JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

Dynamic Aspect-Oriented Programming With AspectWerkz

Jonas Bonér
Senior Software Engineer
Alexandre Vasseur
Software Engineer

java.sun.com/javaone/sf

Session Goal

What you will learn

Learn how to apply AOP in J2EE™

How to integrate AOP into your development process

Speaker’s Qualifications

Jonas Bonér

- Senior Software Engineer at JRockit™ Group, BEA Systems
- Founder of the AspectWerkz AOP framework
- Speaker at JavaPolis 2003, eWorld 2004, AOSD 2004, BEA User Group 2004

Speaker’s Qualifications

Alexandre Vasseur

- Software Engineer at JRockit™ Group, BEA Systems
- Co-founder of the AspectWerkz AOP framework
- Speaker at eWorld 2004, AOSD 2004,

BEA User Group 2004

Agenda

AOP crash course

Demos:

- Hello World
- J2EE integration
- J2EE aspects
  - Role-based security
  - Unit Of Work

Agenda

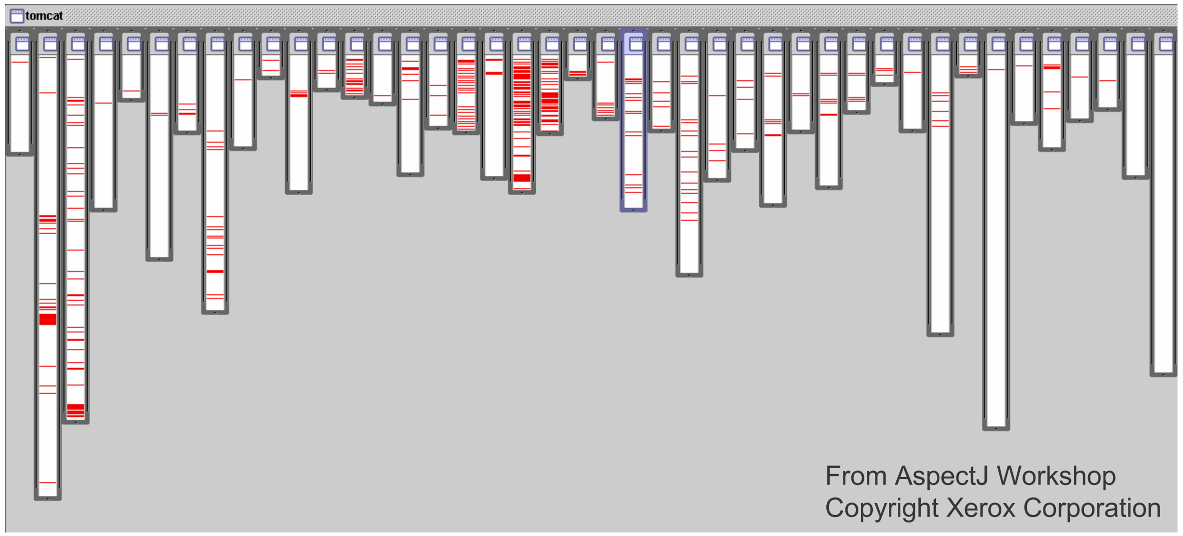
AOP crash course

Demos:

- Hello World
- J2EE integration
- J2EE aspects
  - Role-based security
  - Unit Of Work

AOP crash course

- OOP fails to address 'cross-cutting concerns'
  - Introduces 'code tangling' and 'code scattering'
  - Makes software harder to write, understand, reuse and maintain

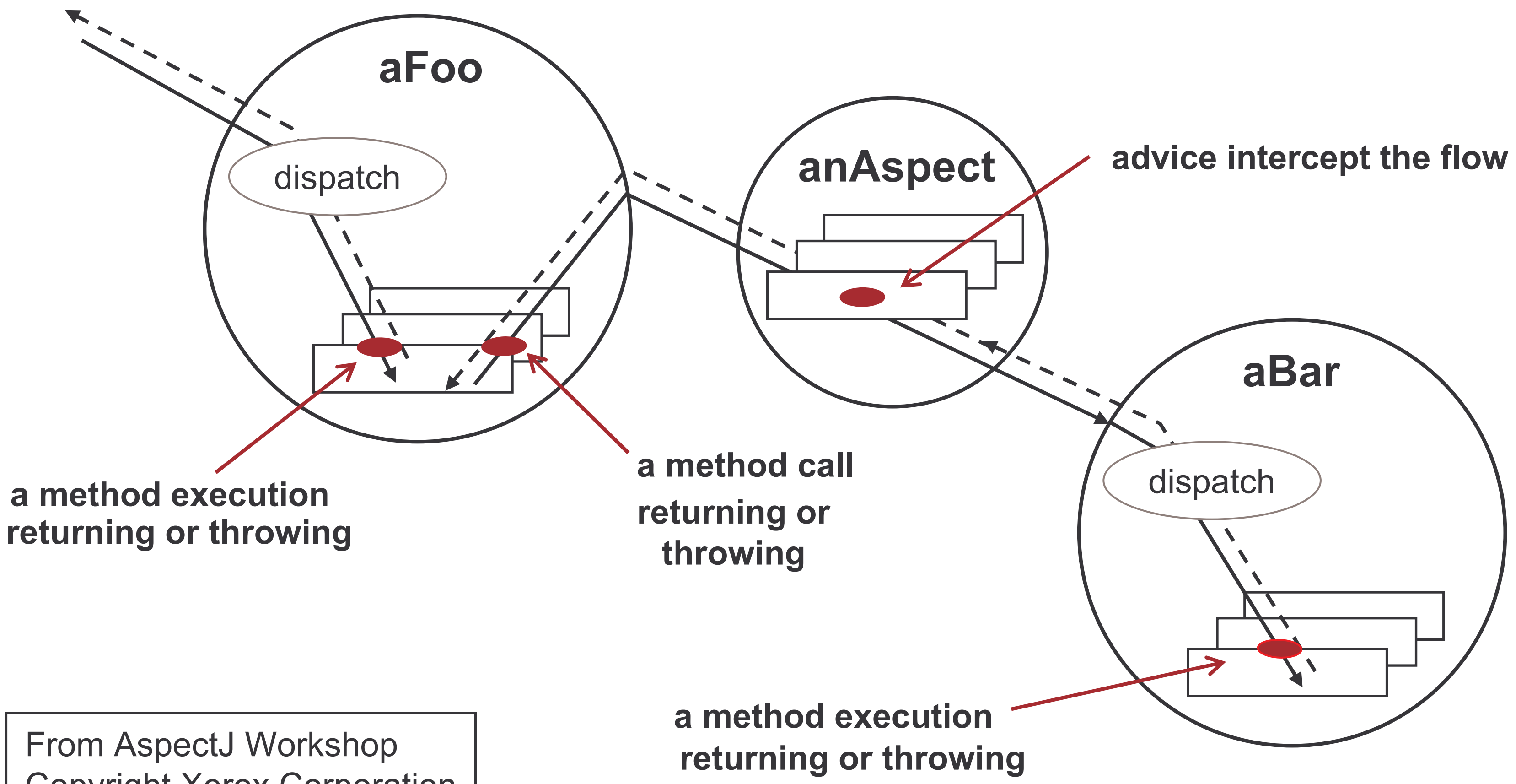


Logging in `org.apache.tomcat` is not modularized

- AOP enables 'Separation Of Concerns'
- The Aspect modularizes a crosscutting concern

Join points

- Well-defined points in the program flow


aFoo
dispatch
anAspect
advice intercept the flow
aBar
dispatch
a method execution returning or throwing
a method call returning or throwing
a method execution returning or throwing

From AspectJ Workshop
Copyright Xerox Corporation

AOP crash course

Core concepts

1. Define well-defined points in the program flow
   – Join points
2. Pick out these points
   – Pointcuts
3. Influence the behaviour at these points
   – Advice, Introductions
4. Weave everything together in a functional system
   – Weaver

AOP crash course (with AspectWerkz)

- Regular expression based pattern language

- Defined as Annotations

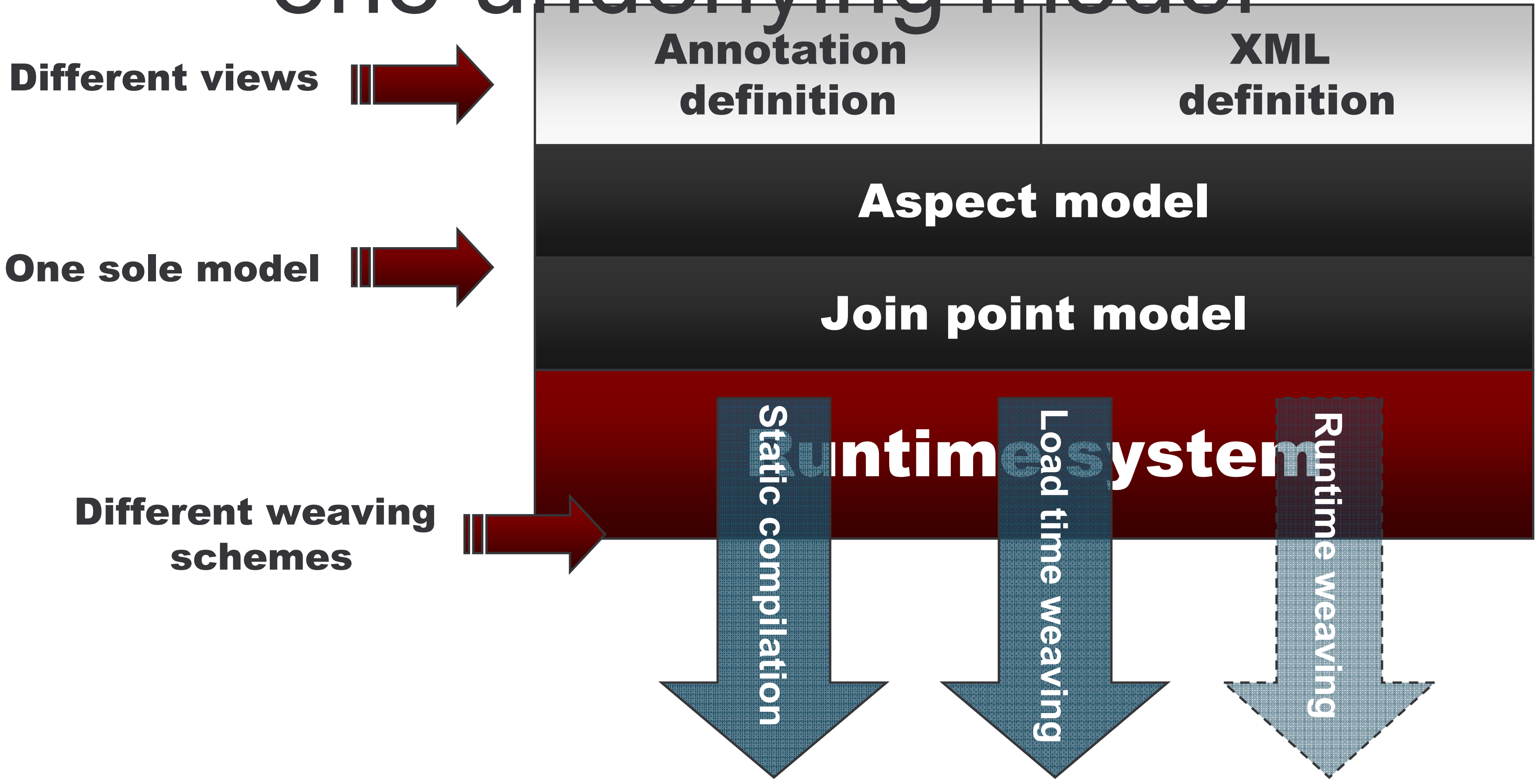
```
@Expression
   execution(@TxRequired *.*(..))
   call(* Interface+.*(..))
   get(Baz foo.Bar.field)
   etc.
```

- Pointcuts are composable

```
OR, AND, NOT, cflow(PCD)

call(* Interface+.*(..)) AND within(foo.Bar)
```

Two definition schemes – one underlying model



AOP crash course

Main points

- AOP brings a new theory to address cross-cutting concerns
- Different AOP solutions support different weaving schemes
- AspectWerkz is a Java 1.5 ready solution
  - Aspects are Annotated Java™ classes
  - Activated or refined through a simple XML deployment descriptor

Agenda

AOP crash course

Demos:

- Hello World
- J2EE integration
- J2EE aspects
  - Role-based security
  - Unit Of Work

Hello World Aspect

```
public class MyAspect {
    public void beforeGreeting(JoinPoint jp) {
        System.out.println("before greeting...");
    }
    public void afterGreeting(JoinPoint jp) {
        System.out.println("after greeting...");
    }
}
```

XML definition

```
<aspectwerkz>
  <system id="AspectWerkzExample">
    <aspect class="testAOP.MyAspect">
      <pointcut name="greetMethod"
 expression="execution(* testAOP.HelloWorld.greet(..))"/>
      <advice name="beforeGreeting" type="before"
              bind-to="greetMethod"/>
      <advice name="afterGreeting" type="after"
              bind-to="greetMethod"/>
    </aspect>
  </system>
</aspectwerkz>
```

Method name for advice

Annotation definition

```
public class MyAspect {

  @Before("execution(* testAOP.HelloWorld.greet(..))")
  public void beforeGreeting(JoinPoint jp) {
    System.out.println("before greeting...");
  }

  @After("execution(* testAOP.HelloWorld.greet(..))")
  public void afterGreeting(JoinPoint jp) {
    System.out.println("after greeting...");
  }
}
```

DEMO

Agenda

AOP crash course

Demos:

- Hello World
- J2EE integration
- J2EE aspects
  - Role-based security
  - Unit Of Work

Around advice : The proceed method

- The **`JoinPoint`** class has a **`proceed()`** method:

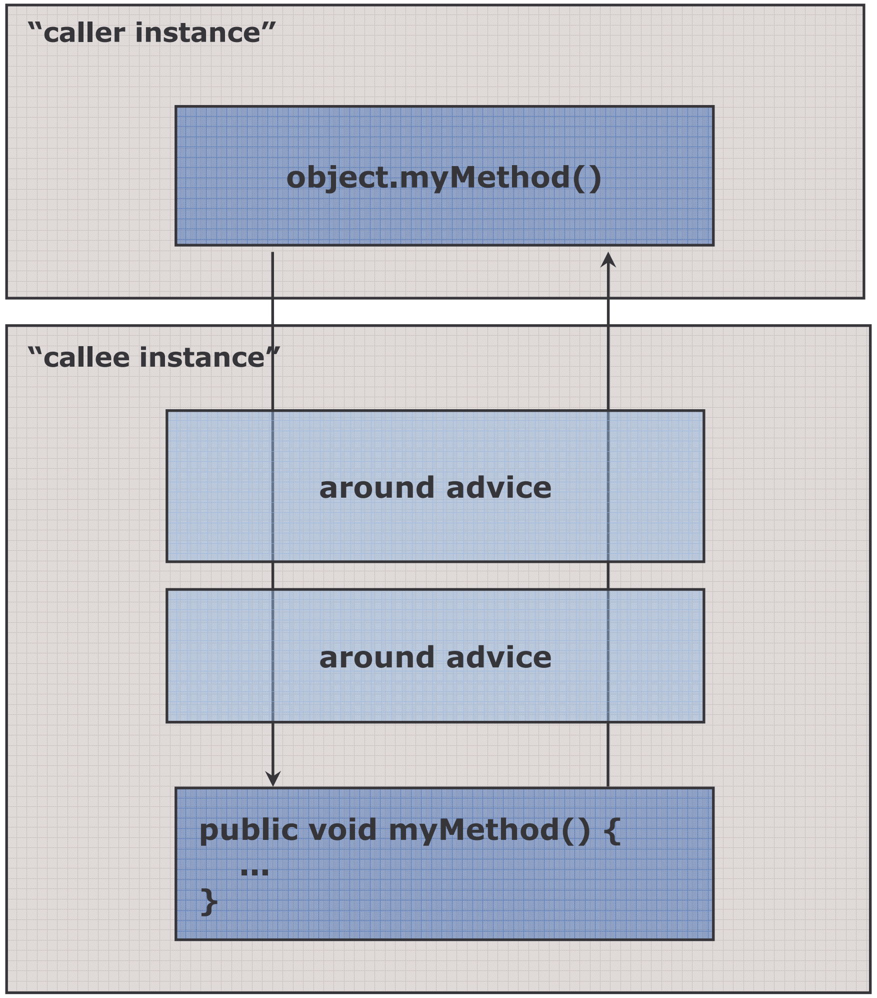
```
Object result = joinPoint.proceed();
```

- Only works in *Around advice*
- It either invokes:
  - The next advice in the chain, or
  - The target join point (method, field, constructor etc.)

- It returns the result from the join point invocation

Around advice

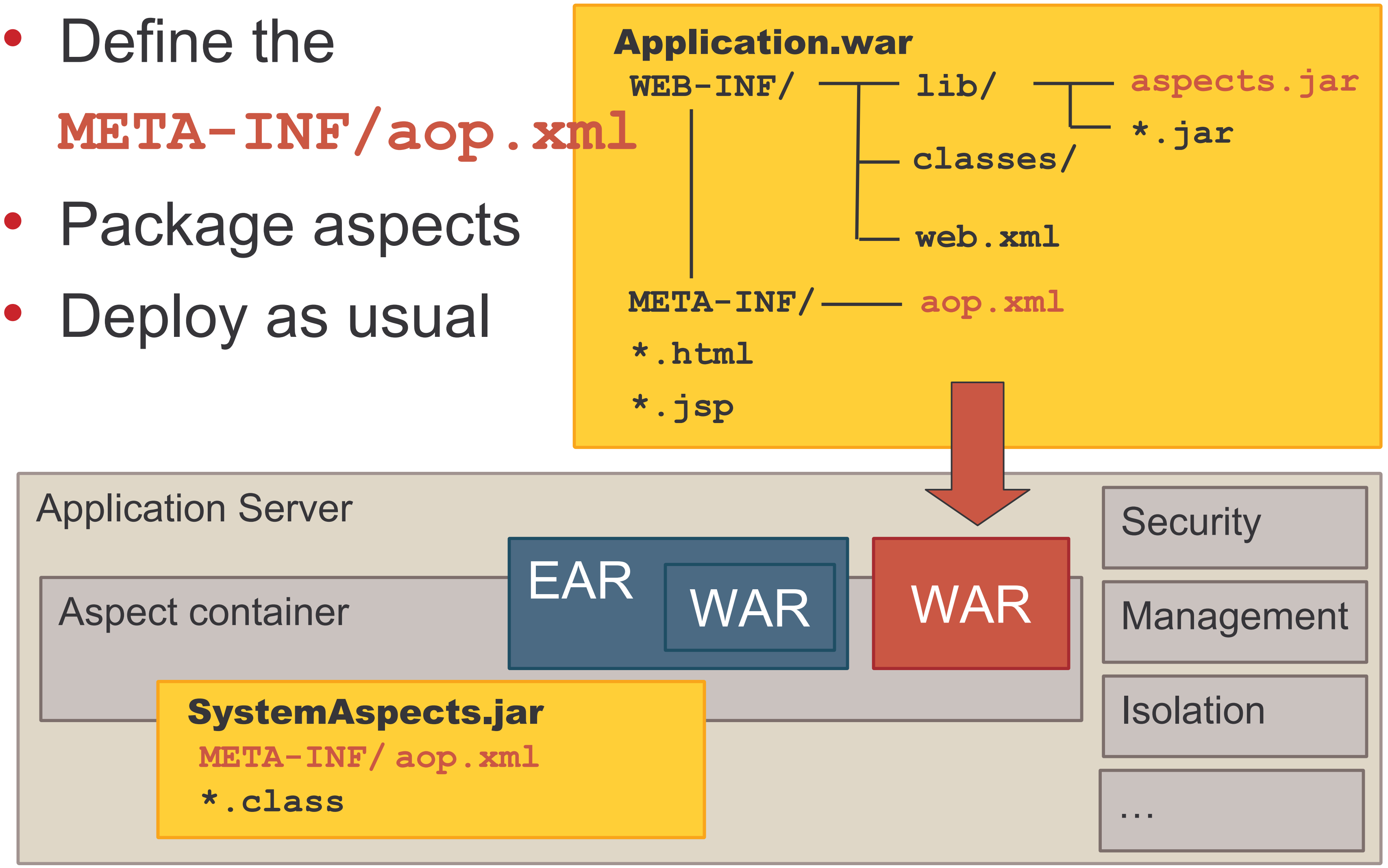
```
public Object aroundAdvice(JoinPoint joinPoint)
     throws Throwable {
     // do stuff
     Object result =
          joinPoint.proceed();
     // do more stuff
     return result;
}
```



Aspect Container integration in J2EE™

AspectWerkz's Aspect Container

- Define the
  META-INF/aop.xml
- Package aspects
- Deploy as usual



DEMO

Agenda

AOP crash course

Demos:

- Hello World
- J2EE integration
- J2EE aspects
  - Role-based security
  - Transaction demarcation
  - Unit Of Work

J2EE aspects

Example: Address book web application

- Requirements
  - Login and logout
  - List user’s contacts
  - Add a contact
  - Remove one or more contacts

- Services
  - Authentication
  - Authorization
  - Persistence of the address books
  - Transaction integrity

J2EE aspects

- AOP makes it possible to capture the concern in one single modular unit (the Aspect)

- Security Aspect that implements
  - Authentication
  - Authorization (on method or even field level)
  - Role-Based using JAAS (pluggable)

- Use of abstraction makes the Aspect reusable

J2EE aspects

Authentication advice

```
/** @Around authenticationPoints */
public Object authenticateUser(JoinPoint joinPoint)
    throws Throwable {
    if (m_subject == null) {
        // no subject => authentication required
        Context ctx = ... // principals and credentials
        m_subject = m_securityManager.authenticate(ctx);
    }
    Object result = Subject.doAsPrivileged(
        m_subject, new PrivilegedExceptionAction() {
            public Object run() throws Exception {
                return joinPoint.proceed();
            };
        }, null
    );
    return result;
}
```

J2EE aspects

Authorization advice

```
/** @Around authorizationPoints */
public Object authorizeUser(JoinPoint joinPoint)
   throws Throwable {
   MethodRtti rtti = (MethodRtti)joinPoint.getRtti();
   if (m_securityManager.checkPermission(
      m_subject,
      rtti.getMethod())) {
      // user is authorized => proceed
      return joinPoint.proceed();
   }
   else {
      throw new SecurityException(...);
   }
}
```

J2EE aspects

Integration in the web application

- Extend the **`AbstractRoleBasedAccessProtocol`** aspect and define the pointcuts:
  - **`authenticationPoints`**
  - **`authorizationPoints`**

- Authenticate on all methods with the
  - **`@Authenticate`** annotation

- Authorize on all methods with the
  - **`@Authorize`** annotation

J2EE aspects

Define the pointcuts in the XML descriptor

```
<aspectwerkz>
 <system id="security">
  <aspect class="aspect.RoleBasedAccessProtocol">
   <pointcut
     name="authenticationPoints"
     expression="execution(@Authenticate * *.*(..))"/>
   <pointcut
     name="authorizationPoints"
     expression="execution(@Authorize * *.*(..))"/>
  </aspect>
 </system>
</aspectwerkz>
```

DEMO

Agenda

AOP crash course

Demos:

- Hello World
- J2EE integration
- J2EE aspects
  - Role-based security
  - Unit Of Work

Unit Of Work

- Unit Of Work
  - Common pattern in enterprise application architectures
  - Implements a transaction
  - Keeps track of new, removed and dirty objects
- Will be used to implement:
  - Transaction demarcation for Plain Old Java Objects (POJOs)
  - Persistence handling for POJOs

The Unit Of Work API

```
public class UnitOfWork {
    public static UnitOfWork begin() {...}
    public void commit() {...}
    public void rollback() {...}

    // registers the transactional objects
    public void registerNew(Object obj) {...}
    public void registerRemoved(Object obj) {...}
    public void registerDirty(Object obj) {...}

    // template methods
    public void doBegin() {...}
    public void doCommit() {...}
    public void doPreCommit() {...}
    public void doPostCommit() {...}
    public void doRollback() {...}
    public void doDispose() {...}
}
```

Problems with non AOP solution (1)

- Is a cross-cutting concern
- Introduces code scattering
- Introduces code tangling

Problems with non AOP solution (2)

- For example, this code:

```
AddressBook book = new AddressBook(...);
book.addContact(contact);
```

- Would have to be replaced by:

```
UnitOrWork unitOfWork = UnitOfWork.begin();
try {
   AddressBook book = new AddressBook(...);
   unitOfWork.registerNew(book);
   book.addContact(contact);
   unitOfWork.registerDirty(book);
   unitOfWork.commit();
} catch(Exception e) {
   unitOfWork.rollback();
}
```

Enter Aspect-Oriented Programming

- Can make the **`UnitOfWork`** completely transparent

Advice: RegisterNew

- Registers the newly created instance

```
/** @Around transactionalObjectCreationPoints */
public Object registerNew(JoinPoint joinPoint)
    throws Throwable {
    Object newInstance = joinPoint.proceed();
    if (UnitOfWork.isInUnitOfWork()) {
        UnitOfWork unitOfWork = UnitOfWork.getCurrent();
        unitOfWork.registerNew(newInstance);
    }
    return newInstance;
}
```

Advice: RegisterDirty

- Registers an object as dirty just before a field is modified

```
/** @Before transactionalObjectModificationPoints */
public void registerDirty(JoinPoint joinPoint)
    throws Throwable {
    if (UnitOfWork.isInUnitOfWork()) {
        Signature sig = joinPoint.getSignature();
        UnitOfWork unitOfWork = UnitOfWork.getCurrent();
        unitOfWork.registerDirty(
            joinPoint.getTargetInstance(),
            sig.getName()
        );
    }
}
```

Advice: ProceedInTransaction

```
/** @Around transactionalMethods */
public Object proceedInTransaction(JoinPoint joinPoint) {
    if (UnitOfWork.isInUnitOfWork()) {
        return joinPoint.proceed();
    }
    UnitOfWork unitOfWork = UnitOfWork.begin();
    final Object result;
    try {
        result = joinPoint.proceed();
        if (unitOfWork.isRollbackOnly()) {
            unitOfWork.rollback();
        } else {
            unitOfWork.commit();
        }
    } catch (Throwable throwable) {
        throw handleException(throwable, unitOfWork);
```

Exception handling

```
private Throwable handleException(
    Throwable throwable,
    UnitOfWork unitOfWork) {
    if (throwable instanceof RuntimeException) {
        unitOfWork.rollback();
    }
    else {
        unitOfWork.commit();
    }
    return throwable;
}
```

- Uses the same approach as in EJB
  - Rollback on **`RuntimeException`**

Unit Of Work Listeners

- The `UnitOfWorkListener` API:
  - `public void doBegin()  {...}`
  - `public void doCommit()  {...}`
  - etc.
- These allows listeners to define what to do at specific points:
  - TX begin
  - TX commit
  - TX pre-commit
  - TX post-commit
  - TX rollback
  - TX dispose
- JTA, Hibernate, Berkeley DB, JDBC…

JTA integration

```
public class JtaAwareUnitOfWorkListener extends UnitOfWorkListener {
    ... // declare the member TX manager and the TX
    public boolean isRollbackOnly() {
        return m_transaction.isExistingTransaction()
               && m_transaction.isRollbackOnly();
    }
    public void doBegin() {
        m_transaction = s_txManager.getTransaction();
    }
    public void doRollback() {
        s_txManager.rollback(m_transaction);
    }
    public void doCommit() {
        s_txManager.commit(m_transaction);
    }
}
```

How to use it?

- Just define the three pointcuts
  - `transactionalObjectCreationPoints`
  - `transactionalObjectModificationPoints`
  - `transactionalMethods`

For More Information

- Thursday, 10h45 – TS-1391
  AOP in J2EE environments
- http://docs.codehaus.org/display/AW/Hello+World
- Attend AOSD.05 in Chicago
- http://aspectwerkz.codehaus.org
- http://blogs.codehaus.org/people/jboner
- http://blogs.codehaus.org/people/avasseur

Q&A

JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

Dynamic Aspect-Oriented Programming With AspectWerkz

Jonas Bonér
Senior Software Engineer
Alexandre Vasseur
Software Engineer

java.sun.com/javaone/sf