Aspect-Oriented Programming (AOP).

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano

Outline

   - Aspect-Oriented Programming: Motivations.
   - Crosscutting Concerns and Aspect-Oriented Programming.

2. AspectJ
   - AspectJ by Examples.
   - An Introduction to Join Points, Pointcuts and Advices.
   - Pointcuts and Advices in Details.

   - Benefits in Adopting Aspect Oriented Programming.
   - When to Adopt AOP.
   - References

Aspect-oriented Programming (AOP).

An Example of Good Modularity: XML Parsing.

XML parsing in org.apache.tomcat:
- red shows relevant lines of code;
- nicely fits in one box.

Aspect-oriented Programming (AOP).

An Example of Bad Modularity: Logging.

Where is logging in org.apache.tomcat?
- red shows lines of code that handle logging;
- are not in just one place, not even in a small number of places.
Logging is not modularized at all.
Aspect-Oriented Programming (AOP).

The Cost of Tangled Code.

Redundant code:
- same fragment of code in many places.

Difficult to reason about:
- non-explicit structure;
- the big picture of the tangling isn’t clear.

Difficult to maintain the code:
- have to find all the code involved; and
- be sure to change it consistently; and
- be sure not to break it by accident.

Walter Cazzola

Aspect-Oriented Programming.

The AOP Idea.

Crosscutting is inherent in complex systems.

Crosscutting concerns:
- have a clear purpose;
- have a natural structure;
  - defined set of methods, module boundary crossings, points of resource utilization, lines of dataflow ...

So, let’s capture the structure of crosscutting concerns explicitly:
- in a modular way;
- with linguistic and tool support.

Aspects are well-modularized crosscutting concerns.

Aspects are two things:
- concerns that crosscut;
  ➔ design level
- a programming construct that enables crosscutting concerns to be captured in modular units.
  ➔ implementation level

AOP helps in improving the modularity of the crosscutting concerns.

AspectJ is an aspect-oriented extension to Java that supports general-purpose aspect-oriented programming.

Walter Cazzola

Aspect-Oriented Programming.

Looking Ahead.

We will face the problem from two different perspectives:

Problem Analysis
- crosscutting in the design, and how to use AspectJ to capture that.

AspectJ Language
- crosscutting in the code;
- mechanisms that AspectJ provides.
An Overlay onto Java:
- manages where a crosscutting occurs in the computational flows (join points); and
- a mechanism that manages the crosscutting concerns composition at the specified join points (weaver).

Extensions to the Java Language:
- pointcuts: predicates that pick out join points and values at those points;
- advice: additional action to take at join points in a pointcut;
- inter-class declaration aka introduction aka open classes;
- aspect: a modular unit of crosscutting behavior:
  - comprised of advice, inter-class declarations, pointcuts, field, constructor and method declarations.

The code that implements these classes looks like:

```java
class Line implements FigureElement{
    private Point p1, p2;
    public Point getP1() { return p1; }
    public Point getP2() { return p2; }
    public void setP1(Point p1) { this.p1 = p1; }
    public void setP2(Point p2) { this.p2 = p2; }
    public void moveBy(int dx, int dy) { p1.moveBy(dx, dy); p2.moveBy(dx, dy); }
}
class Point implements FigureElement {
    private int x = 0, y = 0;
    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    public void moveBy(int dx, int dy) { x += dx; y += dy; }
}
```

The display is a sort of a collection of FigureElements, that:
- move periodically;
The display must:
- be periodically refreshed; and
- coordinate asynchronous events.
Several applications manifest a similar behavior:
- session liveness;
- value checking.
Aspect-J.

Join Points

**Join Points** are key points in the dynamic call graph.

- Let suppose that the statement `l.moveBy(1,3)` moves the line `l`.

```
void Line.setP1(Point) ||
void Line.setP2(Point);
```

whenever an instance of Line receives a «void setP1(Point)>> or «void setP2(Point)>> method call.

**A pointcut** (also called primitive) is a sort of predicate on join points that:

- can match or not match any given join point; and
- optionally, can pull out some of the values at that join point.

```
call(void Line.setP1(Point))
```

it matches if the join point is a method call with this signature.

Several kinds of join points

- method and constructor call;
- method and constructor execution;
- field get and set;
- exception handler execution;
- static and dynamic initialization;

Pointcut: Composition.

pointcuts compose like predicates, using &&, || and !.
Aspect-Oriented Programming (AOP).

AspectJ

Pointcuts: a Means of Identifying Join Points (Cont’d).

User-Defined Pointcuts

The programmer can define his own pointcuts, named user-defined or named pointcut.

The user-defined pointcuts can be used in the same way as primitive pointcuts.

The primitive pointcut, can also be:
- call, execution;
- get, set;
- handler;
- initialization, staticinitialization.

An aspect defines a special class that can crosscut other classes.

aspect DisplayUpdating {
  pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

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

Note that the DisplayUpdating is a complete running code.

AspectJ.

Advice: after().

The after advice specifies the action to take after the computation under join points.

**After Advice**

```java
pointcut move():
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point));

after() returning: move() {
  «code here runs when the method move returns»
}
```

Walter Cazzola
Aspect-Oriented Programming.
Slide 17 of 57


An aspect defines a special class that can crosscut other classes.

```java
aspect DisplayUpdating {
  pointcut move():
    call(void Line.setP1(Point)) || call(void Line.setP2(Point));
  after() returning: move() { Display.update(); }
}
```

Note that the DisplayUpdating is a complete running code.

Walter Cazzola
Aspect-Oriented Programming.
Slide 19 of 57

To Update the Display without AOP.

What would you expect?
- update calls are tangled through the code;
- “what is going on” is less explicit.

```java
public class Line extends FigureElement {
  private Point p1, p2;
  public Point getP1() { return p1; }
  public Point getP2() { return p2; }
  public void setP1(Point p1) {
    this.p1 = p1;
    Display.update();
  }
  public void setP2(Point p2) {
    this.p2 = p2;
    Display.update();
  }
  public void moveBy(int dx, int dy) {
    p1.moveBy(dx, dy); p2.moveBy(dx, dy);
    Display.update();
  }
  }
```

Walter Cazzola
Aspect-Oriented Programming.
Slide 20 of 57
aspectj

Using Values at Join Points.

Pointcuts can:
- cut across multiple classes; and
- use interface signatures.

Pointcuts can explicitly expose certain values and advice can use such values.

```java
pointcut move(FigureElement figElt):
  target(figElt) &&
  (call(void FigureElement.move(int, int)) ||
   call(void Line.setP1(Point)) ||
   call(void Point.setX(int)) ||
   call(void Point.setY(int)));

after(FigureElement fe) returning: move(fe) {
  «fe is bound to the figure element»
}
```

Walter Cazzola

Explaining Parameters of User-Defined Pointcut Designator.

Variables are bound by user-defined pointcut declaration
- pointcut supplies value for variable;
- value is available to all users of user-defined pointcut.

```java
pointcut move(Line l):
  target(l) &&
  (call(void Line.setP1(Point)) ||
   call(void Line.setP2(Point)));

after(Line line) returning: move(line) {
  «line is bound to the line»
}
```

Walter Cazzola

Explaining Parameters of Advice.

Values are «pulled»
- right to left across ´:´; left to right side
- from pointcuts to user-defined pointcuts;
- from pointcuts to advice, and then advice body.

```java
pointcut move(Line l):
  target(l) &&
  (call(void Line.setP1(Point)) ||
   call(void Line.setP2(Point)));

after(Line line) returning: move(line) {
  «line is bound to the line»
}
```

Walter Cazzola
Aspect-Oriented Programming (AOP).

**AspectJ**

**Target()**: Primitive Pointcut Designator.

**target(type name | formal reference)**

It does two things:
- exposes target;
- predicate on join points - any join point at which target object is an instance of type name (a dynamic test).

**target()** can be used in a polymorphic pointcut:
- does not further restrict the join points;
- does pick up the target object.

---

Walter Cazzola
Aspect-Oriented Programming.
Slide 25 of 57

---

Aspect-Oriented Programming (AOP).

**AspectJ**

**Aspect-Oriented Software Development: Few Hints.**

**AspectJ by Examples.**

**An Introduction to Join Points, Pointcuts and Advices.**

**Pointcuts and Advices in Details.**

---

**target(Point)**

**target(Line)**

**target(FigureElement)**

---

Aspect-Oriented Programming (AOP).

**AspectJ**

**Aspect-Oriented Software Development: Few Hints.**

**AspectJ by Examples.**

**An Introduction to Join Points, Pointcuts and Advices.**

**Pointcuts and Advices in Details.**

---


---

Walter Cazzola
Aspect-Oriented Programming.
Slide 28 of 57

---

Without AspectJ.

---

With AspectJ.

---

Clear display updating module:
- all changes in single aspect;
- evolution is modular.

---

No locus of "display updating":
- evolution is cumbersome;
- changes in all classes;
- have to track & change all callers.
Aspect-Oriented Programming (AOP).

AspectJ.

Aspects Crosscut Classes.

Figure

+ moveBy(int, int)
+ makePoint(...)  
+ makeLine(...)

Point

+ getX(): int
+ getY(): int
+ moveBy(int, int)
+ setX(int)
+ setY(int)
+ moveBy(int, int)

Line

+ getF1(): Point
+ getF2(): Point
+ setF1(Point)
+ setF2(Point)
+ moveBy(int, int)

FigureElement

+ moveBy(int, int)

AspectJ.

Advice.

Advice is an additional action to take at join points.
- before: before proceeding at join point;
- after returning: after returning a value to join point;
- after throwing: after throwing a throwable to join point;
- after: after returning to join point in either ways;
- around: on arrival at join point gets explicit control over when and if program proceeds.

Pre-condition (Using the Before Advice).

aspect PointBoundsPreCondition {
  before(int newX):
    call(void Point.setX(int)) && args(newX) {
      assert(newX >= MIN_X);
      assert(newX <= MAX_X);
    }
  before(int newY):
    call(void Point.setY(int)) && args(newY) {
      assert(newY >= MIN_Y);
      assert(newY <= MAX_Y);
    }

  private void assert(boolean v) {
    if (!v) throw new RuntimeException();
  }
}
AspectJ
Contract Checking: An Example of Using Advices (Cont’d).

**Post-Condition (Using the After Advice).**

```java
aspect PointBoundsPostCondition {
  after(Point p, int newX) returning:
  call(void Point.setX(int)) && target(p) && args(newX) {
    assert(p.getX() == newX);
  }
}

private void assert(boolean v) {
  if (!v) throw new RuntimeException();
}
```

**Condition Enforcement (Using the Around Advice).**

```java
aspect PointBoundsEnforcement {
  void around(int newX):
  call(void Point.setX(int)) && args(newX) {
    proceed(clip(newX, MIN_X, MAX_X));
  }
}

target = p, args = newX
```

**Property-Based Crosscutting.**

Crosscuts of methods with a common property
- e.g., public or private methods, that returns a certain value, and so on;
- logging, debugging, profiling and so on.

Example, to log the exceptions raised by public methods.

```java
aspect PublicErrorLogging {
  Log log = new Log();
  pointcut publicInterface(): call(public * it.unimi.dico..*.*(..));
  after() throwing (Error e): publicInterface() { log.write(e); }
}
```

Consider code maintenance:
- another programmer adds a public method, i.e., extends public interface - this code will still work;
- another programmer reads this code. “what’s really going on” is explicit.

For each around advice with the signature:

```
<T> around(T1 arg1, T2 arg2, ..., Tn argn)
```

There is a special method with the signature:

```
<T> proceed(T1, T2, ..., Tn)
```

Available only in an around advice.

This means “run what would have run if this around advice had not been defined”.

Walter Cazzola
Aspect-Oriented Programming. Slide 33 of 57

Walter Cazzola
Aspect-Oriented Programming. Slide 34 of 57

Walter Cazzola
Aspect-Oriented Programming. Slide 35 of 57

Walter Cazzola
Aspect-Oriented Programming. Slide 36 of 57
AspectJ.
Wildcarding in Pointcuts.

`target(Point)`
- any type in `Point`

`targe(graphics.geom.Point)`
- any type in `graphics.geom`

`target(graphics.geom.*)`
- any type in any sub-package of `graphics`

`call(void Point.setX(int))`
- any public method on `Point`

`call(public * Point.*(..))`
- any public method on any type

`call(void Point.getX())`
- any getter

`call(void Point.getY())`
- any getter

`call(Point.new(int, int))`
- any constructor

`call(new(..))`
- any constructor

"*" is wildcard
"." is multi-part wildcard

This is wildcarding in pointcuts.

AspectJ.
Other Primitive Pointcuts.

`this("type name")`
- Any join point at which currently executing object is an instance of "type name".

`within("type name")`
- Any join point at which currently executing code is contained within "type name".

`withincode("method/constructor signature")`
- Any join point at which currently executing code is specified method or constructor.

Field reference or assignment join points.

AspectJ.
Fine-Grained Protection: Raising a Run-Time Error.

We want to ensure at run-time that any creation of figure elements goes through the factory methods.

```java
class Figure {
    public Line makeLine(Line p1, Line p2) { new Line(...); }
    public Point makePoint(int x, int y) { new Point(...); }
    ...
}
```

```java
aspect FactoryEnforcement {
    pointcut illegalNewFigElt():
        (call(Point.new(..))) || call(Line.new(..)) && !withincode(* Figure.make*(..));
    before(): illegalNewFigElt() {
        throw new Error("Use factory method instead.");
    }
}
```

AspectJ.
Fine-Grained Protection: Raising a Compile-Time Error.

We want to ensure at compile-time that any creation of figure elements goes through the factory methods.

```java
class Figure {
    public Line makeLine(Line p1, Line p2) { new Line(...); }
    public Point makePoint(int x, int y) { new Point(...); }
    ...
}
```

```java
aspect FactoryEnforcement {
    pointcut illegalNewFigElt():
        (call(Point.new(..))) || call(Line.new(..)) && !withincode(* Figure.make*(..));
    declare error: illegalNewFigElt(): "Use factory method instead.";
}
```
We want to ensure at compile-time that any change to the state of a line goes through the setter methods.

class Line implements FigureElement{
    private Point p1, p2;
    public Point getP1() { return p1; }
    public Point getP2() { return p2; }
    public void setP1(Point p1) { this.p1 = p1; }
    public void setP2(Point p2) { this.p2 = p2; }
    public void moveBy(int dx, int dy) { p1.moveBy(dx, dy); p2.moveBy(dx, dy); }
}

static aspect SetterEnforcement {
    declare error:
        set(Point Line.*) && ! withincode (void Line.set*(Point)) {
            "Use setter method.;"
    }
}

A set of statements (exploitable by the advices) to introspect the join points consistently with Java.

- `thisJoinPoint`
- `Signature getSignature()`
- `Object[] getArgs()`

aspect PointCoordinateTracing {
    before(int newVal):
        set(int Point.*) && args(newVal) {
            System.out.println("At " + thisJoinPoint.getSignature() + " field is set to " + newVal + ";");
    }
}

By using `thisJoinPoint` makes it possible for the advice to recover information about where it is running.

cflow("pointcut designator"),
cflowbelow("pointcut designator")
All join points within the dynamic control flow of (below) any join point in pointcut designator.

aspect DisplayUpdating {
    pointcut move(FigureElement figElt): target(figElt) && call(void FigureElement.moveBy(int, int)) {
        move(figElt) && !cflowbelow(move(FigureElement));
    }
    pointcut topLevelMove(FigureElement fe) : move(fe) && !cflowbelow(move(FigureElement));
    after(FigureElement fe) returning: topLevelMove(fe) { Display.update(fe); }
}
AspectJ.
Inheritance and Specialization.

Pointcuts can have more than an advice:
- a pointcut declaration couldn’t have an advice associated;
- many advices can be associated to the same named pointcut;
Abstract pointcuts can be specialized.
- aspects can have abstract pointcuts and concrete advice on them.

```java
abstract aspect Observing {
    protected interface Subject {}
    protected interface Observer {}
    public void addObserver(Subject s, Observer o) { ... }
    public void removeObserver(Subject s, Observer o) { ... }
    public static List getObservers(Subject s) { ... }
    abstract pointcut changes(Subject s); // changes
    after(Subject s): changes(s) {
        Iterator iter = getObservers(s).iterator();
        while ( iter.hasNext() ) notifyObserver(s, ((Observer)iter.next()));
    }
    abstract void notifyObserver(Subject s, Observer o);
}
```

Walter Cazzola
Aspect-Oriented Programming. Slide 45 of 57

AspectJ.
The Concrete Reuse: DisplayUpdating New Version.

```java
aspect DisplayUpdating extends Observing {
    declare parents: FigureElement implements Subject;
    declare parents: Display implements Observer;
    pointcut move(Subject s):
        target(s) &&
        (call(void FigureElement.moveBy(int, int)) ||
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int)));
    pointcut changes(Subject s):
        move(s) && !cflowbelow(move(Subject));
    void notifyObserver(Subject s, Observer o) {
        ((Display)o).update(s);
    }
}
```

Walter Cazzola
Aspect-Oriented Programming. Slide 46 of 57

AspectJ.
Example: Plug & Play Tracing.

Simple tracing
- exposes join points and uses very simple advice.
An unpluggable aspect
- core program functionality is unaffected by the aspect.

```java
class Point {
    void set(int x, int y) {
        TraceSupport.traceEntry("Point.set");
        this.x = x; this.y = y;
        TraceSupport.traceExit("Point.set");
    }
}
```

Walter Cazzola
Aspect-Oriented Programming. Slide 47 of 57

AspectJ.
Tracing without AspectJ.

```java
class TraceSupport {
    static int TRACELEVEL = 0;
    static protected PrintStream stream = null;
    static protected int callDepth = -1;
    static void init(PrintStream _s) {stream=_s;}
    static void traceEntry(String str) {
        if (TRACELEVEL == 0) return;
        callDepth++;
        printEntering(str);
    }
    static void traceExit(String str) {
        if (TRACELEVEL == 0) return;
        callDepth--;
        printExiting(str);
    }
}
```

Walter Cazzola
Aspect-Oriented Programming. Slide 48 of 57
Aspect-Oriented Programming (AOP).

AspectJ Overview.

Tracing with AspectJ.

```java
aspect PointTracing {
  pointcut trace():
    within(it.unimi.dico.*) &&
    execution(* *(..));

  before(): trace() {
    TraceSupport.traceEntry(thisJoinPoint);
  }

  after(): trace() {
    TraceSupport.traceExit(thisJoinPoint);
  }
}
```

All modules of the system use the trace facility in a consistent way:
- entering the methods and
- exiting the methods

We get the following benefits:
- turn debugging on/off without editing classes;
- debugging disabled with no run-time cost;
- can save debugging code between uses;
- can be used for profiling, logging;
- easy to be sure it is off.

Aspects in the Design

Objects are no longer responsible for using the trace facility
- trace aspect encapsulates that responsibility, for appropriate objects.
If the Trace interface changes, that change is shielded from the objects
- only the trace aspect is affected.
Removing tracing from the design is trivial
- just remove the trace aspect.

Aspects in the Code

Object code contains no calls to trace functions
- trace aspect code encapsulates those calls, for appropriate objects.
If the Trace interface changes, there is no need to modify the object classes
- only the trace aspect class needs to be modified.
Removing tracing from the application is trivial
- compile without the trace aspect class.
Is there a concern that:
- crosscuts the structure of several objects or operations;
- is beneficial to separate out

... Crosscutting.
A design concern that involves several objects or operations.
Implemented without AOP would lead to distant places in the code that:
- do the same thing, e.g., `traceEntry("Point.set")`;
- do a coordinated single thing, e.g., timing, observer pattern
  - harder to find these.

... Beneficial to Separate out.
Exactly the same questions as for objects: does it improve the code in real ways?
Separation of concerns
- e.g., think about service without timing.
Clarifies interactions, reduces tangling
- e.g., all the `traceEntry` are really the same
Easier to modify/extend
- e.g., change the implementation of tracing;
  - e.g., abstract aspect re-use.
Plug and play
- tracing aspects unplugged but not deleted.

Good modularity, even in the presence of crosscutting concerns
- less tangled code, more natural code, smaller code;
- easier maintenance and evolution
  - easier to reason about, debug, change;
- more reusable
  - more possibilities for plug and play;
  - abstract aspects;
- may lead to good implementations, measured by
  - code size, tangling, coupling and so on.
References

