Programming Languages and Concepts

Siegfried Benkner

Research Group Scientific Computing

Universität Wien

Contents

- **Aspect-Oriented Programming**
- **What is AOP?**
- **Benefits of AOP**
- **Aspect J – Overview**
- **AOP and related approaches**

Literature

- A.Colyer, A.Clement, G. Harley, M.Webster.

  Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison-Wesley, 2004.

- Ramnivas Laddad.

  AspectJ in Action, Second Edition, Manning Publications, 2009.

- **AspectJ/Eclipse Documentation**: http://www.eclipse.org/aspectj/docs.php

Aspect Oriented Programming

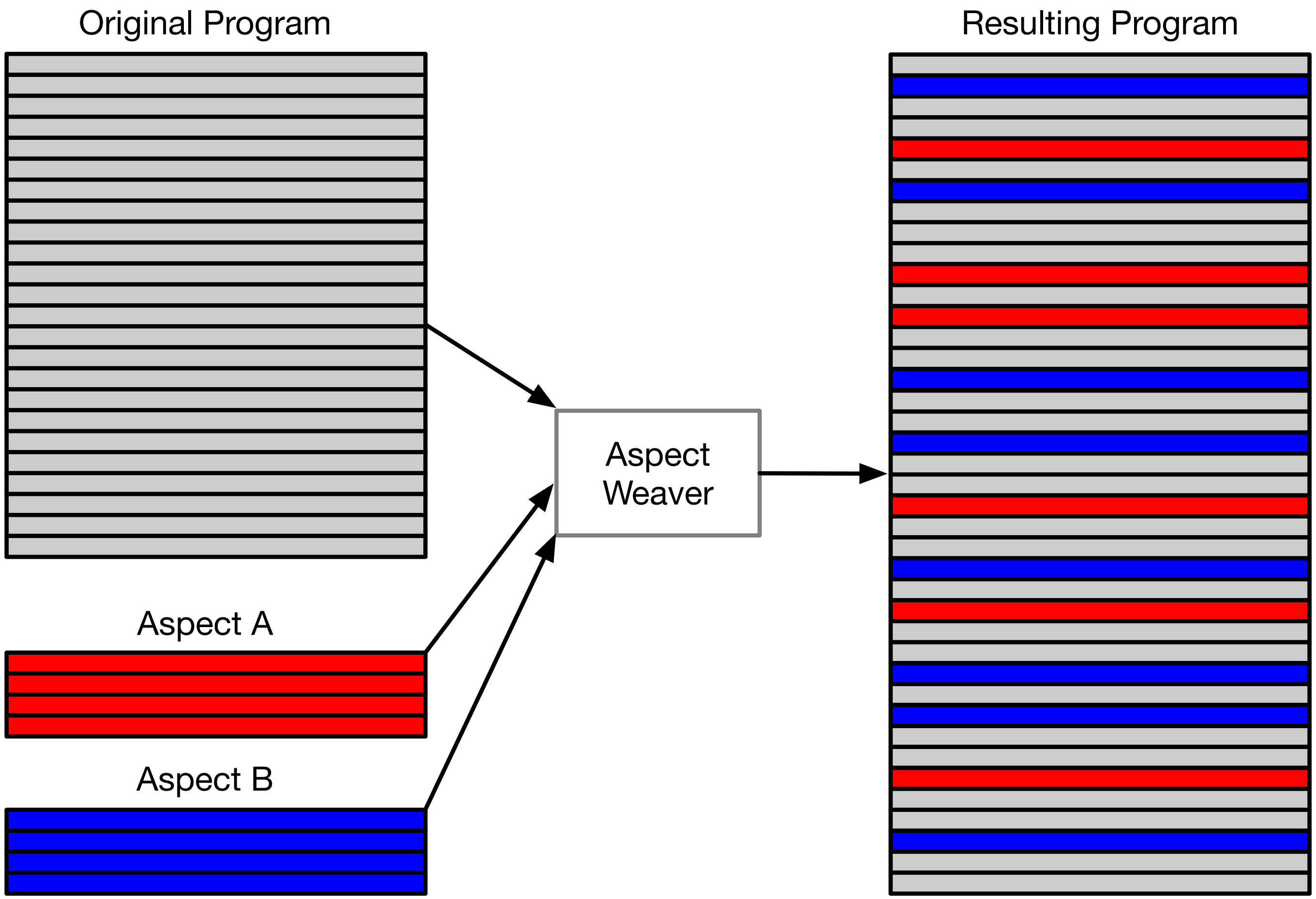
- Aspect-oriented Programming (AOP) is a programming paradigm that aims to increase modularity and improve code design and quality based on separation of cross-cutting concerns.

- AOP allows adding additional behavior (i.e. new code) to an existing program without modifying the program itself.

- The term aspect-oriented programming was coined in 1997.

Aspect Oriented Programming

- AOP allows behaviors that are not central to the business logic (e.g., logging) to be added to a program without cluttering the code.



Aspect Oriented Programming

- Separates different concerns of an object into their own aspects.
- AOP helps to remove
  - infrastructure code and
  - replicated code that is scattered across an entire system.
- Used to apply cross-cutting concerns uniformly to object-oriented code
  - Logging, security, transaction management, …

Aspect Oriented Programming

- **Key contributions**
  - Modular cross-cutting structure
  - Non-hierarchical decomposition
  - Compositional cross-cutting

Bad Modularity

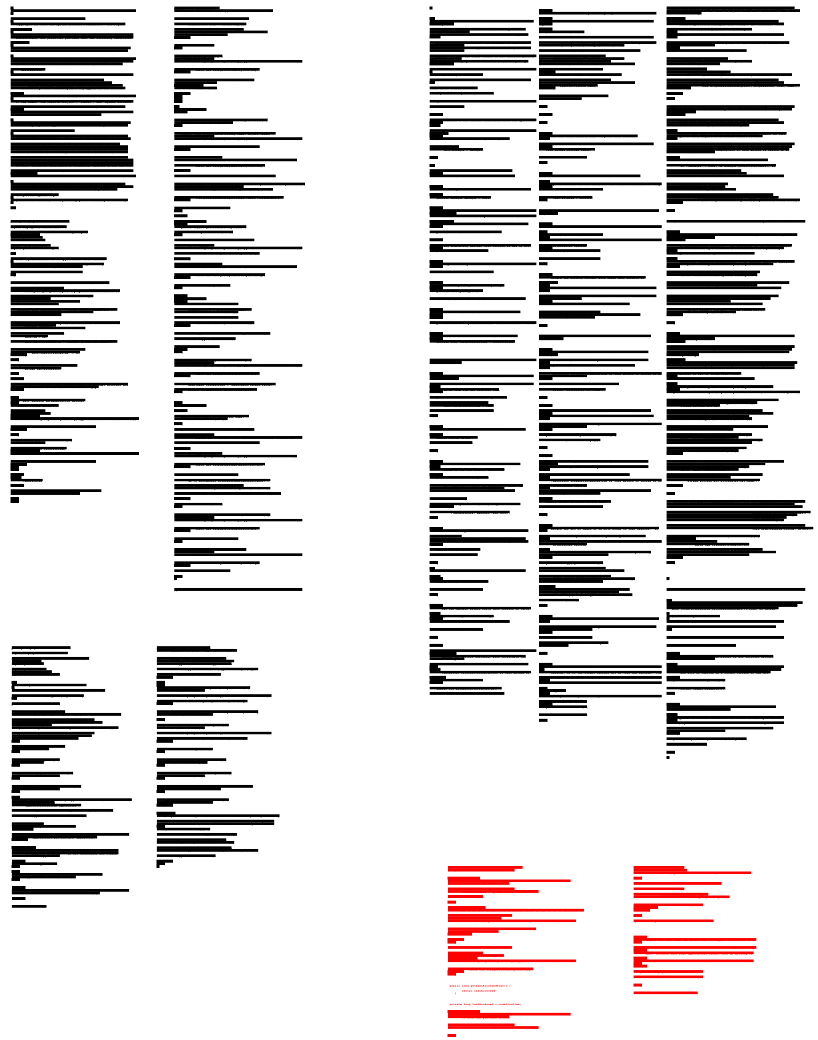
- Scattering – code addressing a certain aspect spread around.
- Tangling – code in one region addresses multiple concerns.
- Scattering and tangling (S&T) tend to appear together; they describe different facets of the same problem.

The cost of scattered/tangled code

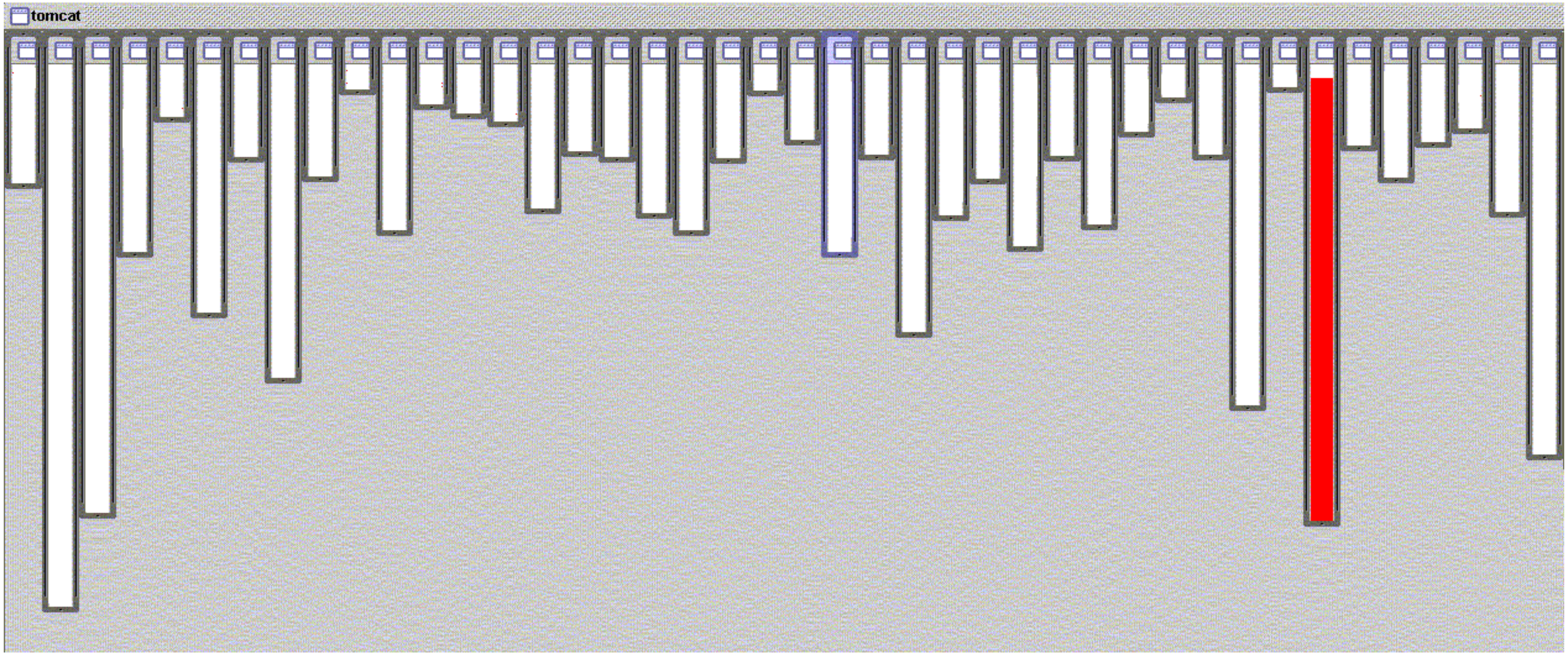
- **Redundant code**
  - same fragment of code in many places
- **Difficult to reason about**
  - non-explicit structure
  - the big picture isn’t clear
- **Difficult to change**
  - have to find all the code involved
  - and be sure to change it consistently
  - no help from OO tools

Good modularity

- Separated – implementation of a concern can be treated as relatively separate entity.
- Localized – implementation of a concern appears in one part of program.
- Modular – above + has a clear, well defined interface to rest of system.

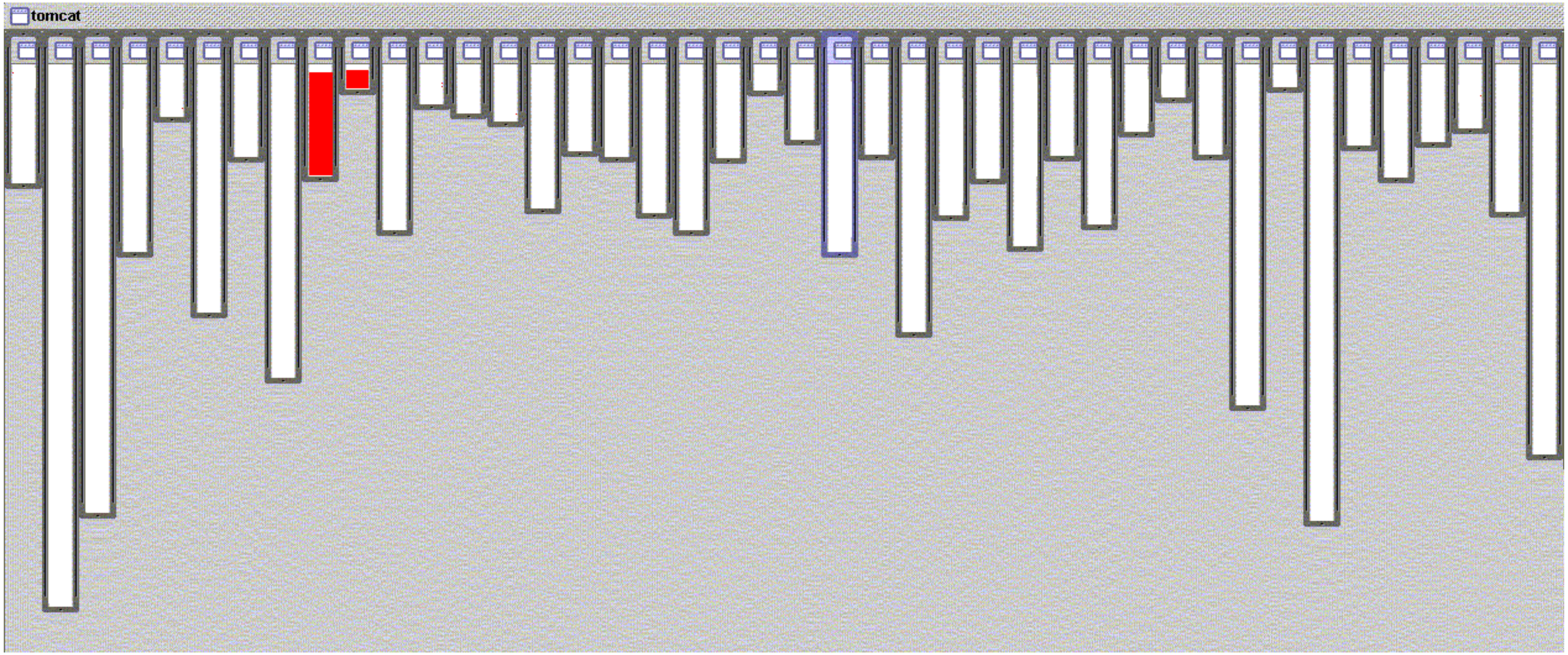


Good modularity



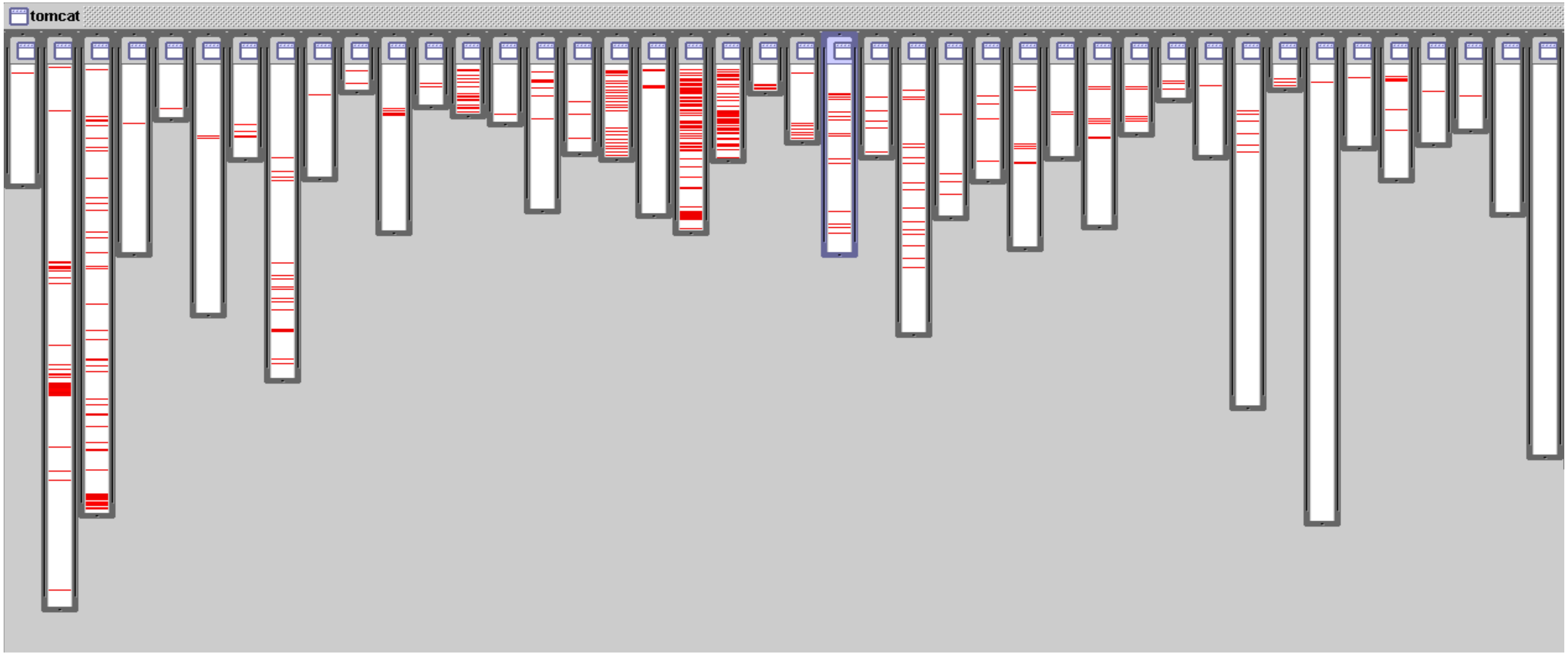
- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box

Good modularity



- URL pattern matching in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in two boxes (using inheritance)

Bad modularity



- Logging is not modularized
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places

Aspects

- Logging
- Security
- Transaction management
- Performance monitoring
- Optimization
- Persistence
- Synchronization
- Thread safety
- Policy enforcement
- Debugging support
- and many application-specific aspects

AOP – Concepts

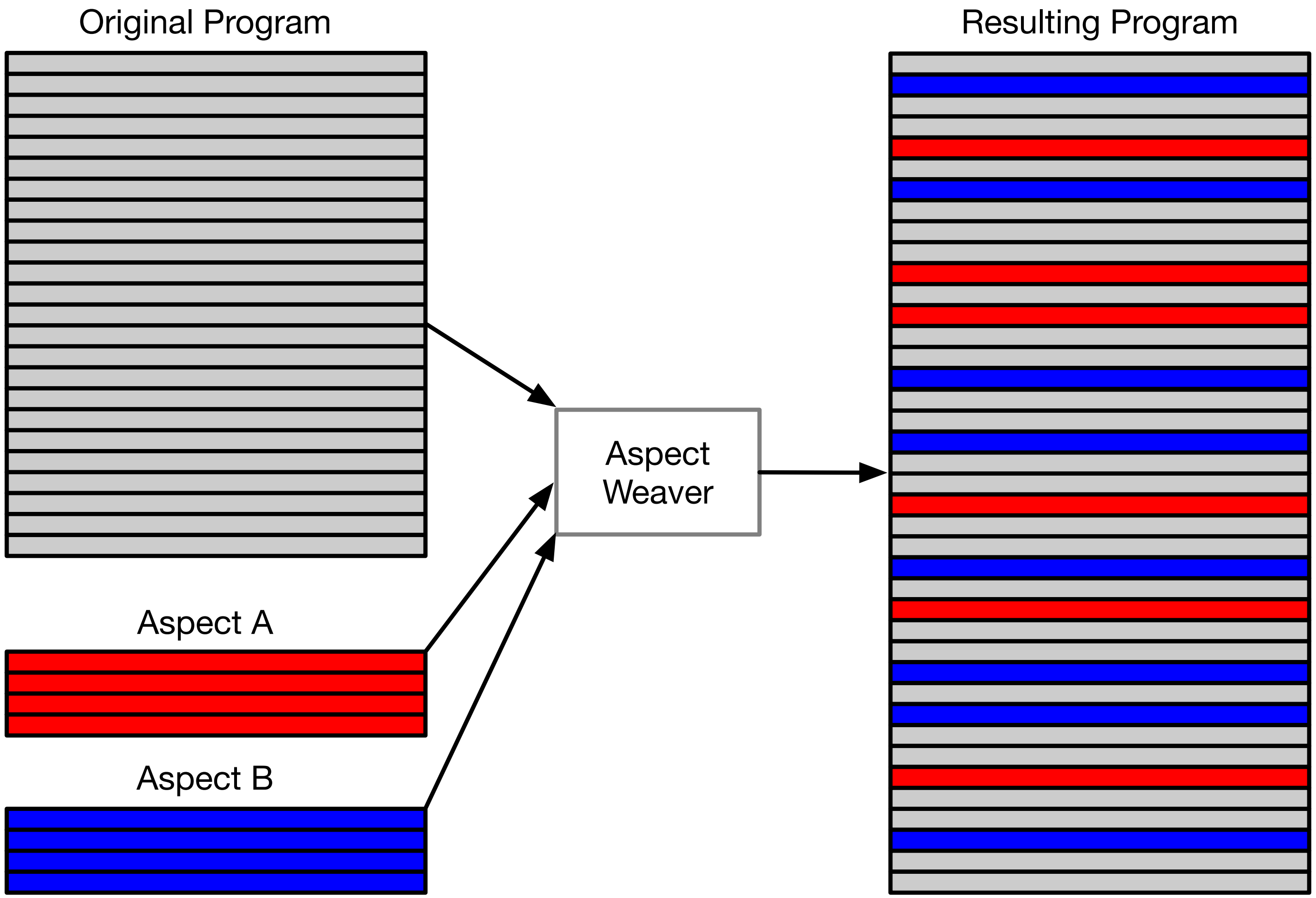
- Join point
  - A well-defined point in the execution of a program (in the dynamic call graph), e.g., execution of a method, initialization of a class, where additional code should be joined (run).
- Pointcut
  - A set of join points. E.g., all method calls to class B within class A.
- Advice
  - Additional behavior, i.e., code to run automatically at all join points in a particular pointcut. Can be marked as before, after, or around.
- Inter-type declarations (aka. Introductions)
  - Adding functionality to a class in place (as opposed to extending it)
- Aspect
  - Construct that captures all above.

Aspects

- Modular unit of crosscutting implementation
- An AspectJ aspect is a crosscutting type consisting of
  - advice on pointcuts (e.g., code to be executed at join points)
  - lexical introduction of behavior into other types (if any)
- Like classes, aspects
  - can have internal state and behavior,
  - can extend other aspects and classes, and
  - can implement interfaces

Aspect Weaver

- An aspect weaver weaves the code (advice) specified by aspects into a program at the specified points (pointcuts).
- Weaving can be performed either at compile time, load time, or run time.



A Simple Example

Insert **`println()`** calls for tracing methods.

```
public class A {
  public int m1() {

      //...
      m2();

      return 0;
  }
  public void m2() {

      //...

  }
}
```

A Simple Example - "Manual Approach"

Insert **`println()`** calls for tracing methods.

```
public class A {
  public int m1() {
      System.out.println("Enter A.m1");
      //...
      m2();
      System.out.println("Exit  A.m1");
      return 0;
  }
  public void m2() {
      System.out.println("Enter A.m2");
      //...
      System.out.println("Exit  A.m2");
  }
}
```

Problem: println() calls have to be added for each method call.

A Simple Example – Using AOP

Insert **`println()`** calls for tracing methods.

```
import org.aspectj.lang.*;

public aspect SimpleLogger {

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

  before() : logMethods() {
     Signature sig = thisJoinPointStaticPart.getSignature();
     System.out.println("Enter "+sig.getDeclaringType().getName()+"."
                         +sig.getName());
  }

  after() : logMethods() {
     Signature sig = thisJoinPointStaticPart.getSignature();
     System.out.println("Exit  "+sig.getDeclaringType().getName()+"."
                         +sig.getName());
  }
}
```

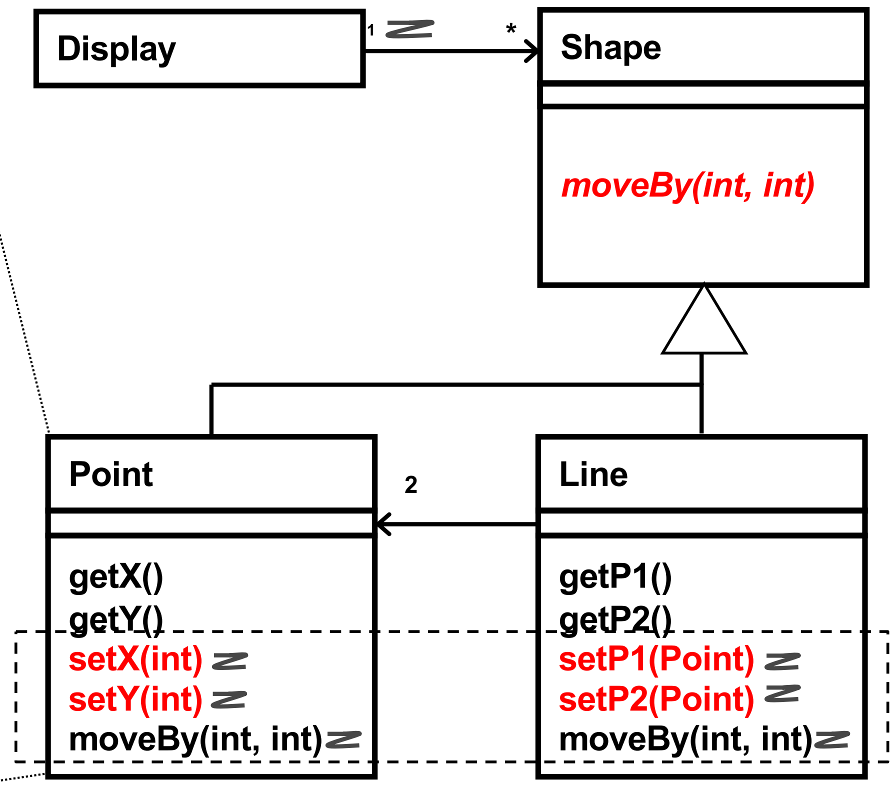
Advantage: single aspect for arbitrary complex programs

Example – Crosscutting Concerns

Fair design modularity

but poor code modularity

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

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

  void setX(int x) {
    this.x = x;
    display.update(this);
  }
  void setY(int y) {
    this.y = y;
    display.update(this);
  }
  ...
}
```

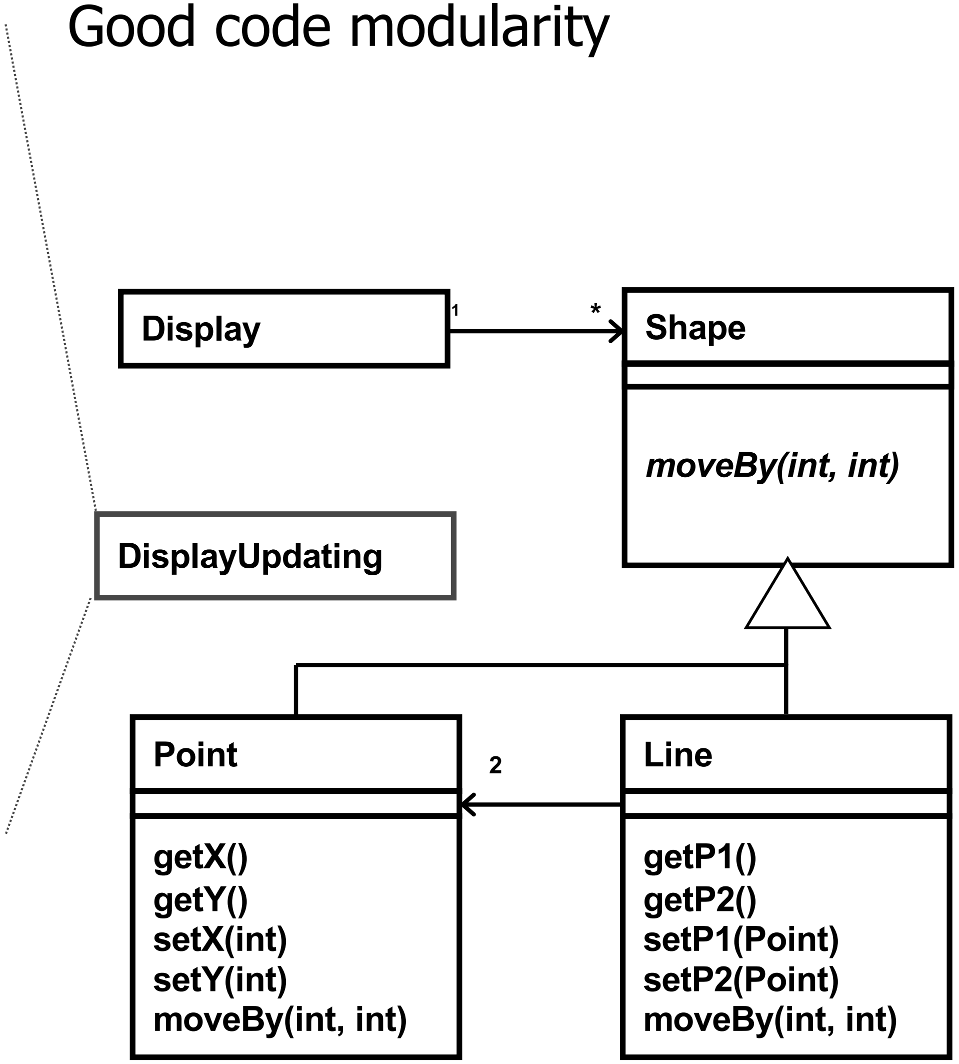


Example – Crosscutting Concerns

```
aspect DisplayUpdating {

  private Display Shape.display;

  pointcut change():
       call(void 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(Shape s) returning: change()
                         && target(s) {
   s.display.update();
  }
}
```

Good design modularity

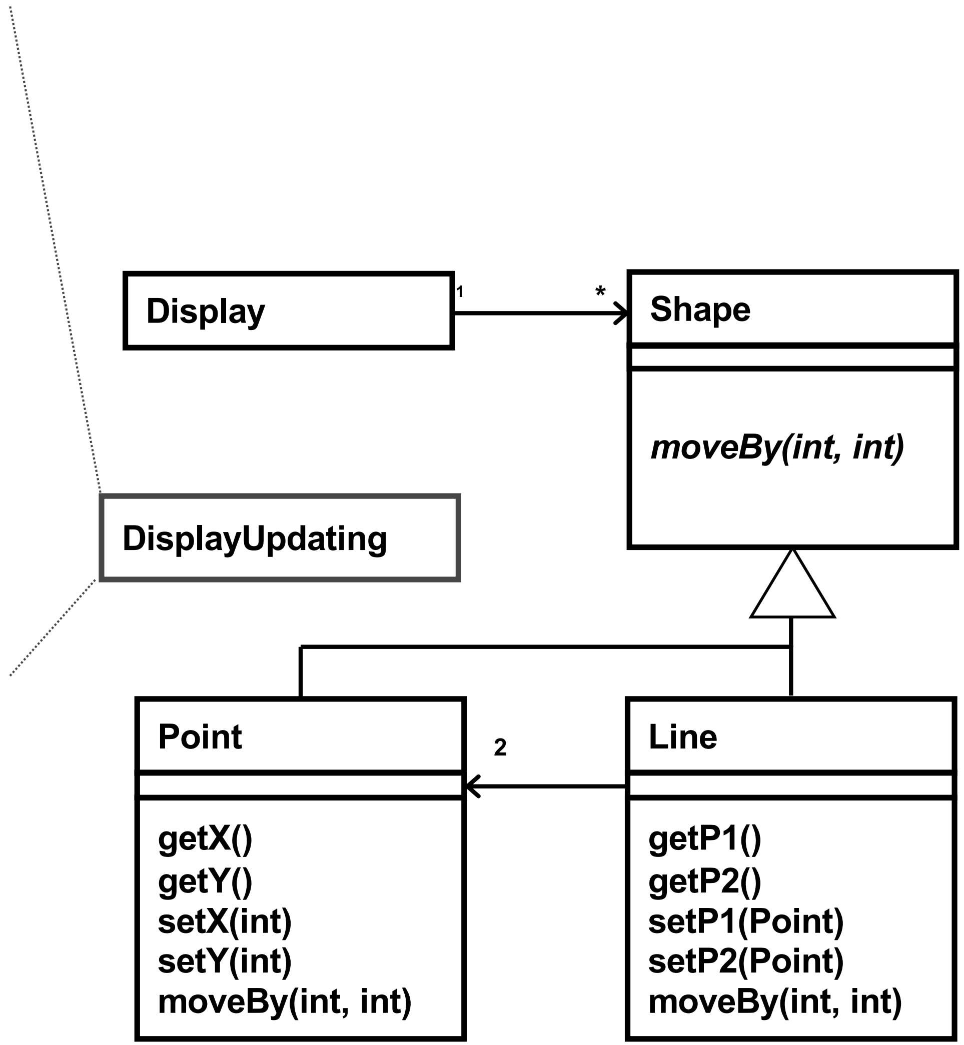
Good code modularity



Example – Crosscutting Concerns

Code looks like the design

```
aspect DisplayUpdating {

  private Display Shape.display;

  pointcut change():
       call(void Shape.moveBy(int, int))
    || call(void Shape+.set*(..));

  after(Shape s) returning: change()
                         && target(s) {
   s.display.update();
  }
}
```



AspectJ

- An aspect-oriented extension to Java
  - outputs .class files compatible with any JVM
  - all Java programs are AspectJ programs
  - originally developed by Xerox PARC (G. Kiczlas et al.)
  - now under the Eclipse project: http://eclipse.org/aspectj

    Current version: AspectJ 1.9.22.1, 11-May-2024
- IDE support
  - Eclipse, IntelliJ, Spring AOP, …
- Freely available implementation
  - Compiler is Open Source

Join Points

- Joint points are **well-defined points in the dynamic call graph**.

- AspectJ provides for many kinds of join points.
  - when a particular method body executes
  - when a method is called
  - when an exception handler executes
  - when the object currently executing (i.e. this) is of some type
  - when the target object is of some type
  - when the executing code belongs to (is within) a certain class
  - when the join point is in the control flow of another join point

Call Graph

- A call graph represents calling relationships between methods (subroutines, procedures, functions) in a program.
- Each node in the graph represents a method and each edge (f, g) indicates that method f calls method g. Thus, a cycle in the graph indicates recursive method calls.

- A dynamic call graph describes the call relationships of a certain program execution, and thus can be different for different executions.

- A static call graph represent every possible run of the program (which is usually an over-approximation).
- Constructing the exact static call graph is an undecidable problem!

Example – Method Call Join Points

- A method call join point includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including return (either normally or by throwing an exception).

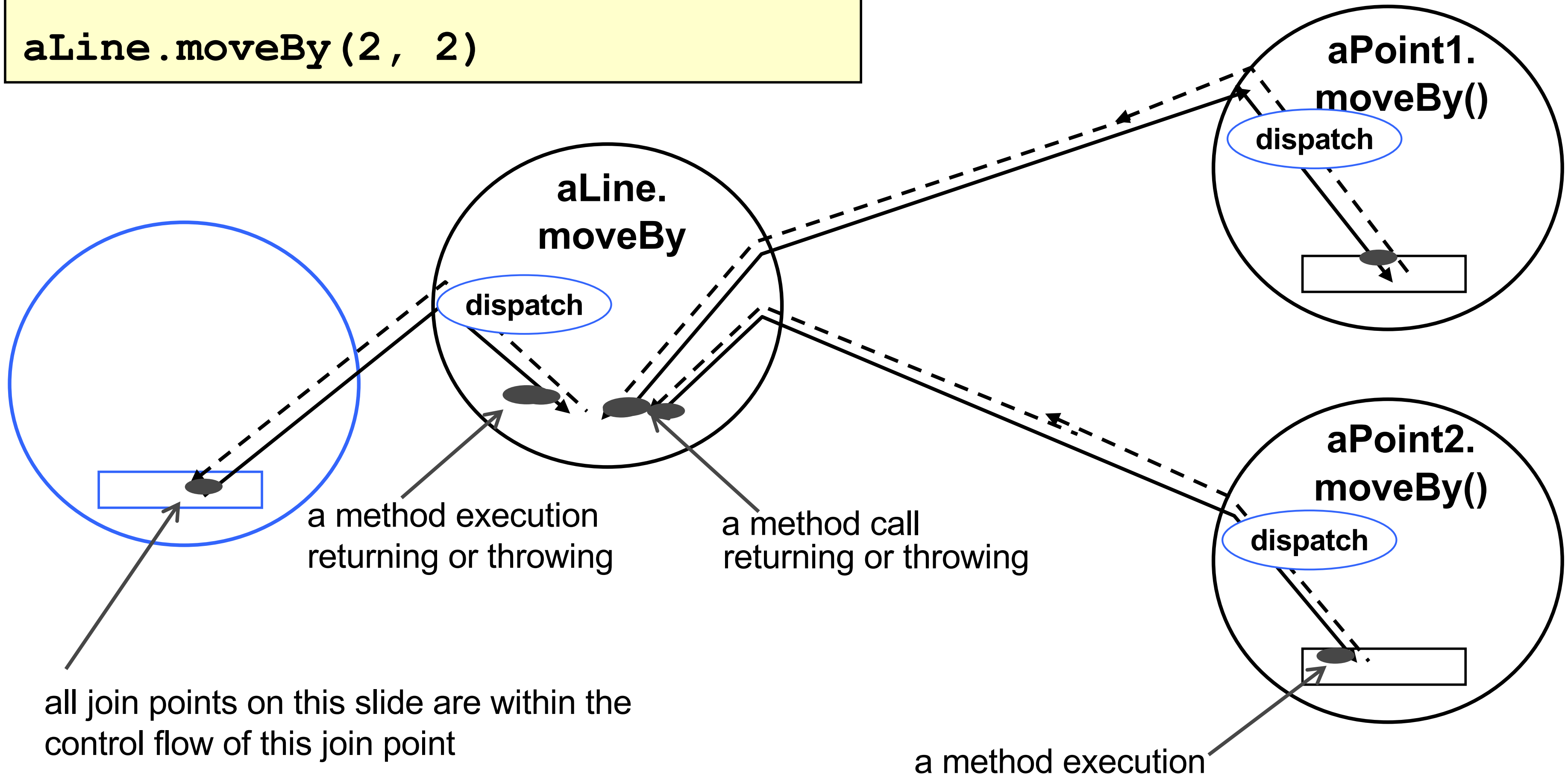
- Each method call at runtime is a different join point.

- All the join points that happen while executing the method body, and in those methods called from the body, execute in the dynamic context of the original call join point.

Dynamic Call Graph

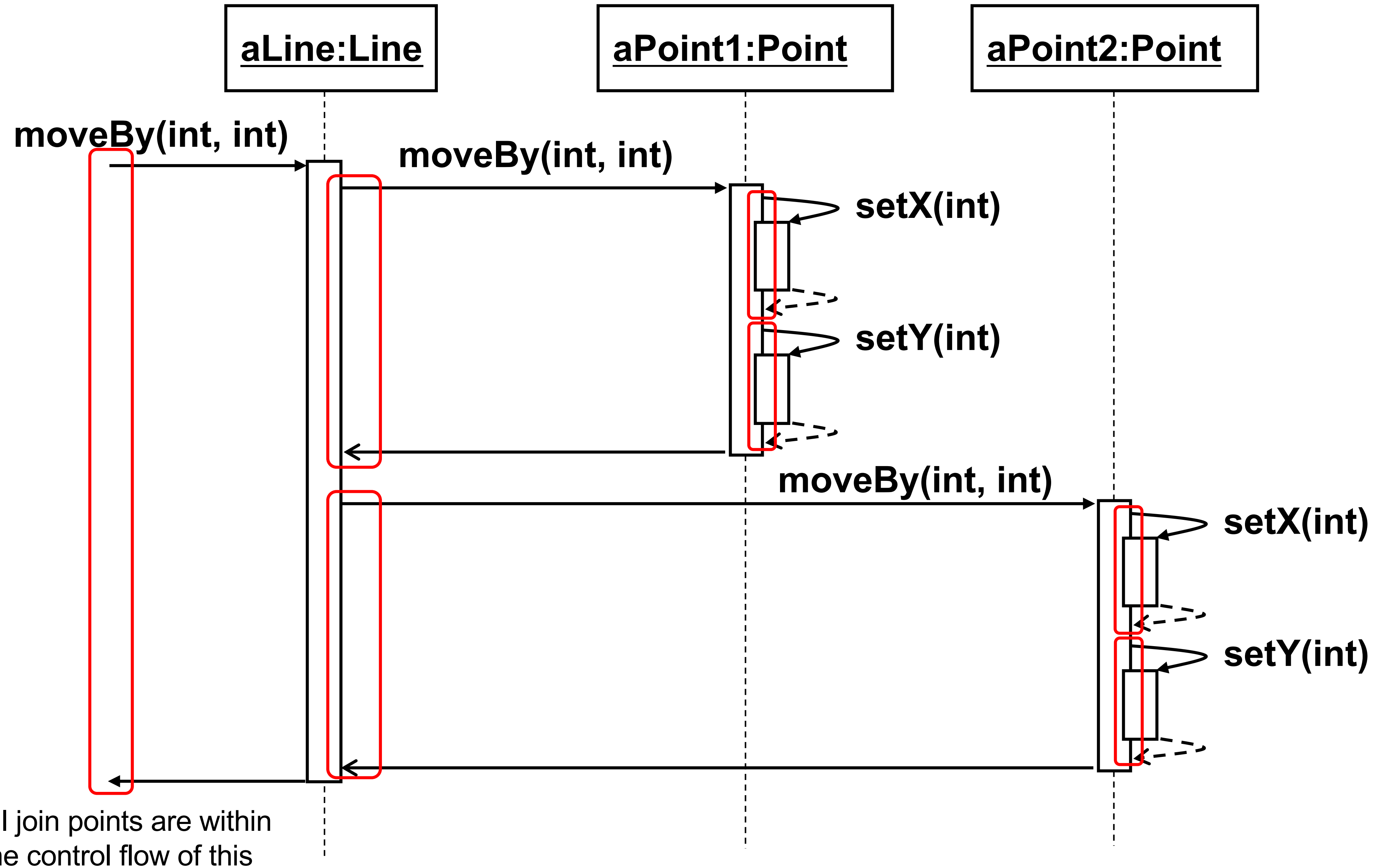
Join points are key points in the dynamic call graph.

```
// method call

aLine.moveBy(2, 2)
```



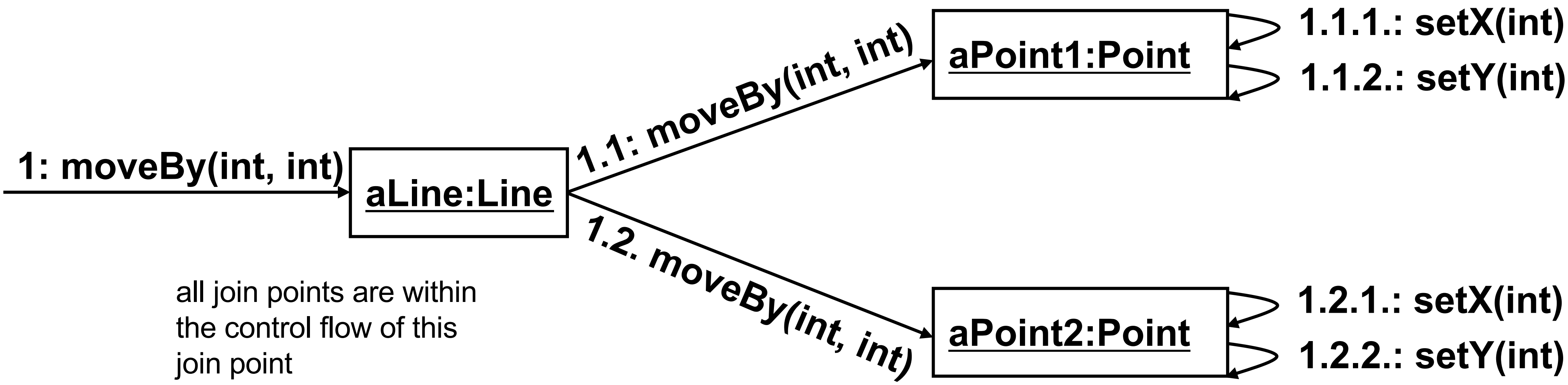
Source: G. Kiczales

UML Sequence Diagram



all join points are within the control flow of this join point

UML Collaboration Diagram



Pointcuts

- Pointcuts pick out certain join points in the program flow.

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

- picks out each join point that is a call to a method that has the signature

  void Point.setX(int).

Pointcuts

- Pointcuts can be named (or anonymous) and can be constructed from other pointcuts with `&&`, `||`, and `!`.

```
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));
```

Pointcuts

- Pointcuts can also be specified based on properties of methods, e.g., by using wildcards.

```
call(void Figure.make*(..))
```

Pointcut Wildcards

"*" is wild card
".." is multi-part wild card

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

call(void Point.setX(int))
call(public * Point.*(..))       // any public method on Point
call(public * *(..))             // any public method on any type

call(void Point.setX(int))
call(void Point.setY(*))
call(void Point.set*(*))
call(void set*(*))               // any setter

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

Pointcuts

```
// method body execution
execution(void Point.setX(int))

// method call
call(void Point.setX(int))

// exception handler execution
handler(ArrayOutOfBoundsException)

// currently executing object of some type
this(SomeType)

//target object of some type
target(SomeType)

//executing code belongs to a certain class
within(SomeClass)

//within control flow
cflow(call(void Test.main()))
```

Pointcut Parameters

- By specifying parameters, context information of each join point picked out by the pointcut can be accessed by any advice that uses the pointcut.
- Any advice that uses this pointcut has access to the context information.
- All pointcut parameters must be bound at every join point picked out by the pointcut.

s is a pointcut parameter

```
pointcut services(Server s): target(s) && call(public * *(..));

   before(Server s): services(s) {
     if (s.disabled) throw new DisabledException();
   }
```

- Any advice that uses the pointcut **`services`** has access to a **`Server`** from each join point picked out by the pointcut.
- That **`Server s`** comes from the target of each matched join point.

Advice

- An advice for a point cut specifies the code to run at each join point picked out by this point.

- Three different types:
  - Before advice runs before the program proceeds with the join point.
  - After advice runs after the program proceeds with that join point.
  - Around advice can perform custom behavior before and after the method invocation.

Before Advice

- Before advice on a method call join point runs before the actual method starts running, just after the arguments to the method call are evaluated.

```
// before-advice for pointcut move()
before(): move() { System.out.println("about to move"); }
```

After Advice

Three types.

- after returning runs after a method returns normally
- after throwing runs if a method throws an exception
- after runs after returning or throwing (like finally).

```
// after advice for pointcut move()
after() returning: move() {
   System.out.println("just successfully moved"); }
```

Around Advice

- Around advice can perform custom behavior before, after, or instead of the matched method invocation.
- On arrival at join point gets explicit control over when and if program proceeds.

```
void around(Point p, int x): target(p) &&
                             args(x) &&
                             call(void setX(int)) {
   if (p.assertX(x)) proceed(p, x);
   p.releaseResources();
}
```

- This around advice traps the execution of the join point; it runs instead of the join point.
- The original action associated with the join point can be invoked through the special **`proceed`** call.

Inter-Type Declarations

- Can declare members (fields, methods, and constructors) that are introduced into other types.
- Can declare that other types implement new interfaces or extend a new class.

```
aspect FaultHandler {
   private boolean Server.disabled = false;

   private void reportFault() {
     System.out.println("Failure! Please fix it."); }
   public static void fixServer(Server s) { s.disabled = false; }
   ...
}
```

Inter-Type Declaration:
add field **disabled** to Server

An Example Aspect

```
aspect FaultHandler {
   private boolean Server.disabled = false;
   private void reportFault() {
     System.out.println("Failure! Please fix it."); }

   pointcut services(Server s): target(s) && call(public * *(..));

   before(Server s): services(s) {
     if (s.disabled) throw new DisabledException();
   }

   after(Server s) throwing (FaultException e): services(s) {
     s.disabled = true;
     reportFault();
   }
}
```

inter-type declaration

matches all calls to public methods of Server objects

advice

AOP – Generating Assertions

- Assertions enable an informal design-by-contract style of programming.
  - Preconditions

    What must be true when a method is invoked.
  - Postconditions

    What must be true after a method completes successfully.
  - Class invariants

    What must be true about each instance of a class.

- Assertion usually serve as test code integrated directly into a program. They are disabled by default. To enable assertion the –ea option needs to be specified.

Example: Preconditions

```
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;
  }
}
```

Example: Postconditions

```
aspect PointBoundsPostCondition {

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

  after(Point p, int newY) returning:
      call(void Point.setY(int)) && target(p) && args(newY) {
    assert p.getY() == newY;
  }
}
```

AspectJ - Weaver

- **Compile-time weaver**
  - Aspect and class source code → class files

- **Binary weaver** ("linker")
  - Aspect and class source/binary files → class files

- **Load-time weaver**
  - Aspect and class binary files
  - Weave class files when being loaded into VM

AspectJ applied to a large middleware system

- Java code base with 10,000 files and 500 developers.
- AspectJ captured logging, error handling, and profiling policies.

Existing policy implementations

- affect every file
  - 5-30 page policy documents
  - applied by developers
- affect every developer
  - must understand policy document
- repeat for new code assets
- awkward to support variants
  - complicates product line
- don't even think about changing the policy

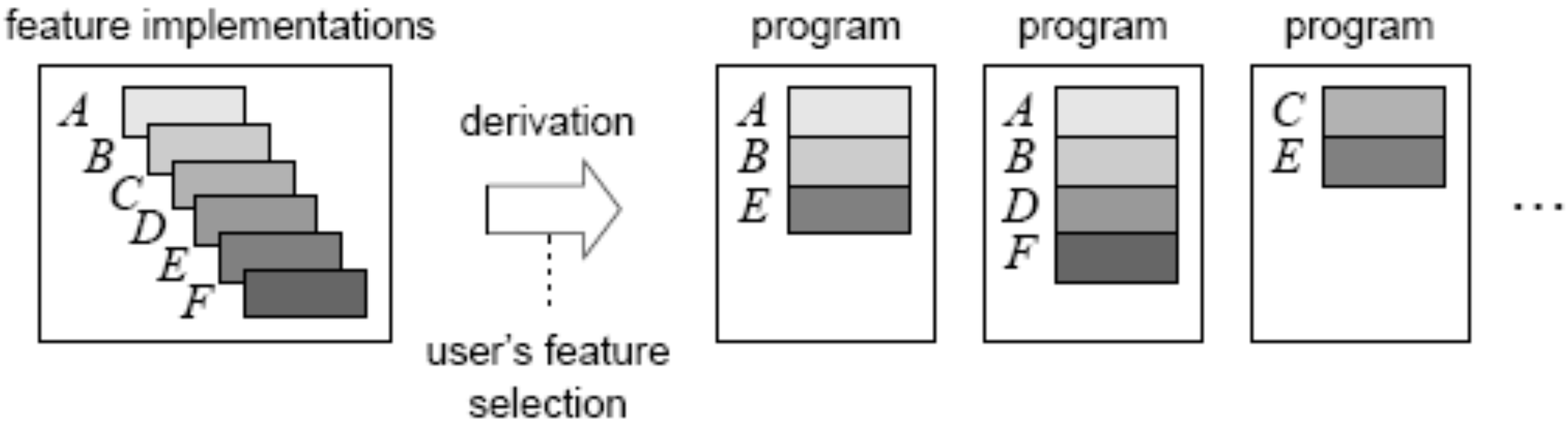
Policies implemented with AspectJ

- one reusable crosscutting module
  - policy captured explicitly
  - applies policy uniformly for all time
- written by central team
  - no burden on other 492 developers
- automatically applied to new code
- easy plug and unplug
  - simplifies product line issues
- changes to policy happen in one place

Source: G. Kiczlas

AOP and Related Approaches

- Generative Programming
- Metaprogramming, Metaobject Protocols
- Feature-Oriented Programming (FOP)
  - decompose a software system in terms of the features it provides
  - construction, customization, and synthesis of large-scale software systems
  - the set of software systems generated from a set of features is also called a software product line.



http://www.jot.fm/issues/issue_2009_07/column5/