Experience with statically-generated proxies for facilitating Java runtime specialisation

K. Renaud

Abstract: Issues pertaining to mechanisms which can be used to change the behaviour of Java classes at runtime are discussed. The proxy mechanism is compared to, and contrasted with other standard approaches to this problem. Some of the problems the proxy mechanism is subject to are expanded upon. The question of whether statically-developed proxies are a viable alternative to bytecode rewriting was investigated by means of the JavaCloak system, which uses statically-generated proxies to alter the runtime behaviour of externally-developed code. The issues addressed 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 JavaCloak proxy mechanism places on the system.

1 Introduction

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

1. Tailoring source code [1].
2. Providing a customised Java virtual machine (JVM) [2–4].
3. Using wrapper/proxy objects. The wrapper/proxy pattern was identified by Gamma *et al.* in their seminal book on design patterns [5]. Dynamically-generated proxies were used by Welch and Stroud in their Dalang system [6].
4. Bytecode engineering:
   (a) Pre-load-time: Providing bytecode rewriting tools to allow programmers to make changes to class bytecode [7,8].
   (b) Load-time: Providing a custom 'classloader' which integrates a meta-object protocol (MOP) with the original bytecode. (The 'classloader' loads classes as required by an application. The JVM allows a programmer to extend this class to specialise its behaviour.) These approaches are illustrated in the Javassist and Kava systems [9, 10].

Each of the above approaches has both advantages and disadvantages and we intend to discuss the pros and cons of these mechanisms. Many of the approaches have been tried and tested but the effects of one, namely that of statically-generated proxies, has not yet been quantified. This research was done to investigate both the viability of using statically-generated proxies for runtime specialisation, and the suitability of Java for expediting such specialisation. To this end the JavaCloak system was developed, using Java. JavaCloak uses statically-developed proxies to alter system runtime behaviour.

© IEE, 2002
IEE Proceedings online no. 20020909
DOI: 10.1049/ip-sen:20020909
Paper received 18th July 2002

The author is with the Department of Computing Science, University of Glasgow, Glasgow, Scotland, UK

2 Runtime specialisation

Source-code tailoring cannot really be considered to be a runtime specialisation technique although it is often used as a mechanism for post-implementation adjustment of behaviour. 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 insert code inconsistently throughout the program.

Customised JVMs provide 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.

The latter runtime specialisation mechanisms, bytecode engineering and proxies, will be discussed in the following two Sections.

2.1 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 three categories: (i) those that support bytecode rewriting [7, 8]; (ii) those that provide a MOP [11]; and (iii) component evolution toolkits such as BCA that dynamically adapt code at runtime [12].

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's object Serialisation 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. The changes must be applied every time a class requiring it is recompiled (to ensure the rewriting has been correctly applied to the newly generated bytecode) and there is the extra effort of determining whether or not a new version requires amending.

In addition, systems that require bytecode to be rewritten, such as ObjectStore's PSE Pro [13], may burden the programmer with the management of two sets of classes. For example, one class B may depend on another class A that has to be post-processed. Class A should therefore be compiled first and post-processed before class B is compiled. Tracking these kinds of relationships for significant bodies of code is a non-trivial problem. This kind of bytecode rewriting is simply performed at too low a level of abstraction. A higher level of abstraction is required and thus the MOP approach has been developed [10].

MOPs [14] are a powerful programming paradigm for associating new behaviour with a program. MOPs can be used to apply either behavioural or structural reflection. Behavioural reflection changes only the behaviour of the system at runtime. Structural reflection allows a program to change, at runtime, the definition of a class, a function or a record. The authors of [11] have defined a MOP for Java that rewrites bytecode at load time to achieve behavioural reflection. 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 [11]. Kava also supports overriding of field access and allows exception handling to be intercepted so that exceptions can be overridden.

Javassist [9] is a bytecode rewriting tool based on structural reflection. It also makes use of a MOP. However, as pointed out by Welch and Stroud, it does not support reflection on methods inherited from superclasses [10]. This problem has been solved by Kava.

The last category, component evolution toolkits, is illustrated by the system described by Keller and Hölzle in [12]. It is targeted at integrating Java classes with other, non-compatible, classes. This system is based on the dynamic adaptation of bytecode as the class is loaded into the JVM. The programmer identifies the class which should have its bytecode modified 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 potentially 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.

2.2 Proxies

Our definition of proxy (note that in this work the terms 'proxy' and 'wrapper' are synonyms) used in this paper accords with that of Gamma *et al.* [5]. Proxies may augment, or report on, the behaviour of the original classes, applying purely behavioural reflection. Most runtime specialisation is done in order to add non-functional properties to the system, so that the functionality of the original class will often still be required. Thus proxies will probably be required to invoke methods on instances of the original, matching classes. Proxies can be set up at different times:

1. Compile-time: Such proxies are generated statically, compiled and then loaded by the JVM instead of the original classes. To use compile-time proxies the following two problems have to be overcome:
   - Dual existence: It is simple enough to instruct the JVM to load the proxy class instead of the base-class by changing the location in the 'classpath', but this will only work if the proxy class has the same fully qualified class name as the base-class. However, if the two classes have the same name, the tricky part is the loading of the base-class once the proxy class has been loaded since the JVM will not see the need to reload a class definition once it already has one for a particular class.
   - Bridging: If the previous problem is overcome and the JVM is somehow tricked into loading two different definitions for the same class name, the following problem is that of allowing the proxy to access the base classes. The JVM generally expects a class definition to be unchanging. Any use of the two different class definitions in the same context will routinely generate a 'ClassCast' exception.
2. Load-time: Such proxies are generated on-the-fly by providing the JVM with a customised 'classloader'. This 'classloader' generates the wrapper and provides it in place of the original class [6]. The substitution is thus done when the application requests an instance of a class that is being wrapped, causing a different definition of the class to be loaded. Welch and Stroud have discussed the limitations of load-time generated proxies in [6].

One area where proxies do work well is in the 'wrapping' of interfaces. The Java 2 platform (version 1.3) defines dynamic 'Proxy' classes [16] to do this. These classes implement a list of interfaces invoked at runtime when the class is created. However, a 'Proxy' class has some limitations which the user will have to bear in mind:
   - Dynamic proxy code inherits from 'Proxy' when it is generated within the 'classloader', which means that such proxies cannot form part of another inheritance structure in a single-inheritance language such as Java.
   - A programmer-defined invocation handler must be used to dispatch the method invocation to the wrapped instance at runtime, requiring the skills of an expert programmer.
   - The delegation to proxies is restricted to interface methods. Field and exception access is thus not possible using these proxies. This is a distinct limitation.
3. Runtime: This can only be done by means of a reflective JVM [16], which allows the already-loaded bytecode to be altered so that a proxy can be substituted for the original class.

In cases where very simple behavioural changes are required it may be that proxies are less intimidating to the average programmer than MOPs. The JavaCloak system [17] was an experiment to investigate the possibilities of statically-generated proxy classes. JavaCloak has worked well for simple cases. However, some implementation challenges were encountered and these will be discussed in Section 4. The following Section will introduce the JavaCloak system.

3 The JavaCloak system

JavaCloak does not require access to base-class source code and generates proxies using the 'java.lang.reflect' package. The proxies intercept all method invocations to the base-class and delegate calls to the base-class. The programmer can customise the proxy source code so as to change the runtime behaviour of the class that the proxy class wraps up. The preceding Section mentioned two problems related to the use of statically-generated proxies. These have been overcome in JavaCloak as follows:

1. Dual existence: One runtime problem that JavaCloak has 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 'classloader' that loaded the class definition. If the two classes are loaded by two different 'classloaders' the JVM considers them to be different classes, even though the fully-qualified class names are the same.

2. Bridging: Once the two classes are in the JVM it is essential that instances of the two classes be kept strictly separate, so that they function within their own context, as it were. If instances of these two classes are used in the same piece of code, e.g. one instance is passed to one of the methods of the other instance, a 'ClassCast' exception will be raised. JavaCloak uses a runtime manager to maintain a strict separation between the two types of objects. This is achieved by storing the type of the original class, by name only, in a character string and manipulating it via the reflection mechanism defined in 'java.lang.reflect'.

The key enabling features of JavaCloak, shown in Fig. 1, will be described below:

1. Proxies: Proxies are generated for the base-class by means of the 'java.lang.reflect' package. The programmer can then specialise the runtime behaviour of the class in a finely-grained and flexible manner. (Shown as PC in Fig. 2)

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. So, for example, when the system is being run without the proxy, the programmer would start it from the command line as follows:

'java Application'

and when the system should run with the proxies, the programmer starts it as:

'java -Dconfig = run.cfg Application'

The 'run.cfg' file will tell the JavaCloak runtime manager where to find the base-class class files.

3. A customised classloader: JavaCloak makes use of a specially defined 'classloader', embedded within the runtime manager, to load the original classes at runtime (Shown as wcl in Fig. 2). 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 as shown above.

The JVM tests type-equivalence by comparing both the class name and the instance of the 'classloader' that loaded the class. Thus if two different 'classloaders' are used to load classes with the same name the JVM does not consider the classes to be identical.

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, consider the method call depicted in Fig. 3. Say the proxy PC_i invokes a method, M, and passes an instance of a proxy class, PC_j, as a parameter. The proxy class, PC_j, has no meaning to the original class, C_i, and, if passed to it, would cause the system to generate an exception. The JavaCloak runtime manager offers a facility for substituting original objects of type C_j so that such parameters are 'unwrapped' before they are passed to the method in the original class C_i.

In the same way the original object may pass a reference to a wrapped object, of type C_k, back to the proxies as a return value from M. The runtime manager offers a facility to substitute the proxy, of type PC_k, for such objects so that they do not cause 'ClassCast' exceptions in the application (which has no concept of the original class C_k).

An example will now be introduced to illustrate the concepts in the rest of this Section. Consider a bank that has a system for handling all accounts. Assume the 'SavingsAccount' class offers the interface given in code fragment 1.

```
public class SavingsAccount extends Account{
   public SavingsAccount(float interestRate) {}
   public SavingsAccount linkAccounts(SavingsAccount sa) {}
   public float getBalance() {}
   public void addInterest() {}
   public float getTaxableInterest() {}
```

Code fragment 1: SavingsAccount class

Consider now that new legislation comes into being which only charges tax on savings account interest where the account balance is more than £1000. Since this change only affects one method, 'getTaxableInterest()', of one class, 'SavingsAccount', in the system, the JavaCloak approach may be more suitable than more intricate and powerful mechanisms such as Kava. A proxy class is generated for the 'SavingsAccount' class. The 'getTaxable-Interest()' method is shown in code fragment 2.

```
public float getTaxableInterest(){
1: Object[] params = new Object[] {};
2: float result = 0;
3: try {
4: Object real_c = WM.wm.unwrap(this);
5: result = ((Float)methods[1].invoke(real c,params)).
   floatValue();}
6: catch(Exception ee) {deal with exceptions}
7: return result;}
```

Code fragment 2: Proxy SavingsAccount class

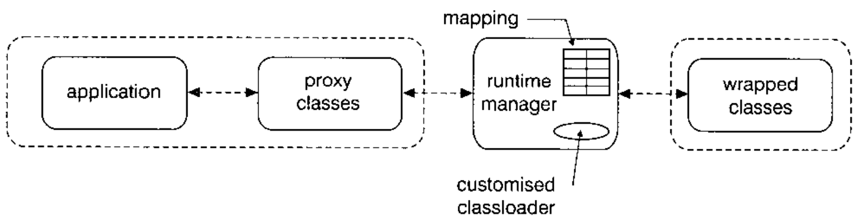

Fig. 1 *JavaCloak architecture*

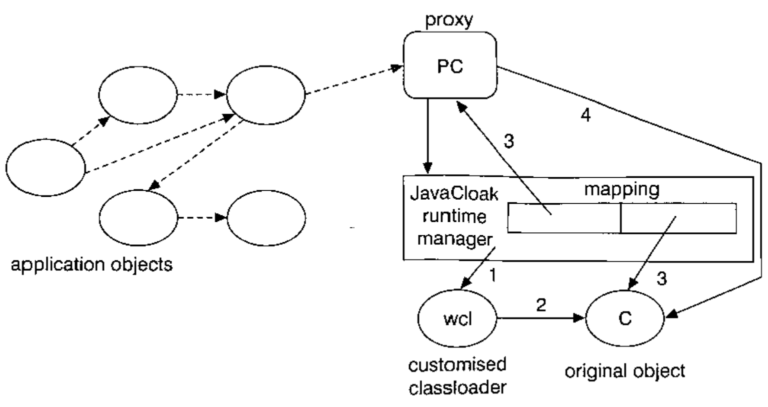

Fig. 2 *Interaction at runtime*

The required behavioural changes can be incorporated into this code by the programmer, a process that requires no additional programmer skills. In general terms 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. Specifically, to implement the new tax law the programmer adds one line to the 'getTaxableInterest()' method, line 1 in code fragment 3.

After the customisation of the proxy source code has been performed, and it has been compiled, instances of the proxy objects are created at runtime when the application instantiates an instance of the original class. The proxy code then delegates all calls made on its instance's 'public' methods to equivalent methods defined on the instance of the original class. The process becomes more complex when parameters and return values are involved, hence the inclusion of the JavaCloak runtime manager.

```
public float getTaxableInterest() {
1: if (getBalance() < 1000) return 0;
2: Object[] params = new Object[] {};
3: float result = 0;
4: try {
5: Object real_c = WM.wm.unwrap(this);
6: result = ((Float)methods[1].invoke(real_c,params)).float
Value();}
7: catch(Exception ee) {deal with exceptions}
8: return result;}
```

Code fragment 3: Adapted proxy SavingsAccount class

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

1. Uses the customised 'classloader' to load the original class definition from the location specified in the runtime variable. (lines 2 and 3 in code fragment 4).

```
public SavingsAccount(float p0){
1: try {
2: Wrappingclassloader wcl = WM.wm.getclassloader();
3: Class real = wcl.findClass("bank.SavingsAccount");
4: if(!this.getClass().getName().equals(real.getName()))
   return;
5: assignMethods(real);
6: Object[] params = new Object[] {new Float(p0)};
7: o = constructors[0].newInstance(params);
8: ref id = WM.wm.register(this, o);}
9: catch(Exception ee) {deal with exceptions}}
```

Code fragment 4: Proxy SavingsAccount constructor

2. Creates an instance of the original class. (line 7 in code fragment 4).
3. Inserts an entry into the mapping table linking the proxy to the original object. (line 8 in code fragment 4).
4. Returns a reference to the original object to the proxy. (line 8 in code fragment 4).

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

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 (line 1 in code fragment 5).
2. The proxy invokes the method on the original object. (line 5 in code fragment 5).
3. If the method returns a value the proxy receives the return value and stores it in a variable of type 'Object':
   (a) 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 object that invoked the method. (line 6 in code fragment 5).
   (b) Otherwise the return value is 'cast' to the correct type and returned to the application. (line 5 in code fragment 2).

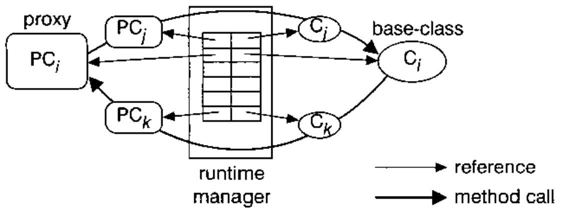

Fig. 3 *Unwrapping and wrapping proxy instances*

```
public SavingsAccount linkAccount(SavingsAccount p0){
Object[] params = new Object[] {WM.wm.unwrap(p0)};
SavingsAccount result = null;
try {
Object real_c = WM.wm.unwrap(this);
Object        out = (SavingsAccount)methods[0].invoke
   (real_c,params);}
result = (SavingsAccount)WM.wm.getWrapper(out);
catch(Exception ee) {deal with exceptions}
return result;}
```

Code Fragment 5: The linkAccounts method

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 JavaCloak proxies for specialisation. The term 'reflection' is generally used to denote the ability of a system to change its behaviour by examining certain properties at runtime and adapting the system depending on these properties. Some authors would refer to JavaCloak's activities as reflection, and others would refer to it as runtime specialisation. The authors of [6] use the term reflection to refer to their proxy-based approach, and their example will be followed from here onwards.

4 Proxy-related problems

During the development of JavaCloak some technical problems were encountered. 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 limits the use of the JavaCloak mechanism to non-system classes since these classes are loaded by the JVM automatically and not by means of a search of the 'classpath'. Changing the behaviour of system classes seems to violate the spirit of these classes and this is therefore not a serious difficulty. Other generic proxy-enabled reflection-related problems will be discussed in the following Sections.

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 runtime 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. The way JavaCloak maintains two identical classes is explained in Section 3. There are some other tricky problems related to this apparently transparent substitution, some of which have been solved, and others of which remain.

4.1.1 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 only via programmer-provided '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 of the original class directly. Given the current definition of Java, providing the application with access to 'public' fields consistently and transparently in the presence of JavaCloak proxies is not possible. This argument also applies to fields marked 'protected' and those that are scoped at the package level in Java.

In following the proxy approach, we need 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. There is no support at sourcecode level in Java to be able to implement the 'watchers', thus the only approach is to assume (and limit the programmer) to a clean object-oriented programming model.

4.1.2 Inheritance: This is a tricky problem for JavaCloak. JavaCloak proxies request the runtime manager to create a matching instance of the base-class in the proxy constructor. This is the logical place to do it. Inheritance causes a problem here. Consider the situation where JavaCloak provides a proxy for class B, which inherits from class A, which also has a proxy. If each constructor creates a link to a matching base-class instance there will be multiple links between the proxy world and the wrapped world, so that the inheritance structure, while correct in the proxy world, is amiss in the wrapped world.

To illustrate this point, consider the situation as shown in Fig. 4. We have a class A, which is extended by a class B, in turn extended by class C. The application instantiates an instance of C. Since the system loads the JavaCloak proxy rather than the original class, C is loaded. Since B and A are also wrapped, C's constructor will also create an instance of both B and A. If each constructor automatically

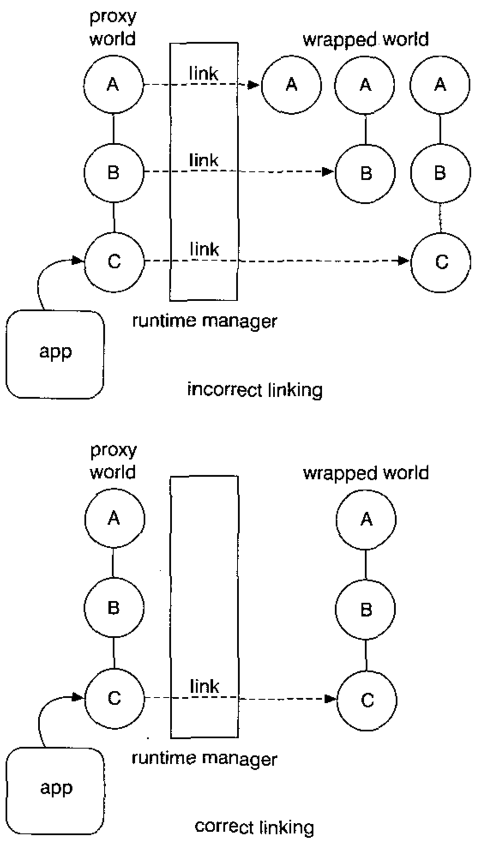

Fig. 4 *Inheritance across worlds*

creates an instance of the matching original class, the links will be as shown on the top of the Figure.

The multiple links will cause havoc when proxy objects invoke superclass methods, because any changes made by methods invoked on C, which make state changes, will not be reflected in the superclasses of the original objects, because the associated original objects are totally unrelated to one another.

The position should be as shown on the bottom, where the inheritance structures in the two worlds are totally unconnected, except at the initial explicit level within the instantiated proxy instance. JavaCloak must ensure that superclass constructors do not request the runtime manager to create matching original objects if invoked from subclass constructors.

A related problem concerns inheritance from proxy classes. If class C now has a subclass, D, 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 D invokes on the (proxy) superclass on the matching original class. The difficulty occurs when one considers methods that D defines. Should D also be provided with a proxy? The SOM approach, which makes use of meta-objects, requires D to have a related metaclass which is a subclass of the metaclass related to C [18]. They thus require any subclasses of 'wrapped' classes to also be 'wrapped'. These types of issues are not trivial to solve and a satisfactory solution is yet to be found.

4.2 Self and encapsulation

The JavaCloak approach implements the proxy class and the wrapped class separately. This leads to two problems, the self [19] and the encapsulation problem.

The self problem arises because the meaning of self (or 'this' in Java programs) is different in the proxy and in 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 [19] 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 [6].

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 of the return value 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 operation 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 serialisation does, then the proxy and the real object will be serialised, 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 that, at the JavaCloak implementation level, we cannot make use of these methods to manage the proxy. However, the JavaCloak management makes use of a 'java.util.Hashtable' which needs to be able to call the 'hashCode' and 'equals' methods on the proxy. This requires additional objects to be registered with the lookup mechanism which then operate as a place-holder for the proxy when performing a lookup on it, via the hash table. The objects themselves cannot be used, so these tokens represent the real objects in this case.

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 [6] cite a number of difficulties inherent in 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 note that some researchers feel that this type of action allows adaptive runtime redefinition of exceptional behaviour [20]. They argue that while exception handling should not be re-definable it might be desirable to add additional behaviour that takes place when an exception is raised at the base level.

Table 1: Overhead of calling through a JavaCloak proxy

Constructor	getBalance	addInterest	getTaxableInterest	linkAccounts
82.05 311.9	1.459 2.255	1.578 2.0875	1.4655 2.494	1.0125 3.365
Percentage increase: 280%	54%	32%	70%	232%

Table 2: Comparing the proxy approaches

	Wraps classes	Proxy inherits from	Programmer skills required	Overriding mechanism	Static or dynamic	Mediates field access
Dalang	Yes	Meta-object class	Average	Meta-object	Both	Yes
Java proxy	Only interfaces	Proxy	Expert	Invocation handler	Dynamic	n/a
JavaCloak	Yes	Base-class superclass	Fair to average	Changing proxy code	Static	No

Decisions about this are best left to the individual programmer and JavaCloak does not presume to dictate on this issue.

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 appraised of the problem in case the exception was caused by a bug in the reflection code. JavaCloak has dealt with this problem by incorporating a 'BugHandler' into the runtime system. If a JavaCloak-specific exception is detected in the proxy, a bug report is generated and the user is requested to email it to a given address.

4.5 Cost

This Section describes the typical runtime performance overhead of using a JavaCloak proxy. There is a cost related to behavioural reflection. Golm and Kleinöder identify the following costs which are affected by reflection [21]:

- Reflective method call: One extra level of abstraction is added here by JavaCloak. This also includes the cost of unwrapping of parameters before sending to the wrapped class and wrapping of returned objects.
- Installation: The costs related to making a method reflective. In JavaCloak the price is paid at compile-time.
- Memory: Two versions of each reflective class are held in memory, the proxy and the wrapped class. The runtime manager also maintains tables mapping these two worlds to each other. The JavaCloak proxies are fairly lightweight though, since they do not contain any class-specific variables or fields.

In addition there is an initial setup time to be considered. In the constructors JavaCloak reflects over the methods, and registers the proxy and the actual object with the JavaCloak runtime mechanism. JavaCloak also initialises the wrapped and proxy classes and essential JavaCloak runtime classes and maintains the mapping table in the runtime manager.

The major impact seems to convert mainly to time, at runtime, when it matters. JavaCloak's impact on the system needed to be measured. 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 code fragment 1 and a proxy for it was generated by JavaCloak. A small test program was written that created an instance of 'SavingsAccount' and invoked methods 'getBalance', 'addInterest', 'GetTaxableInterest' and 'linkAccounts' 100 times. An average of the time taken for each method was recorded. This program was run 20 times, both with and without the proxies, and an average of the elapsed time taken. All the figures given are in milliseconds 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.

The overhead at the constructor is as expected. 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 'SavingsAccount' is passed back from itself by the 'linkAccounts' method, we need to map this value to its corresponding proxy. Since 'SavingsAccount' actually returns an instance of itself that does not have an equivalent proxy instance JavaCloak has to generate a proxy instance for this return object and enters it into the mapping table before returning it to the caller.

It is obvious from the Tables that JavaCloak proxies introduce a substantial overhead. It may be that this overhead is unimportant as it would be if, for example, applications run at night when an extra few milliseconds are negligible. In real-time systems obviously this performance penalty could become an issue.

5 Conclusions

JavaCloak has demonstrated that it is possible to use statically-generated proxies to specialise runtime behaviour. During the development of JavaCloak many technical problems were encountered, some of which were solved. The other two proxy approaches, Dalang and the Java proxy approach, were compared to JavaCloak. Table 2 compares and contrasts the three major proxy approaches.

The Java proxy approach is not a serious competitor since it takes the easy route of only providing proxies for interfaces. Since Java is a single-inheritance language this limitation neatly prevents any inheritance problems. The Dalang approach has its strengths and weaknesses. It is more powerful than JavaCloak since it is able to operate both statically and dynamically and can mediate access to fields, which JavaCloak cannot do. However, it does break the inheritance structure of the base-class by inheriting from the meta-object class, something JavaCloak does not do. JavaCloak's impact on performance is high, as pointed out in Section 4.5, but this may not be an issue depending on the nature of the application. Its main strength is its ease of use to the inexperienced programmer.

6 Acknowledgments

I acknowledge the contributions of Huw Evans at the University of Glasgow. His collaboration and insights were essential during the course of this work. I also thank Gordon Blair and Duncan Pemberton at Lancaster University for their helpful comments on a previous draft of this paper.

7 References

1 WU, Z.: 'Reflective Java and a reflective component based transaction architecture'. Proceedings of OOOPSLA'98. Workshop on Reflective Programming in C++ and Java, Vancouver, Canada 18–22 October 1998, pp. 6–10
2 DE O GUIMARÃES, J.: 'Reflection for statically typed languages'. Proceedings of ECOOP'98, Brussels, Belgium 20–24 July 1998, pp. 440–461
3 GOLM, M.: 'Design and implementation of a meta architecture for Java'. Master's thesis, University of Erlangen, Germany, 1997

4 OLIVA, A., and BIZATO, L.E.: 'The design and implementation of guarana'. Proceedings of USENIX conference on object-oriented technology, San Deigo, CA, USA

5 GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J.: 'Design patterns: elements of reusable object-oriented software' (Addison-Wesley, Reading, MA, 1994)

6 WELCH, I., and STROUD, R.: 'Dalang—A reflective extension for Java. Technical Report CS-TR-672, Computing Science Department, University of Newcastle Upon Tyre, September 1999

7 COHEN, G.A., CHASE, J.S., and KAMINSKY, D.L.: 'Automatic program transformation with JOIE'. Proceedings of the USENIX 1998 Annual Technical Conference, Berkeley, CA, USA, 15–19 June 1998, pp. 167–178

8 DAHM, M.: 'Byte code engineering'. Proceedings of JIT'99, Duesseldorf, Germany 1999

9 CHIBA, S.: 'Load-time structural reflection in Java'. Proceedings of the 14th European Conference on Object-oriented programming. ECOOP 2000, Sophia Antipolis and Cannes, France, 12–16 June 2000, pp. 313–336

10 WELCH, I., and STROUD, R.J.: 'Kava: Using bytecode rewriting to add behavioural reflection to Java'. Proceedings of USENIX Conference on Object-oriented Technology, San Antonio, Texas, USA, 29 January–Februrary 2001

11 WELCH, I., and STROUD, R.: 'From Dalang to Kava: the evolution of a reflective Java extension', *Lect. Notes Comput. Sci.*, 1999, **1616**, pp. 2–21

12 KELLER, R., and HÖLZLE, U: 'Binary component adaptation', *Lect. Notes Comput. Sci.*, 1998, **1445**, pp. 307–329

13 Objectstore Enterprise Edition Homepage. http://www.object-store.net/objectstore/, accessed May 2000. Excelon Corporation

14 KICZALES, C., DES RIVIERES, J., and BOBROW, D.G.: 'The art of the meta-object protocol' (MIT Press, Cambridge, MA, USA, 1991)

15 Java 2 Platform, Standard Edition online documentation. http://java.sun.com/j2se/1.3/docs/api/index.html/, accessed April 2000

16 OGAWA, H., MATSUOKA, K.S.S., MARUYAMA, F., SOHDA, Y., and KIMURA, Y.: 'OpenJIT: an open-ended, reflective JIT compiler framework for Java', *Lect. Notes Comput. Sci.*, 2000, **1850**, pp. 362–382

17 RENAUD, K., and EVANS, H.: 'Javacloak: engineering Java proxy objects using reflection'. Proceedings of NET.OBJECTDAYS 2000, Messekongresszentrum Erfurt, Germany, 9–12 October 2002

18 DANFORTH, S., and FORMAN, I.R.: 'Reflections on metaclass programming in SOM', *ACM SIGPLAN Notices*, 1994, **29**, (10), p. 440

19 LIEBERMAN, H.: 'Using prototypical objects to implement shared behavior in object-oriented systems'. Conference proceedings: Object-oriented programming: systems, languages, and applications OOPSLA'86, pp. 214–223

20 CHIBA, S., and MASUDA, T.: 'Designing an extensible distributed language with a meta-level architecture'. Proceedings of the ECOOP '93 European Conference on Object-oriented programming, LNCS 707, July 1993, Kaiserslautern, Germany, pp. 483–502

21 GOLM, M., and KLEINÖDER, J.: 'Jumping to the meta level. Behavioural reflection can be fast and flexible', *Lect. Notes Comput. Sci.*, 1999, **1616**, pp. 22–39