Make Reflection Practical to Use

Zhixue Wu
APM Ltd.
28 Oct. 1998
Transaction Framework

- **Goal:** a transactional architecture with
  - high transparency to application developers
  - high performance
  - flexibility and scalability
  - fast application development

- **Approach**
  - three-tier architecture
  - component technology
  - reflection and introspection
**Deliverables & Current Status**

- A visual component builder tool (beta) ---> (1.0) ---> (2.0)
- A compiler for generating reflection class (beta) --->(1.0) ---> (2.0)
- A system component container (alpha) --->(beta) --->(1.0) --->(2.0)
- A set of concurrency control metaobjects (TPL)
- An object transaction service (75%) --->(beta) --->(1.0b) --->(2.0b)
- A demonstration example ( ) --->(alpha) --->(1.0a) --->(2.0a)
- An architecture report (beta)
- Integration with FlexiNet --->(alpha)
- Packaging to EJB Jar --->(1.0b) --->(2.0b)
- Programming guide
Problem

Reflection is a powerful tool for providing system flexibility and adaptability however some practical issues make it difficult to use
**Behavioural Reflection**

- The behaviour of method invocation can be customised by programmers via metaobjects
  - method invocation is intercepted by a metaobject
  - extra processing can be done before and after method execution
  - meta information for classes, objects, and parameters is accessible
  - values of parameters can be manipulated at meta level

```java
void credit(double amount)
{balance = balance+amount}
```

```
Object metaMethod(Method appMtd, int cId, Objects args[]) throws Throwable
```

- `set_lock;`
- `back_up;`
- `release_lock`
- `free_back_up`

---

© 1998 ANSA Consortium
**Add System Capability via Reflection**

- Business logic is implemented in application objects
- System capabilities are implemented in meta objects
- Integration through metaobject binding (static or dynamic)
- New strategy can be applied through changing metaobject binding

![Diagram with optimistic CC method, pessimistic CC method, hybrid CC method, binding switch, and application object]
Challenges in Programming
Metaobjects

- Generic programming
- Obscure way for accessing meta information
- Difficult to provide multiple capabilities
- Hard for metaobject reuse
- Impossible to use third-party products
- Consistency concerns for dynamic binding
  - between old and new metaobjects
  - system states
Provide Multiple Capabilities

- Multiple inheritance
- Multiple binding
- Multilevel reflection
- Metaobject chain
- Construct a new metaobject
Multiple Inheritance

- Name collision
- Multiple inheritance not supported in Java
Multiple Metaobject Binding

- The application objects will be called multiple times
  - cannot ensure the correct functionality of the application object
Multilevel Reflection

- Each level shifts an invocation to a higher level
- Poor performance:
  - multiple packaging and unpackaging
  - multiple interceptions
- Order:
  - in which order
- Integration is more complex than stacking
- Semantics
Metaobject Chain

- Performance: better than multilevel reflection
- Need to make changes to some metaobjects
- Order problem
- Integration is more complex than stacking
A Two-Layer Approach

- Separate composition and control from functionality implementation
- First-layer metaobject:
  - interception method invocation
  - coordinate second-layer metaobjects
- Second-layer metaobjects:
  - provide a particular capability
- Contract interface between first and second layer metaobjects
  - ensure a second layer metaobject reusable
  - ensure consistency between new and old metaobject
**Metaobject Structure**

- **TransactionalMetaobject**
  - **ConcurrencyControl**
    - TplCC
    - TmStmpCC
  - **PersistencyControl**
    - LazyPersist
    - BusyPersist
  - **RecoveryControl**
    - MethodBased
    - StateBased

**Base Level Objects**
- DepositAccount

**Meta Space**
- Meta Regions
Dynamic Binding

- Change metaobject binding at runtime
  - to cater for environment changes
  - to improve performance by making use runtime information

- When to make change
  - the rules
  - how to ensure the rules

- How to ensure consistency
  - new and old metaobject
  - system states
**Selector**

- A dedicated active object within a first-layer metaobject
- Define the rules for changing metaobjects
- Register to relevant events
- Make decision on when to change metaobject binding
- Perform binding changes if required

**Events**
- Environment events, user interception events, runtime statistics events

**Diagram:**
- **Selector**
  - **Event sources**
  - **Meta Regions:**
    - **Concurrency Control**
    - **Persistency Control**
    - **Recovery Control**
  - **Meta Space**
  - **Base Level Objects:**
    - **TransactionalMetaobject**
    - **DepositAccount**
A Reflective Transaction Architecture

- Container provides a first layer metaobject: `transactionalMetaobject`
- Three second-layer metaobject interfaces: `persistency`, `concurrency`, `recovery`
- Each interface may have multiple implementations
- Application deployer choose metaobjects for a application
- “Off-the-shelf” metaobjects can be used
Summary of the Two-Layer Approach

- Separate composition, interception and control from implementations of subtasks
  - enable easy integration of multiple metaobjects
  - make second-layer metaobject much easier to implement
  - enable metaobject reuse
  - the contract interface ensures compatibility between metaobjects

- First-layer metaobject is responsible for composition, interception and control

- Construct first-layer metaobjects as components
  - easy composition
  - easy customisation