POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE • CB3 0RD UNITED KINGDOM
+44 1223 515010 • Fax: +44 1223 359779 • Email: apm@ansa.co.uk • URL: http://www.ansa.co.uk

ANSA Phase III

Design of Reflective Java

Zhixue Wu and Scarlet Schwiderski

Abstract

The purpose of the Reflective Java project is to make it easier for a Java-powered system to provide nonfunctional capabilities to its applications transparently and flexibly. At the same time, it also makes it easier for application software to adjust its behaviour to meet new requirements. For example, an applet can be customised at its execution site to fulfil particular requirements dynamically.

The project provides a reflective extension to the Java language, in which method calls are made open-ended; a simple preprocessor that translates reflective programs into standard Java programs and generates classes for binding a Java object to its metaobject.

This documentation discusses the design of Reflective Java and describes its programming environment. Some examples are also presented to demonstrate meta level programming.

APM.1911.00.01	**Draft**	23rd December 1996
	Technical Report	

Distribution:
Supersedes:
Superseded by:

Copyright © 1996 APM Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Design of Reflective Java

Design of Reflective Java

Zhixue Wu and Scarlet Schwiderski

APM.1911.00.01

23rd December 1996

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by APM Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

APM Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Copyright © 1996 APM Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

APM Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

Page	No.	Title
1	**1**	**Introduction**
3	**2**	**The Metaobject Protocol (MOP) Approach**
3	2.1	Options
3	2.2	Overview of the MOP approach
5	**3**	**The Design**
5	3.1	Reflection properties of Reflective Java
6	3.2	Binding
6	3.2.1	Static and dynamic binding
6	3.2.2	The idea
7	3.2.3	The reflection class
8	3.3	Programming in Reflective Java
8	3.3.1	Application programming
8	3.3.2	Meta level programming
9	3.3.3	End-user programming
10	3.3.4	Dynamic object binding
10	3.4	Java[TM] Core Reflection
12	**4**	**Binding Specification**
12	4.1	Method category
13	4.2	Binding specification language
14	4.3	**The Syntax**
15	**5**	**The Meta Level Programming**
15	5.1	Metaobject class
16	5.2	Accessing meta data
17	5.3	Category numbers
17	5.4	Dealing with exceptions
18	5.5	Accessing information of the caller
19	**6**	**Summary**

1 Introduction

In the ANSA programme, the *Reflective Java* project makes some features of Java reflective, thus enabling Java-powered systems to be customised dynamically, flexibly and transparently. This documentation describes the design of Reflective Java.

Java has been adapted as a programming language for the Internet because of its ability to simplify the development of flexible, portable applications with high-level graphical interfaces. Using Java, a user can write a program once and run it anytime, on any platform. This is a key requirement for the Internet, where one program should be capable of running on any computer in the world.

Java has solved the problem of program portability in the hardware world. However, in practice there is also a portability problem in the software world, i.e. porting a program to a different infrastructure. For example, an application program in a sequential environment cannot be used without change in a concurrent environment, and an application program in a centralised system cannot be used without change in a distributed system.

Generally, an application program needs to meet two kinds of requirements: *functional requirements*, which are concerned with the main purpose of an application (i.e. what it does), and *non-functional requirements*, which are concerned with its fitness to fulfil this purpose (i.e. how it does it). For example, non-functional requirements may be concerned with concurrency control, failure resilience, data persistence, and data presentation.

The portability problem in the software world is due to the non-functional application requirements. Since non-functional requirements fulfil a general purpose, it would be ideal to implement them in an application-independent fashion. In this way, they can be reused across a wide range of application domains. On the other hand, functional requirements should be implemented without considering any non-functional requirements. Thus, they can be reused in different software environments. By achieving a clear separation between functional and non-functional requirements, the portability problem in the software world can be resolved.

Unfortunately, Java does not support this capability at present. Java takes the Application Programming Interface (API) approach to implement non-functional requirements. Although the API approach works well for providing some basic functionality such as I/O, network and GUI (i.e., functionality that is application-independent), it is problematic to use it in order to provide some advanced functionality, such as concurrency control and data persistence (functionality that is more application-dependent).

- During the design and implementation of the API for a specific non-functional capability, it is often required to make a choice from several possible implementation strategies. The resulting API could usually only implement a fixed, single point in the whole design space. Therefore, it cannot match all application demands, no matter how well it is designed.

- The API approach cannot provide functionality transparently to application programs, because the API must be explicitly invoked by application programs. This increases the complexity of the application source code and makes it hard to reason about its correctness. Furthermore, it makes it hard for an application program to adapt to a new strategy for providing a particular non-functional capability. This would involve changes to the application's source code.

A weakness caused by the above problems is that configuring the engineering qualities of Java applications cannot be done without changing the source code. Thus, it is impossible for the execution site to control the engineering qualities of Java applets.

In summary, although Java provides the "write once, run anywhere" capability, it fails to provide the "write once, run on any infrastructure" capability. Therefore, Java's strength of portability is not reflected fully in practice.

The goal of the Reflective Java project is to enable Java to provide non-functional capabilities to its applications transparently and flexibly.

2 The Metaobject Protocol (MOP) Approach

2.1 Options

One could imagine that by making use of Java's interface and/or inheritance mechanisms, it would be possible to implement a non-functional capability under different strategies. Application programmers could then choose the right strategy suitable for their specific application. In practice, the problems mentioned in the last section cannot be solved in this way. Since different strategies usually require application programs to invoke different subsets of the methods provided, adapting to a new strategy still implies changing the application programs.

Some researchers have shown that the reflection / metaobject protocol (MOP) approach is appropriate for adding non-functional capabilities to a system [SW95, Zim96]. Taking this approach, it is possible to achieve a clear separation of functional and non-functional capabilities, while still providing the necessary flexibility for application programmers to customise the system according to the needs of the particular application.

There are different options for introducing reflection to a language like Java. The first option is to make changes to the language itself. However, this would confuse those programmers, who are not familiar with the reflection concept or, who do not need this functionality. Therefore, this option is not appropriate. The second option is to extend Java without making any changes to the language or its compiler. Normal programs do not need to use the new concept, but it provides a better approach for those programs which need to provide capabilities of the Java language, and those which would like to control the engineering qualities of Java applications.

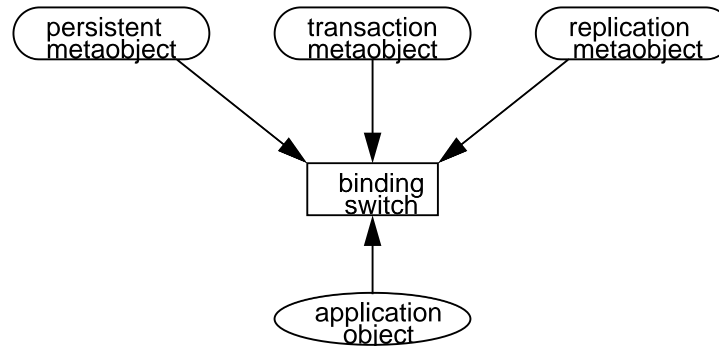
2.2 Overview of the MOP approach

The combination of reflection [{Mae87] and object-oriented programming in the form of a metaobject protocol [KdRB91] provides a mechanism that is needed to produce flexible and reusable implementations of non-functional requirements. Using a metaobject protocol, it is possible to change the behaviour of an application object by applying new kinds of metaobject. This makes it possible to implement non-functional requirements in a way that is transparent to the application program, while still remaining flexible enough to meet the demands of a wide range of applications.

The basic idea behind this approach is that functional requirements are satisfied by application objects implemented by application programmers, while non-functional requirements are satisfied by metaobjects implemented by system programmers. Different ways of implementing particular non-functional requirements are realised by different metaobjects. The actual behaviour of an application object is determined not only by the object that implements its functionality, but also by the metaobject which it is associated

with. The association can be thought of in terms of a binding between the application object and the metaobject. Non-functional capabilities can be added to an application object by simply binding it to an appropriate metaobject, namely a metaobject which implements the desired non-functional requirements.

Figure 2.1: The idea of the metaobject protocol approach



For example, as illustrated in Fig. 2.1, an application object can become a persistent object whose state survives the session in which it was created through binding it to a persistent metaobject. It can become a replicated object whose state is stored in multiple sites of the system for tolerating system failures through binding it to a replication metaobject. In this way, non-functional requirements can be addressed in a way that is transparent to application objects. The behaviour of an application object can be changed without re-implementing the object, but by simply changing its metaobject binding.

3 The Design

Reflective Java is a reflective programming environment that enables a Java-powered system to provide non-functional capabilities to its applications transparently and flexibly, by supporting the Metaobject Protocol (MOP) approach.

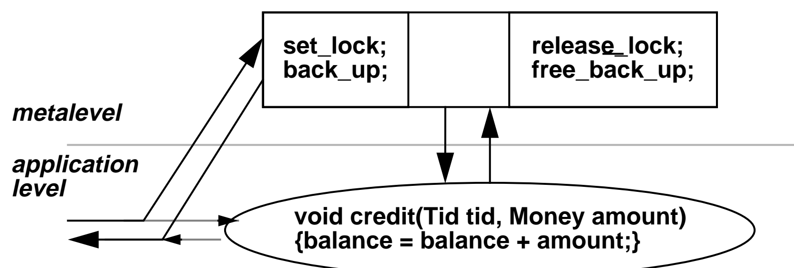
3.1 Reflection properties of Reflective Java

Supporting a fully reflective programming language means that all the properties of the language should be accessible and changeable by programmers (in a controlled way). However, making Java fully reflective would be a very complex task. Also, not all the properties of the language need or even should be made reflective, for example, because of security reasons. Therefore, in order to keep Reflective Java simple and safe, it only provides limited reflection properties, namely properties that are simple but powerful enough to solve the problems discussed in Section 1.

In Reflective Java, only one property of Java becomes reflective, namely the method invocation. This enables programmers to redefine the behaviour of method invocations according to their particular needs. The basic idea is that method invocations to an object can be intercepted and passed to the corresponding metaobject. The metaobject decides how to deal with the invocation. In this way, system programmers can make method invocations behave according to their needs.

Reflective Java enables programmers to change the behaviour of method invocations by specifying what should be done before or/and after “normal” method execution: `metaBefore` is executed before a method is invoked, and `metaAfter` is executed afterwards. For example, Fig. 3.1 shows how to provide concurrency control to an object that is implemented for a sequential environment. The object is bound to a metaobject that implements `metaBefore` for locking the object and making a back-up of its state, and `metaAfter`, for unlocking the object and freeing the back-up.

Figure 3.1: Behaviour of a reflective method call



Other typical non-functional requirements can also be satisfied by taking advantage of reflective method invocation, for example, remote object invocation, data persistence and data replication.

3.2 Binding

An important issue when making a programming language reflective is, how to bind an object to a metaobject so that the behaviour of the object can be controlled by the metaobject.

3.2.1 Static and dynamic binding

An object can be bound to a metaobject statically or dynamically. Static binding is done at compile time and cannot be changed subsequently. Dynamic binding allows an object binding to a metaobject to be changed at runtime.

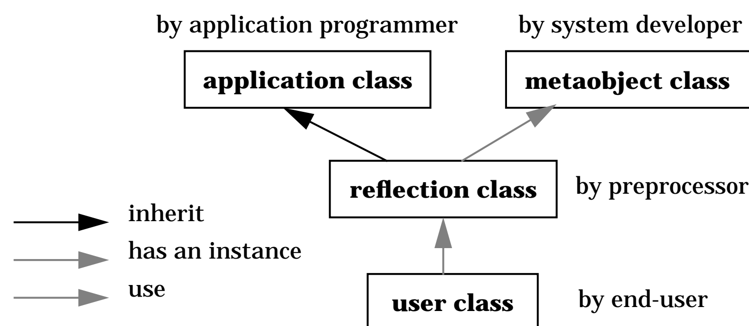
Static binding is simple and easy to realise, but it means that the behaviour of an object is determined statically and cannot be changed dynamically. This would be a severe limitation for some applications which require an object to behave according to its environment. For example, in a concurrent system an object is initially bound to a metaobject implementing an optimistic concurrency control protocol. At runtime, however, it turns out that there are too many conflicts for the optimistic protocol to work properly. Hence, it is required to change the object binding to a metaobject implementing a pessimistic concurrency control protocol.

Allowing dynamic binding would provide great flexibility to applications. It enables applications to dynamically change their strategies, and even introduce new strategies without shutting the system down. This is important for some real-time systems, where system shutdown is impossible. However, achieving this kind of flexibility requires the reflective language to support dynamic loading and linking.

3.2.2 The idea

Dynamic binding is supported in Reflective Java so that great flexibility can be provided to applications. Object binding in Reflective Java is done through a reflection class that is specific to an application class.

Figure 3.2: Object binding



The general idea is to make a reflection class a subclass of an application class. The reflection class has a `metaobject` variable that is an instance of a metaobject class. Therefore, the operations in the application class can be

overwritten in the reflection class. An overwritten operation is implemented such that any incoming invocation is passed to the metaobject denoted by the `metaobject` variable, where the invocation is interpreted and dealt with.

A reflection class is to be generated automatically for an application class by the *Reflective Java preprocessor*.

In order to benefit from reflection, end-users have to use instances of a reflection class, instead of instances of an application class.

3.2.3 The reflection class

For each application class that becomes reflective, the Reflective Java preprocessor produces a reflection class in the following way:

- the reflection class is a subclass of the application class
- the reflection class has one variable `metaobject` that is an instance of `MetaObject` class; the root of all metaobject classes
- the reflection class has one instance variable that records the name of the metaobject class to which the application class intends to bind
- the reflection class provides a pair of public methods, `getMeta` and `changeMeta`, for programmers to check and change the metaclass of an object
- the reflection class has a constructor which calls the constructor of the superclass (i.e. the application class), loads the metaobject class through the class loader, and creates an instance of it
- the reflection class overwrites the methods which are to become reflective

As stated earlier, each metaobject class must provide two methods: `metaBefore` and `metaAfter`. The `metaBefore` method intercepts the beginning of an application object's method invocation, and `metaAfter` intercepts the end. For each method of an application class that is to become reflective, the reflection class overwrites that method with a method having the following structure:

- packing the parameters of the method in order to pass them to the metaobject
- calling the `metaBefore` method with the following information about the application method: the identifier of the method, the parameter package, and the method category
- unpacking the parameters returned by `metaBefore` (in case the parameters have been changed by the metaobject)
- calling the "original" application method taking into account the returned parameters
- packing the parameters of the method plus the result of the above method call in order to pass them to the metaobject
- calling the `metaAfter` method with the following information about the application method: the identifier of the method, the parameter package, the result package, and the method category
- unpacking the result returned by `meta_after` (in case the result has been changed by the metaobject)
- returning the result to the original caller

3.3 Programming in Reflective Java

Reflective Java enables Java-powered systems to provide non-functional capabilities to its applications transparently and flexibly. The approach taken by Reflective Java is to use the MOP approach. In this way, it is possible to make a clear separation of functional and non-functional requirements. The application programmer focuses on implementing the functional requirements of the application, not considering extensions with non-functional capabilities. On the other hand, system developers provide suitable metaobject classes representing specific non-functional capabilities. Finally, the end-users choose a metaobject class that contains the required non-functional capabilities. The end-users are also able to change the metaobject class of an object dynamically.

3.3.1 Application programming

Application programmers can develop applications by using the normal Java programming language. It is the goal of Reflective Java that a normal Java program can become reflective by binding it to a metaobject. Therefore, an end-user can download a Java program and use it in a different environment by binding it to a metaobject, although the developer of the application program does not know of Reflective Java and the reflection concept. For example, Fig. 3.3 is a normal Java class that implements a simple bank account. The programmer of this class focuses on satisfying the functional requirements of the application, without considering concurrency control.

Figure 3.3: A simple bank account class

```
class Account {
 public void credit(double m)
  {
    balance = balance + m;
  }

 public double check( )
  {
    return balance;
  }

private double balance;

}
```

3.3.2 Meta level programming

System programmers must be aware of Reflective Java. Although they program in the normal Java programming language, they must do it according to the rules of Reflective Java. For example, any metaobject class must be a subclass of `MetaObject` class. A metaobject class must implement the `metaBefore` and `metaAfter` operations. Fig. 3.4 shows a metaobject class implementing simple locking operations that can be used to implement some basic concurrency control. When an application class is bound to the `Meta_Lock` class, the `metaBefore` operation is executed before an operation of the application is executed, and the `metaAfter` operation is executed afterwards. The `mid` and `cid` parameters in these operations are used to pass the identifier and category of an operation of the application class. Metaobject classes may use this information to decide which action should be taken for a

particular operation. Metalevel programming will be discussed in more detail in Section 5.

Figure 3.4: A simple lock metaobject class

```
class Meta_Lock extends MetaObject{

public void metaBefore(MID mid, CID cid, Arg arg)
   {
    if (cid == 201) lock.set_read_lock();
    else lock.set_write_lock();
   }
public void metaAfter(MID mid,CID cid,Arg arg, Arg rslt)
   {
    if (cid == 201) lock.release_read_lock();
    else lock.release_write_lock();
   };

private Lock lock;

}
```

3.3.3 End-user programming

It is the end-users who are responsible for specifying which application classes should become reflective and should be associated with which metaobject classes. Reflective Java allows end-users to make only a subset of the operations of a class reflective because usually only public operations need this property. Reflective Java provides a simple script language for end-users to specify object bindings. A simple example is shown in Fig. 3.5. A more detailled description will be given in Section 4.

Figure 3.5: An object binding specification example

```
import Account;

refl_class Account: Meta_lock
{
public Account(String nm) throws Throwable:1;
public void init(String nm):201;
public void credit(double amt):201;
public void debit(double amt)throws OverDraw:201;
public double balance( ):202
}
```

Usually the end-users also need to implement some classes to create and manipulate application objects for their particular application. This is done in the normal Java programming language, but must obey the rules of Reflective Java. For example, to make an application object reflective, instead of from the original application class, it must be instantiated from the reflection class created for that application class. End-users can also dynamically check and change an object binding through the `getMeta` and `changeMeta` operations of the reflection class.

Reflective Java separates the binding specification from other Java programs so that it can be changed easily and independently. This also simplifies the work of the Reflective Java preprocessor.

3.3.4 Dynamic object binding

Some applications, such as mobile computing and multimedia systems, require dynamic change to their policies according to real-time conditions. To fulfil this requirement, Reflective Java provides the capability for end-user programs to change an object binding dynamically. For this purpose, it provides a `changeMeta` operation with the following signature:

```
public void changeMeta(String metaclassnm);
```

In order to change an object binding to a new metaobject, a program only needs to call the `changeMeta` operation with the name of the new metaobject class as its input. Normally, a need to change the object binding occurs when the run-time conditions are altered. For example in a multimedia application, large increase in the network traffic load may require reducing the quality of service. In mobile computing, moving from one environment to another may require changing some policies in order to match the new environment. By taking the Reflective Java's approach, all these requirements can be met easily by making an application object binding to an appropriate metaobject accordingly.

3.4 Java™ Core Reflection

In version 1.1 of SUN's JDK [SUN], some reflective feature has been introduced to enables Java code to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions. The API accommodates applications that need access to either the public members of a target object (based on its runtime class) or the members declared by a given class.

The Java™ Core Reflection API provides a small, type-safe and secure API which supports introspection about the classes and objects in the current Java Virtual Machine. If permitted by security policy, the API can be used to construct new class instances and new arrays, to access and modify fields of objects and classes, to invoke methods on objects and classes, and to access and modify elements of arrays.

Although both Java™ Core Reflection and Reflective Java intend to provide reflection to Java, there is a fundamental difference. Java™ Core Reflection provides application programs the capability to introspect meta information about their classes and objects, and use it to access and update them. On the other hand, Reflective Java enables programmers to change the behaviour of the method invocation; it is a property of the Java language, not application program.

By allowing application programs to introspect meta information about their classes and objects, Java™ Core Reflection make them easier to do some generic programming, and provides them with some flexibility. For example, application programs can construct a method invocation dynamically. However, unlike in Reflective Java, application programmers cannot change the behaviour of a method invocation. Another difference from Reflective Java

is that Java™ Core Reflection does not provide a clear separation between base level and meta level programming. It only enables the base level programs to do some meta level work.

Like any other Java applications, applications of Reflective Java can enjoy the benefits of Java™ Core Reflection, particularly at the meta level.

4 Binding Specification

One goal of Reflective Java is to provide the freedom for end-users to configure the engineering qualities of an application, namely, what non-functional capabilities an application should have, and what strategies should be chosen to implement them. Therefore, some mechanisms need to be provided in Reflective Java for end-users to make their choices statically as well as dynamically.

In Reflective Java, the behaviour of an application is determined by both the objects implementing the application and their metaobjects. Since the non-functional capabilities are implemented by metaobjects, the engineering qualities of an application object are determined by its metaobject. End-users can choose a particular non-functional capability by using a particular metaobject. A simple declarative language is provided in Reflective Java for end-users to describe the binding specification.

4.1 Method category

The binding specification language is a declarative language. It is used by end-users to specify the relationship between an application class and a metaobject class. The end-users do not need to specify how to establish or maintain this relationship.

To provide more flexibility, Reflective Java allows end-users to make only part of the operations of a class become reflective. Thus, some of the operations of a reflective object can be non-reflective. Non-reflective operations of an reflective object will maintain their original behaviour, and will not suffer any performance penalty due to reflection.

Metaobjects are usually used to implement non-functional requirements that are for general purpose. Thus, it would be ideal to implement them in an application-independent fashion. However to make metaobject effective, some information about the application classes need to be passed to the metatobject. For example, in the case of implementing a metaobject for concurrency control, information about whether an operation is read-only would help the metaobject to decide what kind of lock should be used. Without such information, the metaobject could only apply write locks for every operation in order to ensure consistency; thus losing concurrency greatly. Generally speaking, the more application information is available in a metaobject, the more effective it becomes. On the other hand, if too much application information is revealed, the metatobject would become too application dependent, thus, losing its generality. A balance must be achieved.

An important issue related to making application information available to metaobjects is how to pass this information. It must be done in an application-independent way because the metaobject will be used by multiple applications; when implementing a metaobject, the system user does not know what application classes will use the metaobject late on. For example, in the above

concurrency control metaobject case, the operation names could not and should not be used to check whether an operation is read-only. Otherwise, the metaobject would become class-dependent.

To solve these problems, Reflective Java introduces the concept of *method category*. The metaobject implementation may perform different actions to different categories. Again, take the concurrency control metaobject as an example, suppose that read-only operations are classified as category `201`, and update operations are classified as category `202`. Then, the implementation of the metaobject can be as follows:

```
if (category_id == 201)
then lock.set_read_lock();
else lock.set_write_lock();
```

In this way, although the metaobject does not know which operation will use it, it can still provide appropriate lock for each operation. Clearly, some form of a protocol needs to be set up between a metaobject class and application classes that will use the metaobject class. Normally, the metaobject class determines what kind of category should be specified according to the requiremented strategies. The end-users specify what category an object operation should belong to accordingly.

4.2 Binding specification language

The main purpose of a binding specification is to specify which metaobject class an application class will be associated with. However, as discussed in theprevious section, to provide flexibility and to make metaobjects more effective, the binding specification contains some associated information as well.

The following example shows a binding specification for an application class called Account:

```
package test;
import Name;

refl_class Account: Meta_Lock {

   public Account(Name nm):1;
   public void credit(double amt):202;
   public void debit(double amt) throws Overdraw:202;
   public double check():201;
}
```

The example specifies that an application class called Account will be associated with a metaobject class called `Meta_Lock`, which is indicated in the third line of the binding specification. The first line indicates that the class `Account` belongs to a package called `test`. This line is optional since some classes may not belong to any package. In the second part of the specification, all the classes that are used in the specification are imported. In the above example, the class `Name` is imported since it is used in the constructor of the class.

The body of each reflection class in the specification consists of the constructors of the class and a number of operation signatures. An operation signature has the following syntax.:

```
[accessSpecifier] [synchronized]
        returnType methodName ([paramlist])
        [throws exceptionsList] :category_id;
```

It corresponds to the method signature in Jave with an extra category part.

Due to using class inheritance to achieve method invocation interception, the current version of Reflective Java does not allow `private, static` and `final` methods become reflective. It is still too early to tell whether this limitation would cause difficulties in practical applications.

One point worth mentioning here is that all the constructors of a class must be specified in the binding specification whether they become reflective or not. In order to be able to identify that a constructor does not intend to become reflective, the category identifier `0` is reserved. That means, by specifying a constructor with category identifier `0`, the Reflective Java preprocessor does not make the constructor reflective.

Furthermore, the category identifiers ranging from `1-100` are also reserved for special uses. They should not be used for normal methods.

4.3 The Syntax

```
specification ::= package imports definitions
package::= empty | IDENTIFIER
imports ::= import*
import ::= 'import' import_name ';'
import_name ::= IDENTIFIER['.'IDENTIFIER ]*
definitions ::= definition*
definition ::= refl_class_header '{' refl_class_body '}'
refl_class_header :: = 'refl_class' class_name ':' meta_class_name
class_name ::= IDENTIFIER ;
meta_class_name ::= IDENTIFIER ['.' IDENTIFIER]*
refl_class_body ::= op_decl*
op_decl :: = op_head ':' category ';'
op_head ::= modifier op_name '(' parameter_dcls ')' throw_clause
          |modifier op_type_spec op_name '(' parameter_dcls ')'
                    throw_clause
          |modifier op_type_spec op_name'('parameter_dcls')''['']'
                    throw_clause
op_name ::= IDENTIFIER
modifier ::= empty | PUBLIC |PROTECTED
throw_clause ::= empty | THROWS throw_list
throw_list ::= throw [',' throw]*
throw ::= IDENTIFIER
op_type_spec ::= type_spec |VOID
parameter_dcls ::= empty| param_dcl [',' param_dcl]*
param_dcl ::= type_spec IDENTIFIER
type_spec ::= base_type | ref_type
ref_type ::= type_name | type_spec '[' ']'
base_type ::= 'byte' |'short' |'int' | 'long'|'char' |'float'
|'double '|'boolean' | 'string'
type_name ::= IDENTIFIER { '.' IDENTIFIER}*
category ::= int
```

5 The Meta Level Programming

The main idea of Reflective Java is to separate the concerns of non-functional requirements from functional requirements so that application programmers can focus on the implementation of functional requirements. Non-functional requirements are implemented by system programmers at the meta level.

Although programming at the meta level is still done in the normal Java programming language, there is some extra information, namely the meta data of application objects, can be accessed. This provides meta level programs with extra power to implement non-functional requirements. On the other hand, meta level programming must obey some rules in order to be usable for application objects. These issues are discussed in this section.

5.1 Metaobject class

System programmers implement non-functional requirements at the meta level in form of metaobject class. Reflective Java has implemented a class called `MetaObject` (see Fig 5.1) that defines the basic data attributes for holding metadata of an application class. It also defines two abstract operations: `metaBefore` and `metaAfter` that must be implemented in user defined metaobject classes.

Figure 5.1: MetaObject Class

```
public class MetaObject {
public AST_Class class_metadata;
protected Object applicationobj;
protected String classname;
public MetaObject() { };
public MetaObject(Object appobj, String objnm)
{
 applicationobj = appobj;
 classname = new String(objnm);
}
 public String metaBefore(int m_id, int c_id,ArgPack args)
    { return "noException":};

 public String metaAfter(int m_id, int c_id, ArgPack args,
                                            ArgPack reply)
     { return "noException":};
}
```

User-defined metaobject classes must be a direct or indirect subclass of the `MetaObject` class, and must implement the two abstraction operations `metaBefore` and `metaAfter`.

Users can also define other local data attributes and operations in order to implement the `metaBefore` and `metaAfter` operations.

5.2 Accessing meta data

At the meta level, the meta data of the application object's class is available, such as the name, parameters, type and category of an object operation. This data can be used, for example, to collect statistics of an application. The values of operation parameters are also available at the meta level. They can be accessed as well as updated at the meta level.

Since all the meta data is set up when a metaobject is created, and is local to the metaobject, it is simple to access it. For example, the name of the application class is `class_metadata.class_name`. The name of the first operation of the class is `class_metadata.oper_list[0].oper_name`.

The interface of a metaobject class consists of two operations: metaBefore and metaAfter. Their signature is as follows:

```
public String metaBefore(int m_id, int c_id, ArgPack args)
public String metaAfter(int m_id, int c_id, ArgPack args,
                        ArgPack rpl)
```

Programmers at the meta level implement certain non-functional requirements through these two operations. The metaBefore operation will be invoked before invoking a base level operation, and the metaAfter will be invoked afterwards.

The `m_id` parameter in both operations is the identification of the base level method and `c_id` is its category number. The importance of the category number has been addressed in Section 4.

The values of operation parameters are passed in and out of a metaobject through the `args` parameter of the `metaBefore` and `metaAfter` operations. Meta level programmer need to unpack `args` in order to get the value of a parameter. Some operations are provided in `ArgPack` for this purpose. For example, suppose the type of the first parameter is `double`, its value can be accessed in the following way:

```
double par1 = (double) args.extractArg(0).extractDouble( );
```

If the type of the second parameter is a reference to an object of class Student, then its value can be accessed like this:

```
Student par1 = (Student)args.extractArg(1).extractObject( );
```

At meta level, programmers can make changes to the values of the parameters. However, in order to make the changes effective in the following invocation of the base level operation, the updated parameters need to be packed back into `args`. Some operations are provided in the `ArgPack` for this

purpose. For example, the first and the second parameter can be packed into `args` in the following way:

```
Any _arg_1 = new Any();
    _arg_1.insertDouble(par1);
      args.insertArg(_arg_1);
Any _arg_2 = new Any();
     _arg_2.insertObject((Object)ctl, "Control");
     args.insertArg(_arg_2);
```

Similarly, the result of an object invocation can be accessed and updated at the meta level, but only inside the `metaAfter` operation.

5.3 Category numbers

In Reflective Java, application object classes and metaobject classes are implemented independently. It is the binding specification that connects them together. One important information in the binding specification is the category number of an object method, because it is the category number that decides what kind of actions will be taken at the meta level when the method is invoked. Therefore, the metaobject class must clearly describe the meaning of each category. For example, category `201` implies a read-only operation, and category `231` implies a commit transaction operation. Thus, the end-users can specify the category number of each operation properly. This can be seen as an agreement between the meta level and the application level.

Another issue to pay attention to in meta level programming is to deal with unspecified category number, in the case that the end-user made some mistake in the binding specification. If an unspecified category identification is passed to the meta level, the metaobject should do nothing but just return so that the behaviour of the method is not changed.

Since some category numbers need to be reserved for some special or common operations, category numbers `0 -- 100` have been reserved for this purpose. That is, the category number of normal operations must be greater than `100`.

5.4 Dealing with exceptions

A problem for meta level programming is dealing with exceptions. Since metaobject classes can be used by many application objects, and therefore may throw many different exceptions the metaobject class cannot predict, it is impossible for the `metaBefore` and `metaAfter` operations to specify these exceptions explicitly in their definition.

To solve the above problem, Reflective Java allows the `metaBefore` and `metaAfter` operations to report an exception through their return result. Then, the reflection class produced by the Reflective Java preprocessor can throw a proper exception regarding to the result. For example, if an `Overdraw` exception needs to be raised in a metaobject operation, instead of throwing it, the metaobject operation returns a string "`Overdraw`". Then the reflection object throw s an `Overdraw` exception.

When there is no exception raised in a metaobject operation, it should return the string "`noException`".

5.5 Accessing information of the caller

Sometimes, to implement a non-functional requirement at meta level requires information about the caller of the base level operation. For example, to implement a transaction service using metaobjects, to decide whether an invocation should be allowed to access an object, the identity of the caller is required.

From the implementation of Reflective Java, it is clear that an metaobject lives in the same environment (thread) as the base level object. Thus, if the caller is from the same environment as the base level object, there is no problem to access the information about the caller. However, if the caller is from another environment, for example in the case of remote method invocation, it is tricky for a metaobject to access the caller's information. However, the problem is not caused by meta level programming, but by remote method invocation. Ideally RMI should pass the environment information about the client to the server side, so that the information can be accessed at the meta level as easily as at the base level. However, the current RMI does not pass the required information, so users need to pass it explicitly, for example through a method parameter.

6 Summary

By making the method invocations of Java become reflective, Reflective Java provides applications with the capability to customise their behaviour in order to meet particular application requirements flexibly and transparently. In this documentation, the design and implementation of Reflective Java has been presented. Some guide is also given for building applications using Reflective Java.

In the project, several demonstration examples have been implemented to check and to show the feasibility of Reflective Java. Some of them are trivial such as displaying data in different form according to users needs, and providing concurrency control to object operations. Some are more complicated such as providing data persistence for Java objects , and implementing an object transaction service for Java. The experiences from these demonstrations are very encouraging. It is clear that the main goal of Reflective Java, that is making Java-powered applications customisable according to particular requirements, has been achieved. All the demonstrations are included in the deliverables.

The key technology used to intercept an object invocation in the current implementation is type inheritance. The advantage of this approach is that there is no need to make any change to the Java language, its compiler or its virtual machine. However, as pointed out in Section 3, it also implies some limitations. For example, a private or final operation cannot be made reflective since it is not allowed for a subclass to inherit such operation from its parent class. However, whether this limitation would cause any inconvenience in practical applications needs to be investigated further.

Reflective Java has provided the flexibility required to implement open and flexible applications. The remaining issue is to develop concepts that allow using the provided flexibility in a comfortable but controlled way. For example, suppose an application object is bound to a metaobject that provides some concurrency control mechanism. It is allowed for the application to dynamically change the binding to another metaobject. However, if the new metaobject does not provide proper concurrency control, it will lead the system into an inconsistent state. Therefore, some mechanisms may be required to avoid misusage of the enabled flexibility.

References

[KdRB 91]

Kiczales G., des Rivieres J. and Bobrow D., The Art of the Metaobject Protocol; MIT Press, 1991.

[LINDEN 93]

van der Linden R. J., *An Overview of ANSA*; **AR.000.00**, APM Ltd., Cambridge U.K., May 1993.

[Mae 87]

Maes P. Concepts and Experiments in Computational Reflection; In Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages and Applications, pages 147-155, 1987.

[SUN]

http://www.javasoft.com:80/products/JDK/1.1/docs/guide/reflection/

[SW 95]

Stroud R. J. and Wu Z., Using Metaobject Protocols to Implement Atomic Data Types; In Proceedings of the European Conference on Object-Oriented Programming, pages 168--189, 1995.

[Zim 96]

Zimmermann C., Advances in Object-Oriented Metalevel Architectures and Reflection; CRC Press, Boca Raton, 1996.