Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

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

ANSA Phase III

Reflective Java: Making Java Even More Flexible

Zhixue Wu and Scarlet Schwiderski

Abstract

Java has become popular as a programming language for the Internet because of its ability to simplify the development of flexible, portable applications with graphical user interfaces. Using Java, users can write a program once and run it on any platform. However, Java lacks a good mechanism to support integration of system software, which is required by new applications such as mobile computing, networked interactive multimedia and network programming.

In this paper, we present the Reflective Java project which aims to enable Java-powered systems to be customised dynamically, flexibly and transparently to cater for the particular requirements of an application, or changes in its run-time environment. Reflective Java enables metalevel programming of systems functions as wrappers around method calls. Using metalevel programming, a clear separation can be made between those parts of an application that are concerned with implementing its basic functionality and those parts that are concerned with addressing system issues. Thus, it becomes possible to change the quality of application delivery through alternative infrastructures without disturbing the application components.

We first explain how to use a metaobject protocol to achieve flexibility and customisability. Then we show how reflection can be obtained without making any change to the Java language, its compiler or its virtual machine. As an example, an object transaction service is described to demonstrate the benefits and feasibility of Reflective Java.

APM.1936.02	**Approved**	18th February 1997
	External Paper	

Distribution:
Supersedes:
Superseded by:

Copyright © 1997 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Reflective Java: Making Java Even More Flexible

Reflective Java: Making Java Even More Flexible

Zhixue Wu and Scarlet Schwiderski

APM.1936.02

18th February 1997

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 „ 1997 Architecture Projects Management 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	Section	Title
1	**1**	**Introduction**
3	**2**	**The Proposed Approach**
3	2.1	Application and system components
3	2.2	Reflection and metaobject protocols
4	2.3	Using MOPs to provide flexibility
5	**3**	**The Design and Implementation**
5	3.1	Reflection property of Reflective Java
6	3.2	The implementation
6	3.3	Binding
7	3.3.1	The binding specification
7	3.3.2	An example
8	3.3.3	The changeMeta operation
8	3.4	Meta level programming
10	**4**	**An Object Transaction Service**
10	4.1	The transaction model
11	4.2	Transaction propagation
11	4.3	Implementing objects
13	4.4	The general model
13	4.5	Summary
15	**5**	**Related Work**
16	**6**	**Summary**

1 Introduction

Due to the emergence of new application areas such as mobile computing and interactive multi-media applications, system software needs to meet ever increasing user demands and expectations. Because such applications have varying and even contrary requirements, the *one size fits all* approach inherent in present day "middleware" becomes obsolete.

Mobile computers define a dynamic computing world with intermittent communications. At different times, computers are used at different access points in the net. Their connectivity may vary both in performance and in reliability; the variation of bandwidth and latency may be large. In order to allow applications to run in such different environments, different configurations of supporting system software may have to be provided in each environment.

Java has become popular as a programming language for mobile applications in the Internet because of its ability to simplify the development of flexible, portable applications with graphical user interfaces [AC96]. 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 a downloaded program should be capable of running on any computer in the world.

However, Java does not provide a good mechanism to support adaptability of system software. Java takes the Application Programming Interface (API) approach to implement underlying services. Although this approach works well for providing basic functionality such as I/O, network and threads (i.e., functionality that is application-independent), it is less suited for providing systems functionality, such as distributed processing, concurrency control and data persistence (i.e., functionality that is more application-dependent).

During the design and implementation of an API for a service, it is often necessary to choose between several possible implementation strategies. The resulting API usually either assumes a particular implementation, or provides choices at a very low level. In the first case, the API is suitable for certain application only; in the latter case, it is hard to understand.

The API approach inextricably ties application code to system code. 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 underlying system in order to cater for changing requirements or conditions. This would involve changes to the application's source code.

For example, a database system can replicate data to achieve fault-tolerance so that failures on an object can be recovered by using its replicas. Alternatively, fault tolerance can be achieved through transactions. A transaction ensures that either all or none of its operations are executed. If a transaction is interrupted by a failure, its partial results are undone. Although both replication and transactions are strategies for fault tolerance,

APIs of them would be quite different. Thus, it requires a change application's source code in order to switch from one strategy to the other.

The *Reflective Java* system implemented by APM Ltd. in the ANSA programme aims to solve these problems through metalevel programming by making Java reflective [Mae87, Zim96]. Metalevel programming allows a clear separation to be made between those parts of an application that are concerned with implementing its basic functionality and those parts that are concerned with addressing systems issues. Thus, metalevel programming makes it easier to tune a system component, for example, in order to cater for new application demands, or to adapt to a new environment.

2 The Proposed Approach

2.1 Application and system components

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 well it does it) [SW95]. Non-functional requirements serve a general purpose and are normally addressed by system components. Typically non-functional requirements are concerned with concurrency control, fault tolerance, and data persistence.

Ideally, the implementation of system components should be transparent to applications. This would allow application components to focus on application requirements. In practice, however, extra code needs to be added to the application program in order to utililise the system capability. For example, to support concurrency control a system component might provide two operations, `set_lock` and `release_lock`, to be invoked by an application program explicitly, if it needs to provide concurrency control. The added code increases the complexity of the application program and makes it harder to reason about its correctness. Furthermore, it makes it harder for an application to adapt to a new system component to cater for a new environment. The above example requires the application program to change its source code if it needs to switch to an optimistic concurrency control method.

In this paper, we advocate the *metaobject protocol* [KdRB91] approach to add system capability to an application program, which overcomes the above problems whilst retaining customisability and flexibility. It enables a system to choose system components suitable for a specific application, and to adjust system components to cater for dynamic environment changes.

2.2 Reflection and metaobject protocols

Reflection [Mae87] is the capability of a computational system to reason about and act upon itself. Unlike conventional system, a reflective system allows users to perform computation on the system itself in the same manner as in the application, thus providing users with the ability to adjust the behaviour of the system to suit their particular needs.

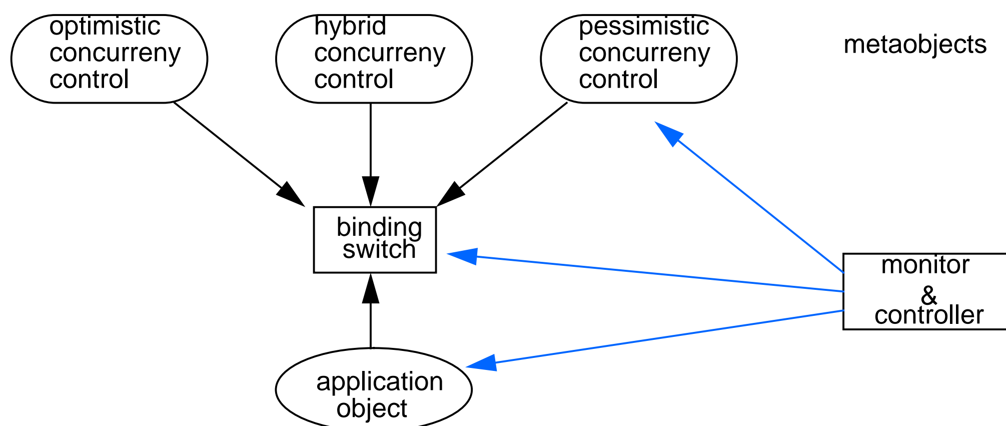
In an object-oriented programming environment, reflection can be realised in the form of metaobjects that represent some internal information and implementation of the system. The interfaces of these metaobjects are called *metaobject protocols (MOPs)* [KdRB91], because they allow application objects to communicate with metaobjects. Through MOPs, users can modify the systems's behaviour and implementation incrementally.

2.3 Using MOPs to provide flexibility

Using metaobject protocols, the actual behaviour of an application object is determined not only by itself, 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. An application object can obtain the capability of a metaobject by binding to it. This makes it possible to provide system capabilities to an application program transparently and flexibly.

The core idea behind this approach is that application requirements are satisfied by application objects, while system issues are addressed via metaobjects. Different ways of addressing a system issue are realised by different metaobjects. Each application object is bound to an appropriate metaobject. In this way, the functionality of an application is determined by its application objects, whilst the quality of application delivery is determined by the associated metaobjects. The quality of application delivery can be changed through alternative metaobjects without making changes to application objects.

Figure 2.1: The idea of the metaobject protocol approach



For example as illustrated in Fig. 2.1, an application object can become usable in a concurrent environment by binding to one of the concurrency control metaobjects. There is no need to make any change to the application object.

A binding between an application object and a metaobject can be changed dynamically according to the run-time conditions. For example, when the conflict rate of concurrently accessing an application object is low, it would be better to bind it to an optimistic concurrency control metaobject. However, when the conflict rate becomes high, the binding can be switched to a pessimistic concurrency control metaobject. By monitoring its components and applications, the system can perform this switching automatically without disturbing application programs.

It is worth pointing out that addressing some system issues does require knowledge of application semantics. For example, in order to ensure data consistency of application objects, the consistency constraints of an application object need to be available to the corresponding metaobject. Therefore if these are not available via reflection, they have to be added as auxiliary information.

3 The Design and Implementation

In order to realise metaobject protocols for providing flexibility to Java applications, we need to make some Java features reflective. Because we want to ensure the pure Java requirement, we take an approach that requires no changes to the Java language, its compiler, or its virtual machine. Therefore, we can ensure that applications using Reflective Java can still run on any platform.

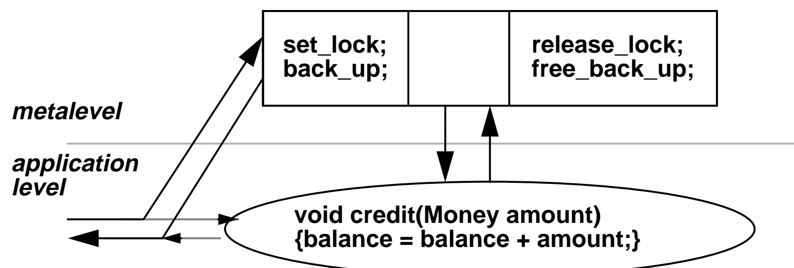
3.1 Reflection property of Reflective Java

Having a fully reflective programming language means that all the properties of the language are accessible and changeable by programmers (in a controlled way). However, making Java fully reflective would be a very complex task, and cannot be done without support from its virtual machine and compiler. Moreover, not all the properties of a 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 enable substitution of alternative system level components.

The current implementation of Reflective Java makes only one property of Java become reflective, namely method invocation. The basic idea is that method invocations can be intercepted and passed to the corresponding metaobject. The metaobject decides how to deal with the invocation. In this way, programmers can make method invocations behave according to their particular needs through the implementation of metaobjects.

Reflective Java enables programmers to change the behaviour of method invocations by specifying what should be done before or/and after “normal” method execution. 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 locks the object and makes a back-up of its state before calling the original object method, and unlocks the object and frees the back-up afterwards.

Figure 3.1: Behaviour of a reflective method call

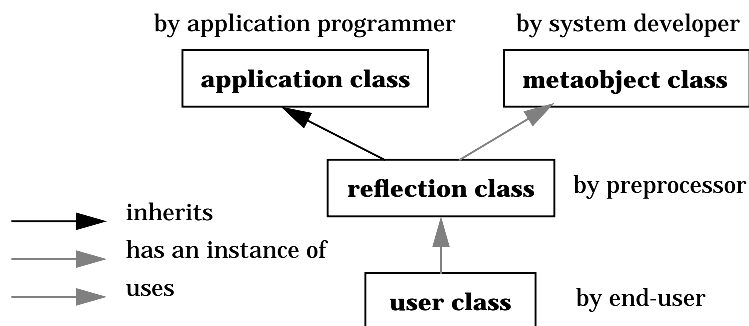


From the example we can see that by using reflection, it is straightforward to provide concurrency control to an object, and it is totally transparent to programmers. Other typical system issues can also be addressed in this way.

3.2 The implementation

The key issue of Reflective Java is to find a way to intercept a method invocation without requiring to make any change to the Java language, its compiler, or its virtual machine. We resolve the issue by making use of class inheritance.

Figure 3.2: Object binding



The general idea is to implement a reflection class as a subclass of an application class. Each operation of an application class is overridden in the reflection class in such a way that an invocation is passed to a metaobject. This is depicted in Fig. 3.2. A reflection class has a `metaobject` variable that references to an instance of a metaobject class.

In this way, if programmers create objects from the reflection class instead of the original application class, the invocation of an object method will be intercepted and dealt with by a metaobject.

A reflection class is generated automatically for an application class by the *Reflective Java preprocessor*. A reflection class also provides a pair of public methods, `getMeta` and `changeMeta`, for checking and changing the metaobject of an application object.

3.3 Binding

There are two kinds of binding: static and dynamic. *Static binding* is done at compile time and cannot be changed subsequently. *Dynamic binding* allows an object to change its binding to a metaobject at run-time.

Static binding is easy to realise, but it does not allow to change the behaviour of an object dynamically. Dynamic binding provides great flexibility to applications by enabling objects to change their behaviour dynamically without stopping. This is an important requirement for new applications. Therefore, dynamic binding is supported in Reflective Java.

3.3.1 The binding specification

A simple declarative language is provided in Reflective Java for end-users to describe the binding specification, that is, to specify the relationship between an application class and a metaobject class.

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

Metaobjects are usually used to address system issues. Thus, it would be ideal to implement them in an application-independent fashion. However to make metaobjects effective, some information about the application classes need to be passed to the metaobject. For instance, 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 this information, the metaobject can only apply `write` locks for each operation; thus losing concurrency to a great extent. 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 metaobject becomes too application dependent, thus, losing its generality. A balance must be achieved.

An issue here is how to pass information of an application object to a metaobject without losing the generality of the metaobject. This 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 later on. For example, in the above case of a concurrency control metaobject, 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 resolve this issue, Reflective Java introduces the concept of *method category* so that a metaobject can perform different actions to different categories. Again, in the concurrency control metaobject 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 locks for each operation. Clearly, some form of a protocol needs to be set up between a metaobject class and application classes. Normally, the metaobject class determines what kind of category should be specified regarding to its implementation. The end-users specify what category an object operation should belong to accordingly.

3.3.2 An example

In summary, a binding specification for an application class describes which metaobject class it intends to bind to, which operations it intends to make reflective, and the category of each operation. 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 `Account` will be associated with a metaobject class called `Meta_Lock`, the category of the `credit` and `debit` operations is 202, and the category of the `check` operation is 201.

3.3.3 The changeMeta operation

To support dynamic binding, Reflective Java provides every reflection class with a `changeMeta` operation with the following signature:

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

In order to change an object binding to a new metaobject, a program calls 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, a large increase in network traffic may require reducing quality of service. In mobile computing, moving from one environment to another may require changing some policies in order to match the new environment.

3.4 Meta level programming

Although programming at the meta level is done in the normal Java programming language some extra information, namely the meta data of application objects, can be accessed. This provides meta level programs with extra power to address system issues. On the other hand, meta level programming must obey certain rules in order to be usable for application objects. For example, any metaobject class must be a subclass of `MetaObject` class. A metaobject class must implement the `metaMethod` operation.

Figure 3.3: A simple lock metaobject class

```
class Meta_Lock extends MetaObject{

public void metaMethod(MID mid, CID cid, Pack arg, Pack rslt)
  {
    if (cid == 201) lock.set_read_lock();
    else lock.set_write_lock();

    rslt = callBaseLevel(mid, arg);

    if (cid == 201) lock.release_read_lock();
     else lock.release_write_lock();
    };

 private Lock lock;

 }
```

Fig. 3.3 shows a metaobject class implementing simple locking operations usable to provide some basic concurrency control. The `mid` and `cid` parameters are used to pass the identifier and the category of an operation of the application class. Metaobject classes may use this information to decide which action to take for a particular operation. The invocation statement `callBaseLevel` executes the corresponding application operation.

4 An Object Transaction Service

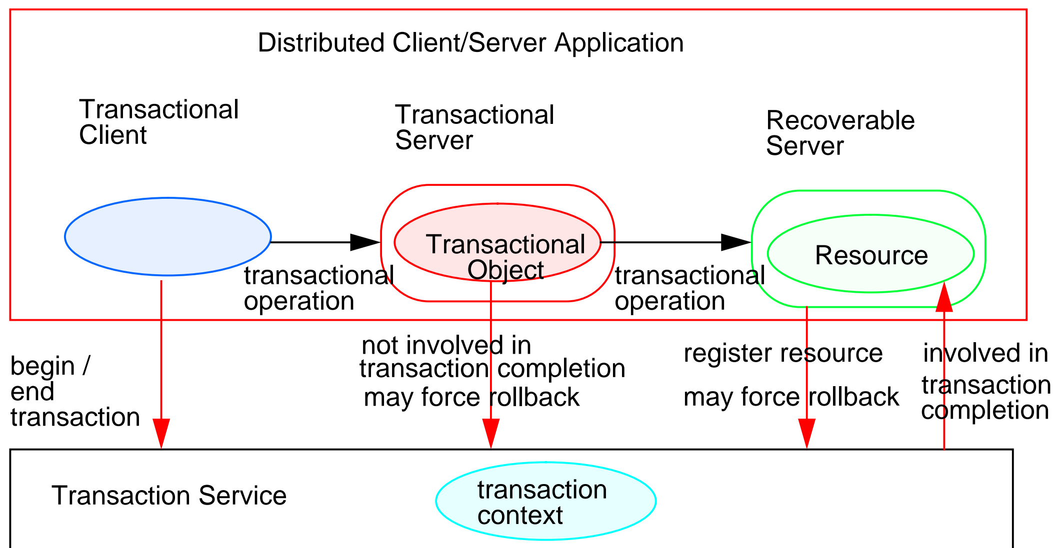
In order to demonstrate and evaluate the benefits of Reflective Java, we have implemented an object transaction service based on the OMG's specification [OMG94]. It is a well-defined transaction model, bringing the transaction paradigm and the object paradigm together.

A major advantage of an object-oriented transaction model is that it enables every object to provide its own concurrency control and recovery, thus providing the possibility for an object to apply a particular concurrency control and recovery policy to cater for its specific requirements. However, this advantage is not exploited fully in the OMG's specification because of a lack of, in our opinion, proper technology. Reflective Java provides the right tool to exploit this advantage.

4.1 The transaction model

Applications supported by the Transaction Service consist of the following entities: Transaction Client (TC), Transaction Objects (TO), Recoverable Objects, Transactional Servers, and Recoverable Servers. Fig. 4.1 shows a simple application which includes these basic elements.

Figure 4.1: Transaction model



A transactional client is an arbitrary program that can invoke operations of many transactional objects in a single transaction. A transactional object is an object whose behaviour is affected by being invoked within the scope of a transaction. A transactional object typically contains, or indirectly refers to, persistent data that can be modified by requests. An object whose data is

affected by committing or rolling back a transaction is called a recoverable object.

A recoverable object must participate in the commitment protocol of the Transaction Service. It does so by registering an object called a `Resource` with the Transaction Service. The Transaction Service drives the commit protocol by issuing requests to the resources registered for a transaction.

4.2 Transaction propagation

The scope of a transaction is defined by a *transaction context* that is shared by the participating objects. A transaction context records all the information related to the transaction, namely the name and identifier of the transaction, the participating objects, and the status of the transaction.

Unlike normal objects, in order to execute an operation a transactional object needs to know the invoker of the operation, i.e. the client transaction which makes the request. Only when it knows the invoker, an object can provide proper concurrency control. An object may also need to access and update a transaction context when executing an operation, for example, to check the status of a transaction or to register itself to a transaction. Therefore, when invoking an operation on a transactional object, the transaction context of the invoker must be propagated to it.

To provide transparency, it would be ideal if the transaction context could be propagated to an object implicitly. For example, we can make the underlying RPC/RMI mechanism pass a transaction context to the server side automatically when it deals with a remote method invocation. The transaction context is then attached to the thread of the transactional object at the server side. Thus, the transactional object can access the transaction context through its local thread. However, this approach requires support from the underlying RPC/RMI mechanism. Our implementation is based on Java's Remote Method Invocation (RMI) mechanism [JAVA 96], which does not provide the required support.

Hence, an explicit approach is taken in our implementation. It requires programmers to declare a transaction context as an explicit parameter of a method so that the RMI mechanism can pass it to the transactional object like any other parameter. Although this approach is not transparent to programmers, it causes little extra work. An object only needs to declare the transaction context as a parameter of every method without any processing in its body. An application program then simply uses the value of the current transaction context, when making an invocation.

4.3 Implementing objects

As described above, objects are responsible for providing their own concurrency control and recovery. Fig. 4.2 shows a skeleton of a recoverable object. All the *italicised* code is for addressing system issues, namely registration, concurrency control and recovery. It is clear that the code for dealing with system issues and the code for implementing application requirements are mixed together. This increases the complexity of the object implementation and makes it harder to reason about its correctness. Furthermore, it also makes it harder for an object to apply a different concurrency control and recovery policy; making changes to the policy means

re-implementing the object. Finally, it also makes it impossible for an object to change its policy dynamically at run-time to cater for changing conditions.

Figure 4.2: Recoverable object

```
class Account extends Resources implements Transactional{
 public void credit( Control ctl,double amt)
  {
   Coordinator co = ctl.get_coordinator( );
   //make sure this object has not been registered for the same transaction
   //make sure this object is involved in only one transaction at a time
   RecoveryCoordinator r = co.register_resource(this);
   balance = balance + amt;
  }
}
```

These problems can be resolved by taking the MOP approach supported in Reflective Java to implement recoverable objects. In this approach, system issues such as concurrency control and recovery can be implemented in metaobjects whilst application requirements are implemented in application objects. By binding to a metaobject, an application object obtains the concurrency control and recovery provided by the metaobject. This clear separation between the code for addressing system issues and the code for implementing application requirements makes the object implementation as simple as in a sequential environment. More importantly, it makes it possible for an object to choose and change its protocol for concurrency control and recovery either statically or dynamically.

Figure 4.3: Recoverable object using MOP

```
class Meta_2pL extends MetaObject {
  public void metaMethod(MID mid, CID cid, Arg args)
  {
  Control ctrl = (Control) args.extractArg(0).extractObject( );
  Coordinator co = ctl.get_coordinator( );
  //ensure this object has not been registered for the same transaction
  //make sure this object is involved only in one transaction at a time
  RecoveryCoordinator r = co ->register_resource(this);
  }
}

class Account {
  public void credit( Control ctl, double amt)
    {   balance = balance + amt; }
 }
```

Fig. 4.3 shows an implementation of a recoverable object by taking the MOP approach. It consists of an application object, and a metaobject that implements concurrency control and recovery. The object implementation is only concerned with the application requirements. There is no code for dealing with concurrency control or recovery.

Note that the `Control` parameter of the `credit` method is used to propagate the transaction context as discussed in Section 4.2.

Although we show the two classes together, the `Account` class can use any metaobject to provide concurrency control and recovery, and `Meta_2pl` can be used by any application object. Furthermore, an application object can switch from one concurrency control and recovery protocol to another at run-time.

4.4 The general model

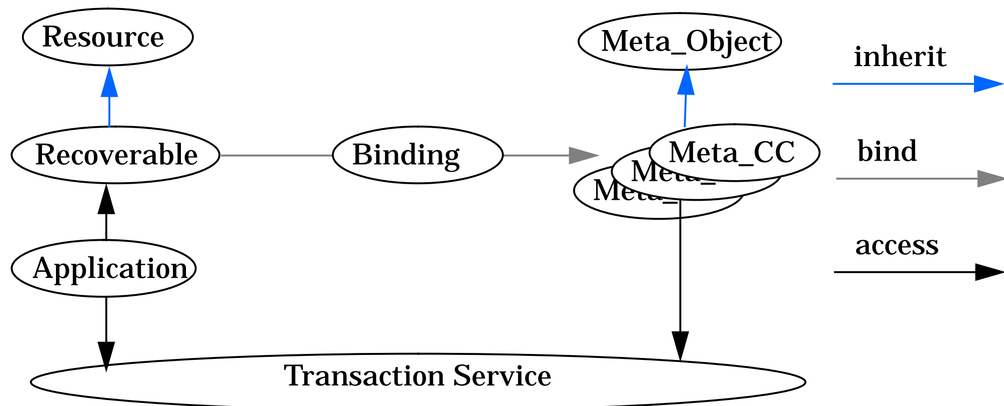
Fig. 4.4 shows our implementation of the transaction model taking the MOP approach. The transaction model uses the 2-Phase-Commitment (2PC) [CP84] protocol to ensure atomic commitment. The interface of the 2PC protocol is defined in the `Resource` class. A recoverable class inherits it by declaring itself as a subclass of the `Resource` class.

A metaobject class `Meta_CC` implements a concurrency control and recovery policy. It also implements the interface of the 2PC protocol. As required by Reflective Java, it must be a direct/indirect subclass of the `Meta_Object` class.

A binding specification describes which metaobject a recoverable class intends to bind to. When an instance of a recoverable class is created, it will be bound to an instance of the corresponding metaobject class automatically. However, this binding can be changed dynamically through the `changeMeta` operation.

An application object uses the operations provided by the Transaction Service to construct a transaction that accesses recoverable objects. The Transaction Service works together with the metaobjects to ensure transactions work properly.

Figure 4.4: Relationship between classes



4.5 Summary

By taking the MOP approach to implement transactional objects, great flexibility has been achieved. An object can choose any available policy to provide concurrency control and recovery. Furthermore, an object can change its policy without making any change to itself, and can do so dynamically without stopping the application. This means that applications can choose a policy suitable for their particular requirements. It also means that it is

possible for an object to adjust its policy dynamically to cater for changing conditions.

The MOP approach also provides a high concurrency transparency to application objects. It requires almost no extra work to make an application object transactional. This relieves application programmers from the heavy burden of dealing with concurrency control and recovery.

By supporting dynamic binding, Reflective Java enables new concurrency control and recovery policies to be introduced to a system at any time. The new policy can be used by a running application without stopping.

Although our transaction service is a simple prototype and some features, such as nested transactions and crash recovery, have not been implemented, it has demonstrated the benefits of Reflective Java.

5 Related Work

CLOS [KdRB91] was the first programming language whose design was based on the metaobject protocol approach. Instead of providing a single fixed language with a single implementation strategy, CLOS provides a surrounding region of alternatives. Users are free to move to whatever point in that region best matches their particular requirements.

Open C++ [Chi95] builds a meta level architecture for C++; it is a compile-time MOP. The Open C++ compiler translates an Open C++ program into a C++ program. The translation process can be customised by meta level programming. However due to its compile-time architecture, meta classes cannot be attached to application classes dynamically at run-time. Thus, it cannot support run-time flexibility.

CodA [McA95] aimed to support distributed applications. It uses fine-grained decomposition into different concepts of application objects: *send, receive, accept, queue, protocol,* and *state*. Thus, different policies can be used to implement a concept.

Like Reflective Java, *MetaJava* [KG96] also intends to support meta programming in Java, but by extending Java's virtual machine. Although this approach may gain performance, it implies that a specific virtual machine is required in order to run a MetaJava program. This contradicts with the Java philosophy.

The *Java™ Core Reflection* [Java96] provides a small, type-safe and secure API which supports introspection about the classes and objects in the current Java virtual machine. There is a fundamental difference between Java™ Core Reflection and Reflective Java. The former provides application programs with the capability to introspect meta information about their classes and objects, and use it to construct new class instances, and update fields of objects and classes. On the other hand, Reflective Java enables programmers to change the behaviour of the method invocation that is a property of the Java language. Therefore Java™ Core Reflection cannot provide the benefits of Reflective Java. Of course, like any other Java applications, applications of Reflective Java can take advantage of Java™ Core Reflection, particularly at the meta level.

Reflection has been used in many application areas: flexible programming [KdRB91], concurrent programming[MWY91], distributed systems[CM93], fault tolerant applications [FNP+95] and soft real-time applications [HT92]. The *Apertos* operating system [Yok92] uses the framework of object/metaobject separation for supporting very large scale, open, distributed systems featuring mobile computing. Using reflection as a general approach to implementing non-functional requirements has been discussed in [Str93]. We believe that by combining the portability of Java and the flexibility introduced through reflection, Reflective Java provides a proper tool to address these application requirements.

6 Summary

By making the method invocations of Java reflective, Reflective Java provides applications with the capability to customise their behaviour in order to meet particular application requirements flexibly and transparently. In this paper, the design and implementation of Reflective Java has been presented. An application example, namely an object transaction service, has been used to show its feasibility.

The key technology used to intercept an object invocation in the current implementation is class 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, 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 override such an operation. 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

[AC96]

Arnold K. and Gosling J., *The Java Programming Language*; Addison Wesley, Reading, MA, 1996.

[Chi95]

Chiba S., *A Metaobject Protocol for C++*; In Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications, pages 285-299, 1995.

[CM93]

Chiba S. and Masuda T., *Designing an Extensible Distributed Language with a Meta-Level Architecture*; In Proceedings of European Conference on Object-Oriented Programming, pages 483-501, 1993.

[CP84]

Ceri S. and Pelagatti G., *Distributed Databases: Principles and Systems*; McGraw-Hill, New York, 1984.

[FNP+95]

Fabre J., Nicomette V., Perennou T., Stroud R. J. and Wu Z., *Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming*; In Proceedings of the 25th IEEE Symposium on Fault Tolerant Computing Systems, 1995.

[HT92]

Honda Y. and Tokoro M., *Soft Real-Time Programming through Reflection*; In Proceedings of International Workshop on New Models for Software Architecture: Reflection and Metalevel Architecture, pages 12-23, 1992.

[JAVA96]

JavaSoft, *JavaTM Remote Method Invocation Specification*; JavaSoft, November 1996.

[KdRB91]

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

[KG96]

Kleinoder J. and Golm M., MetaJava: *An Efficient Run-Time Meta Architecture for Java;* In Proceedings of the International Workshop on Object Orientation in Operating Systems October 1996.

[McA95]

McAffer J., *Meta-Level Programming with CodA*; In proceedings of the European Conference on Object-Oriented Programming, pages 190-214, 1995.

[Mae87]

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.

[MWY91]

Matsuoka S., Watanabe T. and Yonezawa A., *Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming*; In Proceedings of European Conference on Object-Oriented Programming, pages 231-250, 1991.

[OMG94]

OMG, *Object Transaction Service*; OMG document 94.8.4, August 1994.

[Str93]

Stroud R. J., *Transparency and Reflection in Distributed Systems*; in ACM Operating Systems Review, 27(2):99-103, APril 1993.

[SW95]

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

[Yok92]

YoKote Y., *The Apertos Reflective Operating System: The Concept and Its Implementation*; In Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 414-434, 1992.

[Zim96]

Zimmermann C., *Advances in Object-Oriented Metalevel Architectures and Reflection*; CRC Press New York, 1996.