Technologies for Aspect-Oriented Software Development

Eng. Esteban S. Abait
ISISTAN – UNCPBA

Agenda

- Definitions
- Aspect-Oriented Languages
  - AspectJ
- Dynamic Java Frameworks
  - Spring AOP
  - JAC
  - JBoss AOP
- Tools comparison
- Current Limitations

Agenda

- Definitions
- Aspect-Oriented Languages
  - AspectJ
- Dynamic Java Frameworks
  - Spring AOP
  - JAC
  - JBoss AOP
- Tools comparison
- Current Limitations

AOP Definitions
Obliviousness + Quantification (1)

- What is AOP?
- What makes a language “AO”?
  - According to Filman and Friedman [1]

    “AOP = obliviousness + quantification”
- *Obliviousness* states that one can't tell what the aspect code will execute by examining the body of the base code

Developer A

Class *BankAccount*

Developer B

Aspect *SecurityPolicies*

Developer A is oblivious with respect to the applied security policies on the bank account

AOP Definitions
Obliviousness + Quantification (2)

- AOP is thus the desire to make programming statements of the form
  - In programs P, whenever condition C arises, perform action A
- Quantification. What kinds of conditions can we specify?
- Interface. How do the actions interact with the programs and with each other?
- Weaving. How will the system arrange to intermix the execution of the programs with the action?

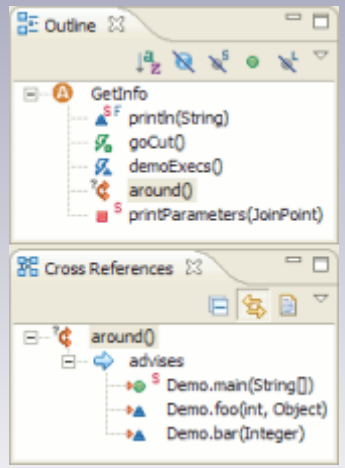
Agenda

- Definitions
- Aspect-Oriented Languages
  - AspectJ
- Dynamic Java Frameworks
  - Spring AOP
  - JAC
  - JBoss AOP
- Tools comparison
- Current Limitations

AspectJ: Introduction

- AspectJ was designed as a *compatible extension* to Java [2]
- This compatibility means:
  - Upward compatibility: all legal programs in Java are legal in AspectJ
  - Platform compatibility: all legal AspectJ programs must run on standar Java virtual machines
  - Tool compatibility: it must be possible to extend existing tools to support AspectJ in a natural way

Programmers compatibility: Programming with AspectJ must feel like a natural extension of programming with Java



Tool compatibility: AJDT provides tool support for AOSD with AspectJ in eclipse. AJDT is consistent with JDT.

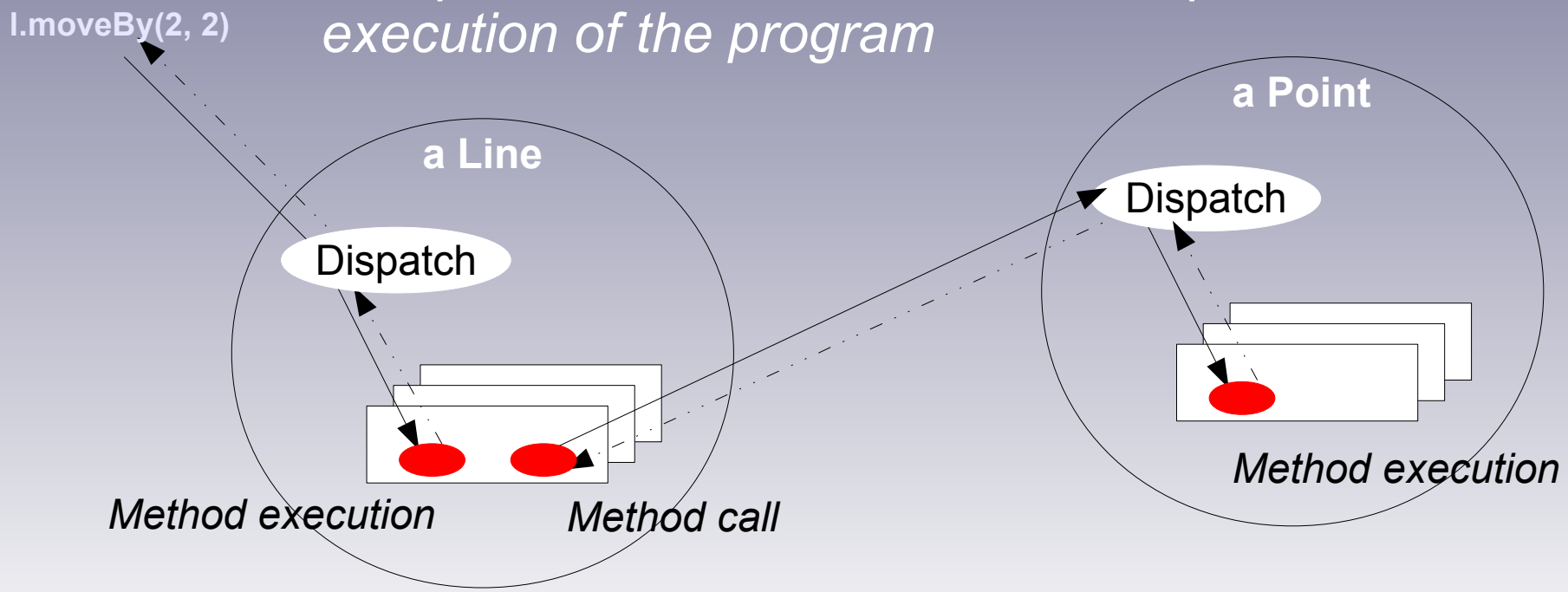
AspectJ: Basic concepts

- What AspectJ adds to Java?
  - A concept: join point
  - A set of constructors:
    - pointcut
    - advice
    - inter-type declarations
    - aspects

```
public aspect AspectoEjemplo {

    //Ejemplo de inter-type declaration
    private Vector Figure.figuras = new Vector();

    pointcut deEjemplo() : execution(* *..*(..));

    before():deEjemplo() {
        //Hago algo
    }

}
```

AspectJ: Join Point Model (1)

- In AOP, join points are the places where the weaver composes the aspects with the base program
- AspectJ supports a dynamic join point model
  - *Join points are certain well-defined points in the execution of the program*



AspectJ: Join Point Model (2)

- Dynamic join points supported by AspectJ

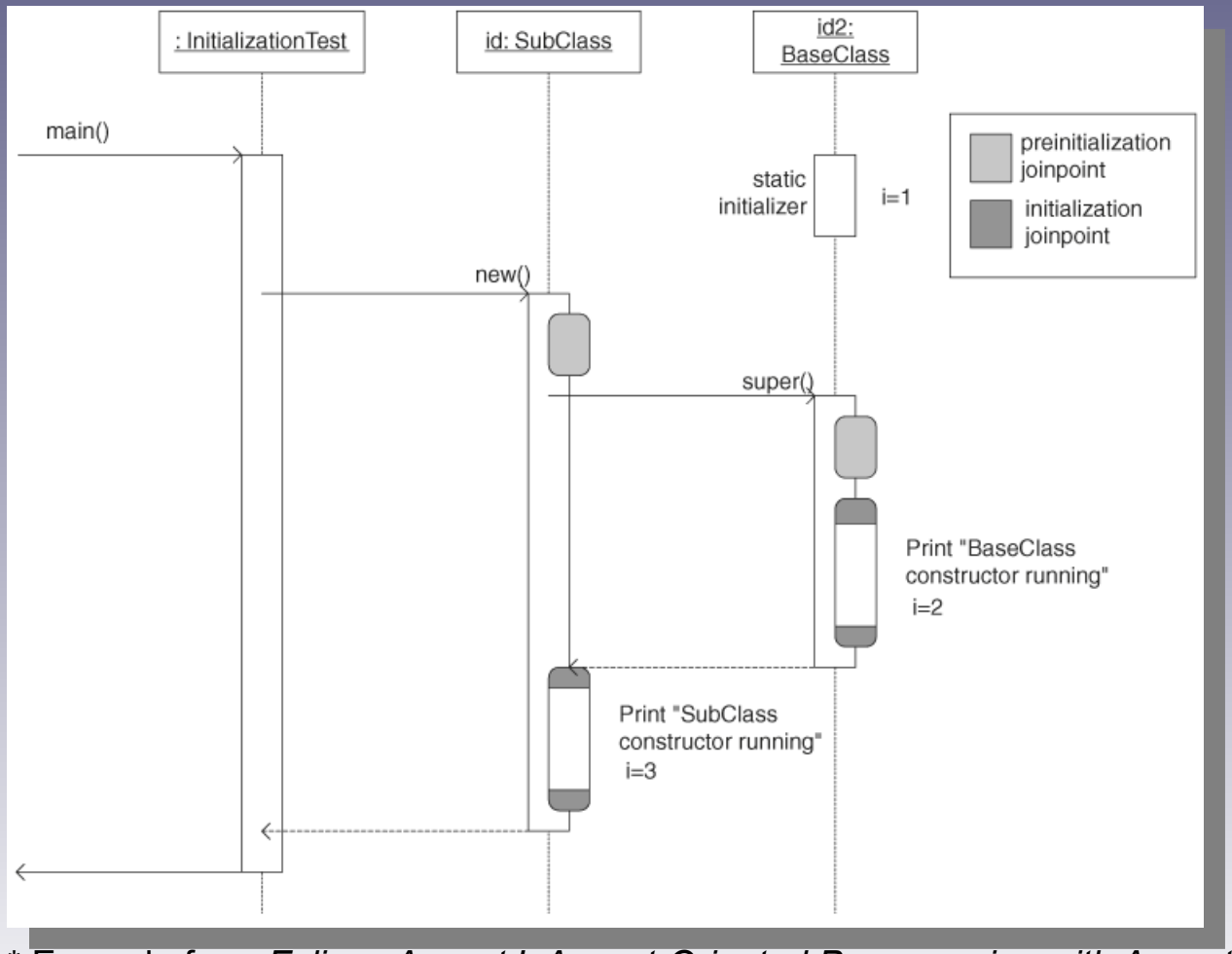
Join point	Code snippet
Method call	l.moveBy(2, 2)
Method execution	void moveBy (int x, int y) { ... }
Constructor call	new Point(2, 2)
Constructor execution	public Point (int x, int y)
Field get	System.out.print(x)
Field set	this.x = 3
Pre-initialization	Next slide
Initialization	Next slide

Static initialization	class A {
	  static { … }
	}

Handler	catch (NumberFormatException nfe) { … }
Advice execution	Other aspects execution

AspectJ: Join Point Model (3)

- Pre-initialization and initialization join points

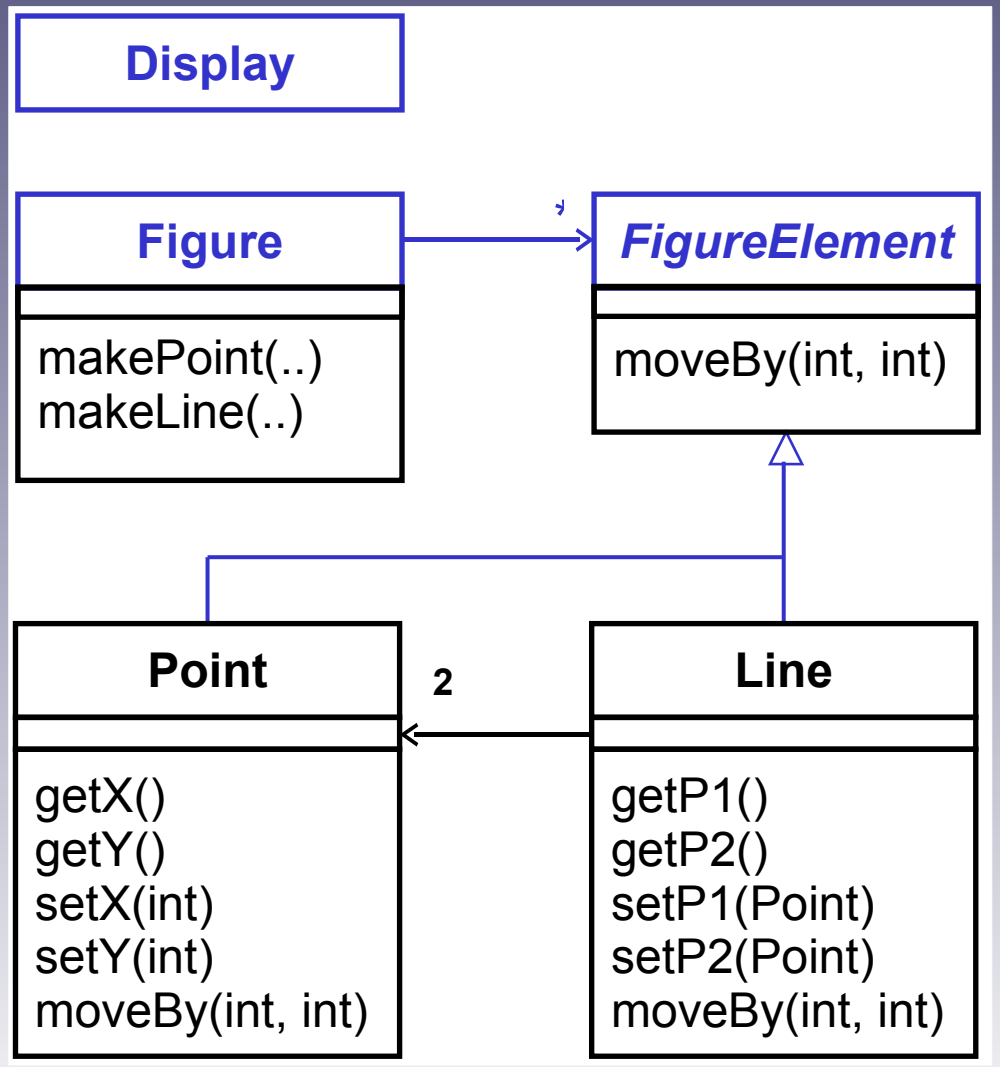


- **Pre-initialization** join point encompasses the period from the entry of the first-called constructor to the call to the super constructor

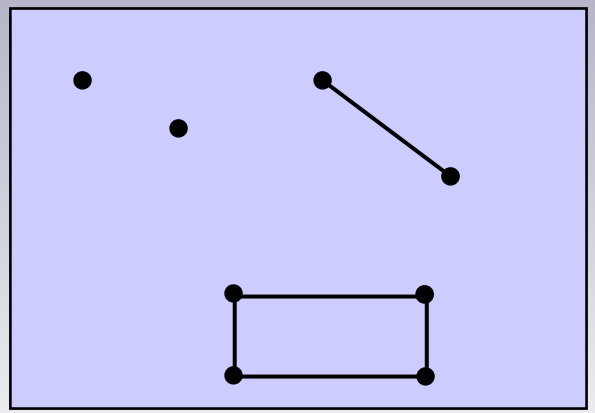
- **Initialization** join point encompasses the period from the return from the super constructor call to the return of the first-called constructor

* Example from *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*

Learning by example: Display update (1)



- **Context:** Figure editor system
- **Problem:** Each time a figure moves the display must be updated
- **Variants:** a) one Display
  b) many Displays



Display

AspectJ
Learning by example: Display update (2)

- First version: non-AOP

```
class Line implements FigureElement{
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update();
  }

  void setP2(Point p2) {
    this.p2 = p2;
    Display.update();
  }

  void moveBy(int dx, int dy) { ... }
}
```

```
class Point implements FigureElement {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Display.update()
  }

  void setY(int y) {
    this.y = y;
    Display.update();
  }

  void moveBy(int dx, int dy) { ... }
}
```

Learning by example: Display update (3)

- Non-AOP implementation: the “display update” concern is *scattered* through all the FigureElement hierarchy
  - Code duplication
  - Hard to evolve and maintain
    - A change in the update policy or in the Display protocol has ripple effects over the FigureElement hierarchy

Developer

- **Requirement:** change the update policy
- **Changes:** all FigureElement subclasses

Learning by example: Identifying join points

- Lets build our first aspect-oriented solution
- We need a way to express all the relevant join point to our problem
  - Pointcut: in AspectJ, pointcuts pick out certain join points in the program flow

```
call (void Point.setX(int))
```

Pick out each join point that is a call to a method **setX** whose class is **Point**, return type is **void** and has only one parameter **int**

```
pointcut move:
  call (void Point.setX(int)) ||
  call (void Point.setY(int))
```

```
pointcut move:
  call (void Point.set*(int))
```

A pointcut can have a name and can be built out of other pointcut descriptors

Learning by example: Implementing the behavior

- Advice: brings together a *pointcut* and a *body of code*
- AspectJ defines three kinds of advices
  - Before advice: runs before the join point is reached
  - After advice: runs after the program proceeds with the join point
  - Around advice: runs as the join point is reached, and has explicit control over whether the program proceeds with the join point

```
pointcut move:
   call (void FigureElement.moveBy(int, int)) ||
   call (void Point.setX(int))                ||
   call (void Point.setY(int))                ||
   call (void Line.setP1(Point))              ||
   call (void Line.setP2(Point));
```

```
after() returning: move() {
   Display.update();
}
```

AspectJ
Learning by example: Putting all together

- Aspects wrap up pointcut, advice and inter-type declarations in a modular unit of crosscutting implementation
- Aspects can also have methods, fields and initializers

```
public aspect DisplayUpdating {
  pointcut move:
    call (void FigureElement.moveBy(int, int)) ||
    call (void Point.setX(int))                ||
    call (void Point.setY(int))                ||
    call (void Line.setP1(Point))              ||
    call (void Line.setP2(Point));

  after() returning: move() {
    Display.update();
  }
}
```

Learning by example: Comparing both versions

AOP version

The "related display" concern is isolated in one unit. No more code scattering

```
class Line implements FigureElement{
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }

  void setP2(Point p2) {
    this.p2 = p2;
  }

  void moveBy(int dx, int dy)
}
```

```
  }
```

```
class Point implements FigureElement {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }

  void setY(int y) {
    this.y = y;
  }
```

```
public aspect DisplayUpdating {
  pointcut move:
    call (void FigureElement.moveBy(int, int)) ||
    call (void Point.setX(int))                ||
    call (void Point.setY(int))                ||
    call (void Line.setP1(Point))              ||
    call (void Line.setP2(Point));

  after() returning: move() {
    Display.update();
  }
}
```

AspectJ: Pointcuts (1)

- A pointcut is a declarative structure that allows matching certain join points exposed by AspectJ
- A pointcut based on explicit enumeration of a set of method signatures is known as *name-based*
- Pointcuts expressed in terms of properties of methods other than its exact name are known as *property-based*

```
pointcut move:
  call (void FigureElement.moveBy(int, int)) ||
  call (void Point.setX(int))                ||
  call (void Point.setY(int))                ||
  call (void Line.setP1(Point))              ||
  call (void Line.setP2(Point));
```

Name-based

Property-based

```
pointcut move:
  call (void FigureElement.moveBy(int, int)) ||
  call (void Point.set*(int))                ||
  call (void Line.set*(Point));
```

AspectJ: Pointcuts (2)

- Pointcuts allows exposing context on a join point
- Without exposing:

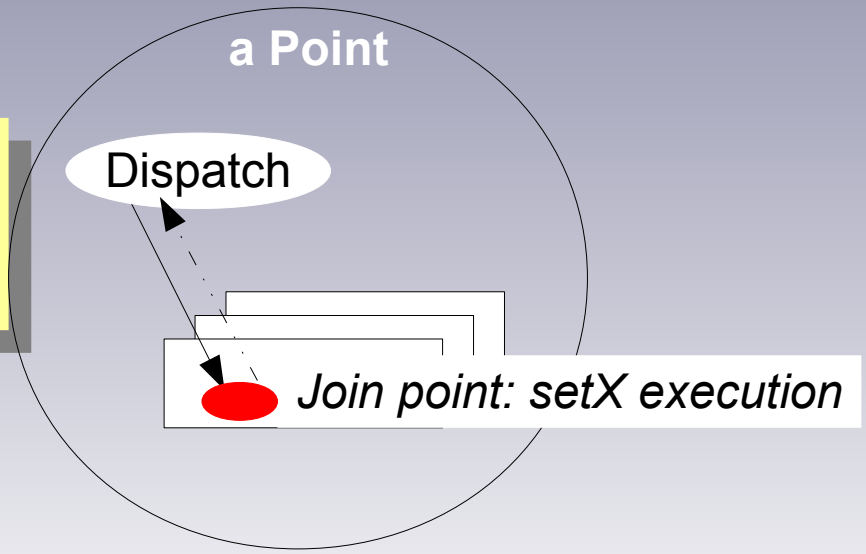
```
pointcut move: call (void Point.setX(int));
```

- Exposing the context:

```
pointcut move (Point p, int newX):
  call (void setX(..)) &&
  target (Point) &&
  args (int);
```

- Extracting the values

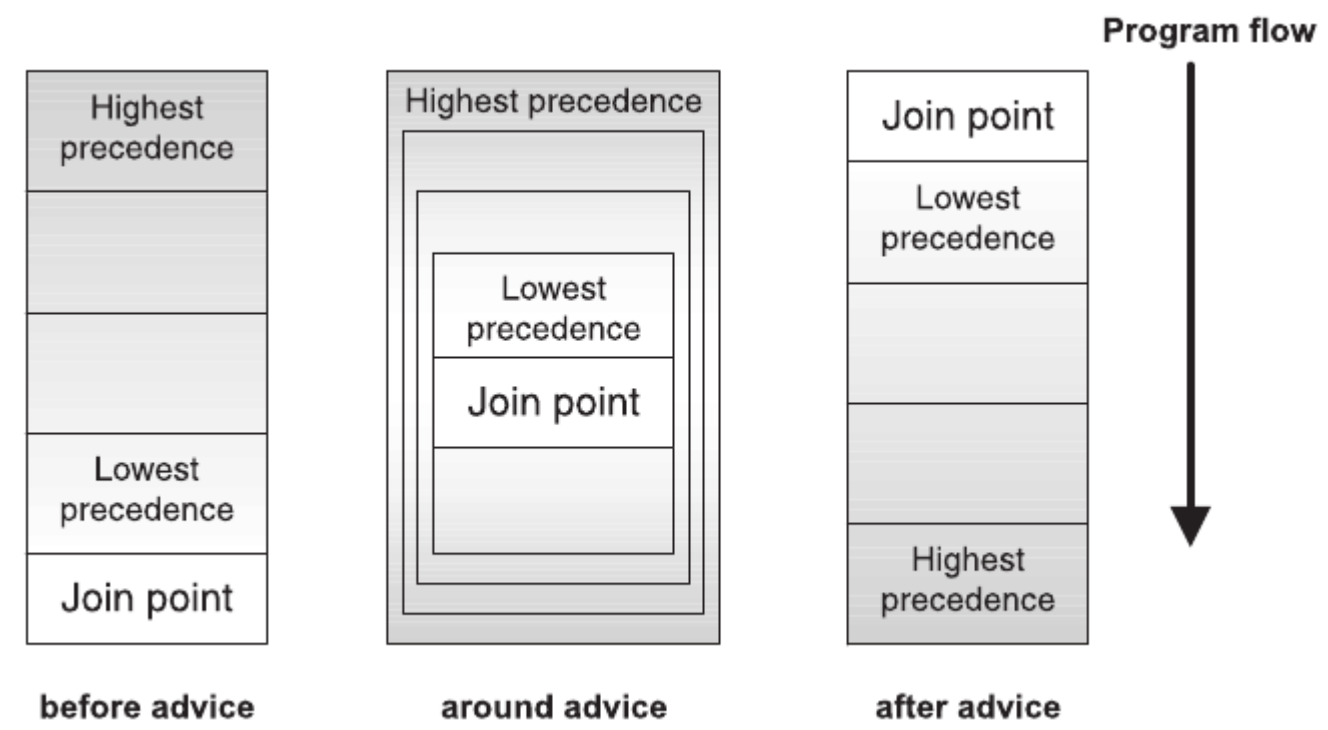
```
before (Point p, int newX) : move (p, newX) {
  System.out.println ("The point " + p +
                      " moved to: " + newX);
}
```



Now the advice can use the data of the join point

AspectJ: Advice precedence (1)

- *What happens when multiples advices affect the same join point?*
  - When this condition arises AspectJ uses precedence rules to determine the execution ordering of the advices



AspectJ: Advice precedence (2)

- How to set the precedence between aspects?
  - Through the **`precedence`** constructor

```
declare precedence : AuthenticationAspect, AuthorizationAspect;
```

  - The aspects on the left dominates the aspects on the right
  - More examples:

```
declare precedence : Auth*, PoolingAspect, LoggingAspect;
```

```
declare precedence : AuthenticationAspect, *;
```

```
declare precedence : *, CachingAspect;
```

AspectJ: Advice precedence (3)

- How to set the precedence between advices defined in the same aspect?
  - Their precedence is determined by their **order** and **type**

```
public aspect InterAdvicePrecedenceAspect {
  public pointcut performCall() : call(* TestPrecedence.perform());

  after() returning : performCall() {
    System.out.println("<after1/>");
  }
  before() : performCall() {
    System.out.println("<before1/>");
  }
  void around() : performCall() {
    System.out.println("<around>");
    proceed();
    System.out.println("</around>");
  }
  before() : performCall() {
    System.out.println("<before2/>");
  }
}
```

Output:

```
<before1/>
<around>
<before2/>
<performing/>
<after1/>
</around>
```

AspectJ: Inter-type declarations (1)

- Through this mechanism AspecJ is able to modify the existent types of the base program
- Inter-type declarations can be used to:
  - provide definitions of fields, methods, and constructors on behalf of other types
  - implement interfaces and to declare super-types

```
declare parents : banking..entities.* implements Identifiable;
```

The declaration of parents must follow the regular Java object hierarchy rules. For example, you cannot declare a class to be the parent of an interface. Similarly, you cannot declare parents in such a way that it will result in multiple inheritance

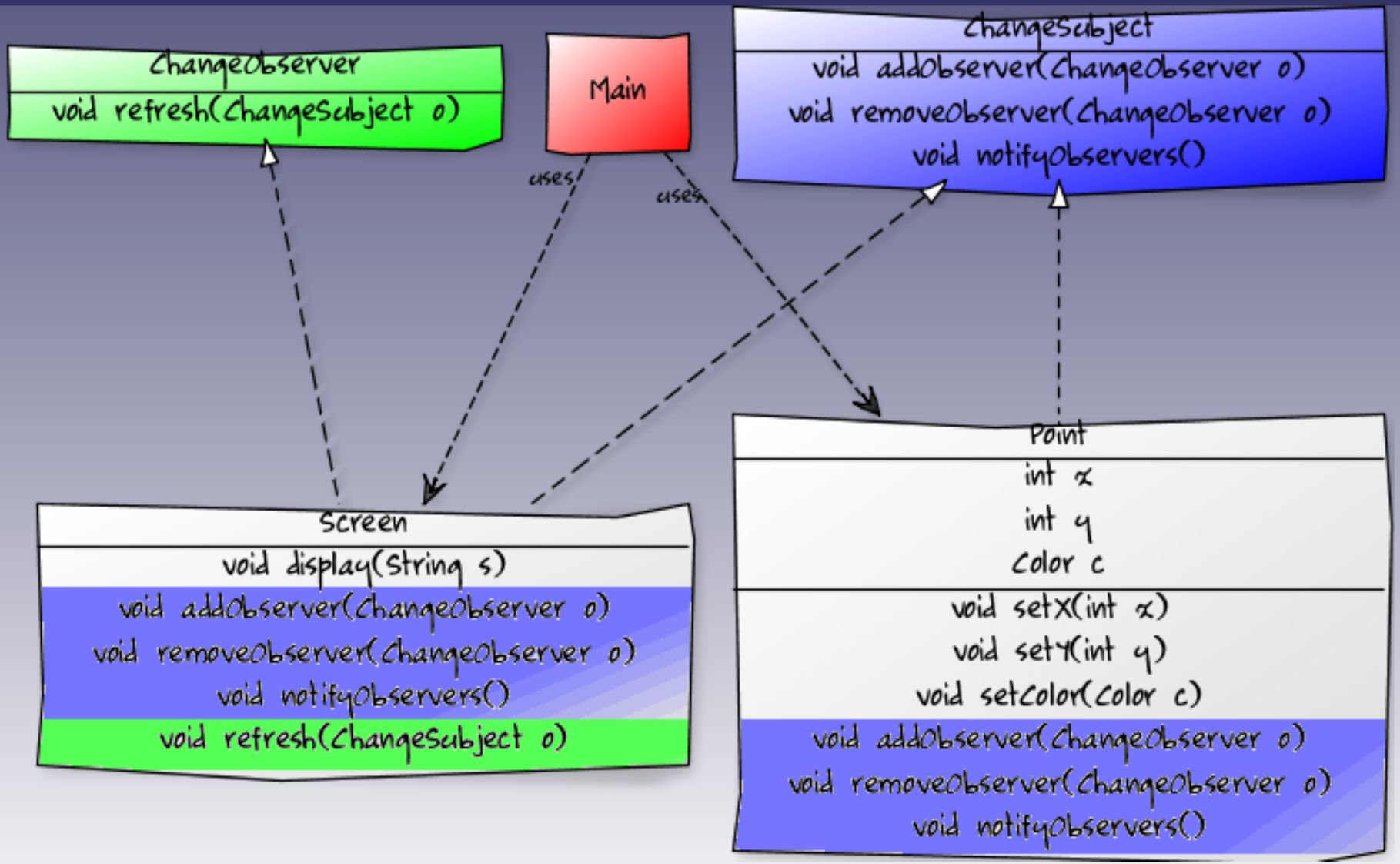
Learning by example: The update problem revised

- Addressing the second version of the display update problem: we want to add more than one view
- Typical solution using OO: the Observer pattern [4]
  - Even when using a design pattern we have crosscuting concerns!
  - The notify action gets scattered through all the hierachy of FigureElements

```
void setP1(Point p1) {
  this.p1 = p1;
  super.setChanged();
  super.notifyObservers();
}
```

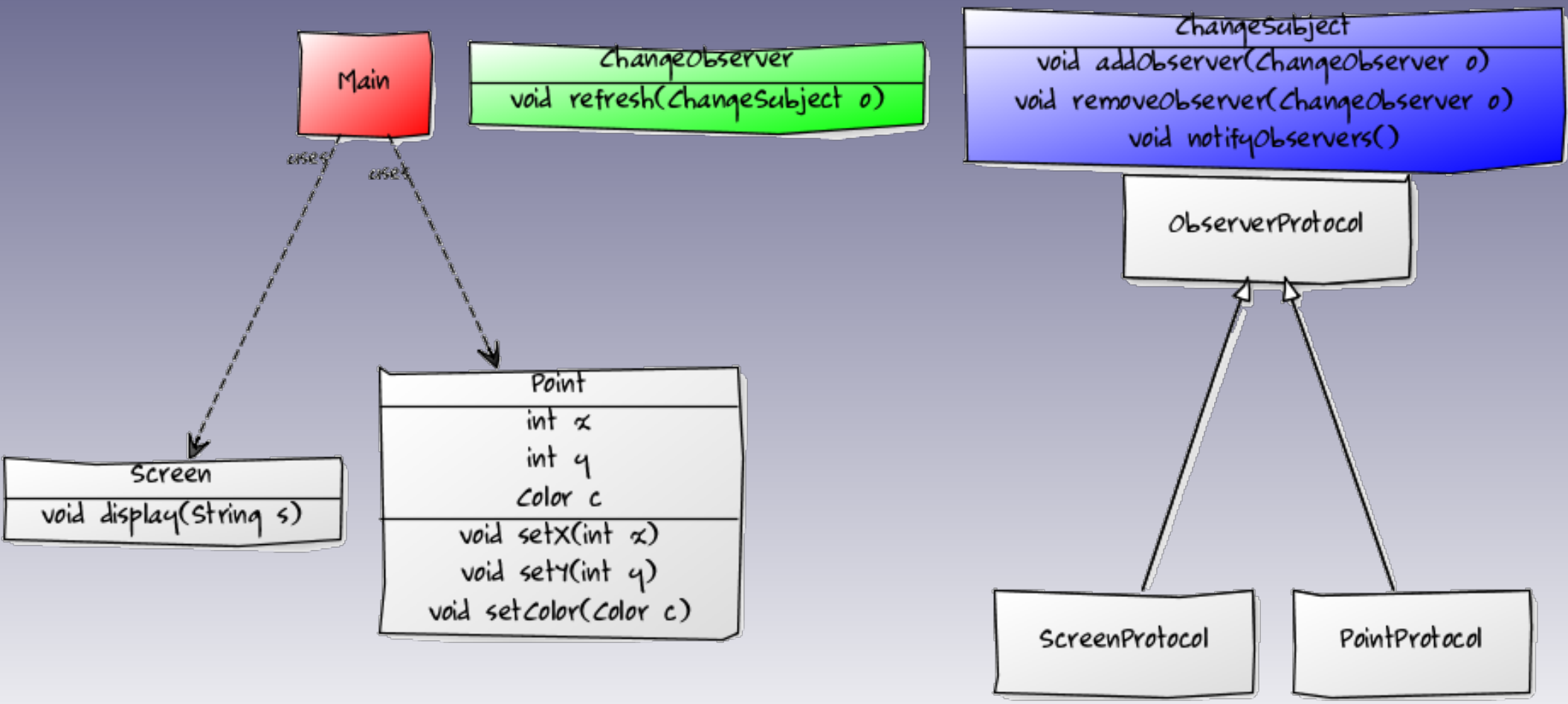
```
void setX(int x) {
  this.x = x;
  super.setChanged();
  super.notifyObservers();
}
```

Learning by example: Observer design pattern (1)



Learning by example: Observer design pattern (2)

- An aspect-oriented design



Learning by example: ObserverProtocol aspect

```
public abstract aspect ObserverProtocol {
    protected interface Subject  {}
    protected interface Observer {}

    private WeakHashMap perSubjectObservers;

    protected List getObservers(Subject subject) {
        if (perSubjectObservers == null) {
            perSubjectObservers = new WeakHashMap();
        }
        List observers = (List)perSubjectObservers.get(subject);
        if ( observers == null ) {
            observers = new LinkedList();
            perSubjectObservers.put(subject, observers);
        }
        return observers;
    }
    public void addObserver(Subject subject, Observer observer) {
        getObservers(subject).add(observer);
    }
    public void removeObserver(Subject subject, Observer observer) {...}

    protected abstract pointcut subjectChange(Subject s);
    protected abstract void updateObserver(Subject subject, Observer observer);

    after(Subject subject): subjectChange(subject) {
        Iterator iter = getObservers(subject).iterator();
        while ( iter.hasNext() ) {
            updateObserver(subject, ((Observer)iter.next()));
        }
    }
}
```

Learning by example: ScreenObserver aspect

```
public aspect ScreenObserver extends ObserverProtocol{

  declare parents: Screen implements Subject;

  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject subject):
    call(void Screen.display(String)) && target(subject);

  protected void updateObserver(Subject subject, Observer observer) {
    ((Screen)observer).display("Screen updated " +
    "(screen subject displayed message).");
  }
}
```

AspectJ: Annotations (1)

- AspectJ 5 supports an annotation-based style of aspect declaration

```
public aspect Foo {}
```

```
@Aspect
public class Foo {}
```

```
pointcut anyCall() :
call(* *.*(..));
```

```
@Pointcut("call(* *.*(..))")
void anyCall() {}
```

```
after() returning : anyCall() {
  System.out.println("phew");
}
```

```
@AfterReturning("anyCall")
public void phew() {
  System.out.println("phew");
}
```

```
declare parents : org.xyz..*
implements Serializable;
```

```
@DeclareParents("org.xyz..*")
Serializable implementedInterface;
```

All declarations are equivalent

AspectJ: Annotations (2)
Capturing join points through meta-data

- Name-based poincut

```
pointcut transactedOps()
    : execution(public void Account.credit(..))
      || execution(public void Account.debit(..))
      || execution(public void Customer.setAddress(..))
      || execution(public void Customer.addAccount(..))
      || execution(public void Customer.removeAccount(..));
```

- Annotation-based poincut

```
pointcut execution(@Transactional * *.*(..));
```

```
public class Account {
    ...
    @Transactional(kind=Required)
    public void credit(float amount)  {
        ...
    }
    @Transactional(kind=Required)
    public void debit(float amount)  {
        ...
    }
    ...
```

AspectJ: Annotations (2)

- What about...
  - obliviousness?
  - reuse?

AspectJ: Kinds of Weaving

- Aspectj's weaver takes as input class files and produces class files as output
- This process can take place at one of three different times
  - Compile-time
  - Post-compile time (link-time)
  - Load-time weaving – since AspectJ 1.5

** More information about how AspectJ's weaving works in "Advice Weaving in AspectJ" Hilsdale and Hugunin [5]*

AspectJ: Load-time weaving (1)

- Configuration through one or more META-INF/aop.xml files located on the class loader search path
- Each file may declare a list of aspects to be used for weaving, type patterns describing which types should woven, and a set of options to be passed to the weaver
- Additionally AspectJ 5 supports the definition of concrete aspects in XML

AspectJ: Load-time weaving (2)

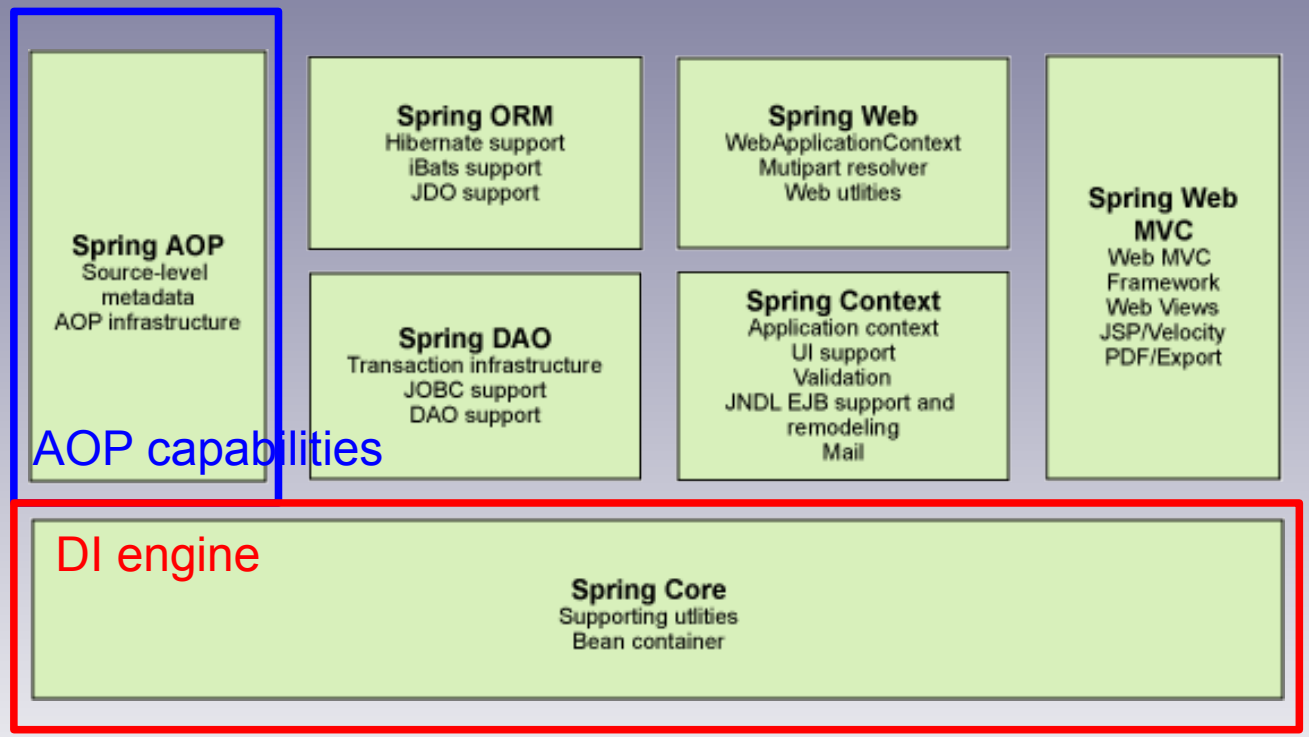
```
<aspectj>
        <aspects>
          <aspect name="com.MyAspect"/>
          <aspect name="com.MyAspect.Inner"/>
          <concrete-aspect name="com.xyz.tracing.MyTracing"
                        extends="tracing.AbstractTracing"
                        precedence="com.xyz.first, *">
          <pointcut name="tracingScope" expression="within(org.maw.*)"/>
          </concrete-aspect>

          <include within="com..*"/>
         <exclude within="@CoolAspect *"/>
        </aspects>

        <weaver options="-verbose">
          <include within="javax.*"/>
          <include within="org.aspectj.*"/>
          <include within="(!@NoWeave foo.*) AND foo.*"/>
          <exclude within="bar.*"/>
          <dump within="com.foo.bar.*"/>
          <dump within="com.foo.bar..*" beforeandafter="true"/>
        </weaver>
</aspectj>
```

Agenda

- Definitions
- Aspect-Oriented Languages
  - AspectJ
- Dynamic Java Frameworks
  - Spring AOP
  - JAC
  - JBoss AOP
- Tools comparison
- Current Limitations

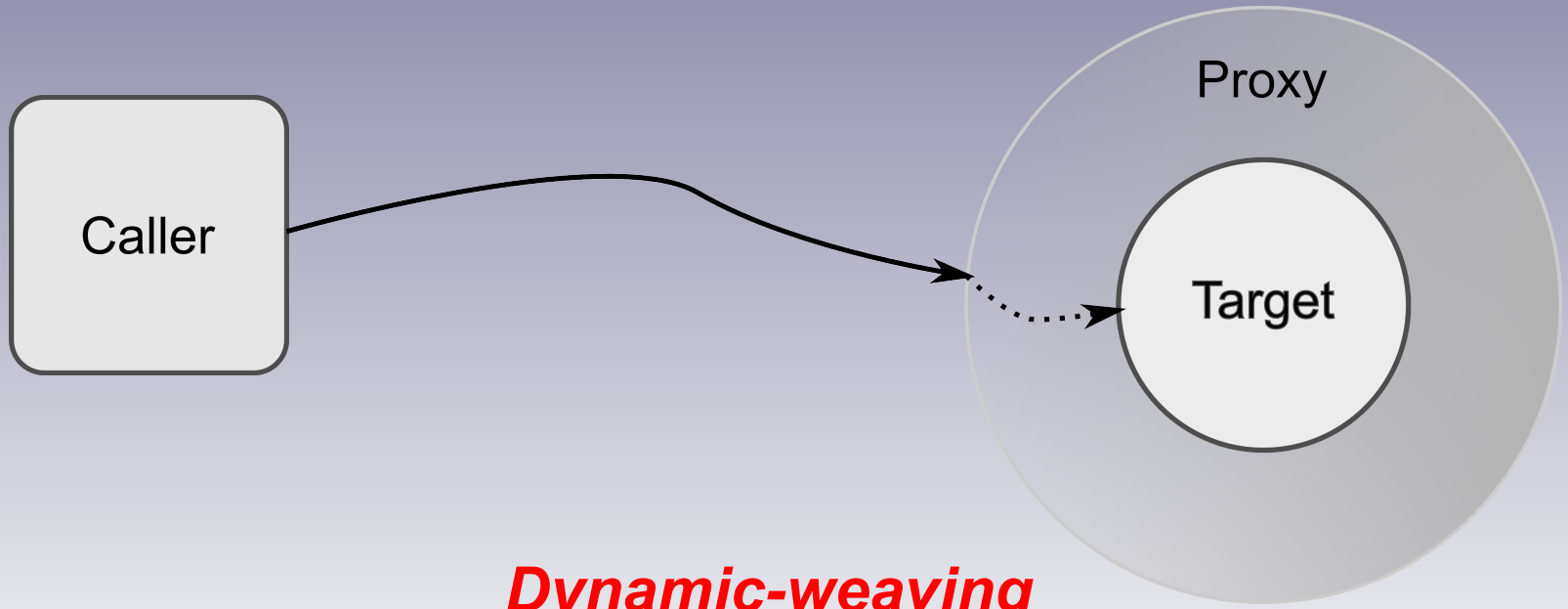
Spring AOP: Introduction

- Goal: to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications [6]



Spring AOP: Proxy-based weaving

- Aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class
- Because it is based on dynamic proxies, Spring only supports method join points



Dynamic-weaving

Spring AOP: Example

- Tracing example on Spring

```
public class TracingAdvice implements MethodBeforeAdvice, AfterReturningAdvice {

    public TracingAdvice() {}

                                Wrapped method  Parameters      Target object
    public void before(Method arg0, Object[] arg1, Object arg2)
            throws Throwable {
        System.out.println("Method: " + arg0.getName());
    }

                                Returned value  Wrapped method  Parameters    Target object
    public void afterReturning(Object arg0, Method arg1, Object[] arg2, Object arg3)
            throws Throwable {
        System.out.println("Method: " + arg1.getName());
    }
}
```

Spring AOP: Advices

- Advice code is written in standard Java classes
  - Benefit: advice code can be reused through different pointcut definitions

Advice type	Interface
Before	org.springframewrok.aop.MethodBeforeAdvice
After-returning	org.springframewrok.aop.AfterReturningAdvice
After-throwing	org.springframewrok.aop.ThrowsAdvice
Around	org.aop.alliance.intercept.MethodInterceptor
Introduction	org.springframewrok.aop.IntroductionInterceptor

Spring AOP: Pointcuts

- Spring provides two classes of pointcuts: regular expressions and AspectJ like pointcuts
- Regular expression pointcuts can be defined using one of the following two classes
  - org.springframework.aop.support.Perl5RegexpMethodPointcut
    - *Useful when using pre-Java 1.4*
  - org.springframework.aop.support.JdkRegexpMethodPointcut
    - Example:

```
<bean id="setterPointcut"
      class="org.springframework.aop.support.JdkRegexpMethodPointcut">
      <property name="pattern" value=".*set*" />
</bean>
```

Spring AOP: Pointcuts + Advisors

- AspectJ-style expressions can be used for defining Spring pointcuts

```
<bean id="setterPointcut"
      class="org.springframework.aop.aspectj.AspectJExpressionPointcut">
      <property name="expression" value="execution(* FigureElement+.set*(..))" />
</bean>
```

- In order to combine a pointcut and an advice, an advisor must be defined

```
<bean id="setterAdvisor"
      class="org.springframework.aop.aspectj.AspectJExpressionPointcutAdvisor">
      <property name="advice" ref="tracingAdvice" />
      <property name="expression" value="execution(* FigureElement+.set*(..))" />
</bean>
```

Spring AOP: @AspectJ Support

- Spring 2.0 interprets the same annotations as AspectJ 5
  - Yet, the AOP runtime is still pure Spring AOP
  - There is no dependency on the AspectJ compiler or weaver

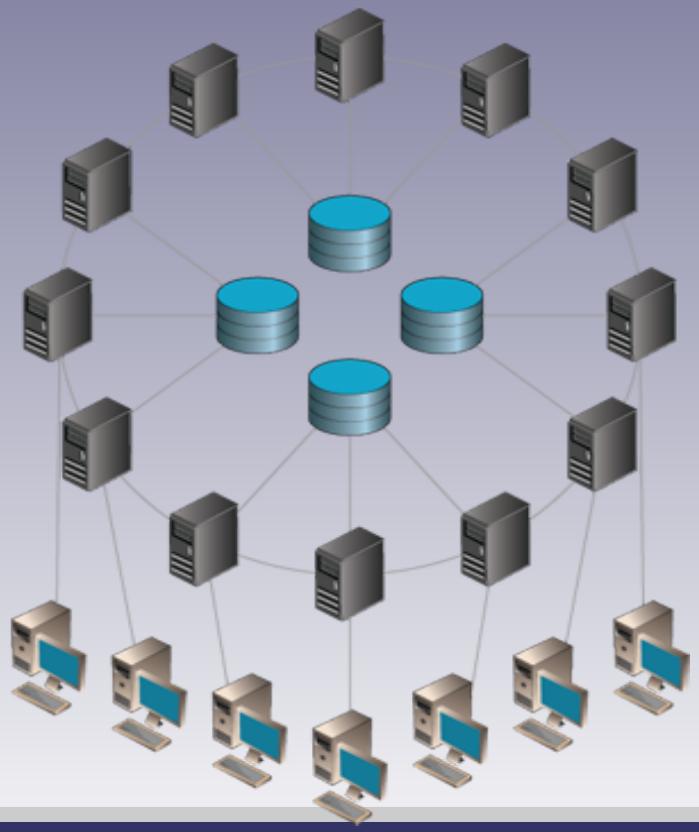
```
@Aspect
public class TracingAspect {

    public TracingAspect() {}

    @Pointcut("execution(* *.set*(..)) && args(x) && !within(TracingAspect)")
    public void setters(int x) {}

    @Before ("setters(x)")
    public void traceBefore(int x) {
        System.out.println("Object's state change to: " + x);
    }
}
```

Agenda

- Definitions
- Aspect-Oriented Languages
  - AspectJ
- Dynamic Java Frameworks
  - Spring AOP
  - JAC
  - JBoss AOP
- Tools comparison
- Current Limitations

JAC – Java Aspect Components

- JAC [7] is a framework for the development of dynamic and distributed aspect-oriented systems
  - Targeted for application running in open and distributed environments
  - Solution for concerns like:
    - fault-tolerance
    - data consistency
    - remote version updating
    - run-time maintenance
    - dynamic server lookup
    - scalability



JAC – Concepts

- JAC defines two levels of aspect-oriented programming
  - *Programming level*: create aspects, pointcuts, wrappers

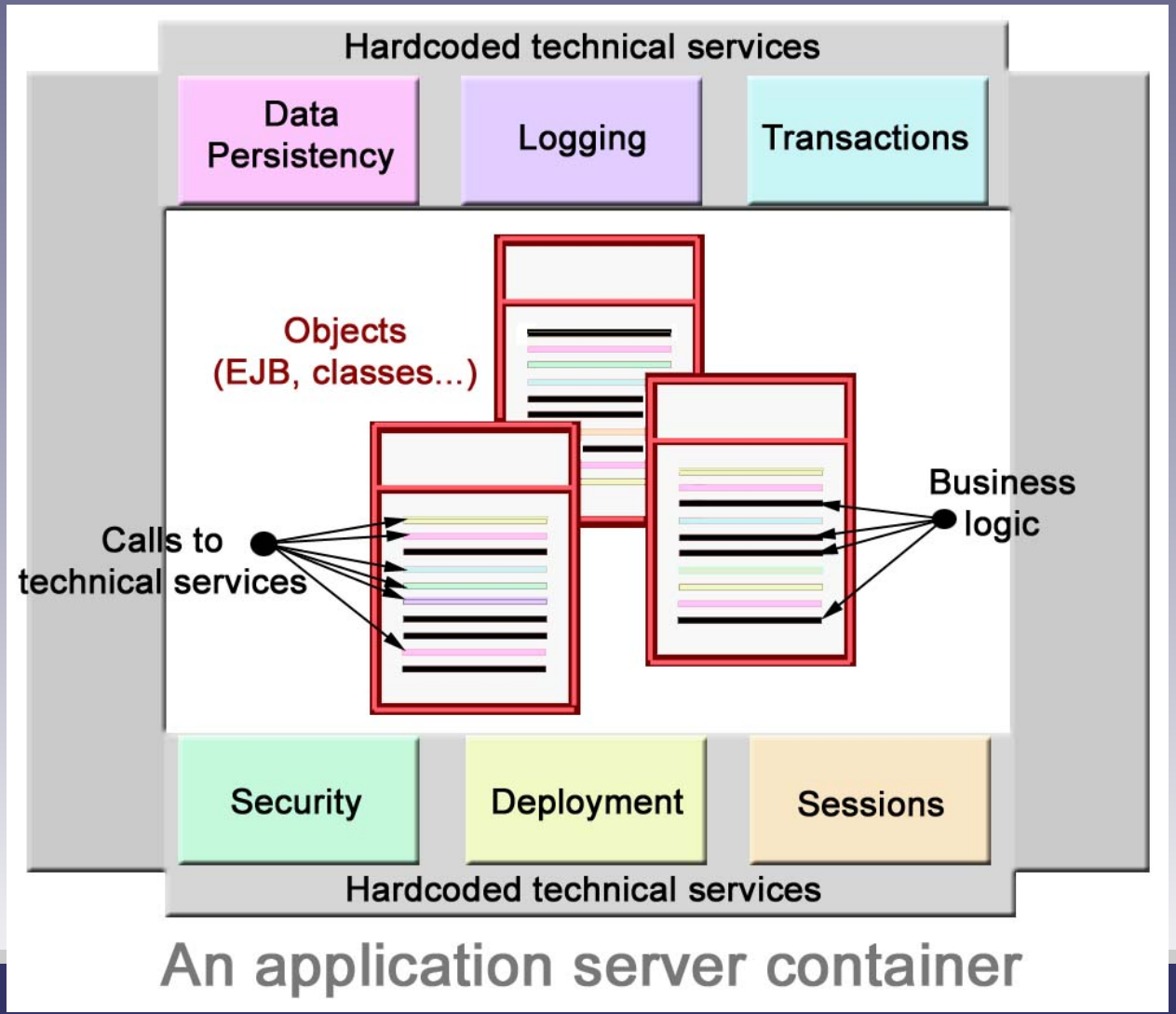
```
public class MyAspect extends AspectComponent {
    public void synchro(ClassItem cl, String method) {
        cl.getMethod(method).setAttribute(synchronized,new Boolean(true));
        pointcut(ALL,ALL,ALL,MyWrapper.class,wrappingMethod,true);
    }
}
```

  - *Configuration level*: customize existing aspects to work with existing applications

```
synchro C m1;
synchro C m2;
```

JAC – Programming level (1)

- Aspect components are hosted by JAC containers



An application server container

- JAC containers host both business components and aspect components
- These base objects may not be located in a single component
- This provides a means to develop distributed applications based on distributed aspects

JAC – Programming level (2)

- Defining an aspect in JAC
- *TraceAspect* class defines an aspect by extending the *AspectComponent* class, which is defined by the JAC API as the root class of all aspects

```
import org.objectweb.jac.core.AspectComponent;
public class TraceAspect extends AspectComponent {
    public TraceAspect() {
        pointcut(
            ".*",
            "aop.jac.Order",
            "addItem(java.lang.String,int):void",
            "aop.jac.TraceWrapper",
            null, false );
    }
}
```

Pointcut
- the three first parameters defines the joinpoints we want to match
- the forth parameter defines the wrapper implementation to use

JAC – Programming level (3)

- Implementing the trace wrapper

```
public class TraceWrapper extends Wrapper {
    public TraceWrapper(AspectComponent ac) {
        super(ac);
    }
    public Object invoke(MethodInvocation mi) throws Throwable {
        System.out.println("-> Before addItem");
        Object ret = proceed(mi);
        System.out.println("-> After addItem");
        return ret;
    }
    public Object construct(ConstructorInvocation ci) throws Throwable
    {
        return proceed(ci);
    }
}
```

JAC – Programming level (4)

- Wrappers contains the code to be executed on a certain join-point that match a given pointcut
- JAC wrappers are defined by extending the Wrapper class
  - Since aspects and wrappers are defined in distinct classes, the wrapper's code is more reusable
- These are always “around” wrappers; no special definition exists for “before” or “after” wrappers

JAC – Configuration level (1)

- Each aspect that is defined with JAC is associated with an aspect-configuration file
- The idea is to separate the code from the initialization values that are used by the code.
  - This improves the reuse of the aspect code

```
public class TraceAspect2 extends AspectComponent {
   public void trace(String objPE, String clPE, String metPE) {
      pointcut(
         objPE, clPE, metPE,
         "aop.jac.TraceWrapper",
         null, false );
   }
}
```

A configurable version of the tracing aspect

JAC – Configuration level (2)

- The configuration file contains the following sentence:

```
trace ".*" "aop.jac.Order" "addItem(java.lang.String,int):void"
```

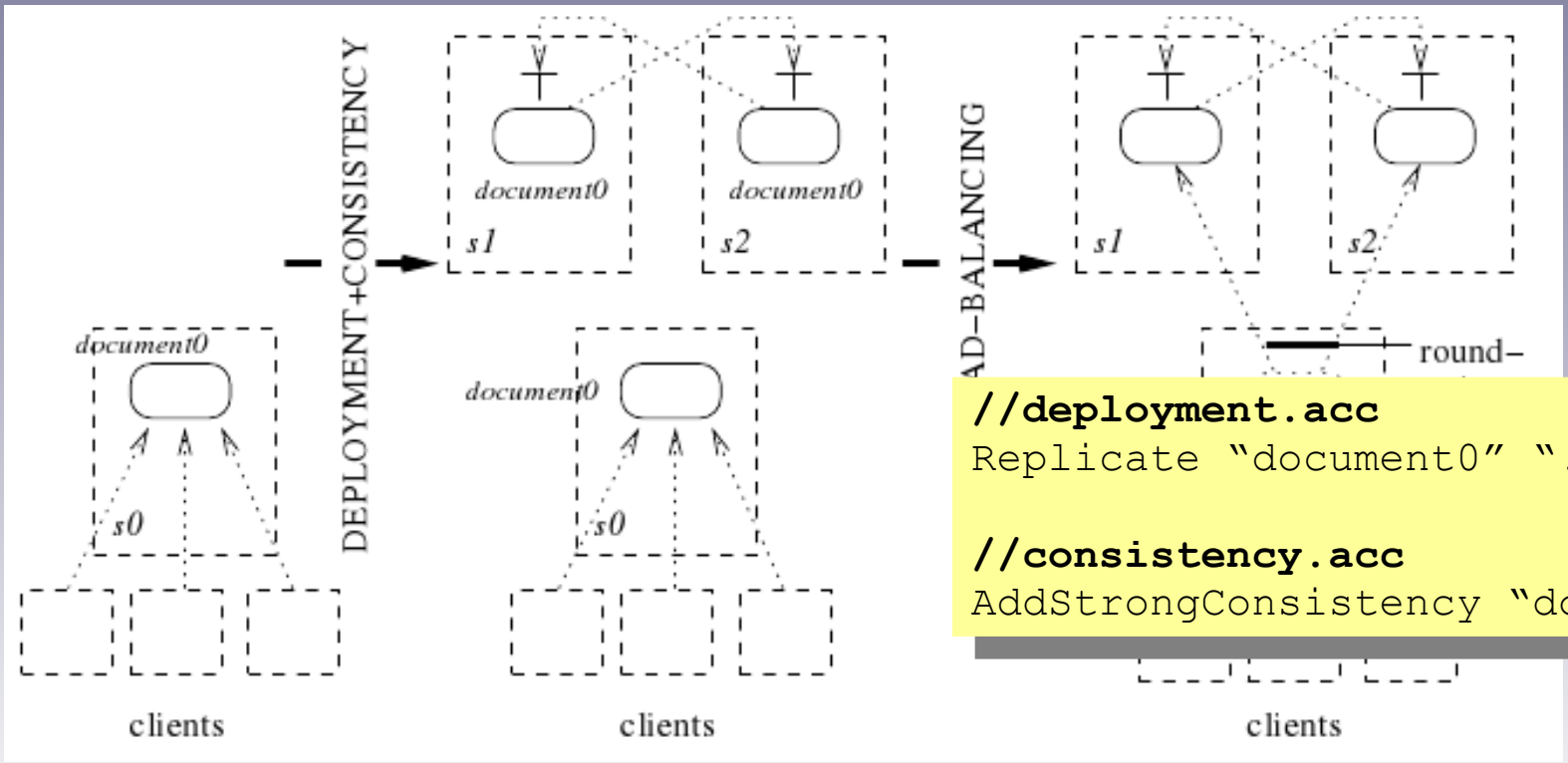
- The aspect-configuration file provides the current value for the definition of the pointcut
- These values can be changed without recompiling the aspect

JAC - Distribution

- JAC supports the distribution of aspects
  - Some aspects may be required in several containers
- Two aspect component are required
  - A deployment aspect, used to create a distributed application from a centralized one (pointcut + RMI)
  - A set of distributed aspects that implement distributed protocols, which enable the different objects to collaborate between the different containers
    - *In JAC, pointcuts can match join points located in different hosts*

JAC – A load-balancing aspect (1)

- Example of distributed application with JAC
  - *document0* object server
  - *s0* initial centralized server




```
//deployment.acc
Replicate "document0" ".*[1-2]"

//consistency.acc
AddStrongConsistency "document0" ".*[1-2]"
```

JAC – A load-balancing aspect (2)

```
public class LoadBalancingAC extends AspectComponent {
    LoadBalancingAC() {
        pointcut("document0", "Document", ".*",
                LoadBalancingWrapper.class, "loadBalance","s0");
    }
    class LoadBalancingWrapper extends Wrapper {
        int count = 0;

        public Object loadBalance() {

            if( replicaRefs.length == 0 ) return proceed();

            if( count >= replicaRefs.length ) {
                count = 0;
            }

            return replicaRefs[count++].invoke(method(),args());
    }
}
```

Host / Container name

Agenda

- Definitions
- Aspect-Oriented Languages
  - AspectJ
- Dynamic Java Frameworks
  - Spring AOP
  - JAC
  - JBoss AOP
- Tools comparison
- Current Limitations

JBoss AOP: Introduction

"JBoss AOP is a 100% Pure Java aspected oriented framework usuable in any programming environment or tightly integrated with our application server" [10]

- Key features:
  - Dynamic weaving
  - Ready-to-use aspect frameworks
    - Design patterns
    - Synchronization
    - Caching
    - Etc
  - Mixins (similar to intertype declarations of AspectJ)

JBoss AOP: Introductions

- Making a non-serializable class Serializable

```
<introduction class="POJO">
   <interfaces>
      java.io.Serializable
   </interfaces>
</introduction>
```

- If the introduced interfaces have methods not implemented by the class, then JBoss will add a default implementation

JBoss AOP: Mixins

- JBoss allows us to force a class to implement an interface and even specify an additional class called a mixin that implements that interface

```
public class POJO {
  private String field;
}
```

```
<introduction class="POJO">
  <mixin>
    <interfaces>
      java.io.Externalizable
    </interfaces>
    <class>ExternalizableMixin</class>
    <construction>new
ExternalizableMixin(this)
    </construction>
  </mixin>
</introduction>
```

An instance of this mixin class will be allocated the first time you invoke a method of the introduced interface

Agenda

- Definitions
- Aspect-Oriented Languages
  - AspectJ
- Dynamic Java Frameworks
  - Spring AOP
  - JAC
  - JBoss AOP
- Tools comparison
- Current Limitations

Tool comparison – Features (1)

- Comparison of Features of AspectJ, JAC, JBoss AOP, and Spring AOP [11]

Feature	AspectJ	JAC	JBoss AOP	Spring AOP
Approach	Language	Framework	Framework	Framework
Weaver implementation	Compile time or load time	Runtime	Runtime	Runtime
Aspect	Keyword *aspect*	Class extending AspectComponent	Regular Java class	Set of advisors
Pointcut	Dedicated syntax	Method pointcut with GNU reg. exp.	XML tags with AspectJ, Perl5 or Java reg. exp.	XML tags with AspectJ, Perl5 or Java reg. exp.
After/Before code	Advice code	Wrapper	Method interceptor	Method interceptor
Intercepted object execution	proceed	proceed	invokeNext	proceed

Tool comparison – Features (2)

- Feature comparison continued

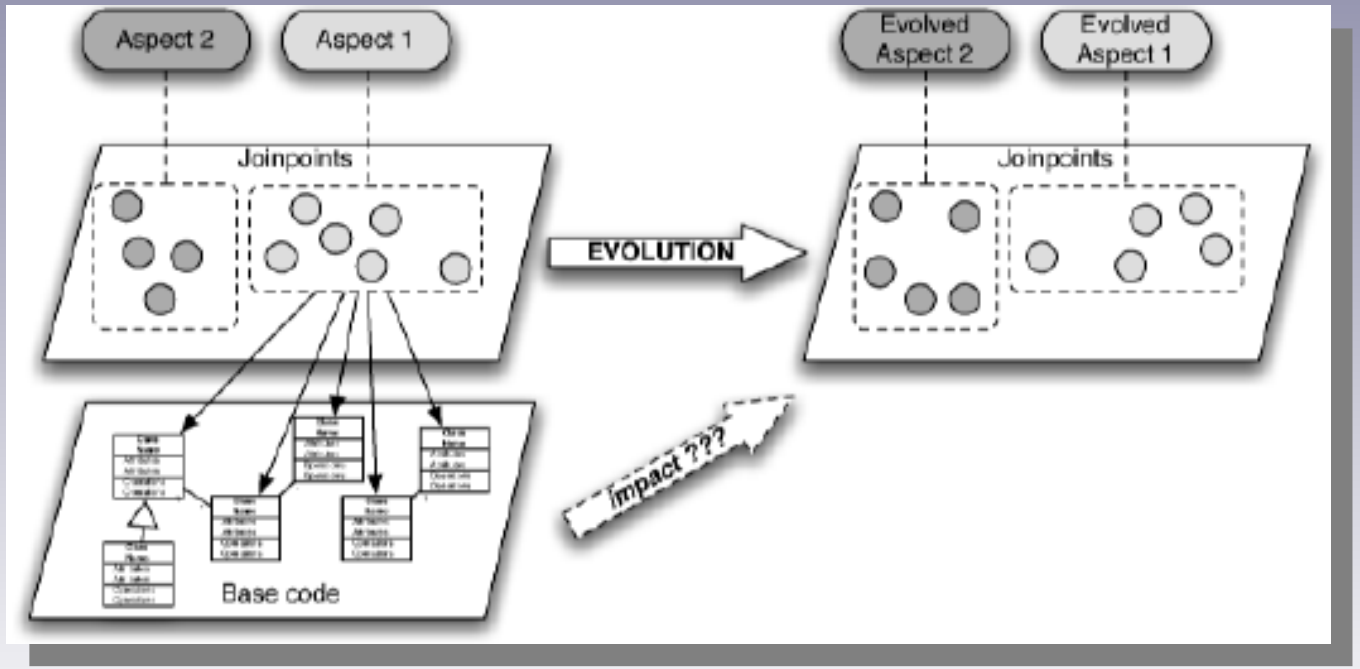
Feature	AspectJ	JAC	JBoss AOP	Spring AOP
Introduction mechanism	Intertype declaration	Role-method	Mixin	Introduction advice (mixin)
Default aspect instantiation model	Singleton	Singleton	One aspect instance per aspectized class	Singleton
Aspect ordering	Keyword *declare precedence*	Property jac.comp. wrappingOrder	Declaration order in the jboss-aop.xml file	org.springframework.core.Ordered interface implementation

Agenda

- Definitions
- Aspect-Oriented Languages
  - AspectJ
- Dynamic Java Frameworks
  - Spring AOP
  - JAC
  - JBoss AOP
- Tools comparison
- Current Limitations

Current limitations: AOL Pitfalls

- Actual aspect-oriented languages and frameworks all share a number of limitations or drawbacks
- Particularly, the *fragile pointcut* [12] and the *aspect composition* [13] problems are the most common



Current limitations: The Fragile Pointcut Problem

- This problem occurs when pointcuts accidentally capture or miss particular joinpoints because of seemingly safe modifications to base code [12]

Capture all changes to the x field

```
pointcut setX (int x) :
    (execution (void FigureElement.setXY (int, int) && args(x, *))
    || (execution (void Point.setX(int)) && args(x));
```

- Several changes can affect the definition of a pointcut
  - Rename method/class/field
  - Move method class
  - Add/Delete method/field/class

Evolution scenarios

```
Add a new method incX(int)
Rename setXY to resetXY
```

Current limitations: Aspect composition problem

- This problem arises when combining into the same application two or more independently developed aspects
  - They may interact in undesirable ways
- Example: given a synchronization and a logging aspect
  - Do the logging aspect log the synchronization code?
  - Do the synchronization aspect synchronize the logging code?

For complex aspects, constructs like `precedence` of AspectJ do not suffice [14]

Questions & Answers

References (1)

1. R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, **Aspect-Oriented Software Development**. Addison-Wesley Professional, October 2004.
2. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, **"An Overview of AspectJ,"** in ECOOP, ser. Lecture Notes in Computer Science, J. L. Knudsen and J. L. Knudsen, Eds., vol. 2072. Springer, 2001, pp. 327-353.
3. R. Pawlak, J.-P. Retaillé, and L. Seinturier, **Foundations of AOP for J2EE** Development. Apress, September 2005
4. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software** (Addison-Wesley Professional Computing Series), illustrated edition ed. Addison-Wesley Professional, November 1994.
5. E. Hilsdale and J. Hugunin, **"Advice weaving in aspectj,"** in AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development. New York, NY, USA: ACM, 2004, pp. 26-35.

References (2)

8. SpringAOP. http://static.springsource.org/spring/docs/2.5.x/reference/aop.html
9. JAC. http://jac.ow2.org/documentation.html
10. JBoss AOP. http://jboss.org/jbossaop/
11. R. Pawlak, J.-P. Retaillé, and L. Seinturier, Foundations of AOP for J2EE Development. Apress, September 2005.
12. Koppen, C., Störzer, M.: PCDiff: Attacking the fragile pointcut problem. In Gybels, K., Hanenberg, S., Herrmann, S., Wloka, J., eds.: European Interactive Workshop on Aspects in Software (EIWAS). (September 2004)
13. W. K. Havinga, I. Nagy, and L. M. J. Bergmans, "An analysis of aspect composition problems," in Proceedings of the Third European Workshop on Aspects in Software, 2006, Enschede, Netherlands.
14. T. Mens and T. Tourwé, "Evolution issues in aspect-oriented programming," 2008, pp. 203-232.

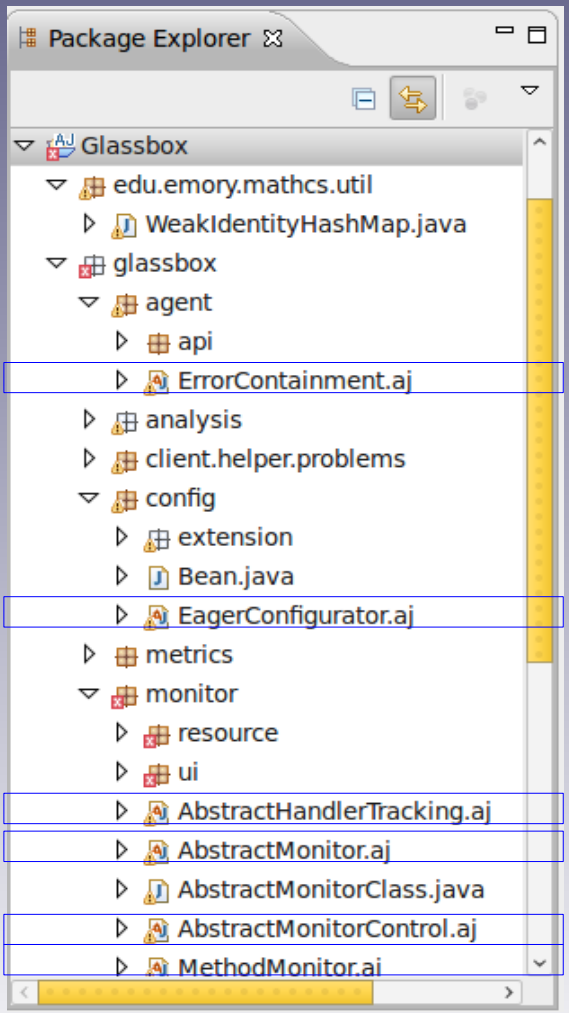
Extras

Code management: Where should the aspect files reside?

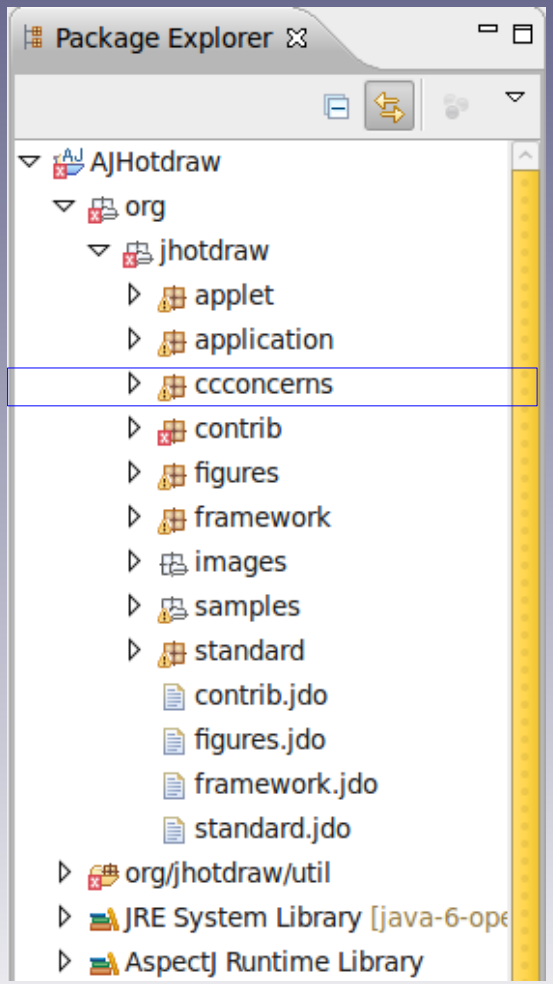
- Should the aspect files be collocated in a separate package?
  - In a large application, there can be some pervasive aspects that could be placed inside an isolated package for aspects [Merson05]
  - Nonetheless, there can be application specific aspects that are used in confined parts of the application [Merson05]

Code management: Where should the aspect files reside?

Glassbox



AJHotdraw



CM

Fecha	Cambio	Autor	Vers.
08/04/2009	Creación	Esteban Sait Abait	1