Aspect Oriented Programming with AspectJ

William G. Griswold
UCSD and formerly aspectj.org

Original AspectJ slides by Gregor Kiczales (UBC) and others
“modularity”

intuitive definition:

a concern is implemented\(^1\) in a modular way if the code for the concern is:

- localized and
- has a clear interface with the rest of the system

\(^1\) coded, designed, modeled …
Traditional Metrics of PL/Programming

• Expressiveness
  – *The conciseness with which a computation can be expressed*
  – Related concepts like high-level, domain-specific
  – A property of the PL, not the human-PL relationship

• Understandability
  – *The relative ease of making sense of a program*
  – Includes human-PL relationship
  – Tends to focus on “quality” of written code, e.g., can be compromised by frequent changes or careless implementation

  https://web.media.mit.edu/~lieber/Lieberary/ZStep/Bridging/Bridging.html
HCI of Programming: “Gulfs”

• **Gulf of Execution**
  – *The difference between the intentions of the user and what the system allows them to do or how well the system supports those actions*
  – For Programming, e.g.,
    • How difficult is it to introduce this new feature into the program (in a modular way, i.e., so that future changes to it will be easy)?

• **Gulf of Evaluation**
  – *The difficulty of assessing the state of the system and how well the artifact supports the discovery and interpretation of that state*
  – For Programming, e.g.,
    • How hard is it to determine that this new feature code does what I intended?

More than “Expressiveness”

Due to Don Norman

A Concise Theory of Object-Oriented

- **Object represents a “thing”**
  - *person, car, date, …*
  - (not two things, not ½ thing)
  - Has single purpose of realizing itself

- **Object responds to messages**
  - (method calls)
  - *Things it does to itself (SRP)*
  - Other objects ask an object to do something to itself via messages

- **Objects are “opaque”**
  - Can’t see each others’ data-vars
  - Messages (calls) are only way to get things done
Consider developing...

a simple drawing application (JHotDraw)
Intuitively thinking of objects?

- Points, Lines...
- Drawing surfaces
- GUI Widgets
- ...

```
Figure
- Point
  - getX()
  - getY()
  - setX(int)
  - setY(int)
  - moveBy(int, int)

- Line
  - getP1()
  - getP2()
  - setP1(Point)
  - setP2(Point)
  - moveBy(int, int)

MoveBy(int, int)
```

Display

Figure

*FigureElement

moveBy(int, int)
What is OOP?

- a learned *intuitive way of thinking*
- design concepts
  - objects, classification hierarchies
- supporting language mechanisms
  - classes, encapsulation, polymorphism…
- allows us to
  - make code look like the design
  - improves design and code modularity

- captures concerns a *function* can’t hold
  - concern = design decision

many other benefits build on these
But some concerns “don’t fit”

i.e., a simple Observer pattern

class Point extends ... {
    private int x = 0, y = 0;

    intgetX() { return x; }
    intgetY() { return y; }

    voidsetX(int x) {
        this.x = x;
        display.update(this);
    }
    voidsetY(int y) {
        this.y = y;
        display.update(this);
    }
}

fair design modularity but poor code modularity
With AOP they do fit

good design modularity
good code modularity

aspect ObserverPattern {

    private Display FigureElement.display;

    pointcut change():
        call(void figures.Point.setX(int))
        || call(void Point.setY(int))
        || call(void Line.setP1(Point))
        || call(void Line.setP2(Point))
        || call(void Shape.moveBy(int, int));

    after(FigureElement s) returning:
        change() && target(s) {
            s.display.update();
        }
}
Ask yourself: could you name a single class “ObserverPattern” in Java?
What is AOP?

- a learned *intuitive way of thinking*
- design concepts
  - aspects, crosscutting structure
- supporting language mechanisms
  - join points, pointcuts, advice…
- allows us to
  - make code look like the design
  - improve design and code modularity
- captures concerns a *class* can’t hold
Thinking OO, We \textit{Intuitively} See

- **Figure**
  - FigureElement
    - makePoint(..)
    - makeLine(..)
    - \textit{moveBy}(int, int)

- **Point**
  - \textit{getX}()
  - \textit{getY}()
  - \textit{setX}(int)
  - \textit{setY}(int)
  - \textit{moveBy}(int, int)

- **Line**
  - \textit{getP1}()
  - \textit{getP2}()
  - \textit{setP1}((Point)
  - \textit{setP2}((Point)
  - \textit{moveBy}(int, int)
Thinking OO, We *Intuitively* See

- **Figure**: `*` FigureElement
  - `makePoint(..)
  - `makeLine(..)
  - `moveBy(int, int)`

- **Point**
  - `getX()
  - `getY()
  - `setX(int)
  - `setY(int)
  - `moveBy(int, int)`

- **Line**
  - `getP1()
  - `getP2()
  - `setP1(Point)
  - `setP2(Point)
  - `moveBy(int, int)`

*factory methods*

*operations that change shapes*
AOP Developers Intuitively See…

- **Figure**
  - `*` to **FigureElement**
    - `makePoint(..)`
    - `makeLine(..)`
    - `moveBy(int, int)`

- **Point**
  - `getX()`
  - `getY()`
  - `setX(int)`
  - `setY(int)`
  - `moveBy(int, int)`

- **Line**
  - `getP1()`
  - `getP2()`
  - `setP1(Point)`
  - `setP2(Point)`
  - `moveBy(int, int)`

- **DisplayUpdating**
- **FactoryEnforcement**
- **Grouping**
- **BoundsChecking**
Dynamic (*runtime* crosscutting)

“join points” — natural events in an OO execution
join points

several kinds of join points
- method & constructor call
- method & constructor execution
- field get & set
- exception handler execution
- static & dynamic initialization

key points in dynamic call graph

suppose l.moveBy(2, 2)

note that the scope of call vs. execution joinpoints is different (caller vs. callee)
join point terminology

key points in dynamic call graph

all join points on this slide are within the control flow of this join point
**pointcuts**

“a means of identifying join points”

A *pointcut* is a set membership 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

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

matches method *call* join points that have this *type signature*
pointcut composition

pointcuts compose as set predicates, using &&, || and !

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

a “void Line.setP1(Point)” call

or

a “void Line.setP2(Point)” call

whenever a Line receives a “void setP1(Point)” or “void setP2(Point)” method call
user-defined pointcuts
defined using the pointcut construct

user-defined (aka named) pointcuts
  – can be used in the same way as primitive pointcuts

pointcut move():
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point));
**after “advice”**

action to take after computation under join points

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

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

DisplayUpdating v1

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();
  }
}
```

box means complete, running code
a multi-class aspect

DisplayUpdating v2

can cut across multiple classes, and use interface signatures

aspect DisplayUpdating {

  pointcut move():
    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));

  after() returning: move() {
    Display.update();
  }
}
without AspectJ

DisplayUpdating v2

• no locus of “display updating”
  – evolution is cumbersome
  – changes in all classes
  – have to track & change all callers
  – we say the concerns are “tangled”

```java
class Point {
    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();
    }
}

class Line {
    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();
    }
}
```
with AspectJ

DisplayUpdating v2

class Line {
    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;
    }
}

class Point {
    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;
    }
}

aspect DisplayUpdating {
    pointcut move():
        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));

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

- clear display-updating module
  - all changes in single aspect
  - evolution is modular
aspects crosscut classes

aspect modularity cuts across class modularity

Display

<table>
<thead>
<tr>
<th>Figure</th>
<th>*</th>
<th>FigureElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>makePoint(..)</td>
<td></td>
<td>moveBy(int, int)</td>
</tr>
</tbody>
</table>

Point

<table>
<thead>
<tr>
<th>Line</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>makePoint(..)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Point</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>makePoint()</td>
<td></td>
</tr>
<tr>
<td>makeLine()</td>
<td></td>
</tr>
<tr>
<td>getX()</td>
<td>getP1()</td>
</tr>
<tr>
<td>getY()</td>
<td>getP2()</td>
</tr>
<tr>
<td>setX(int)</td>
<td>setP1(Point)</td>
</tr>
<tr>
<td>setY(int)</td>
<td>setP2(Point)</td>
</tr>
<tr>
<td>moveBy(int, int)</td>
<td>moveBy(int, int)</td>
</tr>
</tbody>
</table>
aspect using *target* of call

```java
aspect DisplayUpdating v3 {

  pointcut move(FigureElement fe):
    target(fe) &&
    (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)));

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

can expose and use values at join points
only top-level moves

aspect DisplayUpdating {

    pointcut move(FigureElement fe):
        target(fe) &&
        (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 topLevelMove(FigureElement fe):
        move(fe) && !cflowbelow(move(FigureElement));

    after(FigureElement fe) returning: topLevelMove(fe) {
        Display.update(fe);
    }
}
abstract aspect Observing {

    protected interface Subject { }
protected interface Observer { }

public void addObserver(Subject s, Observer o) { ... }
public void removeObserver(Subject s, Observer o) { ... }

abstract pointcut changes(Subject s);

after(Subject s): changes(s) {
    Iterator iter = getObservers(s).iterator();
    for ( Observer obs: getObservers(s) ) {
        notifyObserver(s, obs);
    }
}
abstract void notifyObserver(Subject s, Observer o);
}
abstract aspects for reuse

DisplayUpdating v5

aspect DisplayUpdating extends Observing {

    declare parents: FigureElement implements Subject;
    declare parents: Display implements Observer;

    pointcut changes(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)));

    void notifyObserver(Subject s, Observer o) {
        ((Display)o).update(s);
    }
}
property-based crosscutting

DisplayUpdating v6

aspect DisplayUpdating extends Observing {

    declare parents: FigureElement implements Subject;
    declare parents: Display implements Observer;

    pointcut changes(Subject s):
        target(s) &&
        (call(void FigureElement.moveBy(int,int)) ||
         call(void FigureElement+.set*(..)));

    void notifyObserver(Subject s, Observer o) {
        ((Display)o).update(s);
    }
}

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

neatly captures “set” interface
enforcing design invariants

```java
aspect FactoryEnforcement {

    pointcut newFigElt():
        call(FigureElement.new(..));

    pointcut inFactory():
        within(* Figure.make*(..));

    pointcut illegalNewFigElt():
        newFigElt() && !inFactory();

    declare error: illegalNewFigElt():
        "Must call factory method to create figure elements.";
}
```
AspectJ is now often programmed w/ Annotations

```java
package com.javatpoint;

public class Operation{
    public void msg(){System.out.println("msg method invoked");}
    public int m(){System.out.println("m method invoked"); return 2;}
    public int k(){System.out.println("k method invoked"); return 3;}
}

public class Test{
    public static void main(String[] args){
        ApplicationContext context = new Class
        Operation e = (Operation) context.getBean
        System.out.println("calling msg...");
        e.msg();
        System.out.println("calling m...");
        e.m();
        System.out.println("calling k...");
        e.k();
    }
}

@$Aspect
public class TrackOperation{
    @Pointcut("execution(* Operation.*(..))")
    public void k(){//pointcut name
        @Before("k()")//applying pointcut on before advice
        public void myadvice(JoinPoint jp)//it is advice (before advice)
        {
            System.out.println("additional concern");
            //System.out.println("Method Signature: " + jp.getSignature());
        }
    }
}
```

https://www.javatpoint.com/spring-aop-aspectj-annotation-example
AspectJ is a language, compiler, and environment:
• A small, compatible extension to Java
  – join point model: points in execution, class members
  – JP identification: pointcuts, type patterns
  – semantics at JP's: advice, define members
• Compiles to pure JVM bytecodes
• IDE conveys crosscutting
AOP Summary

- A feature/capability often crosscuts traditional OO modularity
  - distribution of feature code across components
  - high gulfs of execution and evaluation

- AOP enables the code to look like how the programmer thinks about it
  - more modular features
  - lower gulfs of execution/evaluation compared to OO

- IDEs provide PL-like features beyond the PL that further close the gulfs
  - Where might my aspect execute?
  - (What methods might get executed at this call site?)
BACKUP SLIDES
Some Key Tools

- **AspectJ/AspectWerkz**
  - mature
  - IDE support
  - documented, supported
  - *de facto* standard
  - Annotation-style aspect programming

- **Spring**
  - “interceptor-based” AOP; supports AspectJ

- **JBoss/WildFly**
  - “interceptor-based” AOP
aspect DisplayUpdating {

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

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

Example Java elsewhere in program:
FigureElement n1;
Line n2;
n1.setP1(newP1);
n1.setColor(RED);
n1.setP2(newP2);
n2.setDepth(2);
...

Gets compiled to byte codes as:
n1.setP1(newP1);
if (n1.instanceOf(Line))
    Display.update();
n1.setColor(RED);
n1.setP2(newP2);
Display.update();
n2.setDepth(2);
...

If there is a subclass of Line that defines its own setP2(Point), then would have to check type