JavaCloak : Engineering Java™ Proxy Objects using Reflection

Karen Renaud & Huw Evans
University of Glasgow
{karen,huw}@dcs.gla.ac.uk

Abstract.

Java programmers need to be able to locally specialise the run-time behaviour of externally developed code in order to increase software reuse. We describe JavaCloak, a system that supports local specialisation by using proxy objects. The proxy object and the object it wraps (the wrapped object) are typed with the same full-qualified Java class name and a specialised classloader is used by the proxy to ensure the two classes are kept separate. Many issues which arise when using proxies in object-oriented languages, such as the "self problem" and encapsulation problems, are discussed, together with Java-specific problems. The current design and architecture of JavaCloak are described in detail.

1 Introduction

Java software developers need to be able to specialise the run-time behaviour of externally developed code to meet local needs. The original developer cannot identify every possible use of their software and so the local Java developer should be able to specialise its use as part of the run-time configuration of *their* system. When a program is run, we would like to be able to augment its behaviour to include the execution of code we define locally. Giving ourselves control in this way allows us to add arbitrary behaviour to the program, to support system instrumentation, for example. However, in order for the program to retain the correct semantics, it must be able to access and execute all the code that was originally written. As the class is the fundamental building block of a Java program, we would like to be able to change program behaviour on a class by class basis. This will give the programmer the most flexibility.

Such run-time specialisation can be achieved in Java [6] in a number of ways: by changing the Java source code; by post-processing the Java bytecode; by providing a custom classloader; and by providing proxy objects. Changing the Java source code is only an option when the source code is available — which is not likely to be the case when code has been obtained from another organisation and is not a general solution.

Other researchers have investigated tools and techniques that allow Java bytecode to be changed so that access to the source code is not required. These tools can be divided into two categories, those that support generic bytecode rewriting [3, 1] and those that rewrite bytecode for a particular situations, such as to provide a meta-object protocol [12] or to enable dynamically adaptable code at run-time [7].

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 it is compatible with the Java Object-Serialization mechanism so that instances of the rewritten class can be passed across a network or written to disk. However, bytecode rewriting has a number of disadvantages. Firstly, the changes to be made to the bytecode should be recorded somewhere so that they can be re-applied if a new version of the class is compiled. In addition, systems that require bytecode to be rewritten, such as ObjectStore's PSE Pro [2], may burden the programmer with managing two sets of classes. For example, one class (`A`) may depend on another class that has to be post-processed (class `B`). 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. Thirdly, bytecode rewriting by-passes the Java compiler which enforces the Java language semantics. This allows a programmer to get access to all the fields of an object, even if some were marked as `private` in the original source code.

Meta-object protocols [8] are a powerful programming paradigm for associating new behaviour with a program. This is expressed by separating the object-level from the meta-level in an object-oriented programming language. The object-level implements the semantics of the application, whereas the meta-level deals with the language-based semantics of *implementing* that application. For example, for a word processing application, a class at the object-level might be `Paragraph` that handles the manipulation of paragraphs of text. At the meta-level, a programmer can define a binding between the object-level class (`Paragraph`) and a meta-level class, e.g., `MetaParagraph`. The supporting language then delegates to the meta-level any operations that are performed over the `Paragraph` *class*. This includes operations such as method dispatch, field access and object creation. When a method is executed on a `Paragraph` object, the `MetaParagraph` object is checked with first to see how this method should be handled. The meta-level can arrange for another method on a different object to be called instead, thus affecting the behaviour of the program at run-time. The authors of [12] have defined a meta-object protocol for Java that requires bytecodes to be rewritten and so it is subject to the problems identified above. Their system rewrites bytecode at run-time and so it imposes a run-time overhead on the program. This could be optimized to perform the post-processing once, before the program is run, however, in this case, all the classes that the program could possibly use would have to be identified. It is also not clear if the meta-object protocol only allows the addition of new behaviour to a class or if a class may be completely substituted by another one.

The system described by Keller and Hölzle in [7] is targeted at integrating Java classes with other, non-compatible classes. The classes are made compatible by the programmer identifying which class should have its bytecode modified at run-time so as to ensure that the classes can inter-operate. This system also relies on bytecode rewriting, however, their focus is slightly different to other similar bytecode rewriting systems. Keller and Hölzle's system ensures that incompatible classes can be used together by adding, renaming or removing methods, or by changing the class hierarchy.

The Java platform allows the programmer to define custom classloaders so that a program can manage *how* and *from where* classes are loaded. However, certain restrictions are placed on the programmer. Firstly, Java's class equivalence is based on both the name of the class and the identity of the classloader that loaded the class. Given this definition, two classes with the same name (but possibly different implementations) can be loaded by two different classloaders in the same virtual machine. However, if the program uses one class definition in a context where the other class definition is expected, a `java.lang.ClassCastException` will be raised. Our solution uses a specialised classloader, and extreme care has to be taken to ensure that the two sets of classes are kept apart (see section 3.1).

Given the restrictions imposed by the above approaches, we have adopted a solution that exploits proxy objects [4] and the use of a specialised Java classloader. A proxy[1] object is a surrogate object which controls access to the wrapped object by forcing all interaction with the wrapped object via the proxy. We assume that the class to be wrapped has been externally developed, that its source code is not available and that its bytecode is not to be changed. JavaCloak provides a proxy object that has the same fully-qualified classname as the wrapped object. Thus, when a program instantiates an object of the wrapped type, an instance of the proxy is really created, which then instantiates the wrapped object via the specialised classloader. As JavaCloak generates Java source code for the proxy, the programmer is able to alter the run-time behaviour of a wrapped object by changing the code in that object's proxy.

Version 1.0 of JavaCloak as described in this paper, was designed and tested using "classic" version 1.1.5 of Sun Micrsosystem's JDK implementation. The rest of the paper is organised as follows. Section 2 gives an overview of our approach. Section 2.1 shows how a custom classloader was provided to support this system. Section 3 describes how the proxy classes are generated and Section 4 discusses how the proxy classes and the wrapped classes interact. Section 5 describes some pitfalls with the approach and Section 6 describes some uses for JavaCloak. Section 7 describes future work and Section 8 summarises the paper.

[1] Proxy objects are some times referred to as wrappers, however, in this paper we adopt the naming convention of proxy and wrapped objects.

2 System Overview

In this system we would like to be able to execute the code given in the `Run` class (below), which makes use of the class `FString`, a new implementation of a string class. Both these classes have been written externally and we would like to be able to alter the behaviour of the program at run-time by providing an alternative implementation of the class `FString`.

```
package application;
public class Run {
  public static void main(String[] argv) {
    FString str = new FString();

    char[] c = null;
    c = str.update(
           new char[] { '4', '3', '2', '1'} );
    System.out.println(c);
  }
}

package wrapped;
public class FString { /* Original Version */
  char[] data;

  public FString() {
    /* Initialise the data to some known value
       for this example. */
    data = new char[] { '1', '2', '3', '4' };
  }

  public char[] update(char[] c) {
    char[] result = data;
    data = c;

    return result;
  }
}
```

If we execute `Run` outside the environment, `str` is created and the character array `'1', '2', '3', '4'` is initialised. The `update()` method is then called, initialising `data` with a new array of characters characters (`'4', '3', '2', '1'`) but returning the old array. Therefore, when `Run` is executed, `1234` is printed on standard output. When executed inside the environment, the simplified version of the `FString` proxy below is substituted. In section 4, the proxy code will be explained in more detail

```
package wrapped;
public class FString { /* Proxy for FString */
  Object wrappedObject;
  static Class FStringClass =
    WrappingClassLoader.wcl.loadClass(
                    "wrapped.FString");

  public FString() {
    wrappedObject = FStringClass.newInstance();
  }

  public char[] update(char[] c) {
    char[] r = null;

    System.out.println("update() called");

    Method m = null;
    m = FStringClass.getMethod("update",
          new Class[] { char[].class } );
    r = (char[]) m.invoke(wrappedObject,
          new Object[] { c } );

    System.out.println("update() exiting");

    return r;
  }
}
```

Note that name of the proxy class and its package is exactly the same as the original. This is crucial to having this system work. If either of these is different, the externally developed code would not work in the environment. The code in `Run` expects the class `wrapped.FString` to exist and so the fully-qualified package and class name must be used.

The above proxy must provide a public method with exactly the same signature (number and type of parameters, result type and number and type of exceptions) as that provided by the original class. However, internally it can define other `protected` and `private` methods and other fields. The proxy must provide all the methods that the original `FString` defines, including all the public methods of the superclasses it inherits from.

It is possible for the original definition of `FString` to contain non-static fields that are `public`. Code will rely on these public fields being present in the proxy. However, the wrapped code may make use of the public fields within its methods and so it is not feasible to have copies of the fields in the proxy. In this approach we therefore assume a clean object model where there are no `public` non-static fields defined on any class that is being wrapped.

Some Java programs rely heavily on defining `public static` fields in classes to provide global access to object state, for example, `System.out`. This is a

similar problem to that described above. It is not possible[2] to hold a copy of the `static` field in the proxy as the wrapped class may need access to it. As Java does not model field access as method invocation, there is no opportunity in this approach to redirect the access to the static field via the proxy. Therefore, we assume that static fields are always accessed via a `static` method defined on the wrapped class. This then provides the necessary support to call the method on the proxy. Dealing with `public` fields is an area for future work (section 7).

Once the proxy has been compiled, it must be installed in the environment and the `CLASSPATH` changed to load the proxy class when the `FString` class is loaded for the first time. The proxy `FString` will be loaded by the default classloader. The code inside the proxy then explicitly loads the wrapped `FString` class using the `WrappingClassLoader`.

The call to the `loadClass()` method defined on `WrappingClassLoader` takes the string `wrapped.FString` as an argument. This classloader loads the original definition of `FString` from a location that was supplied to the classloader when the environment was run. This is passed via a run-time variable `-Dwrappedpath` which contains a list of directories for the classloader to search to load the wrapped classes.

When an `FString` object is created, the above constructor initialises `wrappedObject`. As `wrappedObject` is of type `java.lang.Object` it is possible to create an instance of the original `FString` object inside the `FString` proxy class. A call to the `update` method defined on the proxy uses the `java.lang.reflect.Method` object to retrieve the `update` method on the original class. The reflected method is then used to call `update()` on the original class, passing the parameter `c` and returning the resulting character array `r`.

2.1 The Class Loader

When a JavaCloak system is started, a runtime Java parameter `wrappedPath` is passed to the virtual machine. This information is used by the `WrappingClassLoader` to find the wrapped class. The JVM internal class loader loads each proxy class once, when it is encountered for the first time. The proxy then explicitly loads the wrapped class using the `WrappingClassLoader`. This maintains two separate class spaces. Any attempt to use a proxy class where a wrapped class is expected results in a `java.lang.ClassCastException` being thrown.

3 Generating the Proxy Classes

The first step in providing this tool is to generate the Java source code for the proxy classes. In order to make this scheme viable, proxy source code must be generated automatically using the Java reflection package. The generated

[2] In the case where the `static` field is defined as `final`, a copy can be made as the field cannot be updated. This assumes it is possible to initialise the field in the same way as the original.

proxy class must be type compatible with the original classes and present the same interface to the application as the wrapped classes. This means that they must provide the same constructors, methods, and public fields. Using the Java introspection mechanism, a full description of the class can be derived, enough to generate a complete "look-alike" proxy class. A generator produces the proxy classes in four separate stages: the skeleton, the public fields, the constructors, and the public methods. The generation of the skeleton and fields is straightforward, but the generation of constructors and methods merits some discussion — which will be done in the following sections.

For the proxy to be type compatible with the wrapped class requires the proxy to have the same fully-qualified class name as the wrapped class, the proxy must support the same set of interfaces as the wrapped class, and the proxy must inherit from a type-compatible superclass, if the wrapped class uses one. Ideally, a programmer should not be able to discover, via reflection, that they are using the proxy and not the wrapped class. However, this cannot be achieved in Java as the reflection mechanism allows a program to discover the names of the class' fields. In future work we hope to explore in more detail the circumstances under which a proxy cannot be substituted for a wrapped class.

3.1 Constructors

A proxy constructor is generated for each constructor in the wrapped class. The constructor establishes a link to an instance of the wrapped class, and maintains a reference to this object to be used in subsequent method invocations, as shown for class `FString` below. This seems straightforward, but there is a complication. The problem is that Java does not allow two different classes with the same name to be used within the same context, and that is precisely what this scheme proposes to do. To circumvent this, a specially written classloader, `WrappingClassLoader` (see section 2.1), is used. The constructor calls the `loadClass` method in the `WrappingClassLoader` to load the wrapped class object (`FStringClass` below). The constructor then creates an instance of that class using the `newInstance` method[3] on the class object to initialise `wrappedObject`.

```
Object wrappedObject;
static Class FStringClass =
       WrappingClassLoader.wcl.loadClass(
                           "utils.FString")
public FString() {
  wrappedObject = FStringClass.newInstance();
}
```

It is important to note that `wrappedObject` is referred to as type `Object`, and not as type `FString`. If `FString` were used here it would refer to the proxy

[3]This only works for constructors with a null parameter. The reflection package is used to handle any parameters passed to a constructor.

as this has been loaded via the default classloader. It is the proxy that then explicitly loads the wrapped class.

Within the proxy world, shown in the previous code segment, the notion of `FString` is the proxy class `FString`. Thus, `wrappedObject` is just an object, a reference into the separate world of the wrapped `FString` which happens to have the same fully-qualified class name as the *proxy* `FString`. You will also note that the `FStringClass` variable is declared as static. This variable describes the class of the wrapped object. This is currently used when invoking methods on the wrapped object. Only one instance of this variable is needed for all proxy objects of a specific class, since the wrapped object class is the same for all instances of a specific proxy class.

Another important detail that must be addressed is caused by the inheritance model of Java. In the proxy class constructors shown above, *each* constructor will create a reference to a wrapped object. Since no-argument superclass constructors are automatically invoked when an instance of a subclass is instantiated, this leads to a situation as shown in Figure 1.

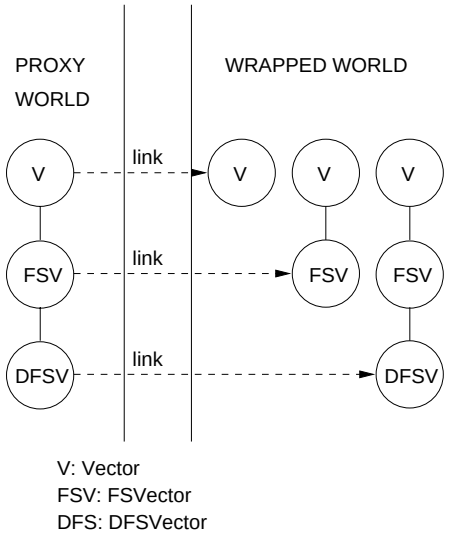


Fig 1. Erroneous Linking

This is prevented by explicitly invoking a proxy superclass constructor which does *not* create a link to the wrapped object. The link should only exist in the object actually being instantiated, not in the superclasses. There is a difficulty in creating such a constructor which will work for all objects. A "proxy" constructor must be created which does not duplicate any possible constructor which already exists for the class. This means it must behave as a sort of *dummy* constructor. In providing such a constructor it is essential that the existing constructors of the class should not be duplicated.

The constructor given below that takes a `Class` as a parameter does *not* create a link to a `wrappedObject`, and is *explicitly* invoked when an instance of a subclass is created, preventing implicit invocation of the null parameter constructor. This prevents the automatic invocation of one of the other constructors, which *would* create a new wrapped object. The correct structure is shown in Figure 2. For example:

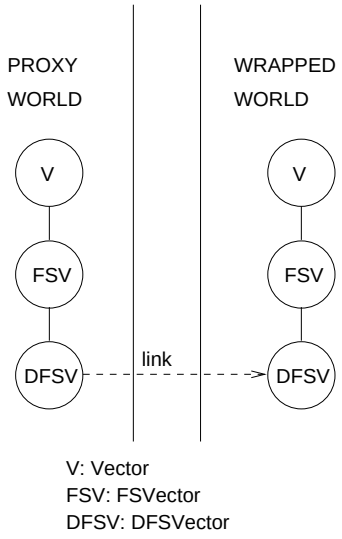


Fig 2. Linking the two worlds

```
Class FSVectorClass =
        WrappingClassLoader.wcl.loadClass(
                        "utils.FSVector");
public FSVector (Class cls) {
  // Do not create a link here
}

public FSVector()
{
  super(FSVector.class);

  wrappedObject = FSVectorClass.newInstance();
}
```

3.2 Methods

A method is generated for each `public` method in the wrapped class. This method invokes the same method on the wrapped object. For example:

```
public void displayString() {
  Method m = FStringClass.getMethod(
              "displayString", new Class[] { } );

  m.invoke(wrappedObject, new Object[] {} );
}
```

Firstly, a reference to a method object (`m`) is obtained that reflects over the method `displayString()`. The method is then invoked on `wrappedObject` by calling `m`'s `invoke()` method. Since this method receives no parameters, the formal parameters for the `getMethod()` call and the array of method invocation parameters is empty.

This is the basic mechanism. However, it has to be made more elaborate to pass method parameters in and to receive any method results. The implementation of the `setName` method in the proxy is as follows:

```
public void setName(String name) {
  Method m = FStringClass.getMethod(
              "setName",
              new Class[] { String.class });

  m.invoke(wrappedObject,
           new Object[] { name } );
}
```

The `setName` method on the proxy receives `name` as a parameter, the reflected `setName` method is retrieved and it is called on `wrappedObject`. The `name` reference is passed as a parameter to the reflected method in the `Object[] name` array. As the class `String` has the same meaning for both the proxy and the wrapped class, this is safe. When a proxy class instance is passed to a proxy method, it has to be unwrapped before it is passed on to the wrapped instance.

3.3 Passing an Instance of a Wrapped Class

In our approach, the proxy class and the wrapped class are implemented separately. This leads to two problems, the "self problem" [9] 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 class. If all references to `this` in the wrapped object are considered to really be references to the proxy, then this is a delegation model, where all field and method accesses within the wrapped class are delegated to the proxy which delivers them to the wrapped instance. If all references to `this` in the wrapped instance are not considered to be references to the proxy but to the wrapped object, this is a forwarding model as only the initial method invocation on the proxy has been forwarded to the wrapped object.

In [12] the authors present two forms of the encapsulation problem. When implementing the proxy and the wrapped class separately the wrapping is only

logical, in the sense that, if the wrapped class can be referenced independently of its proxy, an instance of the wrapped class can be created without creating a wrapping proxy. In our solution, the proxy class and the wrapped class have the same name; no renaming is required as in [12]. In our approach the main run-time difference between the proxy class and the wrapped class is that the proxy class has been loaded by the `WrappingClassLoader`. One way for a programmer to get a direct reference to the wrapped class would be to write their own classloader that explicitly loaded it. To do this, they would have to know the location of the wrapped classes. As this information is not exposed by our system, they could not discover this information by writing a Java program. The wrapped instance can break the encapsulation provided by the proxy by returning a reference to itself (via `this`) within another object. This other object can then be used to extract the direct reference to the wrapped object. This is possible in Java because `this` in the wrapped object refers to the wrapped object and not the proxy.

3.3.1 Passing References to a Wrapped Instance

The encapsulation problems are discussed with reference to the `addElement` method on the `FSVector` proxy below:

```
public Object wrappedObject;
static Class FSVectorClass;

public void addElement(FString name) {
  Method m = FSVectorClass.getMethod("addElement",
              new Class[] { FString.class} );
  m.invoke(wrappedObject,new Object[] {name});
}
```

The references `myString` and `SV` below refer to instances of the proxy classes. When the application calls `addElement`, an instance of the `FString` proxy class is passed as a parameter. This object is not understood by the wrapped class. Indeed, if it is sent, the wrapped class and the proxy class will cause a `java.lang.ClassCastException` to be thrown as the two different classes are being used in the same context.

```
FString myString = new FString();
FSVector SV = new FSVector();
SV.addElement(myString);
```

The proxy object should not be sent, since the wrapped objects only understand other wrapped objects. A solution is to unwrap the proxy, and pass through its `wrappedObject` references. For example, the preceding implementation of `addElement` would be changed to:

```
public void addElement(FString name) { // Proxy FString
  Method m = FSVectorClass.getMethod("addElement",
                new Class[] { name.FStringClass });
    m.invoke(wrappedObject,
              new Object[] { name.wrappedObject });
}
```

In our approach, we have to remove the proxy when calling forward to a method that expects an instance of a wrapped class, because of the Java type system. Therefore, we cannot support a delegation model as described above and are forced to adopt a model based on forwarding. This means that any use of `this` inside the wrapped class will be a direct reference to the wrapped instance and not to its proxy. We have, therefore, lost the ability to associate the proxy with the wrapped instance while the wrapped method is being executed. However, we can reassociate the wrapped object with a proxy assuming this unwrapped reference is passed back to the application via one of our proxies.

If a wrapped method returns an unwrapped instance back to the proxy, the proxy and its `Translator` return to the caller the proxy for that wrapped object. The next section outlines the design of the `Translator` which provides a safe "bridge" between the two sets of classes.

4 Integrating the Two Type Spaces

The previous discussion has shown that some form of translator is required to keep the *proxy* classes and the *wrapped* classes separate. This is done by providing a `Translator` object for each class of proxy. When a method call is received at the proxy, its `Translator` is used to handle the method invocation. This ensures the two different sets of classes are kept separate as the `Translator` only ever refers to the wrapped classes via the reflection mechanism.

By using a translator object, the complexity can be removed from each individual proxy instance and contained within one translator for each class in the wrapped package. It maintains two tables, one to match the corresponding objects in the different sets. The other static table, shared by all instances of the `Translator` class, keeps track of other translators. The table structure is shown in Figure 3. The translator object needs to know about other translators to enable wrapping and unwrapping of parameters and return values.

The `Translator` is used by a proxy during three main phases: proxy instantiation; method dispatch; and at proxy garbage collection time. A `Translator` is created should a proxy require it, otherwise the `Translator` is shared between all instances of a particular type of proxy. When a method is dispatched, the proxy passes to its `Translator` information relating to the method being called. The `Translator` then calls the method on the wrapped object, unwrapping any parameters before the wrapped method is called. The `Translator` also ensures any parameters that are passed from the wrapped method are mapped to the proxy for that object. When the application drops all references to the proxy, the virtual machine calls the object's `finialize` method. This method is over-

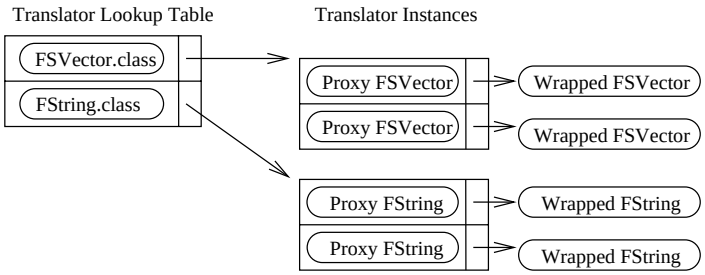


Fig 3. The Translator Tables

ridden so that the applicable entry can be removed from the instance table to ensure the correct garbage collection semantics.

5 Pitfalls

This mechanism does not work for packages which use platform-dependent native code. This is because native classes which are loaded from operating system libraries must be load by the JVM's default ClassLoader. This means it is not possible to separate the two sets of classes and the system fails. This can sometimes be overcome if the package has some sort of mechanism which allows the use of a proxy class, for example, the `java.awt` package and this is discussed in more detail in [10].

Interfaces need to handled in a special way as an interface proxy cannot be created. A class may implement an interface, and pass itself as a parameter to a method on one of the wrapped classes. This causes problems because the classloader uses to load the wrapped class may differ to the classloader used to load the interface. If this is true, a `java.lang.ClassCastException` is raised. Currently, JavaCloak overcomes this by generating a special class that implements the interface, which then operates only with the *wrapped* instances.

Finally, the technique described in this paper manipulates the `CLASSPATH` by placing the proxy classes at the very start. The `CLASSPATH` mechanism is used differently in JDK 1.2, which expects to find any class with a name starting with `java`, in a well-known place, i.e., within the `rt.jar` file provided by Sun. Therefore, it may not be possible to provide proxies for these classes in the same way as described in this paper.

6 Uses for JavaCloak

JavaCloak can be used to provide proxies for many different purposes. The motivating use and one that has influenced the design is the need to report on the use of a specific class, for instrumentation and logging purposes, placing the extra functionality into the proxy.

Proxy objects can also be constructed to support remote access of the wrapped object. A JavaCloak proxy could be provided by the adapting programmer to communicate with a remote object. Therefore, the wrapped object does not have to be changed to support distributed computation as all of this complexity can be placed into the proxy.

Proxy objects are useful whenever the requirements of local specialisation cannot be directly met by the given class and tools, such as bytecode rewriters, are not appropriate.

7 Future Work

As currently defined, a proxy class defines a `static Class` reference which represents the wrapped class at run-time. When a proxy method is invoked, this class is used to retrieve a reflective `Method` object which is used to call the wrapped method. Every time the `getMethod` call is made on the `Class` object, a new instance of the method object is returned, when in fact, only one method object per method is required. This can be achieved by creating a reflected method object for each public method in the state of the proxy. This removes the need for the `static Class` reference representing the wrapped class.

Java programmers like to use `public` references for `static` and `non-static` fields in their classes. In the case of `non-static` fields, it can be argued that this is bad object-oriented programming as encapsulation is broken. A pair of `get()` and `set()` methods are likely to be inlined by the Java compiler or a JIT compiler, therefore removing the run-time overhead of using the method. Controlling access to `static` references via a `static` method is also better object-oriented programming. However, we need to be able to deal with the case when direct access is made and this is an area for future work. Keeping the state of the proxy synchronized with the state in the wrapped object is a challenge and Java may not provide enough support for this at the language level.

The use of JavaCloak together with Java inner-classes, subtyping, exceptions, interfaces, programmer supplied classloaders, the Java 2 security model [5] and Java RMI [11] are areas for future work.

We are currently investigating the release of this system as a tool for Java programmers using Java 2 (JDK 1.2). This will be done after the system described in this paper has been improved and optimised as some operations in the current proxy structure are unnecessary. For example, reflecting over the method of the wrapped class only needs to be done once, within the proxy constructor and not each time the proxy's method is called as has been described here.

8 Summary

We have described JavaCloak, a system for changing the behaviour of objects at run-time via proxy objects. JavaCloak eliminates the need to make changes

to application source code, or its bytecode, as the change in behaviour is encapsulated in separate proxy objects. This means the semantics of the original classes are unaffected and the changed behaviour can be easily removed by simply not using the proxies. JavaCloak requires code in the proxy to provide the new behaviour and the proxy classes and the wrapped classes are kept separate by the use of a specialised classloader that is aware of the location of the wrapped classes and manipulates these classes purely through the Java reflection mechanism.

Acknowledgements

Karen Renaud is supported by a scholarship from the Association of Commonwealth Universities and this work is also funded in part by a grant from the Foundation for Research and Development in South Africa, and the University of South Africa. The authors thank Tony Printezis for reading an earlier draft of this paper.

References

[1] 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.

[2] E. Corporation. ObjectStore Enterprise Edition Homepage. `http://www.object-store.net/objectstore/`, May 2000. Web Document.

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

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

[5] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation.* Addison-Wesley, Reading, MA, USA, 1999.

[6] Java 2 Platform, Standard Edition. `http://java.sun.com/j2se/`, Sun Microsystems, January 1998.

[7] 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.

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

[9] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In N. Meyrowitz, editor, *OOPSLA'86 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 214–223. ACM SIGPLAN, ACM Press, 1986.

[10] K. V. Renaud. Tracking activity at the user interface in a Java application. Technical Report TR-1999-33, Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, G12 8RZ, April 1999.

[11] Java™ remote method invocation specification. Technical Report, Revision 1.5, JDK 1.2, `http://java.sun.com/products/jdk/rmi/`, Sun Microsystems, Inc, Oct. 1998.

[12] 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.