J2EE + AOP = AspectJ2EE

Tal Cohen
Joint work with
Joseph (Yossi) Gil

Technion
Israel Institute of Technology
Untangling the Web We Weave

TRANSACTION MANAGEMENT

LOGGING

BUSINESS LOGIC

SECURITY

method transferAmount(acct A, acct B, amount) {

Clearly distinct concerns

Program requirements

Aspectual Decomposition
(conceptual process)

Aspect Weaving
(mechanical process)

Executable
Aspect-Oriented Programming to the Rescue

• Design the core “business logic”
• Define aspects which are “transformation operators”.
  – Example: a logging operator
• Aspect ≡ Operator ≡ Program Module
  – Operators can be reused for many tasks
  – Should not break the operand
  – Transformation is packaged
• Apply “aspect modules” to adapt the code to support:
  – Logging
  – Security
  – Persistence …
The Enemy of the Good…

- Two conflicting forces in software design:
  - If it is good, you want to add aspects to it …
  - When you change it…
  - In changing it, you might damage it…
  - …now it is not so good anymore!

Le mieux est l’ennemi du bien.
Code Transformation

• An aspect is a “code transformer”.
  – However, there is no simple, unified metaphor or semantics for definition or application of aspects.

• Ideally, we would like the transformer to be:
  – Expressive
  – Modular
  – Structure-preserving, i.e., tightly coupled with the underlying language.
  – Simple

• Bad example:
  – C pre-processor (which does not know anything about C....)

• Good example:
  – mixins -- an abstract sub-class mechanism.
What is “Aspect Application”?

• How can we make AOP more than a fancy pre-processor?

• Uncontrolled and non-systematic change leads to anarchy.
  – Multiple applications of the same transformer?
  – Specification of application order?
  – Parameters to the transformer?
  – Time of transformation? (Compile/Link/Run)
  – Does the non-transformed version remain? Are the two versions compatible?
Our Solution

• **Reduce** all above questions to the familiar domain of OOP
  - Motivation & Inspiration: *Enterprise Applications* and J2EE deployment process.

• In a nutshell, aspect application is a form of subclassing (slightly restricted, slightly generalized…)
  - Strong evidence that “this works”
  - A natural extension of an existing “deployment” process (more later)

Capture the mode of operation of banks, government agencies, insurance companies, hospitals…
  - Make the world go round

Java 2, Enterprise Edition - a modern and popular middleware framework for enterprise applications, implemented by e.g., IBM WebSphere App Server
  - Make the world go round
Aspects as “Reimplements” Operators

A class in OOP is:

1. A type declaration
   \[ C \ p; \]
2. Template for object creation (constructor, field layout etc.)
   \[ p = \text{new } C(); \]
3. An implementation
   \[ p.m(); \]

Simple subclassing:

1. Creates a subtype by adding more functions to the protocol.
2. Extends the structure definition by adding more fields.
3. Modifies the behavior by replacement and refinement

Given a class \( C \), aspect \( A \), the application \( C' = A[[C,p_1,\ldots,p_n]] \) is a new class such that

Actual parameters \( p_1,\ldots,p_n \) make \( C \) fit into the slots that \( A \) expects

variables of type \( C' \)
aspect Logging {
    before every method of the operand {
        print("Method started: ") + currentMethodName);
    }

    after every method of the operand {
        print("Method completed");
    }
}
Applying an Aspect to a Class Hierarchy

Since no new types are defined, this does not disturb polymorphism!
Parameterized Example: A Transaction Aspect

```java
aspect Transactions<
    T, // methods of the operand that require a transaction (possible existing)
    S // methods of the operand that require their own tx (possibly nested)
>
{
    before every method in S { startNewTransaction(); }
    before every method in T
        { if (!inTx) startNewTransaction(); }

    after every method in T ∪ S
        { if (newTxStarted) commit(); }
}
```

- Every application of this aspect must provide values for parameters T and S
# AspectJ2EE as a Distilled Code Transformer

<table>
<thead>
<tr>
<th>Structure template</th>
<th>Class</th>
<th>Inheritance</th>
<th>Mixins</th>
<th>AspectJ2EE Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Define</td>
<td>Extend</td>
<td>Template for structure extension</td>
<td>Template for structure extension</td>
</tr>
<tr>
<td>New Type</td>
<td>Declare</td>
<td>Subtype specific parent</td>
<td>Template for subtyping limited set of parents (operands)</td>
<td>No</td>
</tr>
<tr>
<td>Code implementation</td>
<td>Yes</td>
<td>Modify specific parent</td>
<td>Modify limited set of parents (operands)</td>
<td>Modify any operand</td>
</tr>
<tr>
<td>Parameters</td>
<td>Only with Templates</td>
<td>No</td>
<td>Only for parent class</td>
<td>Expressive binding mechanism</td>
</tr>
</tbody>
</table>
Enterprise JavaBeans (EJBs)

- Represent the **data model** in J2EE applications
- Developers provide the business logic
  - “Transfer amount X from account A to account B”
- J2EE provides everything else
  - **Services**: Security, transactions, persistence…

```java
Account a = AccountFactory.find(12345);
a.deposit(100);
float b = acct.getBalance();
```
J2EE is not as good as AOP

• Closed and pre-defined set of services
  – Need a different service? *Tough Luck!* e.g., to add logging support, you’ll have to use scattered and tangled code.

• Fixed implementation of services
  – e.g., J2EE security is role-based. Need name-based security? *Tough Luck!*

• EJBs are hard to program
  – Just look at the standard specification
  – You mustn’t do this, you mustn’t do that…
    • e.g., “this” may not be passed as a method argument
The Deployment Process

- J2EE services are added to business logic during the deployment process.
- Done without modifying code – only generating support code: Subclasses, Factory classes, RMI stubs/skeletons.

Define EJB Services as Aspects.
Aspect application becomes part of the deployment process.
J2EE Deployment

```
interface javax.ejb.EntityBean
+withdraw()
+deposit()
+getBalance() : float

interface Account
+withdraw()
+deposit()
+getBalance() : float

interface javax.ejb.EJBObject
+create() : Account
+findByPrimaryKey() : Account

interface AccountHome
+create() : Account
+findByPrimaryKey() : Account

interface javax.ejb.EJBHome
+create() : Account
+findByPrimaryKey() : Account

interface javax.ejb.EJBObject
+create() : Account
+findByPrimaryKey() : Account

interface javax.ejb.EntityBean
+withdraw()
+deposit()
+getBalance() : float

interface Account
+withdraw()
+deposit()
+getBalance() : float

User-provided classes

Many, many more classes generated...
```
AspectJ2EE Deployment

Application of Four Aspects to an EJB

The generated classes are subclasses
Join Points Supported by Deployment/Subclassing

Data access

\[ \text{[before | after]} \times \text{[property | private field]} \times \text{[get | set]} \]

Code

\[ \text{[before | after | around]} \times \text{[method | c’tor}^1 \text{ | remote]} \]
\[ \text{[after throwing | after returning]} \times \text{[method | c’tor]} \]

Control flow

\[ \text{[cflow | cflowbelow]} \]
\[ \text{[catch]} \]

1 Standard Java restrictions on constructors apply
The Deployment Descriptor

• XML syntax is used to specify the aspect application details
  – Order of application, parameters, etc.
  – Conceptually based on J2EE’s XML Deployment Descriptors

• For example:

```xml
<entity id="Account">
  <ejb-name>Account</ejb-name>
  <ejb-class>aspectj2ee.demo.AccountBean</ejb-class>
  <aspect>
    <aspect-class>aspectj2ee.core.Lifecycle</aspect-class>
  </aspect>
  <aspect>
    <aspect-class>aspectj2ee.core.Security</aspect-class>
    <pointcut name="secured">execution(*(..))</pointcut>
    <value name="requiredRole">User</value>
  </aspect>
</entity>
```
Remote Calls

• The deployment process also generates RMI stubs and skeletons
• If we allow the user to control the process, he can also control this facet
• In particular, we can define aspects for handling **tier-cutting concerns**
• A single module each (without scattered code) for handling:
  – Encryption/decryption of client-server communications
  – Data compression
  – Memoization
  – Client-side precondition checking
• “Remote Pointcut”, Nishizawa, Chiba and Tatsubori (AOSD ’04)
Summary

1. A smooth embedding of AOP in OOP
2. Proof of the pudding: works in real-world enterprise applications
3. Harmonious marriage of J2EE and AOP
4. New weaving mechanism using the deployment process
5. Realizing tier-cutting concerns with aspects
And they lived happily ever after?

- Integration with WebSphere?
- Wedding AOP with other J2EE sub-technologies
  - Servlets
  - JSP (Java Server Pages)
  - Web services
- User study?
- Refactoring existing project into this model?