


# Programming Language Concepts

## June 18, 2003

Klaus Ostermann

Darmstadt University of Technology  
Software Technology Group  
ostermann@informatik.tu-darmstadt.de

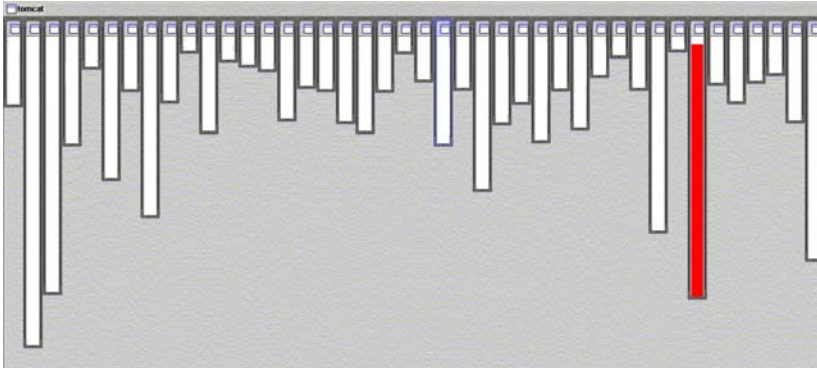


# Aspect-Oriented Programming with AspectJ™

Adapted from slides by the AspectJ team

good modularity

## XML parsing

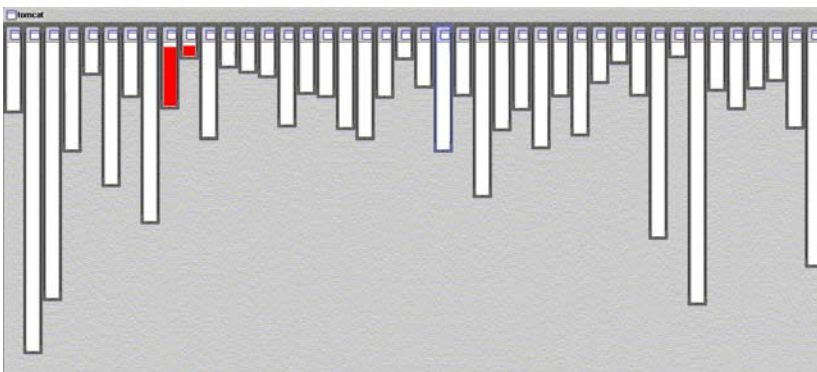


- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

good modularity

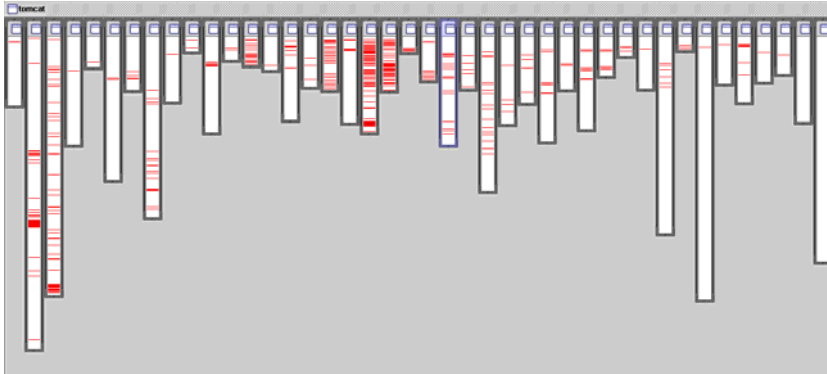
## URL pattern matching



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

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## problems like... logging is not modularized



- logging in org.apache.tomcat
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places

PL\_SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## the cost of tangled 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

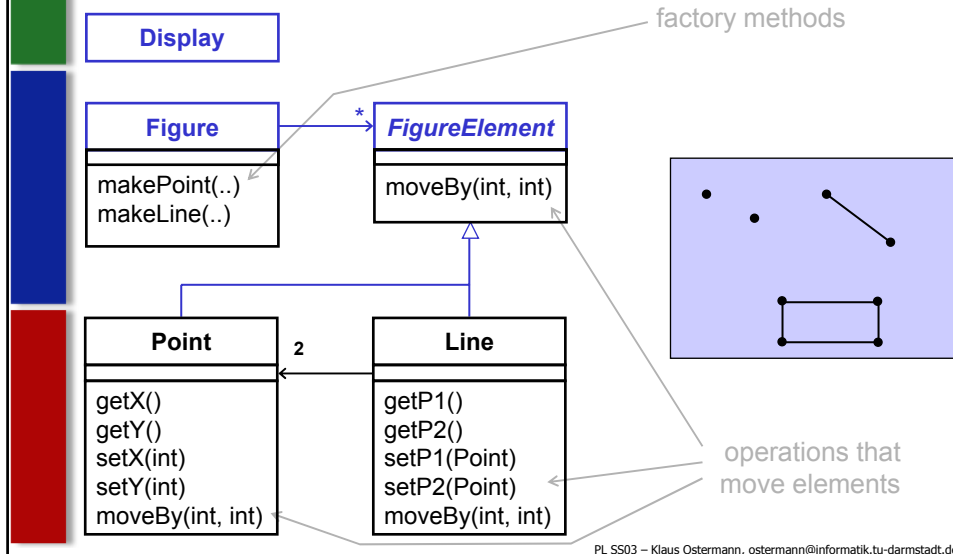
PL\_SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## aspect-oriented programming

- crosscutting is inherent in complex systems
- crosscutting concerns
  - have a clear purpose
  - have a natural structure
    - defined set of methods, module boundary crossings, points of resource utilization, lines of dataflow...
- so, let's capture the structure of crosscutting concerns explicitly...
  - in a modular way
  - with linguistic and tool support
- aspects are
  - well-modularized crosscutting concerns

- AspectJ is:
  - an aspect-oriented extension to Java™ that supports general-purpose aspect programming
- using AspectJ to:
  - improve the modularity crosscutting concerns
    - design modularity
    - source code modularity
    - development process
- aspects are two things:
  - concerns that crosscut [design level]
  - a programming construct [implementation level]
    - enables crosscutting concerns to be captured in modular units

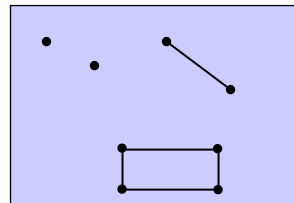
## a simple figure editor



## a simple figure editor

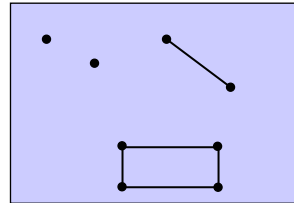
```
class Line implements FigureElement{
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
    void moveBy(int dx, int dy) { ... }
}

class Point implements FigureElement {
    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; }
    void moveBy(int dx, int dy) { ... }
}
```



## display updating

- collection of figure elements
  - that move periodically
  - must refresh the display as needed
  - complex collection
  - asynchronous events

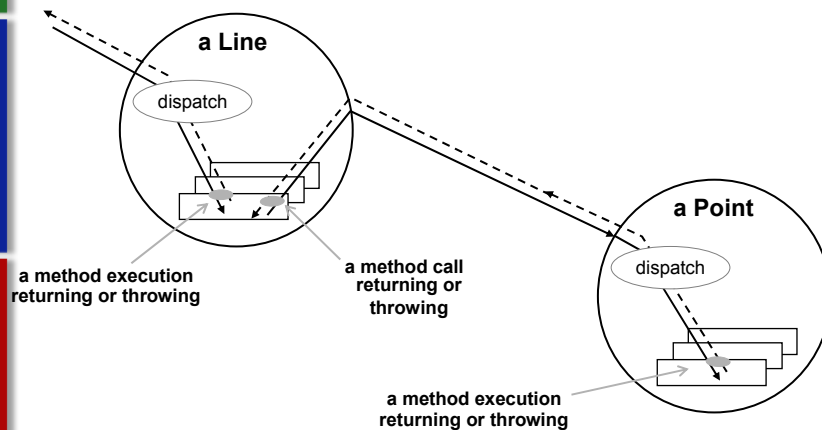


*we will initially assume  
just a single display*

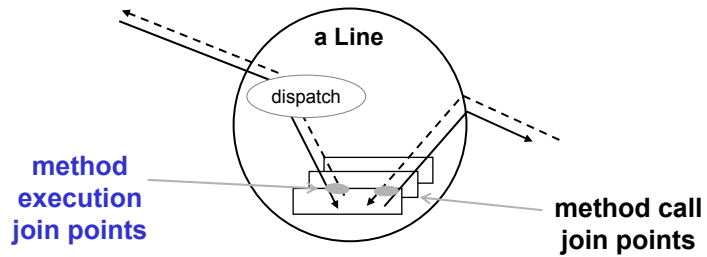
## join points

### key points in dynamic call graph

imagine `l.moveBy(2, 2)`



## join point terminology

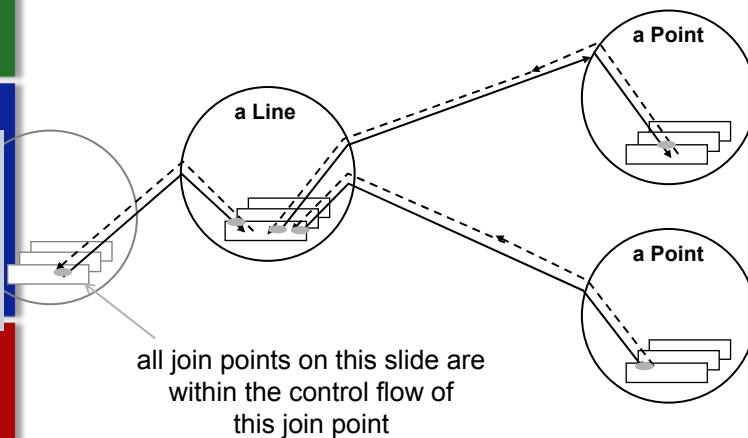


- several kinds of join points
  - method & constructor call
  - method & constructor execution
  - field get & set
  - exception handler execution
  - static & dynamic initialization

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## join point terminology

### key points in dynamic call graph



PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## the pointcut construct names certain join points

each time there is a `<void Line.setP1(Point)>`  
or `<void Line.setP2(Point)>` method call

name and parameters

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

a "void Line.setP1(Point)" call

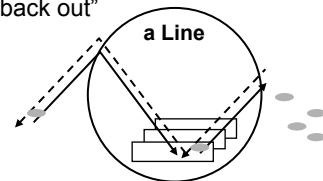
OR

a "void Line.setP2(Point)" call

## after advice

### action to take after computation under join points

after advice runs  
"on the way back out"



```
pointcut move() :  
    call(void Line.setP1(Point)) ||  
    call(void Line.setP2(Point));  
  
after() returning: move() {  
    <code here runs after each move>  
}
```

## 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

```
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();  
    }  
}
```

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

```
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;  
    }  
}
```

## without AspectJ DisplayUpdating v1

```
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();
    }
}

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;
    }
}
```

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## without AspectJ DisplayUpdating v2

```
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();
    }
}

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();
    }
}
```

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## without AspectJ DisplayUpdating v3

```
class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) {
        this.p1 = p1;
        Display.update(this);
    }
    void setP2(Point p2) {
        this.p2 = p2;
        Display.update(this);
    }
}

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(this);
    }
    void setY(int y) {
        this.y = y;
        Display.update(this);
    }
}
```

- no locus of “display updating”
  - evolution is cumbersome
  - changes in all classes
  - have to track & change all callers

## with AspectJ

```
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;
    }
}
```

## with AspectJ DisplayUpdating v1

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

    after() returning: move() {
        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();
    }
}
```

# with AspectJ DisplayUpdating v3

```

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(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)));

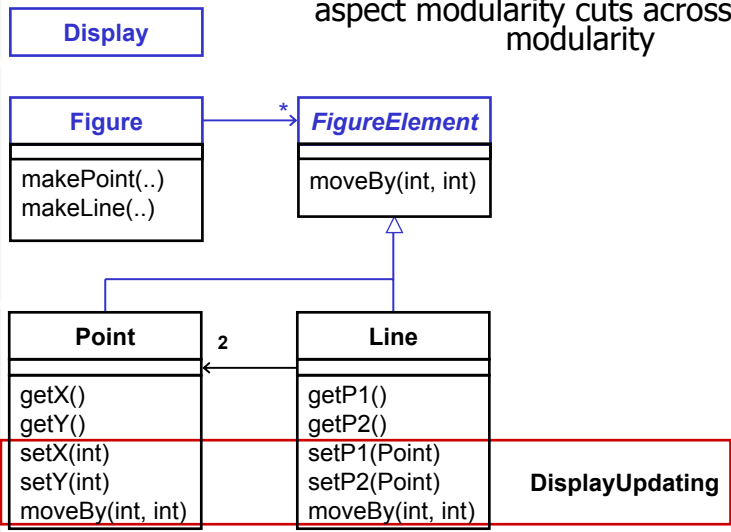
    after(FigureElement fe) returning: move(fe) {
        Display.update(fe);
    }
}

```

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

## aspects crosscut classes

aspect modularity cuts across class modularity



```

aspect DisplayUpdating {

    pointcut move(Object mover, FigureElement movee):
        this(mover)      &&
        target(movee)    &&
        (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(Object mover, FigureElement fe) returning:
        move(mover, movee) {
            Display.update(mover, fe);
        }
}

```

```

class Line {
    private Point _p1, _p2;

    Point getP1() { return _p1; }
    Point getP2() { return _p2; }

    void setP1(Point p1) {
        _p1 = p1;
        // Display.update(this);
    }
    void setP2(Point p2) {
        _p2 = p2;
        // Display.update(this);
    }
}

class Point {
    private int _x = 0, _y = 0;

    int getX() { return _x; }
    int getY() { return _y; }

    void setX(int x) {
        _x = x;
        // Display.update(this);
    }
    void setY(int y) {
        _y = y;
        // Display.update(this);
    }
}

```

```

class Foo {
    ...
    public void mumble(Line l) {
        Display.update(this, l);
        l.setP1(new Point());
        ...
    }
    ...
}

```

## without AspectJ MoveTracking v4

```
class Line {
    private Point _p1, _p2;

    Point getP1() { return _p1; }
    Point getP2() { return _p2; }

    void setP1(Point p1) {
        _p1 = p1;
        Display.update(this);
    }
    void setP2(Point p2) {
        _p2 = p2;
        Display.update(this);
    }
}
```

```
class Point {
    private int _x = 0, _y = 0;

    int getX() { return _x; }
    int getY() { return _y; }

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

```
class Baz {
    ...
    public void mumble(Line l) {
        Display.update(this, l);
        l.setP1(new Point());
        ...
    }
    ...
}
```

```
class Bar {
    ...
    public void mumble(Line l) {
        Display.update(this, l);
        l.setP1(new Point());
        ...
    }
    ...
}
```

```
class XYZZY {
    ...
    public void moveit(Point p) {
        Display.update(this, p);
        p.setX(p.getX()+1);
        ...
    }
    ...
}
```

```
class AnotherClass {
    ...
    public void frotz(Point p) {
        Display.update(this, p);
        p.setY(42);
        ...
    }
    ...
}
```

```
class Foo {
    ...
    public void mumble(Line l) {
        Display.update(this, l);
        l.setP1(new Point());
        ...
    }
    ...
}
```

```
class ... {
    ...
    public void mbar(Line l) {
        Display.update(this, l);
        l.setP1(new Point());
        ...
    }
    ...
}
```

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## property-based crosscutting

```
package com.xerox.a;
public class C1 {
    ...
    public void foo() {
        A.doSomething(...);
    }
    ...
    private void mum() {
        int x = 10;
    }
    ...
}
```

```
package com.xerox.a;
public class C2 {
    ...
    public int frotz() {
        A.doSomething(...);
    }
    ...
    public int bar() {
        A.doSomething(...);
    }
    ...
}
```

```
package com.xerox.b;
public class C3 {
    ...
    public String s1() {
        A.doSomething(...);
    }
    ...
}
```

- crosscuts methods with a common property
  - public/private, return a certain value, in a particular package
- logging, debugging, profiling
  - log on entry to every public method

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## logging again

```
//From ContextManager
```

```
public void service( Request rrequest, Response rresponse ) {
    // log( "New request " + rrequest );
    try {
        // System.out.println("A");
        rrequest.setContextManager( this );
        rresponse.setRequest( rrequest );
        // System.out.print("A");
        // wont request - parsing error
        int status=rresponse.getStatus();

        if( status < 400 )
            status=processRequest( rrequest );

        if(status==0)
            status=authenticate( rrequest, rresponse );
        if(status == 0)
            status=authorize( rrequest, rresponse );
        if( status == 0 ) {
            rrequest.getWrapper().handleRequest( rrequest,
                rresponse );
        } else {
            // something went wrong
            handleError( rrequest, rresponse, null, status );
        } catch (Throwable t) {
            handleError( rrequest, rresponse, t, 0 );
            // System.out.print("B");
        }
        try {
            rresponse.finish();
            rrequest.recycle();
            rresponse.recycle();
        } catch ( Throwable ex ) {
            if( debug>0) log( "Error closing request " + ex );
            // log("Done with request " + rrequest );
            // System.out.print("C");
        }
        return;
    }
}
```

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## modular crosscutting

```
aspect PublicErrorLogging {
```

```
    Log log = new Log();
```

```
    pointcut publicInterface () :
        call(public * com.mycompany..*.*(..));
```

```
    after() throwing (Error e): publicInterface() {
        log.write(e);
    }
}
```

captures public interface  
of com.mycompany  
package

consider code maintenance

- another programmer adds a public method
  - i.e. extends public interface – this code will still work
- another programmer reads this code
  - "what's really going on" is explicit

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

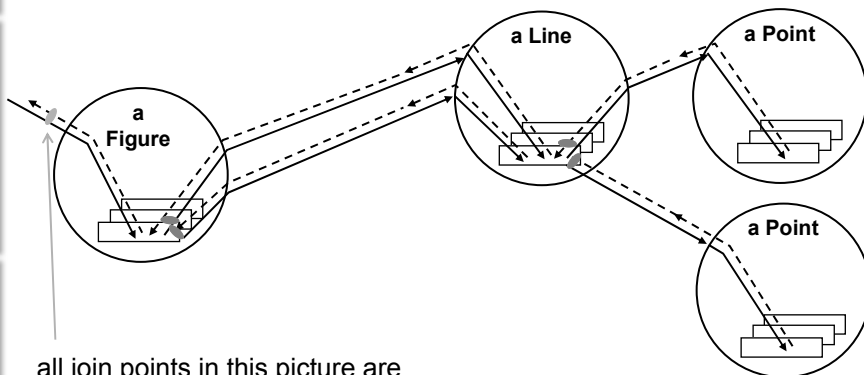
## modular crosscutting

```
aspect PublicErrorLogging {  
    Log log = new Log();  
    pointcut publicInterface ():  
        call(public * com.mycompany..*.*(..));  
    after() throwing (Error e): publicInterface() {  
        log.write(e);  
    }  
}
```

captures public interface  
of com.mycompany  
package

- crosscutting concerns
  - tangled implementation → complex, difficult to maintain
  - modular implementation → can be clear, easy to maintain
- crosscutting concerns per se not complicated!

## join point terminology control flow



all join points in this picture are  
in the control flow of this one

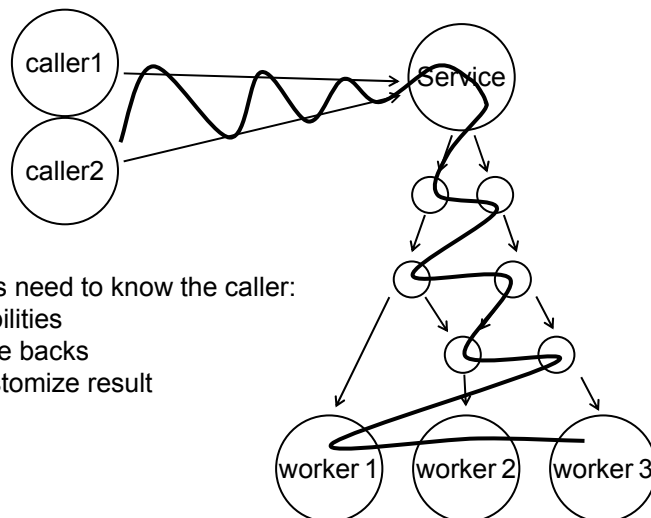
## context sensitive aspects

### MoveTracking v5

```
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 figElt):  
        move(figElt) && !cflow(move(FigureElement));  
  
    after(FigureElement fe): topLevelMove(fe) {  
        Display.update(fe);  
    }  
}
```

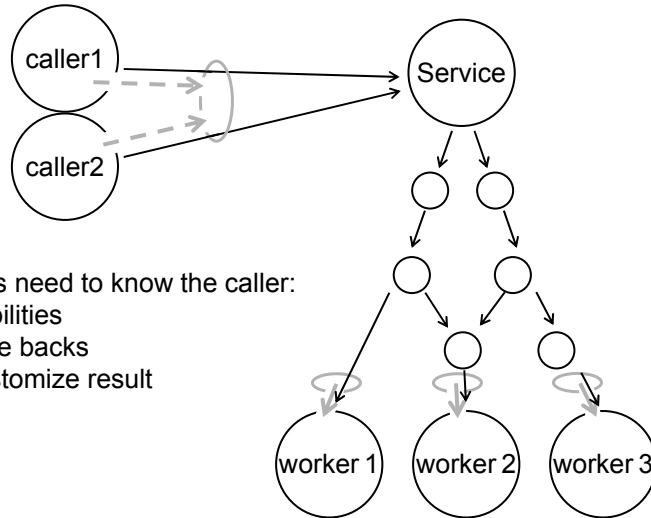
PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## context passing aspects



PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## context passing aspects



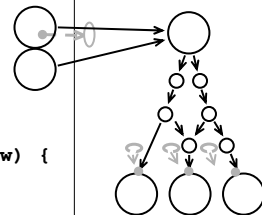
workers need to know the caller:

- capabilities
- charge backs
- to customize result

PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## context passing aspects

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



PL SS03 – Klaus Ostermann, ostermann@informatik.tu-darmstadt.de

## 2 kinds of crosscutting

- aspect
  - a modular unit of crosscutting implementation
  - advice
    - adds functionality to existing operations
    - adds per-aspect state
  - inter-class declarations
    - adds new operations
    - adds per-object state

## inter-class declarations

(like “open classes”)

MoveTracking v6

```
aspect DisplayUpdating {  
    private Object FigureElement.lastMovedBy;  
    public Object FigureElement.getLastMovedBy() {  
        return lastMovedBy;  
    }  
  
    pointcut move(Object mover, FigureElement movee):  
        this(mover)    &&  
        target(movee) &&  
        (call(void Line.setP1(Point)) ||  
         call(void Line.setP2(Point)) ||  
         call(void Point.setX(int)) ||  
         call(void Point.setY(int)));  
  
    after(Object mover, FigureElement movee): move(mover, movee) {  
        Display.update(movee);  
        movee.lastMovedBy = mover;  
    }  
}
```

adds members to target type,  
“owned” by aspect type

public and private are  
with respect to enclosing  
aspect declaration

## there's more...

- more advice
  - before, after (3 kinds), around
- more pointcuts
  - fields, exception handlers, initialization...
- more inter-class declarations
  - declare error, warning, parents
- aspect extension, aspect libraries
- tools...

## expected benefits of using AOP

- good modularity,  
even for crosscutting concerns
  - less tangled code
  - more natural code
  - shorter code
  - easier maintenance and evolution
    - easier to reason about, debug, change
  - more reusable
    - library aspects
    - plug and play aspects when appropriate