

# Kava - A Reflective Java based on Bytecode Rewriting

Ian Welch and Robert J. Stroud

University of Newcastle-upon-Tyne, United Kingdom NE1 7RU  
{I.S.Welch, R.J.Stroud}@ncl.ac.uk,  
WWW home page:<http://www.cs.ncl.ac.uk/people/{I.S.Welch, R.J.Stroud}>

**Abstract.** Current implementations of reflective Java typically either require access to source code, or require a modified Java platform. This makes them unsuitable for applying reflection to Commercial-off-the-Shelf (COTS) systems. The high level nature of Java bytecode makes on-the-fly rewritings of class files feasible and this has been exploited by a number of authors. However, in practice working at bytecode level is error prone and leads to fragile code. We propose using metaobject protocols in order to specify behavioural changes and use standard bytecode rewritings to implement the changes. We have developed a reflective Java called *Kava* that provides behavioural runtime reflection through the use of bytecode rewriting of Java classes. In this paper we discuss the binary rewriting approach, provide an overview of the *Kava* system and provide an example of an application of *Kava*.

## 1 Introduction

We are interested in the problems of applying non-functional requirements to Commercial Off-the-Shelf (COTS) software components. In an environment such as Java, components are usually supplied in a compiled form without source code, and can be integrated into a system at runtime.

Metaobject protocols [12] offer a principled way of extending the behaviour of these components. Metaobjects can encapsulate the behavioural adaptations necessary to satisfy desirable non-functional requirements (NFRs) such as fault tolerance or application level security [1][2][18][19] transparently at the meta-level. Ideally we want to apply these metaobjects to compiled code that executes on a standard Java platform.

The Java 2 Reflection package `java.lang.reflect` provides introspection, dynamic dispatch and the ability to generate proxies for classes on-the-fly. However, this is not sufficient to build a rich metaobject protocol that can be applied transparently. There are a number of alternative extensions for Java that provide more powerful and more transparent reflection. However, they all have flaws that do not make them applicable to the problem of adapting components. These flaws include the requirement for customised Java Virtual Machines (JVMs), limited reflective capabilities, or weak non-bypassability. The term non-bypassability

refers to the binding between the base level and the meta level. For a number of NFRs such as security the meta level should never be able to be bypassed. This is what we term strong non-bypassability. However, in a number of implementations the techniques used to implement the bindings are easily bypassed. We refer to this as weak non-bypassability.

We have produced our own implementation of a reflective extension for Java called *Kava* that provides a rich metaobject protocol, requires only a standard JVM and provides strong non-bypassability. *Kava* implements a runtime behavioural metaobject protocol through the application of standard byte code rewritings, and behavioural adaptation is implemented using Java metaobject classes. This is to be distinguished from structural reflection where a metaobject protocol provides an interface for a programmer who wants to change the actual structure of an object.

The rest of the paper is organized as follows. In section two we provide a review of different approaches to implementing reflection in Java. Section three introduces the *Kava* metaobject protocol. Section four explains the byte code rewriting approach. Section five gives an example of an application of *Kava*. Finally section six provides some conclusions about the general approach.

A prototype implementation of *Kava* has been completed and is available from <http://www.cs.ncl.ac.uk/people/i.s.welch/kava>.

## 2 Review of Reflective Java Implementations

In this section we briefly review a number of reflective Java implementations and attempt to categorize them according to the point in the Java class lifecycle that reflection is implemented.

The Java class lifecycle is as follows. A Java class starts as source code that is compiled into byte code, it is then loaded by a class loader into the JVM for execution, where the byte code is further compiled by a Just-In-Time compiler into platform specific machine code for efficient execution.

Different reflective Java implementations introduce reflection at different points in the lifecycle. The point at which they introduce reflection characterizes the scope of their capabilities. In order to realise reflective control by a metalevel the baselevel system is modified through the addition of traps on operations. These traps are known as metalevel interceptions [26]. For example, in *Reflective Java* method calls sent to the base object are brought under control of an associated meta object by trapping each method call to the base object. These traps are added at the source code stage of the lifecycle making it necessary to have access to the source code. In contrast *MetaXa* adds the traps into the JVM, here the implementation of the dispatch mechanism of the JVM is changed in order to take control over method calls. As the traps are added to the runtime system source code is no longer required. The drawback of this approach is that unlike *Reflective Java* a specialized JVM must be used.

Table 1 summarizes the features of the different reflective Java implementations.

**Table 1.** Comparison of Reflective Java Implementations

Point in Lifecycle	Reflective Java	Description	Capabilities	Restrictions
Source Code	Reflective Java [25]	Preprocessor.	Dynamic switching of metaobjects. Intercept method invocations.	Can't make a compiled class reflective, requires access to source code.
Compile Time	OpenJava [21]	Compile-time metaobject protocol.	Can intercept wide range of operations, and extends language syntax.	Requires access to source code.
Byte Code	Bean Extender [9]	Byte code preprocessor.	No need to have access to source code.	Restricted to Java Beans, requires offline preprocessing.
	Dalang [22][23]	Byte code rewriting as late as loadtime	No need to have access to source code.	Suffers from known problems with class wrapper approach - delegation, identity, weak encapsulation etc.
	Javassist [4]	Byte code rewriting as late as loadtime.	No need to have access to source code.	Focus on metaobject protocol for structural adaptation. Weak encapsulation.
Runtime	MetaXa [7]	Reflective JVM.	Can intercept wide range of operations, Can be dynamically applied.	Custom JVM.
	Rjava [8]	Wrapper based reflection allowing dynamic binding.	Intercepts method invocations, and allows dynamic extension of classes.	Custom JVM - addition of new byte code.
	Guarana [15]	Reflective kernel supported by modified JVM.	Interception of message sends, state access and supports metaobject composition.	Custom JVM.
	java.lang.reflect [20]	Reflective capabilities part of the standard Java development kit.	Runtime introspection, dynamic dispatch, and on-the-fly generation of proxies.	Overall introspection rather than behavioural or structural reflection.
Just-in-time Compilation	OpenJIT [14]	Compile-time metaobject protocol for compilation to machine language.	Can take advantage of facilities present in the native platform. No need for access to source code.	No behavioural metaobject protocol.

All these implementations have drawbacks that make them unsuitable for use with compiled components or in a standard Java environment where the purpose is to add security. Some require access to source code, others are non-standard because they make use of a modified Java platform, and none of the portable approaches support strong non-bypassability.

In contrast, *Kava* does not require access to source code because it is based on bytecode rewriting, doesn't require a non standard Java environment and provides a rich set of capabilities. It also provides strong non-bypassability. Most implementations add traps through renaming of classes, or of methods which means that it may be possible to call the original methods and therefore bypass the meta layer. However, *Kava* actually adds the traps directly into the method bodies avoiding this problem.

Recently a new reflective Java called *Javassist* has been developed that provides a metaobject protocol for structural modification of Java classes and supports a simple metaobject protocol for behavioural reflection. It is the most similar in character to *Kava* of all the reflective Java implementations. However, it uses method renaming to implement the trapping of method calls and therefore doesn't support strong non-bypassability. It also currently doesn't support the same range of reflective capabilities of *Kava*.

### 3 Kava Metaobject Protocol

The *Kava* metaobject protocol provides an interface to the Java runtime object model that gives programmers the ability to modify a Java application's behaviour safely and incrementally. Each object is causally connected with a unique metaobject. The metaobject provides a representation of the structure of the object, and of the behaviour of the object. Metaobjects are implemented in the Java language in the same way that objects are implemented. However, bytecode rewriting techniques are used to create the causal connection between the meta and base levels. This approach allows *Kava* to be used in a standard Java environment, means that it can be applied to compiled Java classes and allows for a rich metaobject protocol. The idea of bytecode rewriting is not new but using it to implement behavioural reflection is a new idea.

When *Kava* creates a reflective Java class then a binding is created between the class and a metaobject class (or metaclass). In *Kava* a metaobject class is a Java class that implements the interface `Metaobject`. Whenever an instance of the class is created then a unique instance of the metaobject implementation is also created. By creating different metaobject implementations the programmer can create standard redefinitions of the Java runtime object model. These different implementations are bound to base level objects in order to customize the behaviour of the base level objects.

In the following sections we first look at the aspects of base level behaviour under the control of the metaobject protocol, and then at how the binding between objects and metaobjects is specified.

### 3.1 Scope of Metaobject Protocol

The aspects of base level behaviour that can be redefined in *Kava* are:

- Method calls to base objects.
- Method calls from base objects.
- State access by base objects.
- Creation of new instances by base objects.
- Initialisation of base objects.
- Finalization of base objects.

The metaobject associated with the base object traps the base level behaviours and specifies what will happen before the behaviour and what will happen after the behaviour. For example, when a base object makes a call to method then the method `beforeSendMethod` is invoked. By redefining `beforeSendMethod` behaviour to take place before the call is dispatched can be defined. An example is shown below. In this example the target of the call, the method itself and the arguments on the call stack are reified respectively as a `Reference`, `Method` and array of `Value`.

```
import kava.*;
public boolean beforeSendMethod(
    Reference target, Method method, Value[] arguments)
{
    System.out.println("invoking " + method.getName());
    return Constants.INVOKE_BASELEVEL;
}
```

Here a tracing message is displayed - the name of the method being invoked. The metalevel then allows the base level behaviour to take place. This is indicated by returning `Constants.INVOKE_BASELEVEL` from the method.

*Kava* also supports extended introspection on the structure of the object. The following aspects of the structure can be determined and manipulated using *Kava* :

- Binding between Metaobject and Object.
- State of the Object.

The implementor of the metaobject can choose to make the metalevel visible to the baselevel by implementing the `getMeta` method and returning a pointer to the metaobject. The default implementation should be to raise a runtime exception if the base level attempts to access the metalevel directly. In the case of implementing security using metaobject protocols we would not want the metalevel to be visible. However, if we were implementing distribution then we may want to access the metalevel in order to adjust parameters such as timeout.

The implementor may also choose to allow the binding between the metaobject and object to be changed. Again the default implementation should be to raise a runtime exception. However, the implementor can define the method

`setMeta` to allow the binding to be changed from one metaobject to another metaobject.

The *Kava* metaobject protocol also supports extended introspection. It allows access to and adjustment of base level state. This is under the control of the Java 2 security model so a security administrator can allow or prevent reflective access to state.

### 3.2 Binding Specification

*Kava* uses a simple binding language to specify which metaobjects are bound to which base level objects or classes. This allows binding information to be removed from the metaobject implementation and reasoned about separately. This adds to the separation of concerns and promotes reuse. It is also more appropriate for situations where source code may not be available, for example with COTS components.

A metaclass definition defines what aspects of the Java object model a metaclass redefines. Only those aspects redefined in the binding specification will be redefined at the metalevel irrespective of whether the metaclass implementation has defined methods for handling other aspects of the base level. This allows a fine granularity for the late binding between the meta and base level. For example, a tracing metaobject may be interested in all method invocations made by an object or only specific method invocations. The binding specification allows the method to be specified by name, or a wildcard to be used (`any-method` for any method, `any-class` for any class and `any-desc` for any parameter list) to indicate that all methods are of interest. An example of a binding between the `MetaTrace` class and the methods `notify` and `setObserver` of the class `Test` is given below. This means that each instance of `Trace` will be bound to an instance of `MetaTrace`.

```
metaclass kava.MetaTrace
{
  INTERCEPT SEND_METHOD( any-class, "notify" , any-desc );
  INTERCEPT SEND_METHOD( any-class, "setObserver" , any-desc );
}
class Test metaclass-is kava.MetaTrace;
```

## 4 Bytecode Rewriting

The *Kava* metaobject protocol is implemented using the technique of byte code rewriting. Byte code rewriting has become an established technique for extending Java both syntactically and behaviourally. For example it has been used to support parametric types [3] and add resource consumption controls to classes [6]. Generic frameworks for transforming byte code such as *JOIE* [5] and *Binary Component Adaptation* [10] have been developed to make coding byte code rewriting easier. However, as pointed out by the authors of *JOIE*, most of these

frameworks lack a high-level abstraction for describing the behavioural adaptations. This makes coding adaptations difficult as it requires a detailed knowledge of byte code instructions and of the structure of class files. Binary Component Adaptation does support a form of a higher-level abstraction in that it has the concept of *deltaClasses* that describe structural changes to a class file in terms of methods for renaming methods, mixin type methods, etc. However, the purpose of the framework is to support software evolution rather than behavioural adaptation. This means that the focus is on adding, renaming or removing methods and manipulating the type hierarchy in order to adapt ill-fitting components to work together rather than describing behavioural adaptation.

The *Kava* metaobject protocol provides a high-level abstraction for adaptation of component behaviour that specifies the change to behaviour in terms of the Java object model and is implemented using byte code rewriting. We exploit the *JOIE* framework to simplify the implementation of this metaobject protocol. The framework frees us from dealing with technical details such as maintaining relative addressing when new byte codes are inserted into a method, or determining the number of arguments a method supports before it has been instantiated as part of a class.

As byte code instructions and the structure of the class file preserve most of the semantics of the source code we can use byte code rewriting to implement metalevel interceptions for a wide range of aspects of the Java Object model such as caller and receiver method invocation, state access, object formalization, object initialisation and some aspects of exception handling. Like compile-time reflection we reflect upon the structure of the code in order to implement reflection. However, we work at a level much closer to the Java machine than most compile-time approaches that deal with the higher-level language. Although this means we cannot extend the syntax of the higher-level language it does mean that we can implement some kinds of reflection more easily than in a traditional compile-time MOP. For example, in the application of *OpenC++ version 2* to adding fault tolerance in the form of checkpointing *CORBA* applications [11] data flow analysis is performed on the source code to determine when the state of the object is updated. With *Kava* no such analysis would be necessary; all that would be required is to intercept the update of state of an object by changing the behaviour of the update field operation in the Java runtime object model. When an update was done a flag could be set indicating that the current state should be checkpointed.

Also since we are often dealing with compiled classes we do not have access to the source code in order to annotate it. There is also an argument that source code annotations are not necessarily a good idea especially when different annotation schemes are used together.

By transforming the class itself we address the problems introduced by the separation of base class and class wrapper (see earlier paper). Instead, standard byte code rewritings are used to wrap individual methods and even bytecode instructions. These micro-wrappers will switch control from the baselevel to metalevel when the methods or byte code instructions are executed at runtime.

The metalevel is programmed using standard Java classes. The metalevel allows the customisation of the Java object model at runtime. The scope of the customisation is determined by which methods and byte code instructions are wrapped at load time, but the exact nature of the customisation is adjustable at runtime.

Byte code rewriting can either be applied at loadtime through the use of an application level classloader [13] or prior to loadtime by directly rewriting class files.

To provide a flavour of this approach we provide an example of the wrapping of access to a field of a base level class. Due to space constraints we present this at a high level using source code instead of byte code. Consider the following field access:

```
myGreeting = "Hello " + name;
```

At the byte code level this is rewritten to:

```
import kava.*;

Reference target = new Reference(this);
Field field = newField("myGreeting");
Value fieldValue = Value.newValue("Hello " + name);

if (meta.beforePutField(target, field, fieldValue)
    == Constants.INVOKE_BASELEVEL)
{
    helloWorld = (String)fieldValue.getObject();
}
meta.afterPutField(target, field, fieldValue);
```

In this example the first line of code specifies that the field belongs to this instance of the base class, the second line specifies the field being updated, and the third line specifies the value the field is being updated with. Then `beforePutField` method of the associated metaobject is invoked with parameters representing the field ("`helloWorld`"), and the marshalled value. At the metalevel the value may be modified in order to adjust the final value that is stored in the field. Alternatively the update of field could be suppressed by returning a `Constants.SUPPRESS_BASE` in which case the base level action does not take place. The last line calls the `afterPutField` method of the associated metaobject with the same parameters as the initial call to the metalevel.

## 5 Application of Kava

*Kava* has wide application potential and should prove well suited to the customising the behaviour of COTS applications that are built from dynamically

loaded components. It provides a high-level abstraction for implementing behavioural adaptations that are expressed as bytecode rewritings. This means that it can be used to reimplement in a principled way behavioural adaptations that have already been implemented through byte code rewriting e.g. enforcing resource controls on applications, instrumentation of applications, visualization support etc. The advantage of using *Kava* would be that these adaptations could be combined as required since they have been built using a common abstraction.

In this section we provide an example application of *Kava* : to prevent downloaded applets from mounting a particular class of denial-of-service attack.

In [16] examples of using bytecode rewriting techniques for protecting against malicious attacks are discussed. In particular the authors provide an example of how bytecode rewriting can be used to protect against denial-of-service caused by a window consuming attack.

The basic idea is that an applet can crash the host system by creating more windows than the windowing system can handle. In order to protect against this attack the system should track the number of windows created and either block or throw an exception when a predetermined limit is exceeded. The class used to generate the top-level windows is the Java library class `Frame`.

The solution provided in [16] is to prevent an applet from invoking the constructor methods of the `Frame` class more than a predefined number of times. They achieve this by subclassing `Frame` to create `Safe$Frame`. `Safe$Frame` is implemented in such a way that it counts the number of top-level windows created and blocks further creation of windows if the predefined limit is exceeded. Then every application class downloaded with the applet is checked for references to `Frame`. When a reference is found then it is replaced by a reference to `Safe$Frame`. The bytecode rewriting is done within a proxy server that intercepts requests for applets.

This approach has the advantage that it is transparent and can work with COTS browsers. The main drawback to the technique that it is difficult to generalize to other classes. For example, another denial-of-service attack might rely upon the applet creating more threads than the system can handle. This would require the creation by hand of a `Safe$Thread` class that monitored and controlled the creation of `Thread` instances.

*Kava* provides a higher level approach to applying such safety mechanisms. The high level approach is to describe a mechanism for limiting the creation of new instances of monitored classes such as `Frame` or `Thread`. Using *Kava* this is done by creating an implementation of a `Metaobject` that performs the resource monitoring. A simplified implementation is shown below:

```
import kava.*;

public class ResourceMonitor implements Metaobject
{

    public void beforeCreation(Reference target, Value[] arguments)
    {
```

```

        incrementUsage(target);
        if (exceededMaximum(target)
        {
            throw new RuntimeException("resource count exceeded by " +
                                     target.getClass());
        }
    }
}

...
}

```

The method `beforeCreation` redefines how new instances are created. First, it calls a method `incrementUsage` that increments a global count of the number of instances of the that class and its superclasses. We count superclasses as an instance of a subclass of a monitored class might be created. Then it calls a method `exceededMaximum` that checks if the maximum limit for the number of instances of any monitored class has been exceeded. If the limit has been exceeded then a runtime exception is thrown and the execution of the applet halts.

The advantage of this approach over the original approach is more general and doesn't require manual generation of new classes if a new class is to be monitored. In [24] we propose a even more general reflective security model for resource consumption control.

## 6 Conclusions

Ideally, a reflective extension for Java that is intended to be used to adapt the runtime behaviour of COTS components should not require access to source code, or modifications to the Java platform, and should provide strong non-bypassability. Unfortunately, existing reflective extensions that we have reviewed do not meet these requirements and provide the scope of control required.

We have implemented a system called *Kava* that overcomes these problems by using bytecode rewriting in order to establish a causal connection between metaobjects and objects. It provides a higher level abstraction for specifying changes to the behaviour of objects than is provided by currently available bytecode rewriting toolkits. It addresses some reflective capabilities not handled by some extensions such as the ability to redefine how base level objects call other base level objects, and offers strong non-bypassability.

To the best of our knowledge *Kava* was the first reflective Java extension to make use of bytecode rewriting in order to implement behavioural reflection. Subsequently there have been similar developments such as *Javassist* although these lack some of the capabilities of *Kava* and do not support strong non-bypassability.

We are currently using the *Kava* prototype to implement a reflective security metalevel for Java applications. Current versions of *Kava* are available from <http://www.cs.ncl.ac.uk/people/i.s.welch/kava>.

## Acknowledgements

This work has been supported by the UK Defence Evaluation Research Agency, grant number CSM/547/UA.

## References

- [1] Ancona M., Cazzola W. and Fernandez E. B. : Reflective Authorization Systems: Possibilities, Benefits and Drawbacks. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science 1606. Springer-Verlag, 1999.
- [2] Benantar M., Blakley B., and Nadain A. J. : Approach to Object Security in Distributed SOM. *IBM Systems Journal*, Vol35, No.2, (1996).
- [3] Agesen, O., Freund S., and Mitchell J. C.: Adding Type Parameterization. In *Proceedings of OOPSLA 1997, Atlanta, Georgia (1997)*.
- [4] Chiba, S. : Load-time structural reflection in Java. To be published in proceedings of ECOOP2000 (2000).
- [5] Cohen, G. A., and Chase, J. S. : Automatic Program Transformation with JOIE. *Proceedings of USENIX Annual Technical Symposium (1998)*.
- [6] Czajkowski, G. and von Eicken, T. JRes: A Resource Accounting Interface for Java. *Proceedings of OOPSLA 1998 (1998)*.
- [7] Golm, M. : Design and Implementation of a Meta Architecture for Java. MSc Thesis (1997). University of Erlangen.
- [8] Guimares, J. : Reflection for Statically Typed Languages. *Proceedings of ECOOP 1998 (1998)*.
- [9] IBM. Bean Extender Documentation, version 2.0 (1997).
- [10] Keller, R. and Holzle, U. : Binary Component Adaptation. *Proceedings of ECOOP 1998 (1998)*.
- [11] Killijian M-O. , Fabre J-C. , Ruiz-Garcia J-C. , and Chiba S. : A Metaobject Protocol for Fault-Tolerant CORBA Applications. *Proceedings of SRDS 1998 (1998)*.
- [12] Kiczales G., des Rivieres J. : *The Art of the Metaobject Protocol*. MIT Press (1991).
- [13] Liang, S. and Bracha, G. : Dynamic Class Loading in the Java(tm) Virtual Machine. *Proceedings of OOPSLA 1998 (1998)*.
- [14] Matsuoka S., Ogawa H., Shimura K., Kimura, Y., and Takagi H. : OpenJIT - A Reflective Java JIT Compiler. *Proceedings of Workshop on Reflective Programming in C++ and Java, UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba, Japan ISSN 1344-3135, (1998)*.
- [15] Oliva, A., Buzato, L. E., and Garcia, I. C. : The Reflective Architecture of Guaran. Available from: <http://www.dcc.unicamp.br/oliva>.
- [16] Shin, I. and Mitchell, J.C. : Java Bytecode Modification and Applet Security. *Stanford CS Tech Report, (1998)*.
- [17] Stroud, R. J. : Transparency and Reflection in Distributed Systems. *Operating Systems Review 27(2): 99-103 (1993)*
- [18] Stroud, R. J. and Z. Wu : Using Metaobject Protocols to Satisfy Non-Functional Requirements. Chapter 3 from "Advances in Object-Oriented Metalevel Architectures and Reflection" (1996), ed. Chris Zimmermann. Published by CRC Press.
- [19] Stroud, R.J. and Wu, Z. : Using Metaobject Protocols to Implement Atomic Data Types, *Proceedings of ECOOP'95(LNCS-925), Aarhus, Denmark, (1995)*.

- [20] Sun Microsystems, Inc. : Java Development Kit version 1.3.0 Documentation (2000).
- [21] Tatsubori, M. and Chiba, S. : Programming Support of Design Patterns with Compile-time Reflection. Proceedings of Workshop on Reflective Programming in C++ and Java, UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba, Japan ISSN 1344-3135, (1998).
- [22] Welch, I. and Stroud, R. J. : Dalang - A Reflective Extension for Java. Computing Science Technical Report CS-TR-672, University of Newcastle upon Tyne, (1999).
- [23] Welch, I. and Stroud, R. J. : From Dalang to Kava - The Evolution of a Reflective Java Extension. Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection 1999, Saint-Malo, France, July 19-21, 1999, LNCS 1616, Springer (1999).
- [24] Welch, I. and Stroud, R. J. : Supporting Real World Security Models in Java. Proceedings of 7th IEEE International Workshop on Future Trends of Distributed Computing Systems, Cape Town, South Africa, December 20-22, 1999, IEEE Computer Society (1999).
- [25] Wu, Z. and Schwiderski, S. : Reflective Java - Making Java Even More Flexible. FTP: Architecture Projects Management Limited (apm@ansa.co.uk), Cambridge, UK (1997).
- [26] Zimmerman, C. : Metalevels, MOPs and What all the Fuzz is All about. Chapter 1 from "Advances in Object-Oriented Metalevel Architectures and Reflection" (1996), ed. Chris Zimmermann. Published by CRC Press.

## Appendix A - Full Metaobject Protocol

```
public interface MetaObject {

// behavioural - can override base action
public boolean beforeGetField(Reference target, Field field);
public void afterGetField(Reference target, Field field,
                          Value value);
public boolean beforePutField(Reference target, Field field,
                              Value value);
public void afterPutField(Reference target, Field field,
                          Value value);
public boolean beforeReceiveMethod(Method method, Value[] arguments);
public void afterReceiveMethod(Method method, Value[] arguments,
                                Value result);
public boolean beforeSendMethod(Reference target, Method method,
                                Value[] arguments);
public void afterSendMethod(Reference target, Method method,
                             Value[] arguments, Value result);

// behavioural - can only modify arguments of base action
// or throw a runtime exception
public void beforeInitialisation(Reference target, Value[] arguments);
public void afterInitialisation(Reference target, Value[] arguments);
```

```

public void beforeFinalization(Reference target);
public void afterFinalization(Reference target);
public void beforeCreation(Reference target, Value[] arguments);
public void afterCreation(Reference target, Value[] arguments);

// introspection
public Object getMeta();
public Value getFieldValue(Field field);
public void setFieldValue(Field field, Value value);
}

```

## Appendix B - Grammar for Binding Language

```

configuration ::= metaclass_defn class_binding_list ;
class_binding_list ::= class_binding_list class_binding_part
| class_binding_part ;
class_binding_part ::= CLASS class_name METACLASS_IS
metaclass_name [RECURSIVE] EOS;
metaclass_defn ::= METACLASS metaclass_name (meta_expr_list);
meta_expr_list ::= meta_expr_list meta_expr_part
| meta_expr_part;
meta_expr_part ::=
INTERCEPT RECEIVE_METHOD (class, method , params) EOS |
INTERCEPT SEND_METHOD (class, method , params) EOS |
INTERCEPT GET_FIELD (class, field_name, method, params) EOS |
INTERCEPT PUT_FIELD (class, field_name, method, params) EOS |
INTERCEPT INITIALISE_INSTANCE (class) EOS |
INTERCEPT INITIALISE_CLASS (class) EOS |
INTERCEPT FINALIZE_INSTANCE (class) EOS |
INTERCEPT FINALIZE_CLASS (class) EOS |
INTERCEPT CREATION (class, method, params} EOS
}

```