



Advanced AOP with AspectJ

Adrian Colyer, CTO, Interface21
adrian.colyer@interface21.com



Agenda

- Aspects and annotations
- Generic abstract aspects
- Inter-type declarations
- Aspects in the domain



Aspects and annotations



Annotation matching

- AspectJ 5 introduces the ability to match based on annotations
- Matching annotated types
 - e.g. `(@SomeAnnotation *)`
- Matching based on annotated members
 - e.g. `execution(@SomeAnnotation * *(..))`



Annotation matching is good for:

- Matching join points that otherwise fit no common or stable pattern
- Robustness in the face of program evolution and refactoring
 - the 'fragile pointcut problem'
- Some kinds of library aspects
 - all a developer has to do is use the annotation
 - no need to learn AOP/AspectJ
- Explicit source code recognition of the concern



Annotation matching not so good for

- Matching large numbers of join points
 - @Traced anyone?
- Matching when there already exist suitable stable properties
- Cases where no good domain-specific annotation exists
- Coupling of code to the annotations used
- JDK 1.4 and below!



Example: Auditing

- Keeping an audit record of who did what
 - User 'joe smith' initiated a withdrawal of 125,000 eur from account 12345 as a result of executing a transfer operation at 09:12:34 02.03.06
- Auditing context typically established early in flow
 - authenticated user
 - operation requested by user



The "design document"

5.0 Auditing

To satisfy regulations we need to keep the following information in an audit trail:

5.1 Changes to financial data

Any operation that changes financial data should generate an audit record comprising the following information:

- * Principal initiating the change
- * ...



Making the code look like the design

```
public aspect Auditing {
    ...
    pointcut auditPoint511_2() :
        <pointcut expression>;
    after() returning : auditPoint511_2() {
        // extract audit data from join point and saved
        // context
        ...
        this.auditingService
            .recordAuditEvent("511_2", auditData);
    }
}
```

← potentially fragile list
of pointcut
expressions here



The Audited Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Audited {
    AuditCode value(); // no default, use req'd
}

public enum AuditCode {
    AC511_2,
    AC511_3,
    ...
}
```



The Audited Annotation

```
public class Account {  
  
    public MonetaryAmount getBalance() { ... }  
  
    @Audited(AuditCode.AC511_2)  
    public void withdraw(MonetaryAmount amount){  
        ...  
    }  
  
    ...  
}
```



Adding behaviour

- We can write a pointcut to match on the annotation
 - this will be stable in the face of refactoring etc.

```
pointcut auditPoint(Audited audited) :  
    Pointcuts.anyOperation() &&  
    SystemArchitecture.inMyApplication() &&  
    @annotation(audited);
```



Adding behaviour

- associate the pointcut with advice

```
before(Audited audited) : auditPoint(audited) {  
    // establish audit data  
    // ...  
    AuditCode code = audited.value();  
    this.auditService.audit(code,auditData);  
}
```



Establishing Audit Context

```
pointcut establishesAuditContext() :  
    Pointcuts.anyPublicOperation() &&  
    SystemArchitecture.inServiceLayer();  
  
before() : establishesAuditContext() {  
    // save context information  
}
```



Establishing Audit Context

- What if service layer operations call each other?
 - we want to capture the user's original entry point...
- Common AspectJ idiom for capturing "top-level" calls:
 - `foo() && !cflowbelow(foo())`
 - any join point matched by `foo()`, but only if it is not in the control flow of another `foo()` join point



Establishing Audit Context

```
pointcut serviceOperation() :  
    Pointcuts.anyPublicOperation() &&  
    SystemArchitecture.inServiceLayer();  
  
pointcut establishesAuditContext() :  
    serviceOperation() &&  
    !cflowbelow(serviceOperation());
```



Storing audit context

- From thisJoinPoint[StaticPart] we can establish service type name and operation name

```
private JoinPoint.StaticPart auditContext;

before() : establishesAuditContext() {
    this.auditContext =
        thisJoinPointStaticPart;
}
```



Thread safety

- There's a problem with this solution:
 - if there are multiple concurrent threads of execution, the auditContext will be overwritten
- One solution is to make the auditContext field a ThreadLocal
- A better solution is to have one instance of the Auditing aspect for each top level service operation
 - then we can just build up as much audit context state as we need at various points in the execution



One instance per control flow

```
public aspect Auditing
    percfow(establishesAuditContext()) {

    // state unique to this control flow...
    private JoinPoint.StaticPart auditContext;

    pointcut establishesAuditContext() :
        serviceOperation() &&
        !cflowbelow(serviceOperation());

    ...
}
```



Learning points

- Can match join points based on annotated types and signatures
- Can bind annotation values and access them in advice (typed advice)
- `foo() && !cflowbelow(foo())` :- idiom for 'top level' join point
- `thisJoinPoint` / `thisJoinPointStaticPart` variables
- aspect instances have a lifecycle you can control by specifying an instantiation model (per-clause)



Contract4J-5

- Uses AspectJ to implement design-by-contract style assertions and invariants

```
@Contract
public class MyClass {

    @Pre("n != null")
    public void setName(String n) { ... }

    ...
}
```



More annotation tricks...

- Enforce project policies on annotation usage
 - declare warning
 - : `execution(@javax.persistence.Column * *(..))`
 - : "use a mapping file instead of annotations";
- Avoid tight-coupling to annotation-driven frameworks
 - declare `@type : com.xyz.service..* : @SomeAnnotation ;`
- Annotation bridging
 - declare `@method`
 - : `@BatchOperation * *(..)`
 - : `@org.osoa.sca.Oneway;`



Generic and abstract Aspects

now this *is* rocket science ;)



Generic aspects

- An *abstract* aspect may be declared as a generic type
 - i.e. with one or more type parameters
- A concrete sub-aspect **must extend a fully-parameterized version** of the super-aspect
 - there are no constructor calls to specify type parameters on concrete aspects



Generic aspects

- Why generic aspects?
 - the type parameters provide an additional way of parameterizing the aspect
 - good for binding types to roles
 - type variables can be used in
 - field and method declarations
 - pointcut expressions
 - declare statements
 - but *not* inter-type declarations



Custom auditing

- Recall the annotation-driven auditing aspect:

```
@Audited(AuditCode.AC511_2)
```

```
public void withdraw(MonetaryAmount amount){...
```

- Auditing requirements can get more complex than this
 - need audit categories, each with their own info – not just a simple code
 - need custom logic to build audit data



Custom auditing

- Allow users to create their own, *business-specific* auditing annotations
- Provide a template method for creating audit entry for a given annotation



Auditing with user annotations

```
public abstract aspect Auditing<A extends Annotation>
percfow(establishesAuditContext()) {

    protected abstract pointcut inAuditingScope();

    protected abstract pointcut establishesAuditContext();

    protected abstract void establishAuditContext(
        JoinPoint jp);

    before() : establishesAuditContext() {
        establishAuditContext(thisJoinPoint);
    }
}
```



Auditing with user annotations

```
pointcut auditPoint(A auditAnnotation) :  
    execution(* *(..)) && inAuditingScope()  
    && @annotation(auditAnnotation);  
  
after(A auditAnnotation) returning :  
    auditPoint(auditAnnotation) {  
        audit(auditAnnotation, thisJoinPoint);  
    }  
  
protected abstract void audit(  
    A auditAnnotation, JoinPoint joinPoint);  
}
```



Usage model

- User defines annotation for an auditing category
- e.g.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Code768AuditEvent {
    int majorCode();
    int minorCode();
    String description() default "";
}
```



Usage model

- Extend generic auditing aspect to use this annotation...

```
public aspect Code768Auditing
extends Auditing<Code768AuditEvent> {
    protected pointcut inAuditingScope() :
        SystemArchitecture.inMyApplication();

    protected pointcut establishesAuditContext() :
        Pointcuts.anyPublicOperation() &&
        SystemArchitecture.inServiceLayer();
}
```



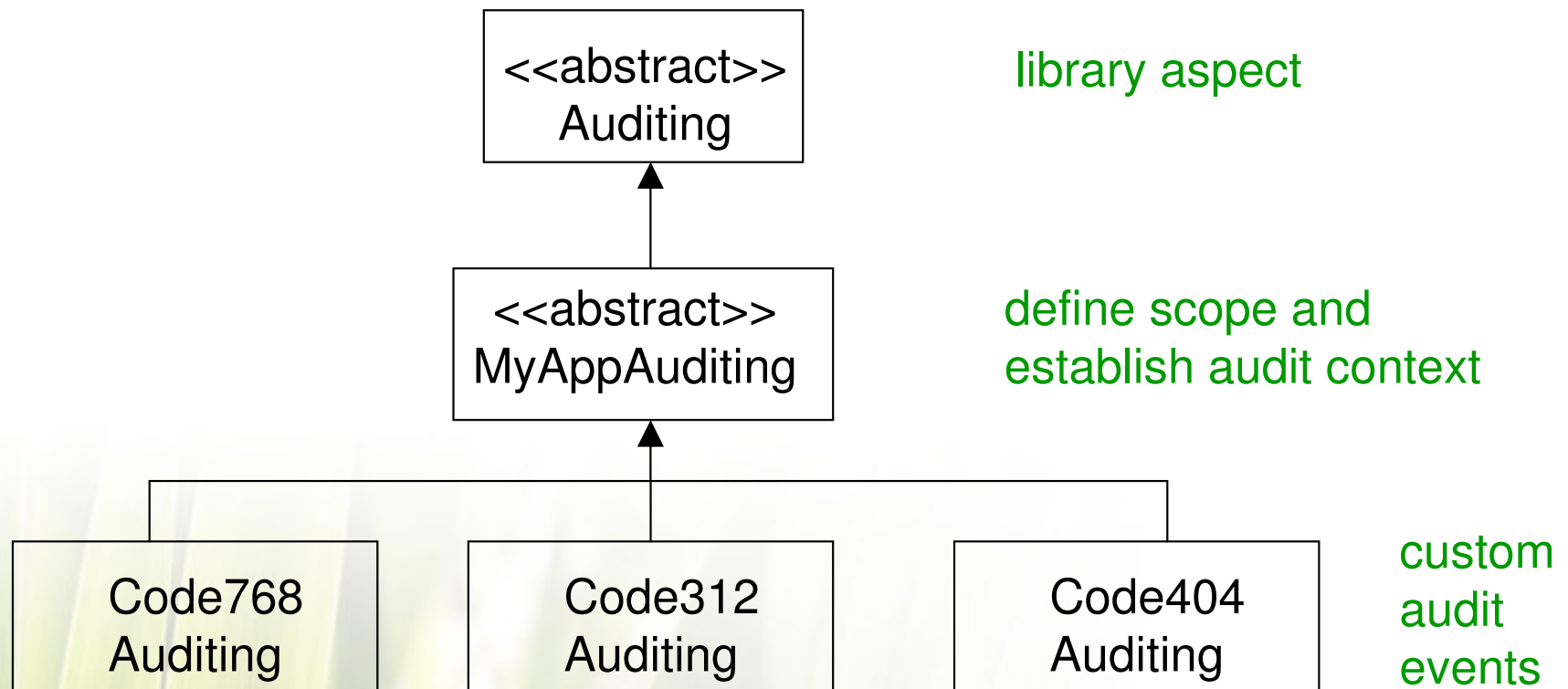
Code 768 auditing

```
protected void establishAuditContext(JoinPoint jp) {  
    ...  
}  
  
protected void audit(  
    Code768AuditEvent event, JoinPoint jp) {  
    // build audit record and send to audit service...  
}  
  
}
```



Custom auditing

- Can have as many subtypes of Auditing as needed
- Perhaps even a hierarchy...





Inter-type Declarations



Read-write Locking Support

```
public aspect ReadWriteLockableSupport {  
    interface ReadWriteLockable { ... }  
  
    declare parents :  
        org.abc..* implements ReadWriteLockable;  
  
}
```

- Provides no default implementation of interface methods as specified
 - so will fail if matched types don't already define the required members



Inter-type declarations

```
public aspect ReadWriteLockableSupport {  
    interface ReadWriteLockable { ... }  
  
    declare parents :  
        org.abc..* implements ReadWriteLockable;  
  
    private ReadWriteLock ReadWriteLockable.rw;  
  
    public void ReadWriteLockable.acquireReadLock() {  
        this.rw.readLock().lock();  
    }  
  
    ...  
}
```

inter-type declarations

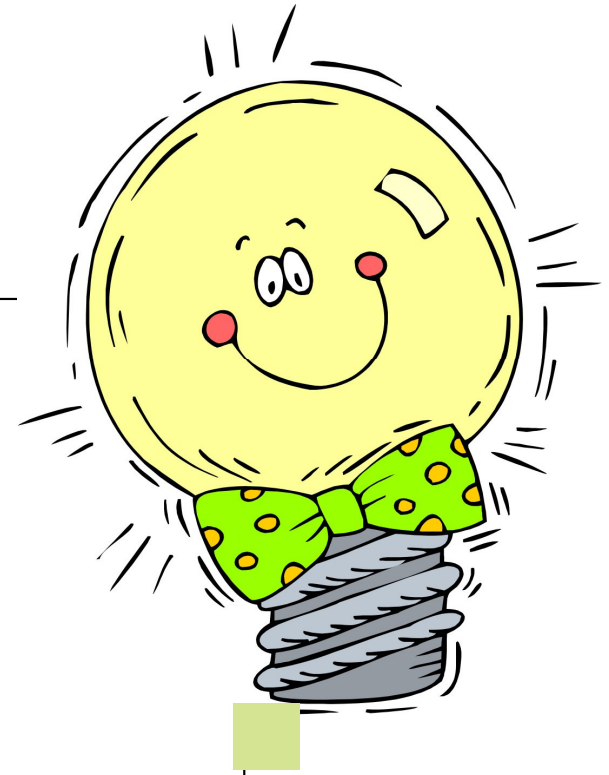


Inter-type declarations

- An inter-type declaration looks like a regular member declaration
 - with the addition of a 'target type' specified before the member name
 - private int x;
 - private int SomeOtherType.x;
- ↑
target type
- Only allowed inside aspects

Tip

- The visibility of an inter-type declaration is with respect to the *aspect*, not the target class. So a private ITD is only visible inside the declaring aspect.





Common uses of ITDs

- ITDs more often used with interfaces than with classes
 - add marker interfaces
 - provide default implementations
 - manage state that is typically private to some aspect and needed by the aspect to perform its function
 - state is held per-instance of the target type



Aspects in the domain



ISafeRunnable

- An example from Eclipse / AJDT
- In Eclipse, plugins are loosely coupled via extension points
- Extenders of a plugin typically provide a class that implements an extension interface
- Platform rules say that you should never allow a rogue plugin to upset others...



ISafeRunnable

```
public interface IXReferenceProvider {
    Class[] getClasses();
    String getProviderDescription();
    ...
    /**
     * Providers are contributed by other
     * plugins, and should be considered
     * untrusted code. Whenever we call such
     * code, it should be wrapped in an
     * ISafeRunnable.
     */
    static aspect SafeExecution {
        ...
    }
}
```



ISafeRunnable

```
static aspect SafeExecution {
    pointcut untrustedCall() :
        call(* IXReferenceProvider.*(..));

    Object around() : untrustedCall() {
        ISafeRunnableWithReturn safeRunnable =
            new ISafeRunnableWithReturn() {
                Object result = null;
                public void handleException(Throwable e) {}
                public void run() throws Exception {
                    this.result = proceed();
                }
                public Object getResult() { return result; }
            };
        Platform.run(safeRunnable);
        return safeRunnable.getResult();
    }
}
```



Design Patterns

- Represent solutions to commonly occurring problems
- Let's keep the problem statements
 - and see if there are any better solutions...
- 17/23 show improvement
 - code locality, reusability, composition, plug/un-plug



Summary

- Aspects and annotations
 - avoid fragile pointcuts
- Generic abstract aspects
 - allow use of domain-specific types
- Inter-type declarations
 - declare members on behalf of other types
- Aspects in the domain
 - not just for infrastructure concerns



Questions...?