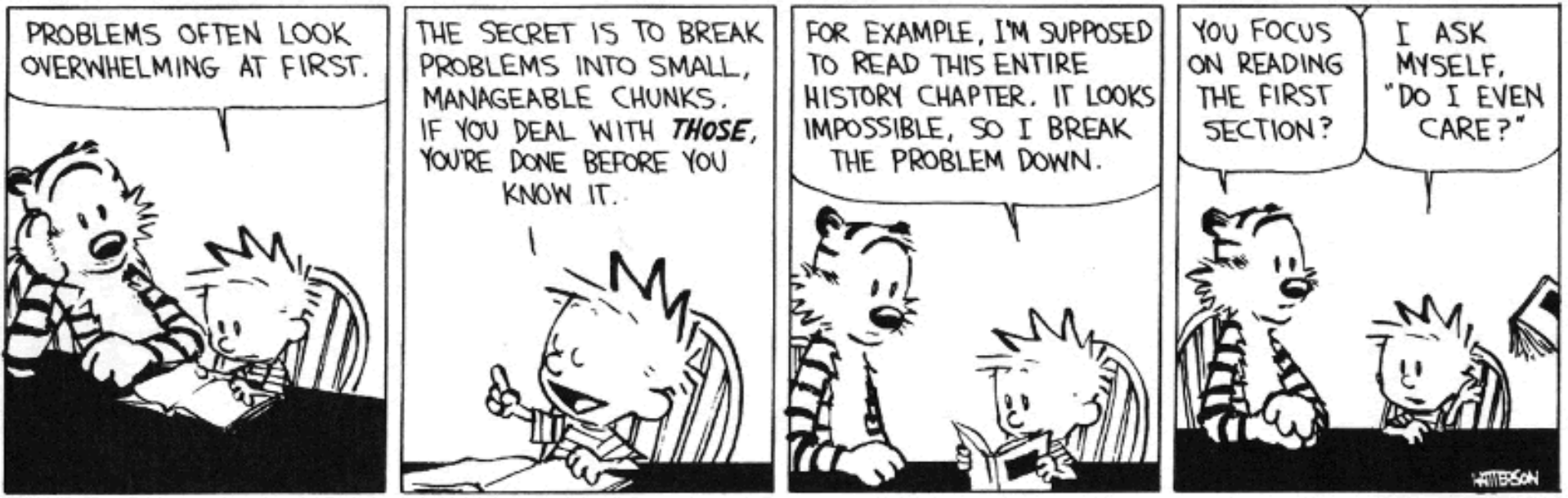
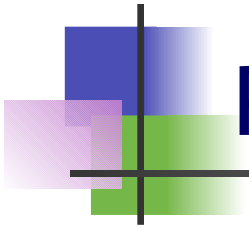


Aspect-Oriented Programming

Wes Weimer

(slides adapted from Dave Matuszek, UPenn)



Programming paradigms

- Procedural (or imperative) programming
  - Executing a set of commands in a given sequence
  - `Fortran, C, Cobol`
- Functional programming
  - Evaluating a function defined in terms of other functions
  - `Scheme, Lisp, ML, OCaml`
- Logic programming
  - Proving a theorem by finding values for the free variables
  - `Prolog`
- Object-oriented programming (OOP)
  - Organizing a set of objects, each with its own set of responsibilities
  - `Smalltalk, Java, Ruby, C++` (to some extent)
- Aspect-oriented programming (AOP) =
  - aka Aspect-Oriented Software Design (AOSD)
  - Executing code whenever a program shows certain behaviors
  - `AspectJ` (a Java extension), `Aspect#`, `AspectC++`, …
  - Does not *replace* O-O programming, but rather *complements* it

Why Learn Aspect-Oriented Design?

- Pragmatics – Google stats (Apr '09):
  - “guitar hero” 36.8 million
  - “object-oriented” 11.2 million
  - “cobol” 6.6 million
  - “design patterns” 3.0 million
  - “extreme programming” 1.0 million
  - “functional programming” 0.8 million
  - “aspect-oriented” *or* “AOSD” 0.3 million
- But it’s growing
  - Just like OOP was years ago
  - Especially in the Java / Eclipse / JBoss world

Motivation By Allegory

- Imagine that you're the ruler of a fantasy monarchy

Motivation By Allegory (2)

- You announce Wedding 1.0, but must increase security

Motivation By Allegory (3)

- You must make changes everywhere: close the secret door

Motivation By Allegory (4)

- … form a brute squad …

Motivation By Allegory (5)

- … clear the Thieves' Forest …

Motivation By Allegory (6)

- … reduce the number of gate keys to 1 …

Motivation By Allegory (7)

- … kill your rival …

Motivation By Allegory (8)

- … double the guards at the gate …

Motivation By Allegory (9)

- … secure the castle hallways …

Motivation By Allegory (10)

- … even reduce the length of the Wedding itself …

Motivation By Allegory (11)

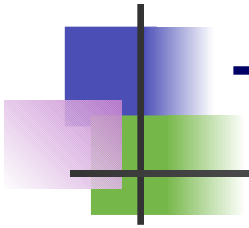
- … you're swamped – you're not happy!

Motivation By Allegory (12)

- It’d be nice to separately advise: “Increase Security”

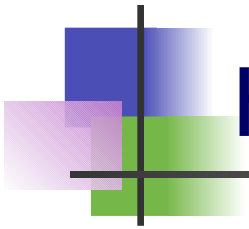
Motivation By Allegory (13)

- Then you'd be a happy monarch!



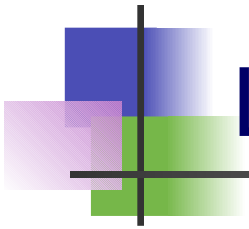
The problem

- Some programming tasks cannot be neatly encapsulated in objects, but must be scattered throughout the code
- Examples:
  - Logging (tracking program behavior to a file)
  - Profiling (determining where a program spends its time)
  - Tracing (determining what methods are called when)
  - Session tracking, session expiration
  - Special security management
  - Error-checking or –handling
- The result is crosscutting code -- the necessary code "cuts across" many different classes and methods



High-Level AOP Goals

- You want to maintain different concerns separately
  - Business logic here
  - Tracing there
  - Security somewhere else
- And yet somehow weave them together to form one unified program that you can run
- Specify rules for integrating them together



Lecture Goals

- What Is Aspect-Oriented Programming
- When Should You Use It
- What Are Join Points
- What Are Pointcuts
- Where Can You Get More Information

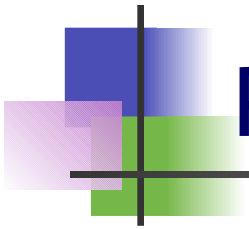
Example – Adding Tracing

```
class Fraction {
        int numerator;
        int denominator;
        ...
        public Fraction multiply(Fraction that) {
            traceEnter("multiply", new Object[] {that});
            Fraction result = new Fraction(
                this.numerator * that.numerator,
                this.denominator * that.denominator);
            result = result.reduceToLowestTerms();
            traceExit("multiply", result);
            return result;
        }
        ...
    }
```

- Now imagine similar code in *every method* you might want to trace

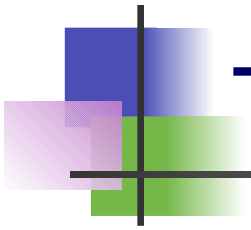
Consequences of crosscutting code

- 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 change
  - Have to find all the code involved...
  - ...and be sure to change it consistently
  - ...and be sure not to break it by accident
- Inefficient when crosscutting code is not needed



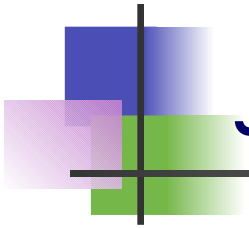
Popular AOP System: AspectJ™

- AspectJ is a small, well-integrated extension to Java
  - Based on the 1997 PhD thesis by Christina Lopes, *D: A Language Framework for Distributed Programming*
  - Widely championed by Gregor Kiczales et al.
- AspectJ "modularizes crosscutting concerns"
  - That is, code for one *aspect* of the program (such as tracing) is collected together in one place
- The AspectJ compiler is free and open source
- AspectJ works with JBuilder, Forté, Eclipse, JBoss, probably others
- Best online writeup: http://www.eclipse.org/aspectj/
  - Parts of this lecture were taken from the above paper



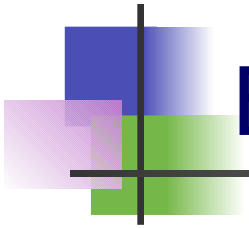
Terminology

- A join point is a well-defined point in the program flow
  - e.g., “when something calls foo()”
- A pointcut is a group of join points
  - e.g., “every call to foo() in Bar.java”
- Advice is code that is executed at a pointcut
  - e.g., “add in this Tracing code”
- Introduction modifies the members of a class and the relationships between classes
- An aspect is a module for handling crosscutting concerns
  - Aspects are defined in terms of pointcuts, advice, and introduction
  - Aspects are reusable and inheritable
- Each of these terms will be discussed in greater detail



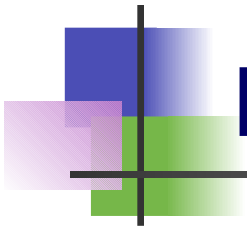
Join points

- A join point is a well-defined point in the program flow
  - Used to specify how to integrate aspects of your program
  - We want to execute some code ("advice") each time a join point is reached
  - We do *not* want to clutter up the code with explicit indicators saying *"This is a join point"*
  - AspectJ provides a syntax for indicating these join points "from outside" the actual code (but this is somewhat illusory)
- A join point is a point in the program flow "where something happens"
  - When a method is called
  - When an exception is thrown
  - When a variable is accessed (and more)



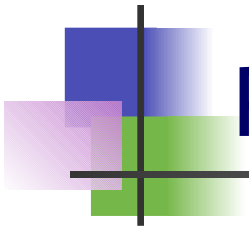
Example designators

- When a particular method body executes:
  - execution(void Point.setX(int))
- When a method is called:
  - call(void Point.setX(int))
- When an exception handler executes:
  - handler(ArrayOutOfBoundsException)
- When the object currently executing (i.e. this) is of type SomeType:
  - this(SomeType)
- When the target object is of type SomeType
  - target(SomeType)
- When the executing code belongs to class MyClass
  - within(MyClass)



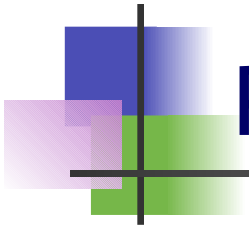
Example 1: Let's Add Tracing

- A pointcut named move that chooses various method calls:
  - pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));
- Advice (code) that runs before (or after) the move pointcut:
  - before(): move() {
    System.out.println("About to move");
    }



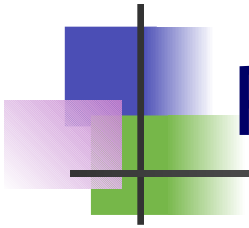
Pointcut designator wildcards

- It is possible to use wildcards to declare pointcuts:
  - execution(* *(..))
    - Chooses the execution of any method regardless of return or parameter types
  - call(* set(..))
    - Chooses the call to any method named set regardless of return or parameter type
    - In case of overloading there may be more than one such set method; this pointcut picks out calls to all of them



Pointcut designators based on types

- You can select elements based on types. For example,
  - execution(int *())
    - Chooses the execution of any method with no parameters that returns an int
  - call(* setY(long))
    - Chooses the call to any setY method that takes a long as an argument, regardless of return type or declaring type
  - call(* Point.setY(int))
    - Chooses the call to any of Point's setY methods that take an int as an argument, regardless of return type
  - call(*.new(int, int))
    - Chooses the call to any classes' constructor, so long as it takes exactly two ints as arguments

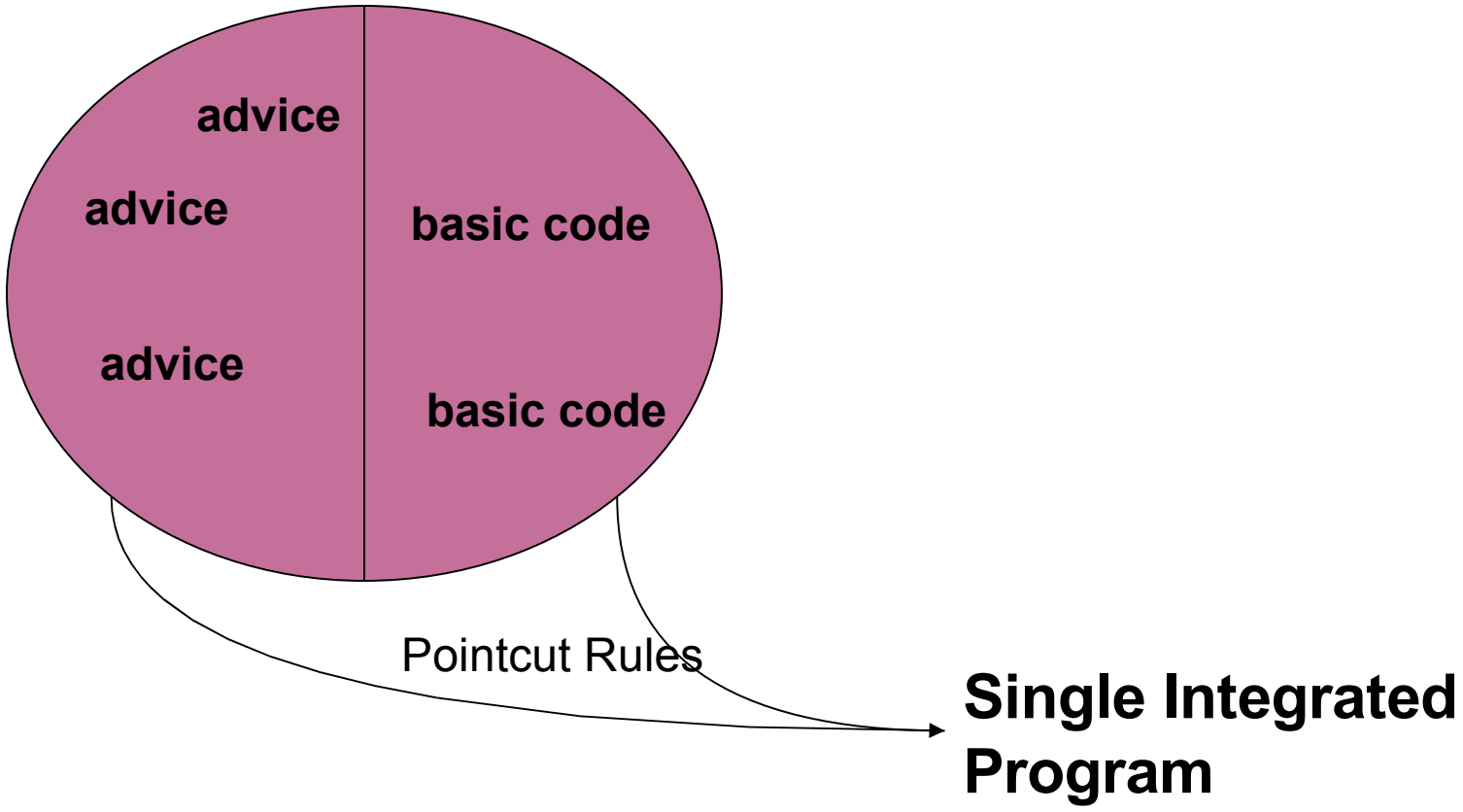


Pointcut designator composition

- Pointcuts compose through the operations **or** ("||"), **and** ("&&") and **not** ("!")
- Examples:
  - target(Point) && call(int *())
    - Chooses any call to an int method with no arguments on an instance of Point, regardless of its name
  - call(* *(..)) && (within(Line) || within(Point))
    - Chooses any call to any method where the call is made from the code in Point's or Line's type declaration
  - within(Line) && execution(*.new(int))
    - Chooses the execution of any constructor taking exactly one int argument, so long as it is inside Line
  - !this(Point) && call(int *(..))
    - Chooses any method call to an int method when the executing object is any type except Point

A Faulty Mental Model

- Many imagine that AOP works like this:



A Problem

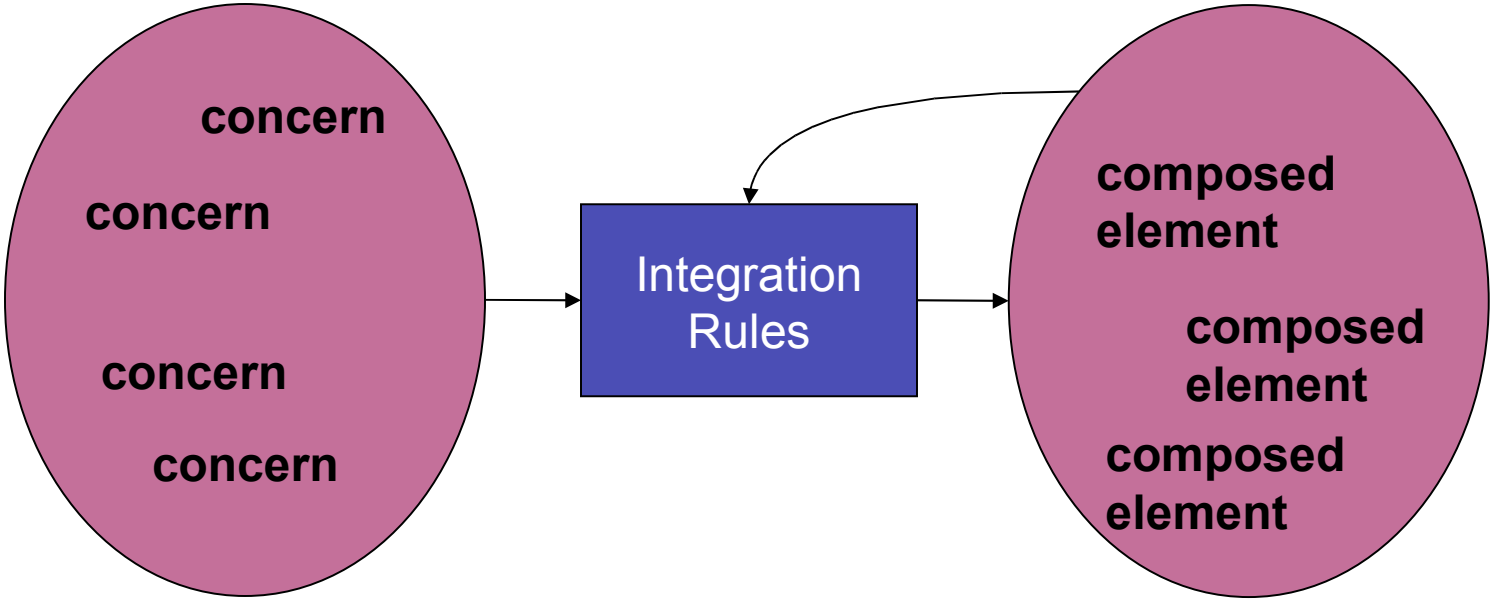
- Consider this Logger:

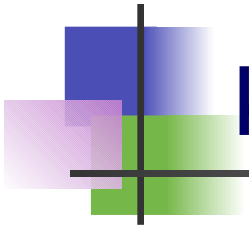
```
aspect Logger {
   before(): call (* *.*(..)) {
        System.out.println("call to " + thisJoinPoint);
   }
}
```

- What might go wrong?

A Better Mental Model

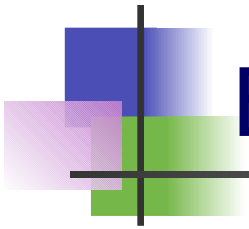
- This idea won't lead you as far astray:





Kinds of advice

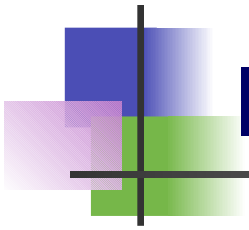
- AspectJ has several kinds of advice; here are some of them:
  - Advice is just like your normal code (cf. AspectWerkz, AspectJ 5)
  - Before advice runs as a join point is reached, before the join point executes
  - After advice on a join point runs after that join point executes:
    - after returning advice is executed after a method returns normally
    - after throwing advice is executed after a method returns by throwing an exception
    - after advice is executed after a method returns, regardless of whether it returns normally or by throwing an exception
  - Around advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point



Example 2: With Parameters

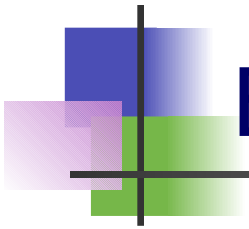
- You can access the context of the join point:
- pointcut setXY(FigureElement fe, int x, int y):
  call(void FigureElement.setXY(int, int))
  && target(fe)
  && args(x, y);

- after(FigureElt fe, int x, int y) returning: setXY(fe, x, y) {
  println(**fe** + " moved to (" + **x** + ", " + **y** + ").");
  }



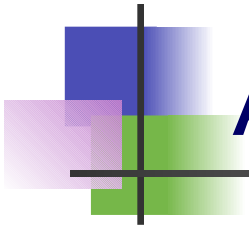
Introductions

- An introduction is a member of an aspect, but it defines or modifies a member of another type (class). With introduction we can
  - add methods to an existing class
  - add fields to an existing class
  - extend an existing class with another
  - implement an interface in an existing class
  - convert checked exceptions into unchecked exceptions



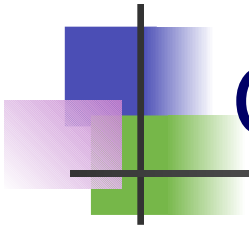
Example introduction

```
aspect CloneablePoint {

   declare parents: Point implements Cloneable;

   declare soft: CloneNotSupportedException:
                              execution(Object clone());

   Object Point.clone() { return super.clone(); }
}
```



AOP Challenges

- It's not all wine and roses
- Debugging is a problem
  - You debug the integrated ("weaved") program – but that doesn't correspond to any particular piece of source
  - Like debugging C++ with macros and templates
- Aspects may depend on each other or themselves
  - This is difficult to reason about
  - What integrated code is really being produced?



Concluding remarks

- Aspect-oriented programming (AOP) is a new paradigm -- a new way to think about programming
- It acknowledges that crosscutting concerns come up in practice
- It provides a way to maintain concerns separately and specify integration rules to weave them together
- AOP is somewhat similar to event handling, where the “events” are defined outside the code itself
- AspectJ is not itself a complete programming language, but an adjunct to Java
- AspectJ does not add new capabilities to what Java can do, but adds new ways of modularizing the code
- Like all new technologies, AOP may--or may not--catch on in a big way

And They Lived Happily Ever After

- You may be skeptical. Any questions?