AspectJ and Aspect-Oriented Programming

Macneil Shonle
mshonle@cs.ucsd.edu

Some slides courtesy the AspectJ group

Semantics of Scoping

- Static scope:
  - looks up the syntax tree at compile time
- Dynamic scope:
  - looks up the dynamic call stack at runtime

Choosy PL Designers Choose Static Scoping

- Each scoping rule seems like it might be more useful than the other for special cases.
- Why would some people believe that static scope is better in the general case? ...
**Choosy PL Designers Choose Static Scoping (continued)**

- Static scope allows for better *Separation of Concerns*; fewer strange bugs would occur
- Strongly supporting *Separation of Concerns* drives programming language design

---

**Overview**

- Separation of Concerns in O-O PLs
- Limits of O-O Modularization
- Aspects: Advanced Modularization
  - Terminology
- The AspectJ Language
  - Semantics of Advice (dynamic)
  - Open Classes (static)

---

**Separation of Concerns in PLs**

- Selection concerns are handled quite nicely in OOP languages with polymorphism

```
if (w.getType() == A)
    drawA(w);
else if (w.getType() == B)
    drawB(w);
else if (w.getType() == C)
    drawC(w);
... 
```

---

**Separation of Concerns in PLs**

- Usually, when SOC is achieved in a PL, we say that the concern is modularized
- Exception handling is another good example of separation of concerns: it creates another channel for error-specific information to flow:

```
Data-flow/Control-flow  method calls, control and iteration constructs
```

```
Error-flow/Error-control  catch blocks, throw
```
Error Handling before Exceptions

- In the C-style of error handling, functions returned error codes that needed to be checked.
- Error handling (identification of errors, accessing data that describes the error, and changing control-flow) was less modular.

```c
err = f(result);
if (err) {
    return err;
} else {
    err = g(result);
    if (err) {
        return err;
    }
}
```

versus

```c
err = f();
```

Modularization

Nicely separated concerns:
- XML Parsing
- Pattern matching
- Collections
- Numerical processing

Language features allow for very nice separation

Limits of Modularization

Some concerns crosscut modular boundaries:
- Caching
- Object marshalling
- Logging
- Thread synchronization

Crosscutting concerns

- A concern crosscuts when it cannot be separated into its own module
- Crosscutting manifests itself in two ways:
  - A concern is tangled if it is mixed with code that addresses a separate concern
  - A concern is scattered if it appears in more than one module.
The Goal of Aspect-Oriented Programming

- Write concerns in local aspects, and have them woven globally into the program by a weaver

The Aspect

- An aspect is a special kind of module that specifically addresses crosscutting concerns
- Aspects encapsulate (and, to an extent, modularize) crosscutting concerns
- The aspect can be woven into the program either at compile-time or run-time

- Red text are synchronization concerns
- **Green text** are Remote Method Invocation concerns
- In this Java program, the concerns are tangled

D, An Early Aspect Language (Crista Lopes, 1995)

Instead of writing this

Concerns are now separated

D, An Early Aspect Language (Crista Lopes, 1995)
### From D to AspectJ

- D is a domain-specific aspect language: it only handles synchronization and marshalling concerns
- D was the inspiration of AspectJ
- AspectJ is an extension to Java and is a general purpose aspect language

### AspectJ Features

- **Advice (The Join-Point Model)**
  - Dynamic
  - Concerned with identifying events that occur when the program is running, and changing or adding to its behavior
- **Open Classes (Intertype Declarations)**
  - Static
  - Adds additional methods or fields to classes, in a way that achieves finer-grained access than just public, private, protected.

### The Join-Point Model

- A **join-point** is a dynamic event that occurs when the program is running

- Example events:
  - A method or constructor is called
  - A method or constructor is executed
  - A field get or set operation
  - An exception handler is entered

### Pointcuts

- A **pointcut** is a set of join-points
- Usual set-theory operations (&&, ||, !)
- Special predicates

- Examples:
  - `call(void Line.setP1(Point))`
  - `call(void Line.setP1(Point)) ⊕ call(void Line.setP2(Point))`
  - `call(void Line.set*(Point))`
Advice
n A pointcut cut can have extra behavior attached to it: i.e., when one of the join-points occurs, execute additional code.

n We call this additional code advice.

```java
aspect DisplayUpdating {
    pointcut move():
        call(void Line.set*(Point));
    after() returning: move() {
        advice
        Display.update();
    }
}
```

Kinds of advice
n before
  a before a join-point occurs, execute the advice body

n after
  a after a join-point occurs, execute the advice body

n around
  a execute the advice body instead of join-point
  a optionally execute join-point when "proceed()" is called

Example: Hello World of AspectJ

```java
aspect HelloLoggingWorld {
    pointcut publicMethods();
        execution(public * mypackage..*(..));
    Object around(): publicMethods() {
        System.out.println("Entering " + thisJoinPointStaticPart);
        Object o = proceed();
        System.out.println("Leaving " + thisJoinPointStaticPart + " returning " + o);
        return o;
    }
}
```

Figure Elements
- Display
- FigureElement
- Line
- Point
Example: Getting Context Information and Parameterizing Advice with it

```java
aspect DisplayUpdating {
    pointcut move(FigureElement figElt):
        target(figElt) &&
        (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);
    }
}
```

Problem: Redundant Display.update Calls

```java
aspect DisplayUpdating {
    pointcut move(FigureElement figElt):
        target(figElt) &&
        (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(fe));
    after(FigureElement fe) returning: topLevelMove(fe) {
        Display.update(fe);
    }
}
```

Problem: Redundant Display.update Calls

```java
aspect DisplayUpdating {
    pointcut move(FigureElement figElt):
        target(figElt) &&
        (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(fe));
    after(FigureElement fe) returning: topLevelMove(fe) {
        Display.update(fe);
    }
}
```

---

Problem: Redundant Display.update Calls

```java
aspect DisplayUpdating {
    pointcut move(FigureElement figElt):
        target(figElt) &&
        (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(fe));
    after(FigureElement fe) returning: topLevelMove(fe) {
        Display.update(fe);
    }
}
```

---

**cflow: Representing Control Flow** (i.e., the dynamic call stack)

- `cflow(Pointcut)`
  - all join points in the dynamic control flow of any
    join point picked out by `Pointcut`
  - basically, if a join point is on the stack, you're in
    that join point's control flow

- `cflowbelow(Pointcut)`
  - all join points in the dynamic control flow below
    any join point picked out by `Pointcut`

---

**Using cflow to Avoid Redundant Display.update Calls**

```java
aspect DisplayUpdating {
    pointcut move(FigureElement figElt):
        target(figElt) &&
        (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(fe));
    after(FigureElement fe) returning: topLevelMove(fe) {
        Display.update(fe);
    }
}
```
Example: Caching

```java
aspect PointCaching {
    private MyLookupTable cache = new MyLookupTable();

    Point around(int x, int y) {
        call(Point.new(int, int)) && args(x, y) {
            Point ret = cache.lookup(x, y);
            if (ret == null) {
                ret = proceed(x, y);
                cache.add(x, y, ret);
            }
            return ret;
        }
    }
}
```

Context-passing: The Wormhole Pattern

Workers need to know the caller:
- capabilities
- charge backs
- to customize result

This is very similar to what dynamic scope provides!

Simulating Dynamic Scope

Workers need to know the caller:
- capabilities
- charge backs
- to customize result

Simulating Dynamic Scope

```java
pointcut invocations(Caller c) {
    this(c) && call(void Service.doService(String));
}
```
Simulating Dynamic Scope

abstract aspect CapabilityChecking {
    pointcut invocations(Caller c):
        this(c) && call(void Service.doService(String));
    pointcut workPoints(Worker w):
        target(w) && call(void Worker.doTask(Task));
    pointcut perCallerWork(Caller c, Worker w):
        cflow(invocations(c)) && workPoints(w);
    before (Caller c, Worker w) : perCallerWork(c, w) {
        w.checkCapabilities(c);
    }
}

Intertype Declarations

n Provides for open classes:

aspect Rendering {
    public void Particle.draw(Graphics g) { ... }
    public void Wall.draw(Graphics g) { ... }
    public void Vector.draw(Graphics g) { ... }
}
Concern-Private Data

Suppose Bar and Czar need to have some access to data that belongs to BaseClass, but Foo should be excluded from touching it.

Aspect Private Intertype Declarations

Private here means only the aspect can access it, the classes can't touch it

```java
aspect SpecialField {
  private BaseClass.MyField field;
  public void BaseClass.process() {
    // can access myField
  }
  public void Bar.process() { // can access myField
  }
  public void Czar. ...
}
```

Where to find AspectJ

http://eclipse.org/aspectj/

(Or just Google “aspectj”)