# Aspect-Oriented Programming and AspectJ

- Aspect-oriented programming is a common buzzword lately
- Papers from ECOOP 1997 (early overview—the manifesto), ECOOP 2001 (overview of AspectJ)
- Kiczales (the team leader) was behind the CLOS MOP



## What is Aspect-Oriented Programming?

Many possible answers:

- a fad
- a way-too-general collection of cross-cutting programming techniques
- a new name for old ideas (generators, domain-specific languages, MOPs)
- the solution to all our problems

My opinion: AOP is just a new name for old ideas, but these ideas are good
  - dominant AOP implementations (e.g., AspectJ) are MOP-like, but they don't need to be

An *aspect* is a piece of functionality that *cross-cuts* functional units of a system



## Simple Working Example (bad, IMO)

- Image processing application with filters
  - filters need to be kept separate
  - filters need to be fused together for optimization (e.g., loop fusion)

```
(defun or! (a b)
  (let ((result (new-image)))
    (loop for i from 1 to width do
      (loop for j from 1 to height do
        (set-pixel result i j
          (or (get-pixel a i j)
              (get-pixel b i j)))))
    result))
```

- If another filter has the same looping structure, the two should be fused together (for locality, max memory consumption, etc.)
- Better example: in distributed apps, synchronization/serialization of data/failure handling are orthogonal to other functionality



## Aspect-Oriented Principles

- A *component* is a part of the implementation that is localized in traditional languages (Java/C/C++, etc.)
- An *aspect* is a part of the implementation that is not well-localized with traditional languages (code ends up being scattered everywhere)
- AOP tries to offer language support for expressing aspects concisely and separately from components
- *Join points*: the points where components interact and aspects can influence them
- Elements of an AOP-based implementation:
  - a component language part
  - an aspect language part
  - an *aspect weaver* applying the aspects to components (really, a generator)

## Image Processing Example Revisited

Essentially, a domain-specific language is designed for image processing

- Component language: implicit loop structure

```
(define-filter or! (a b)
  (pixelwise (a b) (aa bb) (or aa bb)))
```

(`aa`, `bb` are iterators—standard Lisp/Scheme iterator binding semantics)

- Aspect language: operations on nodes in the dataflow graph. E.g., if two loops have the same structure and inputs, fuse them together
- Weaver: represent the component program as a flow graph, run aspect code on it, generate code from higher-level abstractions



## Other Example

- How data is serialized (and, in particular, how much data is copied) in a distributed system is an issue independent of the system's main functionality
- A communication aspect language can allow the programmer to describe how much of an object will be copied
  - again, just a domain-specific language/ generator. You can call it AOP, but the value is in the domain
  - there is a general system called Doorastha that does similar things in Java
- E.g., a digital library may have `Book` and `Repository` classes. We can tell the system to only copy parts of a `Book` object when registering and unregistering it in a remote repository



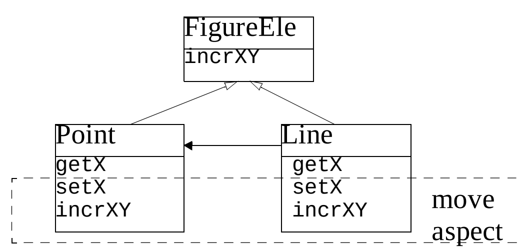
Example aspect program:

```
remote Repository {
  void register (Book);
  void unregister (Book: copy isbn);
// Book class has “isbn” field
  Book: copy isbn lookup(String);
// method: “lookup”, return type: Book
}
```



# AspectJ

- A very nice MOP/general compositional semantic extensibility facility for Java
  - used entirely for interposing code, not changing how the object system works
  - AspectJ is a transparent extension of Java, comes with IDE support (for easier editing, inspection of aspect code)
- To demonstrate, consider an example application: a figure editor



## Join points

- Many possible join points in AspectJ. At:
  - method call (inside calling object)
  - method call reception by an object (any method)
  - method execution (specific method)
  - field access (get/set)
  - constructor call (inside object doing `new`)
  - constructor call reception (any constructor)
  - exception handler execution
  - class initialization (static initializers run)

## Pointcuts

- *Pointcut* = set of join points + values from the context (e.g., the `this` object, method parameters, etc.)

```
call(void Point.setX(int))
```
  - all join points where the method called is `void Point.setX(int)`





## Kinds of Pointcuts

- Pointcuts can be thought of as runtime predicates: when they are true, we are at a join point described by the pointcut.

- Several kinds of pointcuts. E.g.:
  - `call(`*`signature`*`)`
  - `execution(`*`signature`*`)`
  - `get/set(`*`signature`*`)`
    - value can be matched with `args`
  - `args(`*`Type`*`)`
  - `handler(`*`ThrowableClass`*`)`
  - `this/target(`*`Type`*`)`
  - `within(`*`Type`*`)`
  - `withincode(`*`signature`*`)`
  - `cflow(`*`pointcut`*`)`
  - `initialization(`*`ConstrSig`*`)`
  - `staticinitialization(`*`Type`*`)`

- Also: boolean pointcut operators (&&, `||`, etc.) and pointcut constants (user-defined pointcuts)





## Pointcut Example

```
pointcut moves():
   call(void FigureElement.incrXY(int,int))
|| call(void Line.setP1(Point))
|| call(void Line.setP2(Point))
|| call(void Point.setX(int))
|| call(void Point.setY(int));
```

- describes the join points where methods that cause “movement” of a figure are called
  - Note that a “user-defined” pointcut (operator `pointcut`) is used to give a name (`moves`) to the pointcut

## Advice

- *Advice*: specification of aspect code to be interposed at pointcuts
  - before, after, or instead of (`around`) the code at a join point
    - two special cases of “after”: after returning/after throwing (for normal/exception exits)





## Aspects

- Aspects have class-like syntax (and, to some extent, semantics—e.g., for scoping). They can contain pointcuts, advice, and regular class declarations (member vars/methods)

```
aspect MoveTracking {
  static boolean flag = false;
  static boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;
  }

  pointcut moves():
   call(void FigureElement.incrXY(int,int))
   || call(void Line.setP1(Point))
   || call(void Line.setP2(Point))
   || call(void Point.setX(int))
   || call(void Point.setY(int));

  after(): moves() {        // advice
    flag = true;
  }
}
```

## Aspects

- Aspects can have multiple instances
- There are complex rules about how aspect execution (advice application) is ordered
  - the rules take into account Aspect relationships (e.g., if aspect A extends B, then it's considered more specific)
  - there is a `dominates` keyword for aspects that know about each other

Example (uses `MoveTracking` from last slide)

```
aspect Mobility dominates MoveTracking {
  static boolean enableMoves = true;

  around() returns void:
    MoveTracking.moves()
  { if (enableMoves) proceed(); }
}
```

defines an "around" (instead-of) method preventing moves if the flag is not set




## Pointcut Parameters

- Advice and pointcut definitions can have parameters (see empty parentheses in previous examples)
- The parameters can be used in pointcut predicates instead of type variables and take the value *of the instance matching the predicate*
  - this is overloading the existing syntax for an entirely different purpose

```
before(Point p, int nval):
  call(void p.setX(nval)) {
    System.out.println("x value of" + p +
      " will be set to " + nval + ".");
  }
```

To print a message every time the value of `x` for a point changes

Example: Getting the current object




- regular pointcut definition:
  ```
  pointcut foo() :
    instanceof(Point);
  ```
- pointcut with parameter:
  ```
  pointcut foo(Point p) :
    instanceof(p);
  ```
- `p` is the object of class `Point` with which the join point is associated!

Example: Around Advice and Proceed

- We saw `proceed` earlier, but it can also be called with parameters
- To ensure that a method is only called with non-negative int arguments:
  ```
  around(int nv) returns void:
   call(void Point.setX(nv))
  { proceed(Math.max(0, nv)); }
  ```




## Abstract and Generic Aspects

A "virtual type"-like mechanism allows aspect genericity

```
abstract aspect SimpleTracing {
  abstract pointcut tracePoints();
  //yet undefined

  before(): tracePoints() {
    printMessage("Entering",thisJoinPoint);
  }
  after(): tracePoints() {
    printMessage("Exiting",thisJointPoint);
  }

  void printMessage(String s, JoinPoint tjp)
  { ... }
}

aspect XYTracing extends SimpleTracing {
  pointcut tracePoints():
    call(
      void FigureElement.incrXY(int,int));
}
```

- (note the `thisJointPoint` variable and the `JoinPoint` type: they reflectively export details of the AspectJ implementation)

## Wildcards

E.g.,

```
call(* Point.*(..))
call(Point.new(..))
```

## Control-Flow Based Pointcuts

The `cflow` operator is true on points under the dynamic extent of other join points (e.g., while the methods corresponding to these join points are still active on the execution stack)

```
pointcut moves(FigureElement fe):
<see before>;

pointcut topLevelMoves(FigureElement fe):
 moves(fe) && !cflow(moves(FigureElement));
```

## Implementation

The AspectJ compiler inserts code to check and call the right aspects at join points: efficient



## Introductions / Inter-type Declarations

Can declare members and supertypes for existing classes!

A static transformation language. These “introductions” are not advice and are not associated with pointcuts

Add an “`enabled`” field to all `FigureElement`s:

```
- boolean FigureElement.enabled=false;
```

Add a setter method:

```
- public
   FigureElement.setEnabled(boolean b) {
    this.enabled = b;
   }
```

Add superclasses to FigureElement:

```
- declare parents:
    FigureElement extends Drawable
```