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

Zhixue Wu and Scarlet Schwiderski

Abstract

This is a presentation document for Reflective Java.

The purpose of Reflective Java is to make some features of Java reflective, thus enabling Java-powered system to be customised dynamically, flexibly and transparently to suit a particular application. Method calls are made open-ended; a simple pre-processor that translates reflective programs into standard Java programs and generates classes for binding a Java object to its metaobject.

This presentation at first gives an overview of Reflective Java and then demonstrates its benefits via an application, namely an object transaction service.

APM.1931.01 **Approved** 24th January 1997

Technical Report

Distribution:
Supersedes:
Superseded by:

Reflective Java

Zhixue Wu

Scarlet Schwiderski

28 Jan. 1997

Observations

Requirements:

- The one size fits all design strategy becomes obsolete
  - mobile computing, internet programming, and multimedia applications
  - different considerations and requirements
- System software must be made flexible and customisable at run-time
  - many attributes of the application environment vary from time to time, and from place to place

Technologies:

- Object-oriented programming and language theory has suggested methods for building flexible system software components
  - Java
  - reflection and metaobject protocol (MOP)
- It's time to transfer these ideas to mature technology

Java

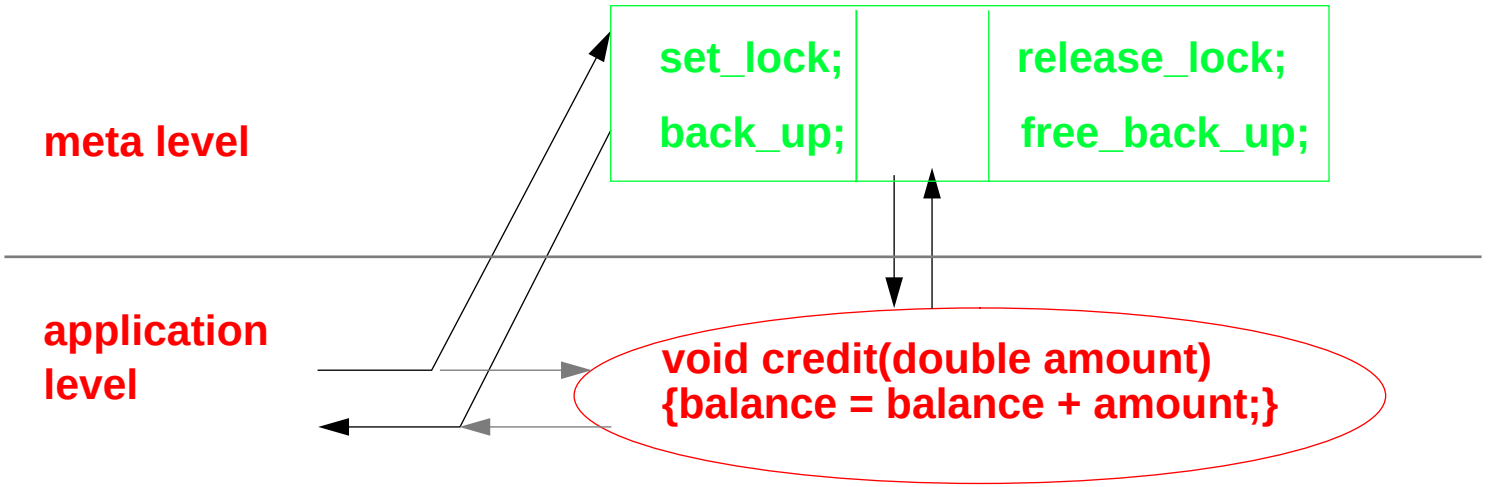
- A simple, object oriented, distributed, interpreted, robust, safe, architecture neutral, portable, high performance, multithreaded, dynamic language
- Advantages
  - object-oriented: separate interface from implementation
  - architecture neutral: write once, run anywhere
  - dynamic loading and linking
- Problems
  - application cannot be decoupled from the choice of non-functional capabilities
  - application is not portable to every infrastructure
  - application cannot adjust its behaviour according to conditions

Reflective Java

- **Enable a Java-powered system to be customised according to the particular application requirements and the run-time environment**
  - **statically at compile time**
  - **dynamically at run-time**
  - **flexibly**
  - **transparently**
- **Make Java reflective**
  - **without any change to the language itself**
  - **without any change to its compiler**
  - **without any change to its virtual machine**

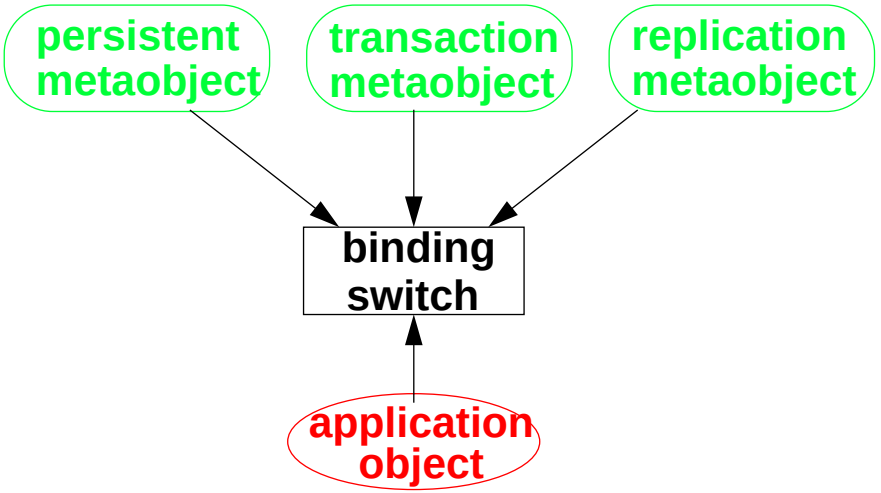
Reflective Method Invocation

- Method invocations are interceptable and changeable by users
  - metaBefore operation
  - metaAfter operation
- Meta data for classes, objects, and parameters is accessible at meta level
- Values of parameters can be manipulated at meta level



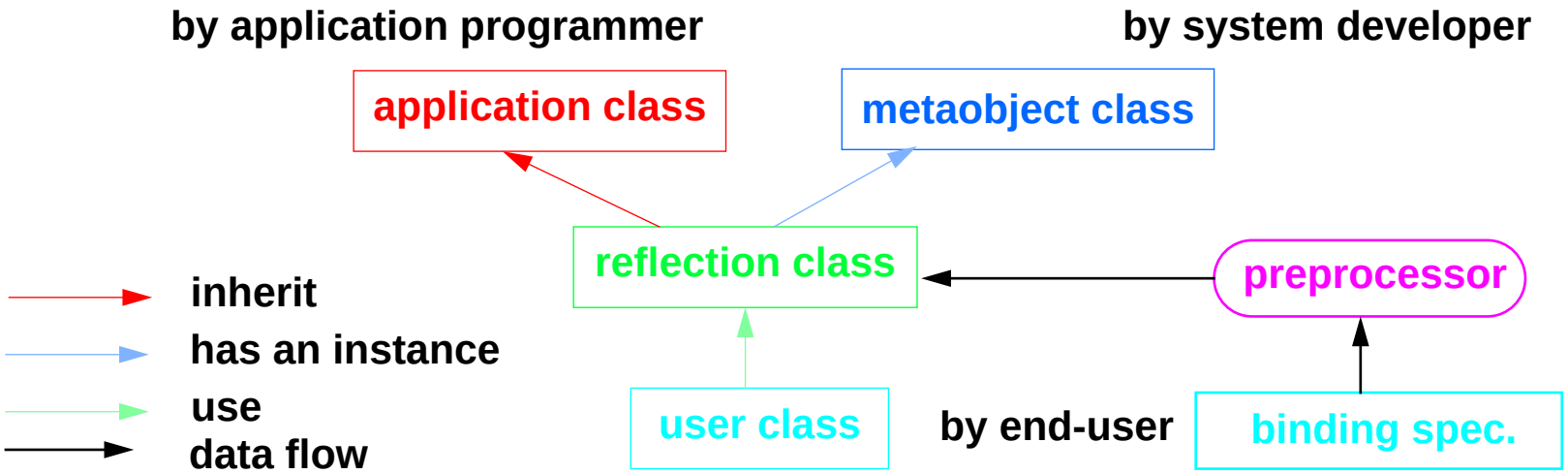
Idea

- Clear separation of functional and non-functional requirements
- Functional requirements are satisfied by application objects
- Non-functional requirements are satisfied by metaobjects
- Non-functional capabilities are added to an application object by binding it to an appropriate metaobject
- Actual behaviour of an application object can be changed by binding it to a different metaobject
- Binding can be changed dynamically



Implementation

- Application classes are implemented by application developers
- Metaobject classes are implemented by system developers
- End-users describe which non-functional capability should be added to an application through a simple declarative language
- A preprocessor generates a reflection class
- The end-user application performs functions through the reflection class

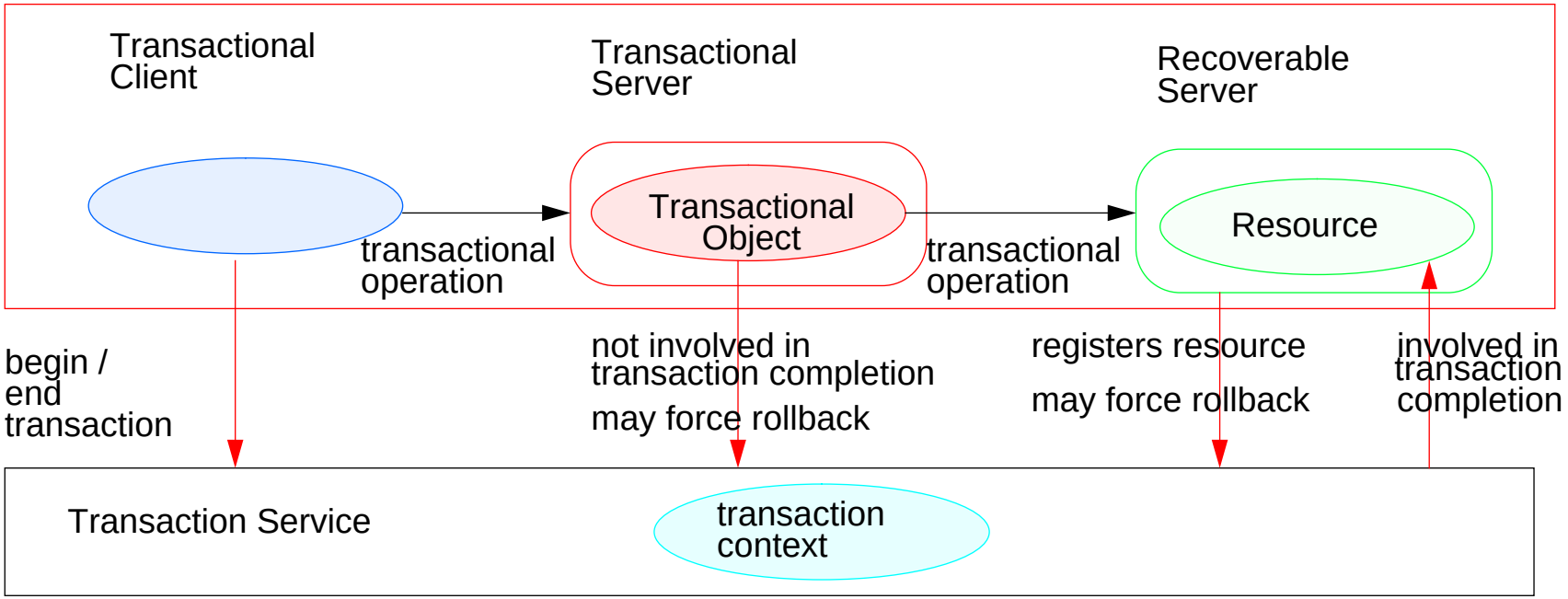


Benefits

- Easy to upgrade product in order to adapt to changes: either in hardware or in application requirements
- Flexibility to customise policies dynamically to suit run-time environment
- High-level transparency to applications
- Free choice of non-functional capability
- Flexible configuration
- Write a Java application once, run it anytime, anywhere, in any environment, with any “-ability”

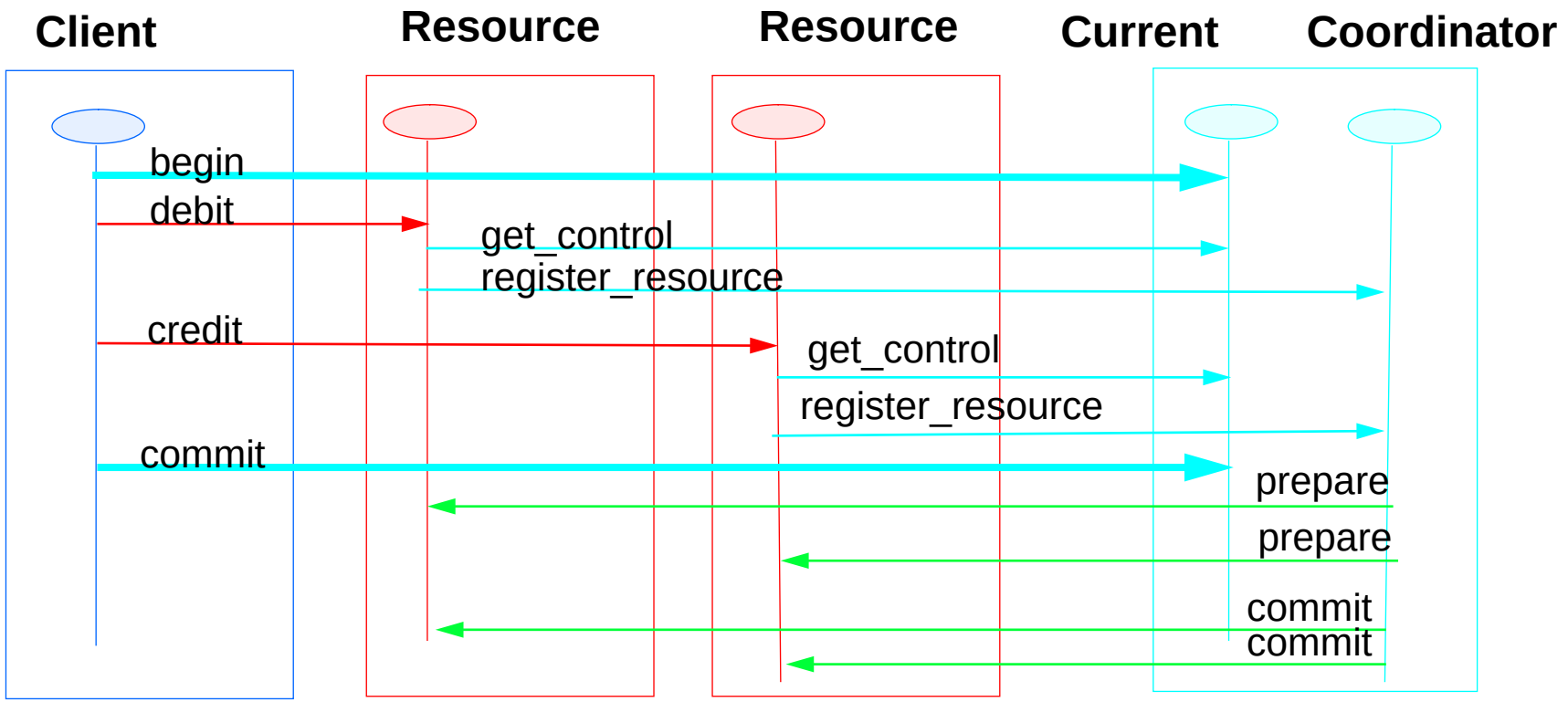
Object Transaction Service

- Based on OMG's specification of Object Transaction Service
- Object-oriented: objects are responsible for concurrency control
- Transaction service: begin, end (2-phase-commitment) transaction



A Transaction Example

- Every object method: deals with concurrency, and registers itself
- Every object: provides prepare, commit, and rollback operations



Recoverable Object

- **An object method**
  - **uses the Control parameter to retrieve the Coordinator object**
  - **registers itself to the transaction service via the Coordinator**
  - **checks whether it is registered for the same transaction**
  - **ensures it is involved only in one transaction at a time**
- **The above is concerned with non-functional requirements**
- **Functional and non-functional code are mixed up**

```
class Account   extends Resource 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 only in one transaction at a time
    RecoveryCoordinator r = co.register_resource(this);
   balance = balance + amt;
 }
 public Vote prepare(...) { .....};
 public void commit(...) {.....};
 public void rollback(...) {.....};
}
```

Recoverable Object (Using MOP)

- **Only functional requirements are implemented in application objects**
- **Non-functional requirements are implemented in metaobjects**
- **Multiple concurrency control methods can be provided**
- **Users can choose a method suitable for their particular application either statically or dynamically**

```
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( );
    //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);
  }
}
```

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

Transactional Client

- **Construct a transaction**
  - **use the begin operation to start a transaction**
  - **a Control object is passed as an explicit parameter of a request**
  - **use the commit operation to end a transaction**
- **Concurrency control method can be changed dynamically**

```
Current txn_crt = new Current( );
Control ctl;

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

```
wu.changeMeta("Meta_2pl");
scalet.changeMeta("Meta_2pl");
txn_crt.begin( );
    ctl = txn_crt.get_control( );
    wu.debit(ctl, 1000);
    scarlet.credit(ctl, 1000);
txn_crt.commit( );
```

Summary

- The benefits and feasibility have been shown clearly
  - flexibility
  - easy to implement: 2 X 3 person months
- Experience
  - transparency: not totally transparent
  - Java's RMI: not flexible, not customisable
  - Java's RMI: pass by copy
- Related work
  - JavaSoft's Reflection API: observation meta data only
  - Meta Java: no separation between functional and non-functional code
  - University of Newcastle: for fault tolerance
  - LAAS: for security and fault tolerance

Information

- APM.1911: Design and Implementation of Reflective Java
- APM.1923: Design and Implementation of Object Transaction Service
- APM.1940: Design and Implementation of a Persistence Service
- Source code release: 23 December 1996
  - ftp.ansa.co.uk
  - phase3/phase3-prototypes/Reflective-Java/reflective-java-01.tar.gz
- Contacts:
  - Zhixue Wu: zw@ansa.co.uk 01223--568930
  - Scarlet Schwiderski: ss@ansa.co.uk 01223--568926