APM

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 and Implementation of A Transaction Service for Java

Zhixue Wu

Abstract

.As part of the Reflective Java project inside the ANSA programme, a transaction service for Java has been implemented. This documentation reports on its design and implementation. The main purpose of this work is twofold: to demonstrate and evaluate the benefits of Reflective Java, and to provide a transaction service for the Java programming language.

The design of the transaction service is based on the OMG's specification for an Object Transaction Service (OMG Document 94.8.4). The key technology behind the implementation is the reflection and metaobject protocol mechanism provided by Reflective Java.

APM.1923.01 **Approved** 24th January 1997

Technical Report

Distribution:
Supersedes:
Superseded by:

Design and Implementation of A Transaction Service for Java

Design and Implementation of A Transaction Service for Java

Zhixue Wu

APM.1923.01

24th January 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

Contents

1 Introduction

As part of the Reflective Java project inside the ANSA programme, a transaction service for Java has been implemented. This documentation reports on its design and implementation. The main purpose of this work is twofold: to demonstrate and evaluate the benefits of Reflective Java, and to provide a transaction service for the Java programming language.

The design of the transaction service is based on the OMG's specification for an object Transaction Service (OTS) [OMG 94]. It is a well-defined transaction model, particularly, bringing the transaction paradigm and the object paradigm together. Another reason behind this decision is that the OMG's specification is a potential industry standard.

A big difference between an object-oriented transaction model and the conventional system-based transaction model is that in the latter, the system provides concurrency control and recovery for all the application objects of the system; whilst in the former, every object provides 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 in the OMG's specification because of lacking, in our opinion, proper technologies.

The reflection and metaobject protocol mechanism supported by Reflective Java provides the right tool to exploit the advantage of providing own concurrency control and recovery. Using Reflective Java [WS 96], a concurrency control method is implemented in a metaobject whilst the functional requirements are implemented in application objects. Providing an object with concurrency control capability can be simply done by binding it to a metaobject supporting concurrency control. It is also very easy to change the concurrency control policy of an object. This can be done by binding an object to another metaobject. Thus, it is very flexible for an object to choose and change its concurrency control policy.

In the next section, we first introduce the transaction model. Then in section 3, we present the classes that implement the transaction service. Section 4 describes how to provide flexible concurrency control to application objects via the metaobject protocol approach. The paper concludes with a summary.

2 Overview of the Transaction Model

This section gives a brief overview of the transaction model defined in the OMG's specification of the Object Transaction Service.

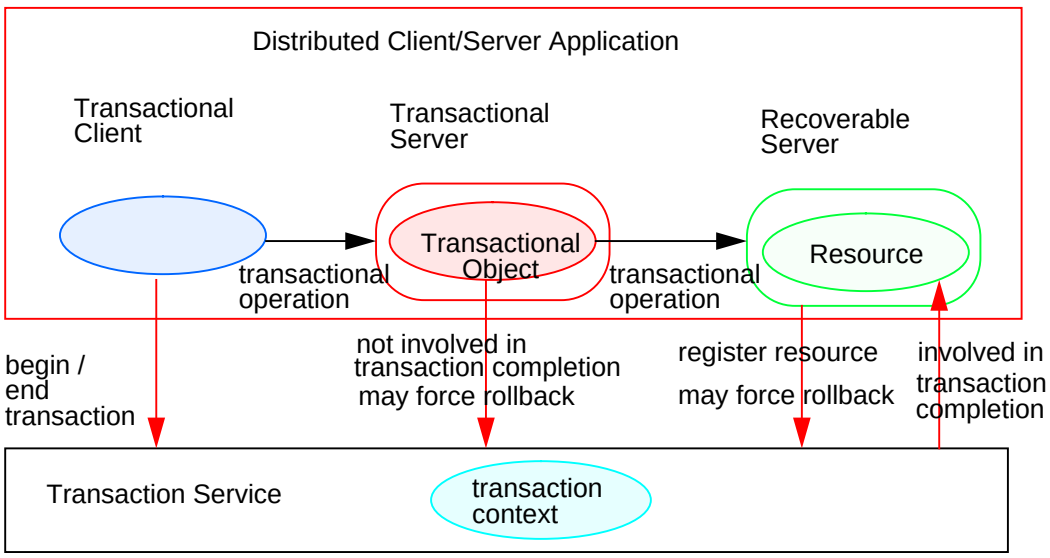
2.1 The transaction model

Applications supported by the Transaction Service consist of the following entities:

- Transaction Client (TC)
- Transaction Objects (TO)
- Recoverable Objects
- Transactional Servers
- Recoverable Servers

Fig. 2.1 shows a simple application which includes these basic elements.

Figure 2.1: 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.

To implement transactional behaviour, an object must participate in certain protocols defined by the Transaction Service. These protocols are used to

ensure that all participants in the transaction agree on its outcome and to recover from failures.

To be more precise, an object is required to participate in these protocols only if it directly manages data whose state is subject to change within a transaction. 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 Transaction Service protocols. 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.

A transactional server is a collection of one or more objects whose behaviour is affected by the transaction, but have no recoverable states of their own. Instead, they implement transactional changes using other recoverable objects. A transactional server does not participate in the completion of the transaction, but it can force the transaction to be rolled back.

A recoverable server is a collection of objects, at least one of which is recoverable. A recoverable server participates in the protocols by registering one or more `Resource` objects with the Transaction Service. The Transaction Service drives the commit protocol by issuing requests to the resources registered for a transaction.

2.2 Transaction service functionality

The Transaction Service provides operations to:

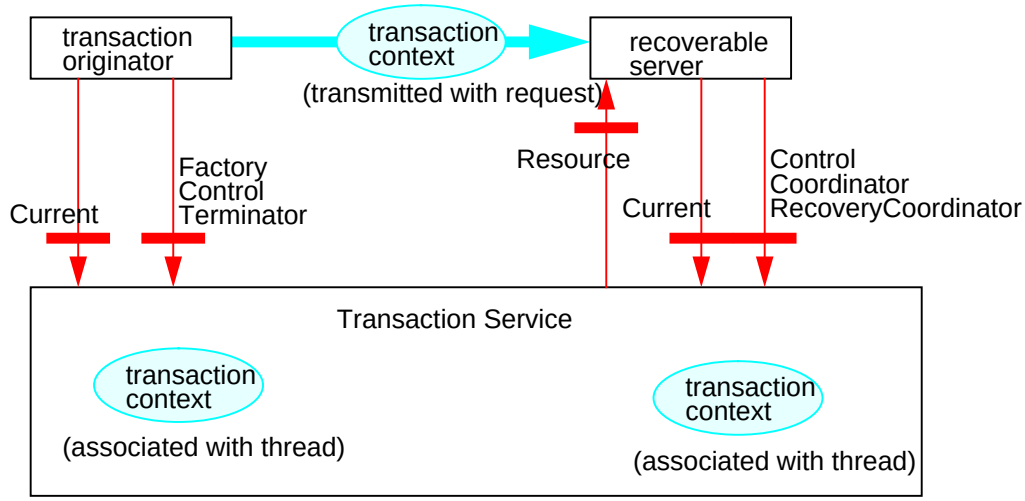
- control the scope and duration of a transaction
- allow multiple objects to be involved in a single, atomic transaction
- allow objects to associate changes in their internal state with a transaction
- coordinate the completion of transactions.

2.3 Transaction service specification

Fig. 2.2 illustrates the major components and interfaces defined by the Transaction Service. The transaction originator is an arbitrary program that begins a transaction. The recoverable server implements an object with recoverable state that is invoked within the scope of the transaction, either directly by the transaction originator or indirectly through one or more components that support the interfaces defined in the specification.

The transaction originator creates a transaction using a `Factory`; a `Control` is returned that provides access to a `Terminator` and a `Coordinator`. The transaction originator uses the `Terminator` to commit or roll back the transaction. The `Coordinator` is made available to recoverable servers, either explicitly or implicitly. A recoverable server registers a `Resource` with the `Coordinator`. The `Resource` implements the two-phase-commit protocol which is driven by the Transaction Service. A `Resource` uses a `RecoveryCoordinator` in certain failure cases to determine the outcome of the transaction and to coordinate the recovery process with the Transaction Service.

Figure 2.2: Components and interfaces of the Transaction Service

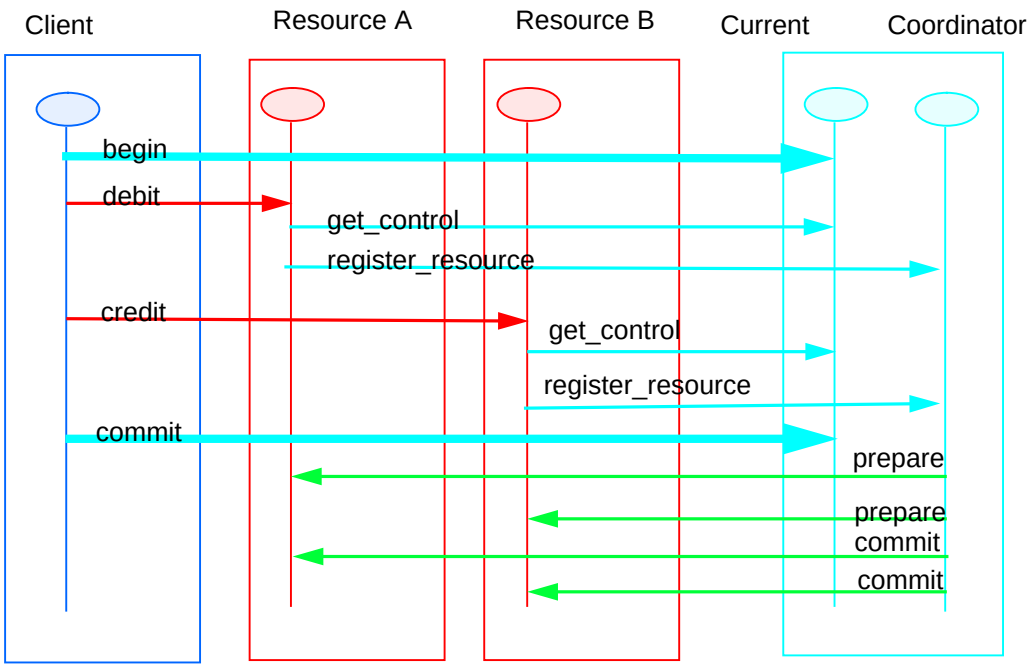


To simplify coding, most applications use the Current pseudo object, which provides access to an implicit per-thread transaction context.

2.4 An example

Fig. 2.3 shows the process of a transaction that intends to transfer some money from one account to another. The operations invoked in each step are also indicated.

Figure 2.3: The process of a transaction



The client starts the transaction by invoking the `begin` operation on the `Current` object associated with the current thread. Then it calls the `debit`

operation on Resource A. When receiving the invocation, Resource A at first obtains the Coordinator object by calling the get_control operation on the Current object. Then it registers itself to the Coordinator object, using the register_resource operation. Finally it performs the debit action. The credit invocation to Resource B is processed in the same way.

After finishing the transfer work, the client ends the transaction by invoking the commit operation on the Current object. Then the Coordinator object starts the first phase of commitment by issuing a prepare command to each registered resource, i.e. Resource A and Resource B. Suppose both resource objects vote for commit, the Coordinator object asks each resource object to commit the transaction by sending a commit request. Otherwise, a rollback request is sent.

3 Transaction Service

This section introduces the classes of the Transaction Service, and discusses some problems we encountered when implementing them.

3.1 Transaction context 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 who 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 help from the underlying RPC/RMI mechanism. Our implementation is based on Java's Remote Method Invocation (RMI) mechanism [JAVA 96]. Unfortunately, it does not provide the required support.

Therefore, 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.

3.2 Updating the transaction context

Another problem we experienced during the implementation is updating the transaction context. The context of a transaction is shared by all transactional objects participating in that transaction. It is forwarded to an object when the transaction makes a request to it. The object may need to make updates to the

transaction context. However in Java's RMI, all parameters of a remote request are passed by copy. Thus, any change made in the server side has no effect in the client side; the changes an object made to the transaction context cannot be passed back to the client side automatically.

To solve this problem, we provide a remote interface for the transaction context so that it can be updated by an object remotely. This will affect the system performance in a large extent, if several updates need to be made for one request. To reduce this problem, we make it possible for an object to update any part of a transaction context in one operation. Then an object can make updates locally and only perform one remote update, just before the end of a method execution.

3.3 Classes

In this section, we present the interface of the classes that implement the Transaction Service. Except the problems discussed in the previous sections, the implementation of these classes is straightforward.

3.3.1 Basic classes

The following classes in the `Transactions` package define the various statuses of a transaction and the types that a transaction can vote:

```
class Status {
 public static final int StatusActive = 0;
 public static final int StatusMarkedRollback = 1;
 public static final int StatusPrepared = 2;
 public static final int StatusCommitted = 3;
 public static final int StatusRolledBack = 4;
 public static final int StatusUnknown = 5;
 public static final int StatusNoTransaction = 6;
}

class Vote {
 public static final int VoteCommit = 0;
 public static final int VoteRollback = 1;
 public static final int VoteReadOnly = 2;
}
```

3.3.2 The Current class

The `Current` class defines operations that allow a client of the Transaction Service to explicitly manage the association between threads and transactions. The `Current` interface also defines operations that simplify the use of the Transaction Service for most applications. These operations can be used to begin and end transactions and to obtain information about the current transaction.

`begin`

A new transaction is created. The transaction context of the client thread is modified. If the client thread is not currently associated with a transaction, the new transaction is a top-level transaction. The `SubtransactionsUnavailable` exception is raised if the client thread already

has an associated transaction, because nested transactions are not supported in our implementation.

```
interface Current {
    void begin( );
    void commit( );
    void rollback( );
    Status get_status( );
    String get_transaction_name( );
    Control get_control( );
}
```

`commit`

If there is no transaction associated with the client thread, the `NoTransaction` exception is raised. Otherwise, the transaction associated with the client thread is completed. The client thread transaction context is restored to its initial status.

`rollback`

If there is no transaction associated with the client thread, the `NoTransaction` exception is raised. Otherwise, the transaction associated with the client is rolled back. The client thread transaction context is restored to its initial status.

`rollback_only`

If there is no transaction associated with the client thread, the `NoTransaction` exception is raised. Otherwise, the transaction associated with the client thread is modified so that the only possible outcome is to rollback the transaction.

`get_status`

If there is no transaction associated with the client thread, the `StatusNoTransaction` value is returned. Otherwise, this operation returns the status of the transaction associated with the client thread.

`get_control`

If the client thread is not associated with a transaction, a `null` object reference is returned. Otherwise, a `Control` object is returned that represents the transaction context currently associated with the client thread.

3.3.3 The Control class

The `Control` class allows a program to explicitly manage or propagate a transaction context. An object of the `Control` class is implicitly associated with one specific transaction.

The `Control` class defines two operations. The `get_terminator` operation returns a `Terminator` object, which supports operations to end the transaction. The `get_coordinator` operations returns a `Coordinator` object, which supports operations needed by resources to participate in the transaction.

```
interface Control {
    Terminator get_terminator( );
    Coordinator get_coordinator( );
}
```

`get_terminator`

An object is returned that supports the `Terminator` class. The object can be used to rollback or commit the transaction associated with the `Control` object. The `Unavailable` exception may be raised if the `Control` object cannot provide the requested object.

`get_coordinator`

An object is returned that supports the `Coordinator` class. The object can be used to register resources for the transaction associated with the `Control` object. The `Unavailable` exception may be raised if the `Control` object cannot provide the requested object.

3.3.4 The Coordinator class

The `Coordinator` class provides operations that are used by the participants of a transaction. These participants are typically recoverable objects. Each object supporting the `Coordinator` class is implicitly associated with a single transaction.

```
interface Coordinator{
    Status get_status( );
    public String register_resource(Resource r)
             throws Inactive, TransactionRolledBack;
    public void rollback_only( ) throws Inactive;
    public String get_transaction_name();
  }
```

`get_status`

This operation returns the status of the transaction associated with the target object.

`register_resource`

This operation registers the specified resource as a participant in the transaction associated with the target object. When the transaction is terminated, the resource will receive requests to commit or rollback the updates performed as part of the transaction. These requests are described in the description of the `Resource` class. The `inactive` exception is raised if the transaction has already been prepared. The exception `TransactionRolledBack` may be raised if the transaction has been marked rollback only.

`rollback_only`

The transaction associated with the target object is modified so that the only possible outcome is to rollback the transaction. The `Inactive` exception is raised if the transaction has already been prepared.

`get_transaction_name`

This operation returns a printable string describing the transaction associated with the target object.

3.3.5 The Resource class

The Transaction Service uses a two-phase commitment protocol to complete a transaction with each registered resource. The `Resource` class defines the operations invoked by the transaction service on each resource. Each object

supporting the Resource class is implicitly associated with a single top-level transaction.

```
class resource {
public int prepare(Control ctl);
public void rollback(Control ctl) throws
            HeuristicCommit, HeuristicMixed, HeuristicHazard;
public void commit(Control ctl) throws NotPrepared,
            HeuristicRollback, HeuristicMixed, HeuristicHazard;
public void commit_one_phase(Control ctl) throws
            HeuristicRollback, HeuristicMixed, HeuristicHazard;
public void forget(Control ctl);
}
```

prepare

This operation is invoked to begin the two-phase commit protocol on the resource. The resource can respond in several ways, represented by the Vote result.

If no persistent data associated with the resource has been modified by the transaction, the resource can return VoteReadOnly. After receiving this response, the Transaction Service is not required to perform any additional operations on this resource. Furthermore, the resource can forget all knowledge of the transaction.

If the resource is able to write (or has already written) all the data needed to commit the transaction to stable storage, as well as an indication that it has prepared the transaction, it can return VoteCommit. After receiving this response, the Transaction Service is required to eventually perform either the commit or the rollback operation on this object.

The resource can return VoteRollback under any circumstance, including not having any knowledge about the transaction. If this response is returned, the transaction must be rolled back. Furthermore, the Transaction Service is not required to perform any additional operations on this resource. After returning this response, the resource can forget all knowledge of the transaction.

rollback

If necessary, the resource should rollback all changes made as part of the transaction. If the resource has forgotten the transaction, it should not do anything.

commit

If necessary, the resource should commit all changes made as part of the transaction. If the resource has forgotten the transaction, it should not do anything.

The NotPrepared exception is raised if the commit operation is performed without performing the prepare operation first.

The heuristic outcome exceptions are used to report heuristic decisions related to the resource. If a heuristic outcome exception is raised, the resource must remember this outcome until the forget operation is performed so that it can return the same outcome in case a rollback or commit is performed again. Otherwise, the resource can immediately forget all knowledge of the transaction.

commit_one_phase

If possible, the resource should commit all changes made as part of the transaction. If it cannot, it should raise the TransactionRolledBack exception.

`forget`

This operation is performed only if the resource raised a heuristic outcome exception to rollback or commit. The resource can forget all knowledge of the transaction.

4 Implementing Objects

In this section, we discuss approaches to implement recoverable objects. First, the problems of the conventional approach are discussed. Then, we show how the problems can be solved by taking the Metaobject Protocol (MOP) approach supported by Reflective Java.

4.1 Object responsibility

In the transaction model specified in OMG's Object Transaction Service, the transaction service is responsible for ensuring that all objects participating in a transaction make the same decision regarding the outcome of the transaction. However, it is the objects that are responsible for ensuring their local consistency, that is, providing their own concurrency control and recovery.

Figure 4.1: Recoverable object

```
class Account extends Resources implements Transactiona{
 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;
  }
}
```

Fig. 4.1 shows a skeleton of a recoverable object. All the grey code is for implementing non-functional requirements. Only black code is for implementing functional requirements. It is clear that the non-functional code makes the object implementation much more complicated.

Allowing objects to provide their own concurrency control and recovery makes it possible for objects to choose protocols suitable for their particular requirements, and could increase the degree of concurrency. However, mixing functional code and non-functional code together causes problems. First, it makes the implementation of an object become more difficult because besides satisfying functional requirements, an object needs to satisfy non-functional requirements. Further, the mixture of two kinds of code makes it hard to apply a different concurrency control and recovery policy to an object. Making change to the policy means re-implementing the object. It also makes it impossible for an object to change its policy dynamically at run-time.

4.2 The MOP approach

The problems described in the last section can be resolved by taking the MOP approach supported in Reflective Java to implement recoverable objects. In this approach, a clear separation between functional and non-functional code can be made so that changes to one do not affect the other. Non-functional requirements, in this case concurrency control and recovery, can be implemented in metaobjects whilst functional requirements are implemented in application objects. By binding to a metaobject, an application object can use concurrency control and recovery provided by the metaobject. This clear separation between functional and non-functional code makes the object implementation as simple as in a sequential environment. More importantly, it makes it easy for an object to choose and change its protocol for concurrency control and recovery either statically or dynamically.

Figure 4.2: Recoverable object using MOP

```
class Meta_2pL extends MetaObject {
  public void metaBefore(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. 4.2 shows an implementation of a recoverable object by taking the MOP approach. It consists of a normal application object that implements the functional requirements, and a metaobject that implements concurrency control and recovery.

It is obvious from Fig. 4.2 that a clear separation between functional and non-functional code has been achieved. The object implementation is only concerned with the functional requirements. There is no code for dealing with concurrency control or recovery. The only non-transparency feature left is the explicit declaration of the parameter `Control` in the operation interface (refer to section 3.1 for an explanation). However, this does not cause any complexity to the object implementation since the object code does not need to deal with it at all.

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, for example, in order to cater for new conditions. In summary, by taking the MOP approach, great flexibility can be achieved.

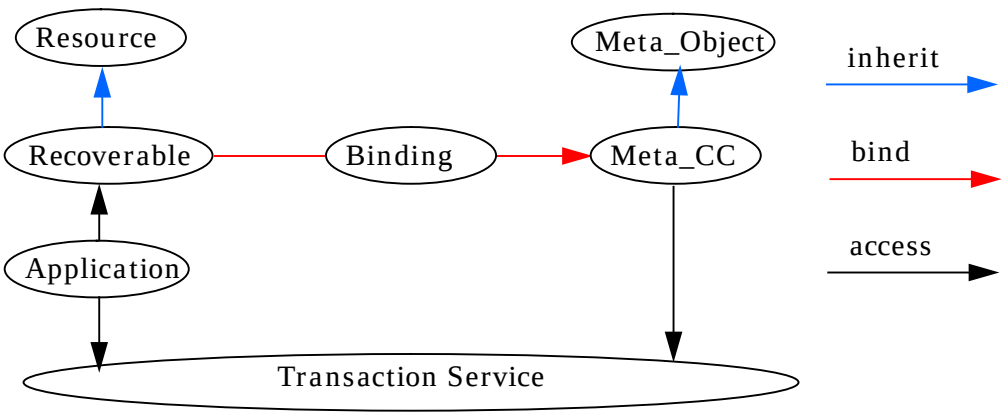
4.3 General model

Fig. 4.3 shows our implementation of the transaction model taking the MOP approach.

The transaction model specified in the Object Transaction Service uses the 2-Phase-Commitment (2PC) protocol to ensure that all objects participating in a transaction make the same decision on its outcome. Every participating object needs to take part in a two phase commitment. The interface of the corresponding operations is given in the `Resource` class (see section 3.3.5). A recoverable class can inherit them by declaring itself as a subclass of the `Resource` class.

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

Figure 4.3: Relationship between classes



A binding specification describes the relationship between a recoverable class and a metaobject class. It specifies that when an instance of that recoverable class is created, it will be bound to an instance of that metaobject class automatically.

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 metaobject that implements concurrency control and recovery to ensure transactions work properly.

4.4 Metaobject implementation

Every object involved in a transaction needs to participate in the two-phase commitment of the transaction. The operations for this are specified in the `Resource` class, and are inherited by all recoverable objects. However, they cannot be implemented in the `Resource` class because the implementation of these operations is closely related to the concurrency control protocol used in a recoverable object. Thus, they should be implemented together with the concurrency control and recovery protocol in a metaobject.

We chooses the 2-phase-locking (2PL) protocol as the first concurrency control policy. It has been implemented in the `Meta_2pl` metaobject class. In 2PL, object operations are classified as either read operations (if they do not make any change to an object), or write operations (if they may update an object).

Before invoking an object operation, a transaction must acquire an appropriate lock for that operation. A transaction can acquire an appropriate lock on an object only if no current transaction holds a conflicting lock on the object. Otherwise, the transaction will block until the object becomes available.

An important issue for a metaobject protocol is to define the category of object operations. Different categories of operations are treated differently in a metaobject. For 2PL, application operations need to be classified as either `read` or `write` operations. Hence, a proper lock can be set when an operation is invoked. Since for the 2PC protocol, every operation needs to be dealt with differently in a metaobject, each of them must have its own category. The category numbers in `Meta_2pl` are defined as follows:

```
101: prepare
102: rollback
103: commit
104: commit_one_phase
105: forget

201: write operations
202: read operations
```

4.5 An example

In this subsection, we present an example in order to show how to define a recoverable object, how to describe a binding specification, and how to construct a transaction.

Taking the MOP approach, a recoverable object deals only with the functional requirements. That is, its implementation is just like that of a normal object. The only difference is that it must be a subclass of the `Resource` class, and each operation must specify a `Control` parameter.

Figure 4.4: A bank Account class

```
class AccountImpl extends Resource implements Account{
    double balance;
    public void credit(Control ctl, double mm)
      { balance = balance + mm; }
    public void debit(Control ctl, double mm) throws Overdraw
      {
       if (balance < mm)
         throw new Overdraw;
       balance = balance - mm;
      }
    public double check(Control ctl)
      {  return balance;}
}
```

A bank `Account` class is shown in Fig. 4.4. It provides three operations: `credit`, `debit`, and `check`. It is clear that there is no code in the implementation for dealing with concurrency control and recovery.

A binding specification describes which metaobject class to which a class would like to bind. Then the instances of the class will be bound automatically

to a metaobject of the specific metaobject class, when it is created. The specification also describes the signature and the category number of every operation that is to become reflective. A binding specification for the Account class is shown in Fig. 4.5.

Figure 4.5: A binding specification

```
refl_class AccountImpl : Meta_2pl {
public void credit(Control ctl, double mm):201;
public boolean debit(Control ctl, double mm) throws Overdraw:201;
public double check(Control ctl) :202;
public int prepare(Control ctl):101;
public void rollback(Control ctl) throws
            HeuristicCommit, HeuristicMixed, HeuristicHazard:102;
public void commit(Control ctl) throws HeuristicRollback,
            NotPrepared, HeuristicMixed, HeuristicHazard:103;
public void commit_one_phase(Control ctl) throws
            HeuristicRollback,HeuristicMixed,HeuristicHazard:104;
public void forget(Control ctl):105;
}
```

Please note that the operations for 2PC are also specified in the binding specification, although they are inherited from the Resource class.

Fig. 4.6 shows a transaction that credits 1000 Pounds in account wu, and then transfers 500 Pounds from account wu to account scarlet.

Figure 4.6: A transfer transaction

```
class Transfer {
   public static void main(String argv[])
     {
      Account wu = new Account();
      Account scarlet = new Account();

      Current txn_crt = new Current();
      Control ctl;

      txn_crt.begin();
           ctl = txn_crt.get_control();
           wu.credit(ctl, 1000);
           wu.debit(ctl, 500);
           scarlet.credit(ctl, 500);
      txn_crt.commit(false);
     }
 }
```

From Fig. 4.6 we can see that it is quite straightforward to construct a transaction. First, the program creates a Current object txn_crt. Then, it invokes the begin operation on the Current object in order to start a transaction. The first thing inside a transaction is to get the control object, because it needs to be used in each operation invocation. Then, the program invokes operations on transactional objects to complete its work. Finally, it uses the commit operation of txn_crt to end the transaction.

5 Summary

A transaction service for the Java programming language has been described in this document. The transaction model is based on the OMG's specification of an Object Transaction Service. The novel point of our implementation is that we take the MOP approach supported by Reflective Java to implement recoverable objects.

By taking the MOP approach, a great flexibility has been achieved. An object can choose any available protocol to provide concurrency control and recovery. Furthermore, an object can change its concurrency control and recovery policy without making any change to itself, and can do so dynamically without stopping the application. This means that users can apply that concurrency control and recovery protocol to an object that is appropriate for their application, for their particular environment, and for their particular requirements. It also means that it is possible for an object to adjust its concurrency control and recovery policy dynamically to cater for changes at run-times. This feature is very useful for applications such as mobile computing, where executing environment changes frequently.

The clear separation between functional and non-functional code also provides a high concurrency transparency to application objects. It requires almost no extra work to make an application object transactional or recoverable. Except for one extra parameter to be declared in the interface of each operation, the implementation of an application object remains the same as the implementation of a normal object. This relieves application programmers from the heavy burden of dealing with concurrency control and recovery.

Furthermore, the clear separation between functional and non-functional code makes it possible for an implementation of a concurrency control and recovery protocol to be used by any application object. Also, it enables new concurrency control and recovery policies to be introduced to an application 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 demonstrates the benefits of Reflective Java.

References

[JAVA 96]

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

[LINDEN 93]

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

[OMG 94]

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

[WS 96]

Wu Z. and Schwiderski S., Design of Reflective Java; APM.1911, APM Ltd., Cambridge U.K., December 1996.