown in Figure 1, will be described below:

1. *proxies* — The pre-runtime generation of proxies ensures that the runtime penalty of using JavaCloak is minimised and secondly that the everyday Java programmer can specialise the runtime behaviour of the

[1] This could also be seen as source-code tailoring but it should be borne in mind that this is much simpler to do since the forwarding model is very simple and the programmer cannot introduce errors into the original code or change the behaviour of that class in any way.

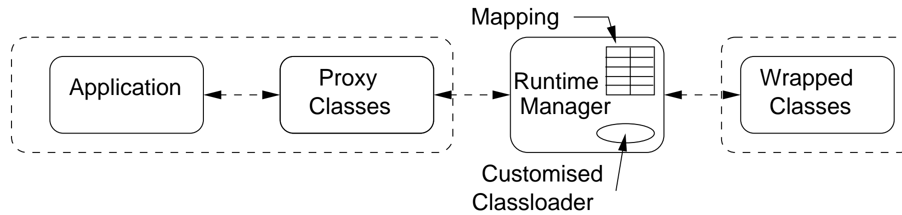

Figure 1: JavaCloak Architecture

class in a finely-grained and flexible manner.

2. *manipulation of the* `CLASSPATH` *at runtime* — the location of the proxy classes is inserted into the `CLASSPATH` instead of the location of the original classes. The location of the original classes is then provided for use by the JavaCloak classloader by means of a runtime system property setting.

3. *a customised classloader* — JavaCloak makes use of a specially defined classloader, embedded within the runtime manager, to load the original classes at runtime. This classloader loads the original definition of the class from a location supplied to the classloader by a runtime variable when the application is executed.

4. *the JavaCloak runtime manager* — this manager is the key to JavaCloak's extra level of abstraction. The runtime manager encapsulates the classloader and loads the original classes when requested by the proxy objects. It also maintains a mapping between proxies and their matching original objects so that any parameters passed between the two can be translated as required. For instance, say a method invocation passes an instance of a proxy class as a parameter. The proxy class has no meaning to the original class and, if passed to it, would cause the system to generate an exception. The runtime manager offers a facility for substituting original objects so that such parameters are 'unwrapped' before they are passed to the method in the original class. In the same way the original object may pass a reference to a wrapped object back to the proxies. The runtime manager offers a facility to substitute the proxy for such objects again so that they do not cause `ClassCast` exceptions in the application (which has no concept of the original classes).

The structure of interaction between the application, proxy and JavaCloak runtime manager is shown in Figure 2. When the proxy is instantiated it asks the runtime manager to instantiate an instance of the original object. The runtime manager then:

1. uses the customised classloader to load the original class definition from the location specified in the runtime variable.

2. creates an instance of the original class.

3. inserts an entry into the mapping table linking the proxy to the original object.

4. returns a reference to the original object to the proxy.

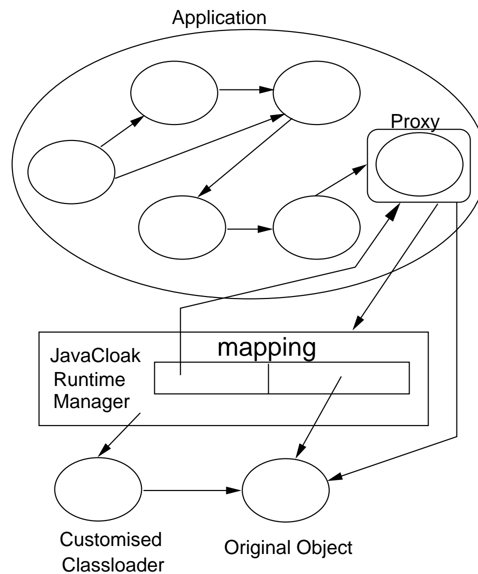

Figure 2: Interaction at Runtime

When a method is invoked on the proxy the following is done:

1. if the method has parameters and the parameter objects are references to proxies then the proxy asks the runtime manager for references to the original objects.

2. the proxy invokes the method on the original object.

3. if the method returns a value:

   (a) the proxy receives the return value and stores it in a variable of type `Object`

   (b) if the return value is an instance of a wrapped class then the proxy asks the runtime manager for a reference to the matching proxy object and the proxy returns the reference to the proxy object to the application.

   (c) otherwise the return value is 'casted' to the correct type and returned to the application.

JavaCloak has been used successfully to wrap classes and to report on access to instances of these classes. The following section will discuss some problems encountered with the use of proxies for reflection.

4 Proxy-Related Problems

The proxy approach is not without its problems. One JavaCloak-specific problem is that JavaCloak makes use of manipulation of the `CLASSPATH` in order to divert the JVM to the proxy classes rather than the original classes. This obviously won't work for system classes which are loaded by the JVM automatically and not by means of

a search of the `CLASSPATH`. This problem could be alleviated by the provision of a customised classloader to the JVM. Other generic proxy-enabled reflection related problems will be discussed in the following subsections.

4.1 Type Safety

It is important for the smooth functioning of a system incorporating JavaCloak proxies that the proxies be type-equivalent to the original classes. Therefore the generated proxy must have the same fully-qualified class name as the original class and it must be loaded at run-time by the correct instance of the classloader. This ensures that the application can use the proxy as if it were the original class, and the inheritance structure is not broken. There are some tricky problems related to this apparent transparent substitution, some of which have been solved, and others of which remain.

4.1.1 Maintaining Two Identical Class Names

One runtime problem that JavaCloak had to overcome is that the JVM needs to have instances of both classes within the system at the same time. One cannot load two different definitions of the same class name into the JVM without some special mechanism. The only way to achieve this apparent conflict is by means of the use of a different classloader for each class. This is because the JVM tests type equivalence by testing both the class name, and the class loader that loaded the class definition. If the two classes are loaded by two different class loaders the JVM considers them to be different classes — even though the fully-qualified class name is the same.

Once the two classes are in the JVM it is essential that instances of the two classes be kept strictly separate. Any attempt to reference the the original class, where previously the proxy class has been referenced by the application, will result in an exception being thrown, and a possible system crash. The JavaCloak runtime manager is used to maintain a strict separation between the two types of objects — the proxies and wrapped objects.

4.1.2 `public` fields

It is possible for the original definition of a class to contain non-static `public` fields. If this is the case, to ensure correctness it must be possible for the field to be accessed directly from the application — and not via `set` and `get` methods. Unfortunately Java does not model field access as method invocation, so there is no opportunity to redirect accesses to the `public` fields via the proxy if the application accesses the fields directly. Providing the application with access to these fields consistently and transparently in the presence of JavaCloak proxies is not simple. JavaCloak needs to ensure that any changes to the original object's `public` fields are reflected in the proxy object's `public` fields as well. In order to do this the JavaCloak system would have to 'watch' these fields and monitor them after each method invocation. This has performance implications which could be non-trivial.

The application may also make changes to the fields in the proxy class which then have to be propagated to the original object. If the application accesses the fields directly and not by means of a method invocation, one would have to 'watch' these variables and propagate the changes to the original objects.

JavaCloak needs either to implement a system of 'watchers' for each `public` field, or assume a clean object-oriented programming model where all field accesses are controlled by means of suitable method calls that can then be used to forward the call to the original object.

4.1.3 Inheritance

This is a tricky problem for JavaCloak. If JavaCloak provides a proxy for class B, which inherits from class A, which also has a proxy, it is important for there not to be multiple links between the proxy *world* and the wrapped *world*. The need to keep these two *worlds* apart was alluded to in Section 4.1.1. JavaCloak must ensure that proxy A's constructor does not request the runtime manager to create a matching original object if it is invoked from B's constructor. If such a link is constructed, the situation will be as shown on the lefthand side of Figure 3. The multiple links will cause havoc when proxy objects invoke superclass methods, and any changes made will not be reflected in the superclasses of the original objects. The position should be as shown on the right — where the inheritance structures in the two worlds are totally unconnected — except at the initial explicit level within the instantiated proxy.

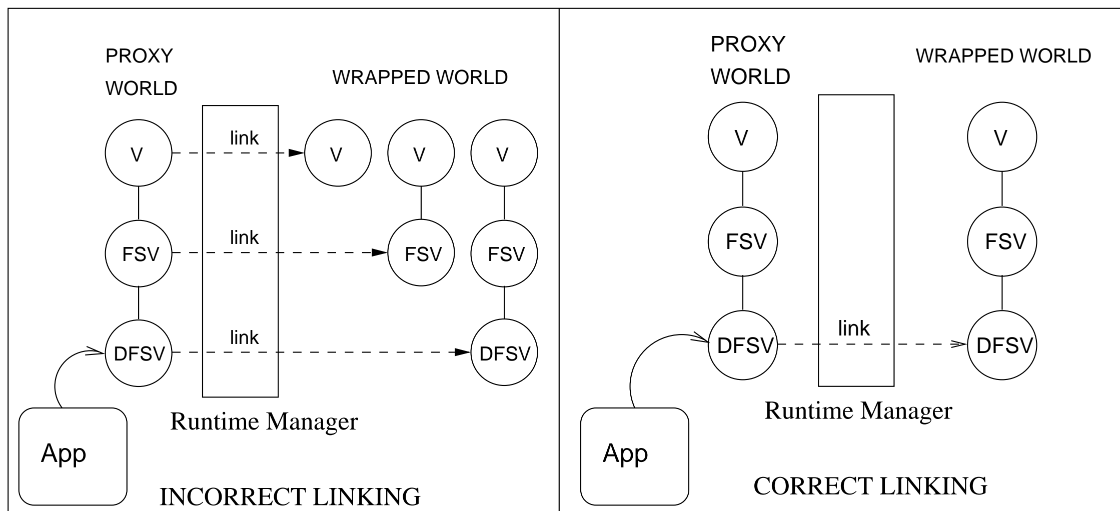

Figure 3: Inheritance across Worlds

A related problem concerns inheritance from proxy classes. If class B now has a subclass, C, which is not wrapped, the correct behaviour of this subclass's methods is debatable. It is a simple enough matter to wrap the superclass and to invoke methods that C invokes on the (proxy) superclass on the matching original class. The difficulty occurs when one considers methods that C defines. Should C also be provided with a proxy? The SOM approach, which makes use of meta-objects, requires C to have a related metaclass which is a subclass of the metaclass related to B [7]. They thus require any subclasses of 'wrapped' classes to also be 'wrapped'. These types of issues are not trivial to solve and JavaCloak has yet to come up with a satisfactory solution.

4.2 Self and Encapsulation

The JavaCloak approach implements the proxy class and the wrapped class separately. This leads to two problems, the *self* [13] and the encapsulation problem.

The self problem arises because the meaning of self (or `this` in Java programs) is different in the proxy and the wrapped instances. Thus the original object could instantiate a new instance of the same class. This object would not have a matching proxy object and therefore the behavioural modification being applied by the proxy would not be applied to this newly instantiated object. Lieberman [13] argues that inheritance-based languages such a Java cannot be used to implement delegation, which is, in essence, what a proxy does. If the programmer needs to intercept accesses to *all* instances of the original class this limitation is a problem but if one wishes merely to intercept all accesses by the *client* program the proxy approach does not present a problem [17].

If the original object returns a reference to itself (`this`), or to another instance of the same class, to the proxy, this will be intercepted by the runtime manager. If a matching proxy already exists the proxy instance will be substituted, and if not, a proxy instance will be instantiated and returned to the proxy. This mechanism works when simple references to wrapped objects are passed back to the caller, but when a direct reference to an original object is embedded in another object that JavaCloak has no control over, then the situation is not solvable: the reference may be private which means JavaCloak cannot access it to perform the required conversion.

Hence in JavaCloak it is possible for a direct reference to an original instance to be passed across the boundary, thus breaking the proxy model. This occurs because a logical proxy model is being used and Java does not allow the redefinition of the meaning of `this` in the original object.

4.3 Identity and Equality

Any existing object identity operations will work in JavaCloak because the identity of the two proxies will be compared, rather than the two original objects. However, if the reference to the proxy is passed to a service that follows the graph of objects reachable from the proxy, as Java's object serialization does, then the proxy and the real object will be serialized, which the programmer may not intend.

In JavaCloak, all invocations on the public methods of an object are intercepted at the proxy, including the `hashCode` and `equals` methods defined on `java.lang.Object`. Thus, in JavaCloak, it is not possible to perform these operations on the proxies themselves as the `hashCode` and `equals` methods are forwarded to the original instance. This means, at the JavaCloak implementation level, that we cannot make use of these methods to manage the proxy. This requires additional objects to be registered with the lookup mechanism which then operate as *tokens* for the proxy when performing a lookup on it.

When calling forward on the `equals` method it is necessary to translate the call from an operation on two proxy objects (`this` and the argument to `equals`), into an operation on two original objects. This is made possible by performing a proxy to original lookup in the JavaCloak runtime manager.

4.4 Transparency

Welch and Stroud [17] cite a number of difficulties inherent to the transparent addition of non-functional requirements by means of reflection — one of which is the handling of exceptions. There are two aspects to be considered:

1. Certain exceptions are declared in the method headers and are therefore 'expected' by the application. One may wish to intercept these exceptions for the purposes of better reporting or more standardised exception handling. Welch and Stroud [17] note that some researchers feel that this type of action allows adaptive runtime redefinition of exceptional behaviour [3]. Welch and Stroud argue that exception handling should not be re-definable but acknowledge the need for it in certain situations such as distributed exception handling.

2. A bigger problem for JavaCloak is that 'unexpected' exceptions may be thrown — caused by the reflection mechanism. These exceptions need to be handled in some consistent way, both in order to minimise disruption of the application, and so that the reflection system developers are apprised of the problem in case the exception was caused by a bug in their code.

5 Performance Results

This section describes the typical run-time performance overhead of using a JavaCloak proxy. The timing results were taken on a lightly loaded Pentium II with version 1.3 of Java 2 from Sun Microsystems. The original class that was wrapped is given in Figure 4[2] and a proxy for it was built. A small test program was written that created a thousand instances of `Square` and invoked methods `draw, moveX` and `getSquares` on each. This program was run twenty times, both with and without the proxies, and an average taken. All the figures given are in seconds for one method invocation or one call to the constructor. The times for the proxy are on the right of the pair in Table 1.

Constructor		draw		moveX		getSquares	
0.0058	0.2799	0.0264	0.0872	0.0278	0.1116	0.0299	0.3905

Table 1: Overhead of Calling Through a JavaCloak Proxy

It is obvious from the table that JavaCloak proxies introduce a substantial overhead — even though the proxies are statically generated. In the constructor this is due to reflecting over the methods, and registering the proxy and the actual object with the JavaCloak runtime mechanism.

The overhead imposed on the invocations stems from needing to track the relationship between the proxy object and the wrapped object. For example, when an instance of `Square` is passed back from itself, we need to map this value to its corresponding proxy. Currently, this is performed by means of a runtime lookup.

[2] The implementation of the methods has been elided to save space as this is not relevant to this measurement.

```
public class Square {
  public  Square() {}

  public void draw() {}

  public void moveX(int delta, Square[][] sq) throws NullPointerException {}

  public Square [][] getSquares() {}
```

Figure 4: Original Class Used to Collect Performance Results

6 Conclusions and Future Work

It is obvious from the previous discussion that the proxy mechanism has serious shortcomings as a reflective Java mechanism. Welch and Stroud [19] report that their Kava bytecode rewriting mechanism currently doubles the cost of instructions. The work reported in this paper has attempted to determine whether the proxy mechanism can ever be a serious competitor to bytecode rewriting. It is clear that the cost associated with statically-generated proxies is even *higher* and thus it can only be considered in cases where the proxy *diverts* method invocations to a remote site — and does not have to carry out time-consuming reflective activities with each and every method invocation. This defeats the purpose of having a complicated reflection mechanism, however, and reverts to a simple use of proxies *in place of* the original classes. The evidence is conclusive and therefore no future development of JavaCloak will be carried out.

References

[1] Java 2 platform, standard edition online documentation. `http://java.sun.com/j2se/1.3/docs/api/index.html/`, April 2000. Web Document.

[2] S. Chiba. Load-time structural reflection in java. In *14th European Conference on Object-Oriented Programming. ECOOP 2000.*, pages 313–336, Sophia Antipolis and Cannes, France., 12 – 16 June 2000.

[3] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 483–502, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[4] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, USA, June 15–19 1998. USENIX Association.

[5] E. Corporation. Objectstore enterprise edition homepage. `http://www.object-store.net/objectstore/`, May 2000. Web Document.

[6] M. Dahm. Byte code engineering. In *Proceedings of JIT'99.*, Duesseldorf, Germany, 1999.

[7] S. Danforth and I. R. Forman. Reflections on metaclass programming in SOM. *ACM SIGPLAN Notices*, 29(10):440–440, Oct. 1994.

[8] J. de O Guimarães. Reflection for statically typed languages. In *ECOOP'98*, Brussels, Belgium, July 20–24 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.

[10] M. Golm. Design and implementation of a meta architecture for java. Master's thesis, Erlangen, 1997.

[11] R. Keller and U. Hölzle. Binary component adaptation. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 1998.

[12] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge (MA), USA, 1991.

[13] H. Lieberman. Using prototypical objects to implement shared behaviour in object-oriented systems. In N. Meyrowitz, editor, *OOPSLA'86 Conference Proceedings: Object Oriented Programmng Systems, Languages and Applications.*, pages 214–223. ACM, 1986.

[14] H. Ogawa, K. S. S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. *Lecture Notes in Computer Science*, 1850:362–382, 2000.

[15] A. Oliva and L. E. Bizato. The design and implementation of guarana. In *COOTS'99 — Proceedings of USENIX Conference on Object-Oriented Technology*, San Deigo, California, USA, May 3–7 1999.

[16] K. Renaud and H. Evans. Javacloak: Engineering Java Proxy Objects using Reflection. In M. Weber, editor, *NET.OBJECTDAYS 2000. Messekongresszentrum Erfurt, Germany*, October 9-12 2000.

[17] I. Welch and R. Stroud. Dalang — A Reflective Extension for Java. Technical Report CS-TR-672, Computing Science Department, University of Newcastle Upon Tyne, September 1999.

[18] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. *Lecture Notes in Computer Science*, 1616:2–21, 1999.

[19] I. Welch and R. J. Stroud. Kava — using byte code rewriting to add behavioural reflection to java. In *COOTS '01 — Proceedings of USENIX Conference on Object-Oriented Technology*, San Antonio, Texas, USA, January 29 – February 2 2001.

[20] Z. Wu. Reflective java and a reflective-component-based transaction architecture. In J. C. Fabre and S. Chiba, editors, *Proceedings of OOOPSLA'98. Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, 18–22 October 1998.