Aspect-Oriented Programming with AspectJ™

AspectJ.org
Xerox PARC

Erik Hilsdale
Gregor Kiczales

with
Bill Griswold, Jim Hugunin, Wes Isberg, Mik Kersten

partially funded by DARPA under contract F30602-97-C0246

aspectj.org

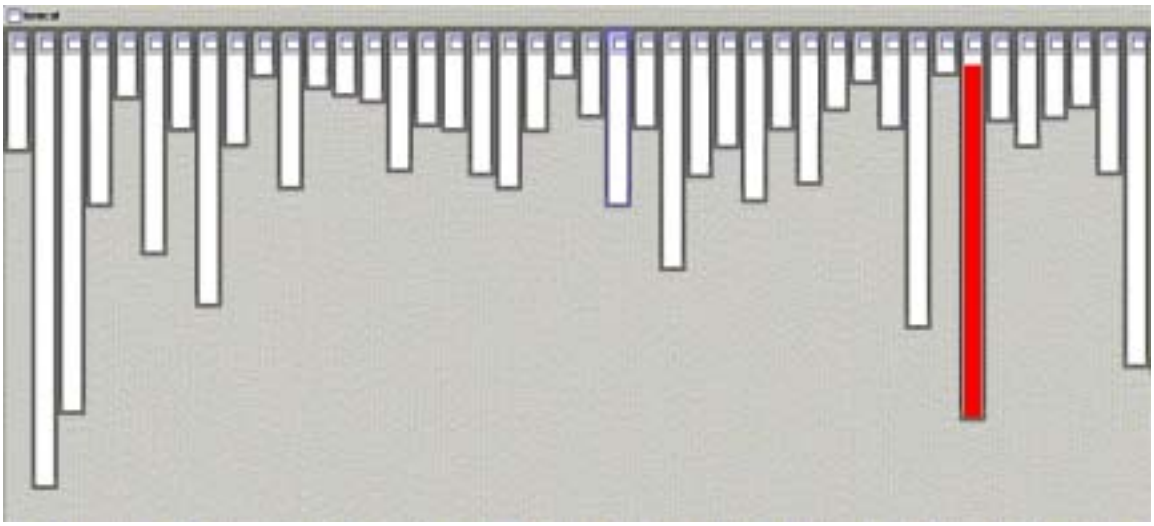
this tutorial is about...

- **using AOP and AspectJ to:**
  - improve the modularity of 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
- **AspectJ is:**
  - is an aspect-oriented extension to Java™ that supports general-purpose aspect-oriented programming

aspectj.org

good modularity

XML parsing

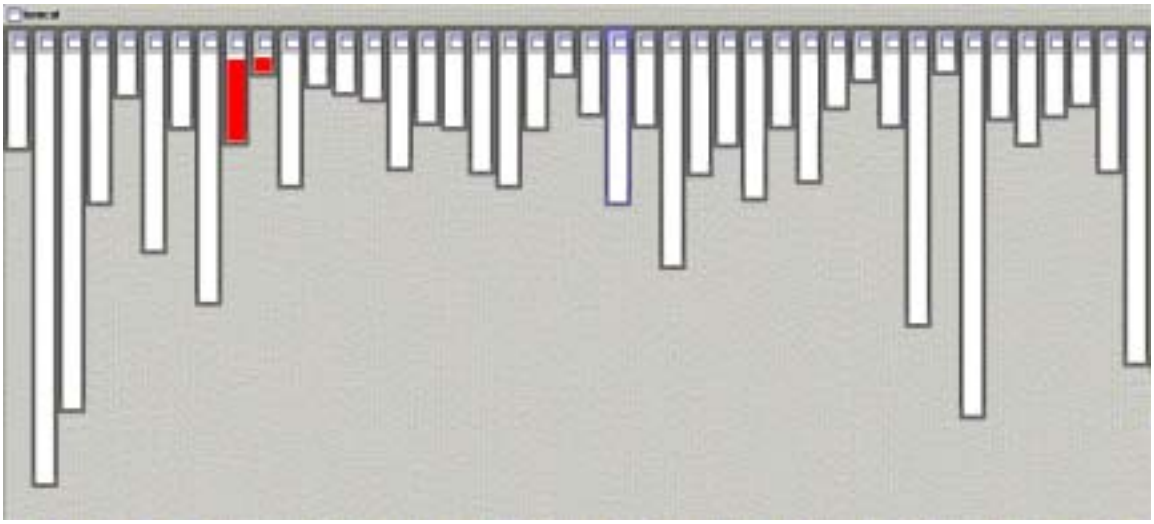


- **XML parsing in org.apache.tomcat**
  - red shows relevant lines of code
  - nicely fits in one box

aspectj.org

good modularity

URL pattern matching

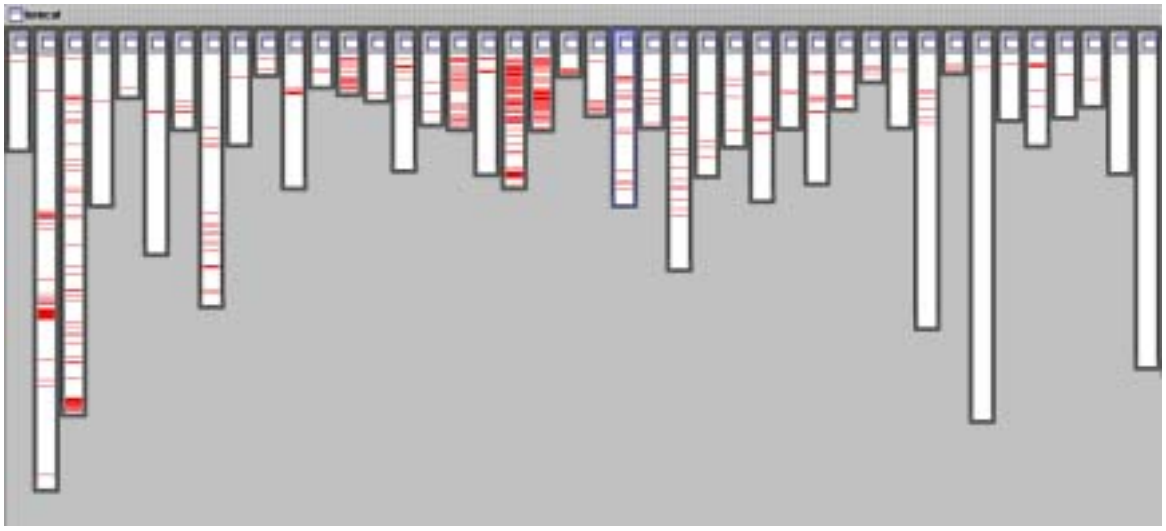


- **URL pattern matching in org.apache.tomcat**
  - red shows relevant lines of code
  - nicely fits in two boxes (using inheritance)

aspectj.org

problems like…

logging is not modularized



- **where is 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

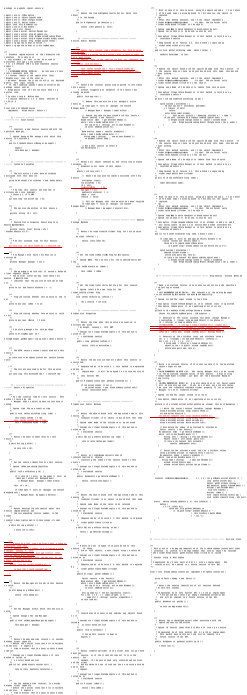
aspectj.org

problems like…

session expiration is not modularized

ApplicationSession

StandardSession



SessionInterceptor

StandardManager

StandardSessionManager

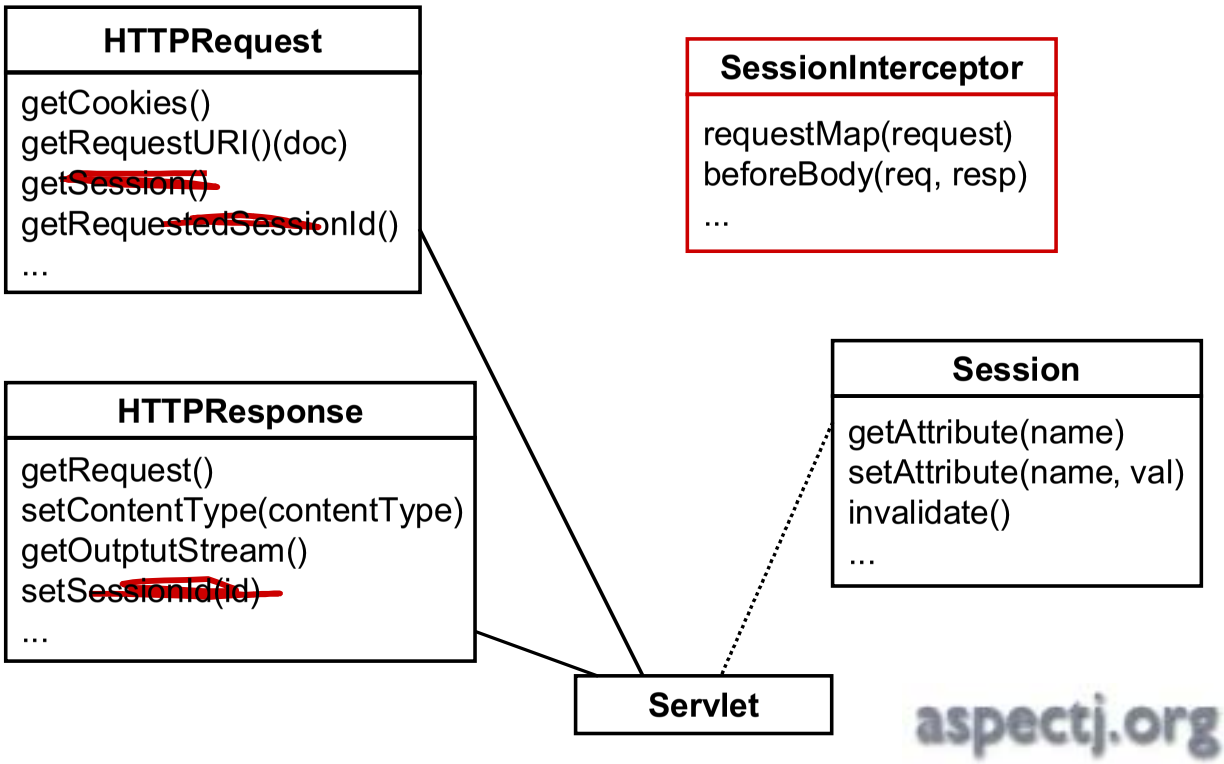
ServerSession

ServerSessionManager

aspectj.org

problems like…

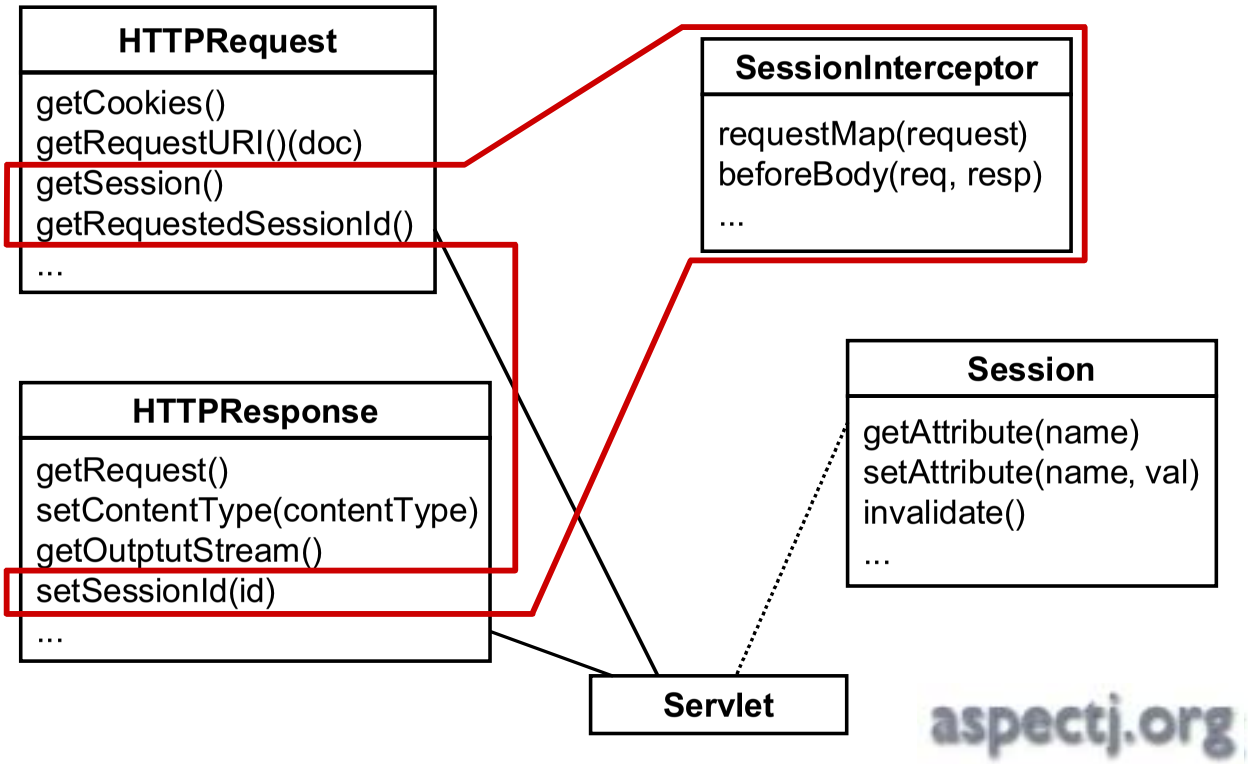
session tracking is not modularized



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

crosscutting concerns



aspectj.org

the AOP idea

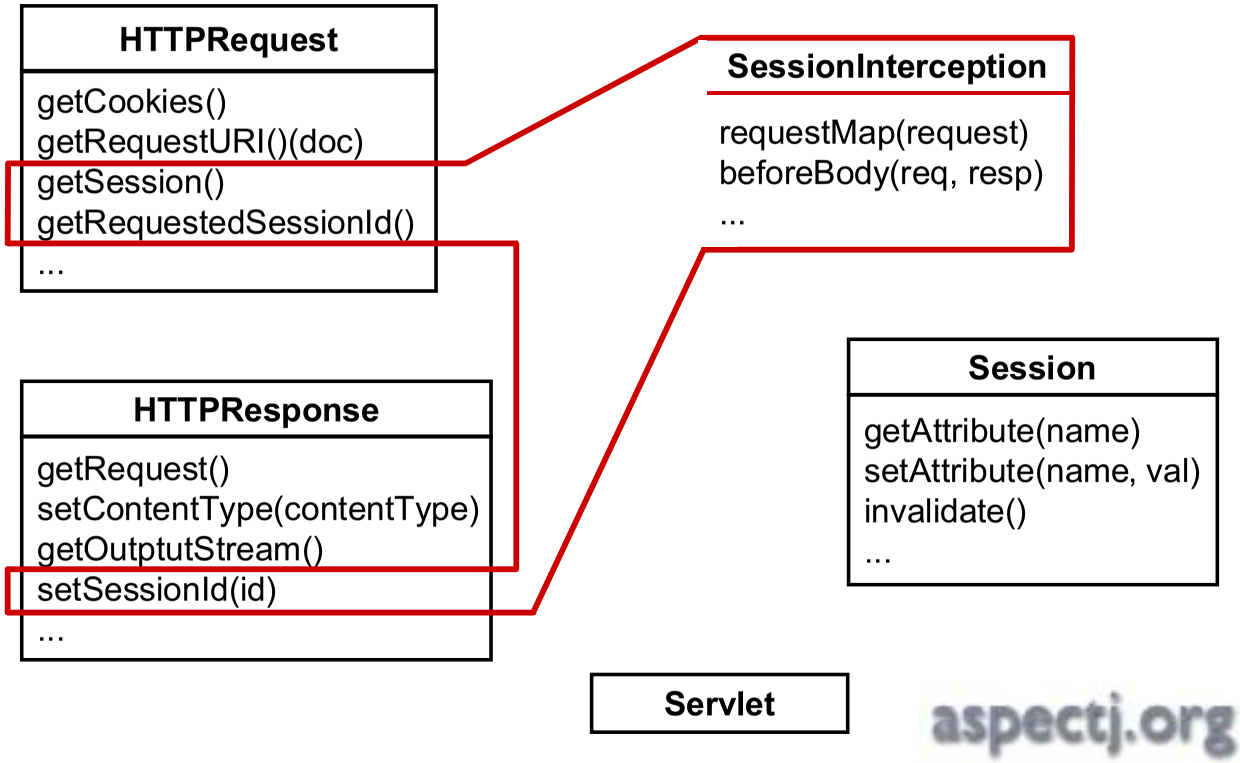
aspect-oriented programming

- **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.org

language support to…

modular aspects



AspectJ™ is…

- **a small and well-integrated extension to Java**
- **a general-purpose AO language**
  - just as Java is a general-purpose OO language
- **freely available implementation**
  - compiler is Open Source
- **includes IDE support**
  - emacs, JBuilder, Forte
- **user feedback is driving language design**
  - users@aspectj.org
  - support@aspectj.org
- **currently at 1.0b1 release**

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

outline

- **I AOP overview**
  - brief motivation, essence of AOP idea
- **II AspectJ language mechanisms**
  - basic concepts, language semantics
- **III development environment**
  - IDE support, running the compiler, debugging etc.
- **IV using aspects**
  - aspect examples, how to use AspectJ to program aspects, exercises to solidify the ideas
- **V related work**
  - survey of other activities in the AOP community

looking ahead

problem structure

Part IV:

crosscutting in the design, and
how to use AspectJ to capture that

AspectJ mechanisms

Part II:

crosscutting in the code
mechanisms AspectJ provides

Part II

Basic Mechanisms of AspectJ

aspectj.org

goals of this chapter

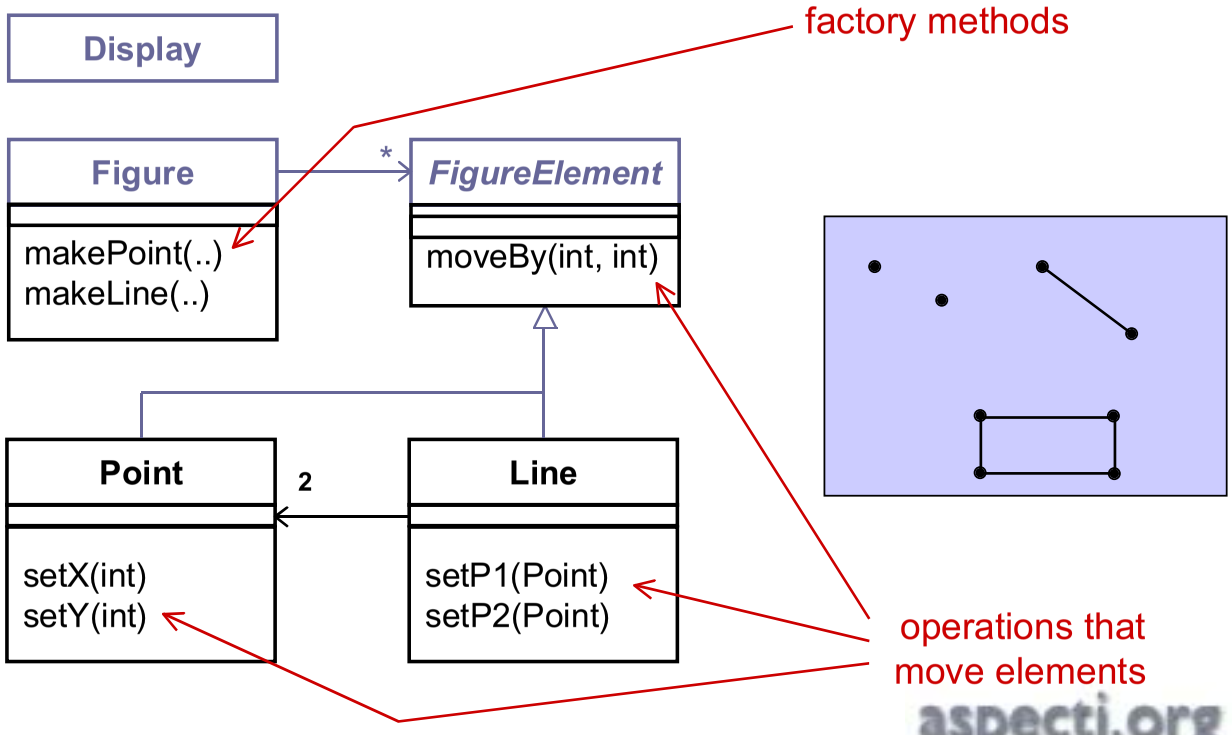
- **present basic language mechanisms**
  - using one simple example
    - emphasis on what the mechanisms do
    - small scale motivation
- **later chapters elaborate on**
  - environment, tools
  - larger examples, design and SE issues

aspectj.org

basic mechanisms

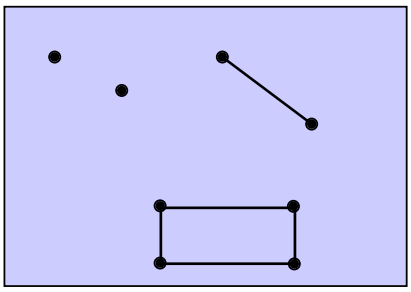
- **1 overlay onto Java**
  - dynamic join points
    - "points in the execution" of Java programs
- **4 small additions to Java**
  - pointcuts
    - pick out join points and values at those points
      - primitive pointcuts
      - user-defined pointcuts
  - advice
    - additional action to take at join points in a pointcut
  - intra-class declarations (aka "open classes")
  - aspect
    - a modular unit of crosscutting behavior
      - comprised of advice, intra-class declarations, field,constructor and method declarations

a simple figure editor



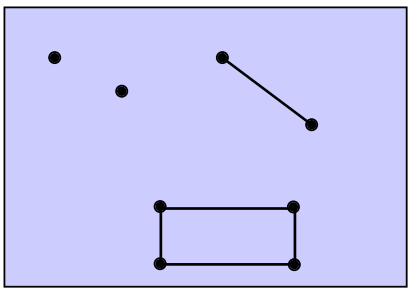
a simple figure editor

```
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

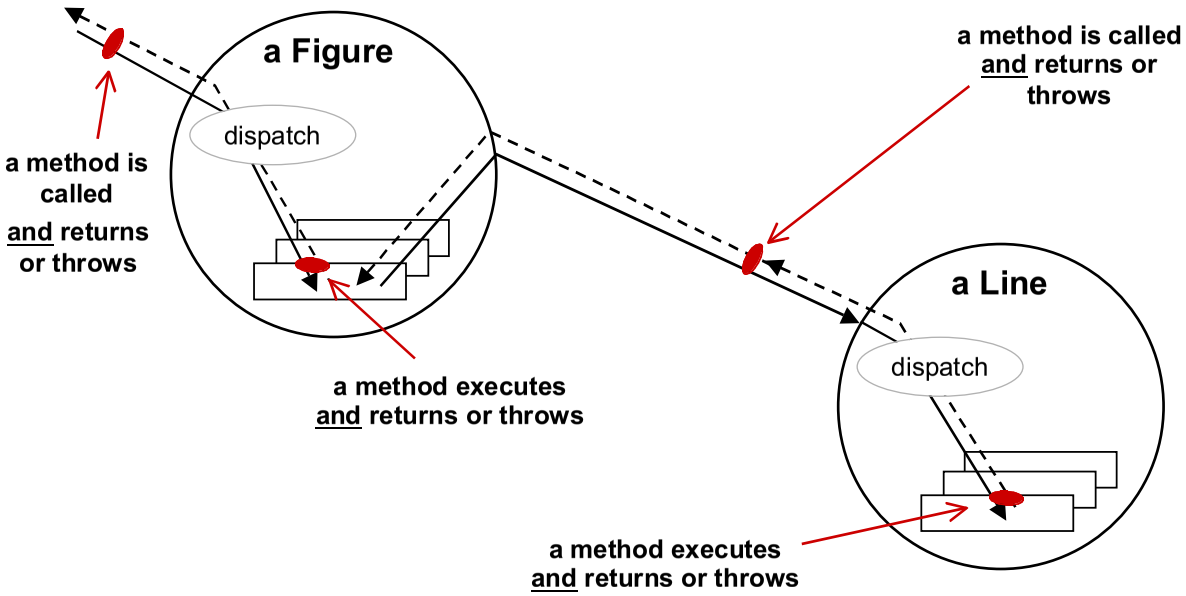
- **collection of figure elements**
  - that move periodically
  - must refresh the display as needed
  - complex collection
  - asynchronous events
- **other examples**
  - session liveness
  - value caching



we will initially assume just a single display

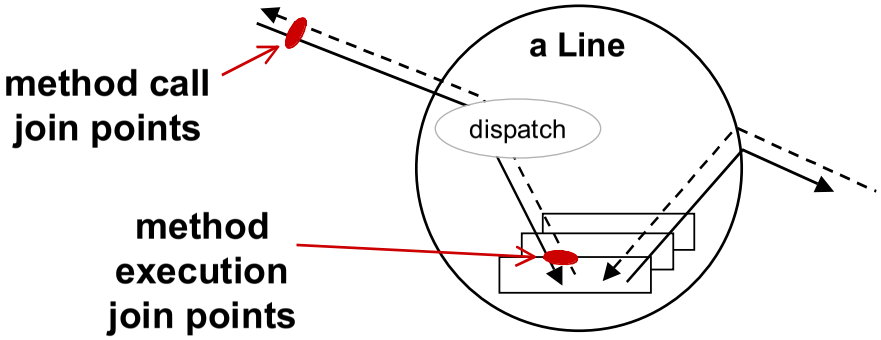
join points

key points in dynamic call graph



aspectj.org

join point terminology



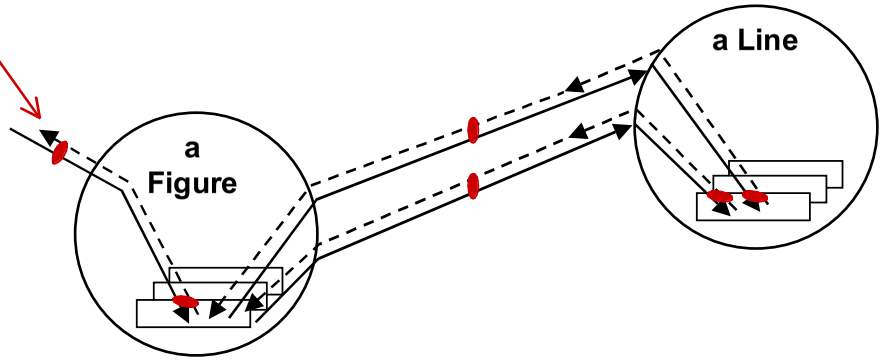
- **several kinds of join points**
  - method & constructor call join points
  - method & constructor execution join points
  - field get & set join points
  - exception handler execution join points
  - static & dynamic initialization join points

aspectj.org

join point terminology

key points in dynamic call graph

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



repeated calls result in new join points

primitive pointcuts

"a means of identifying join points"

a pointcut is like a predicate on join points that:

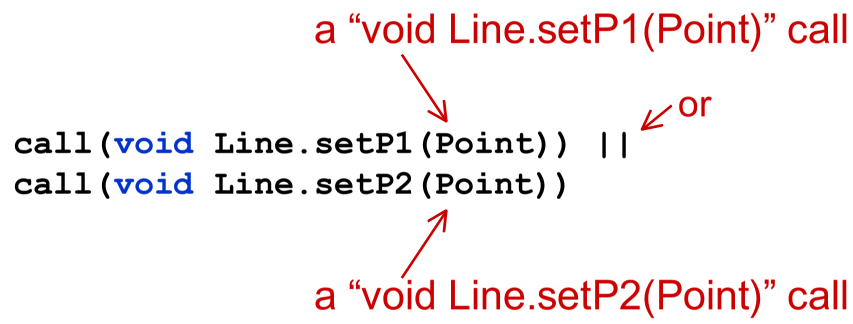
- can match or not match any given join point and
- optionally, can pull out some of the values at that join point

```
call(void Line.setP1(Point))
```

matches if the join point is a method call with this signature

pointcut composition

pointcuts compose like predicates, using &&, || and !



each time a Line receives a
“void setP1(Point)” or “void setP2(Point)” method call

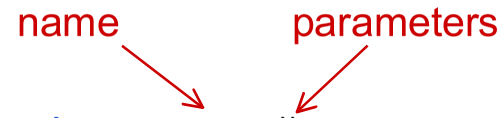
aspectj.org

user-defined pointcuts

defined use the pointcut construct

user-defined (aka named) pointcuts

- can be used in the same way as primitive pointcuts




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

more on parameters and how pointcuts can expose values at join points in a few slides

aspectj.org

pointcuts

summary so far

user-defined pointcut designator

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

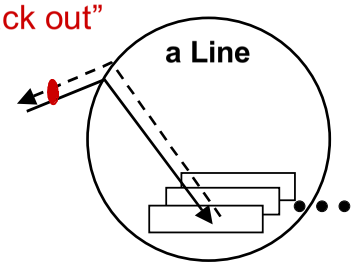
- primitive pointcut designator, can be:
  - **- call, execution** **- this, target, args**
  - **- get, set** **- within, withincode**
  - **- handler** **- cflow, cflowbelow**
  - **- initialization, staticinitialization**

aspectj.org

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>
}
```

aspectj.org

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

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();
  }
}
```

- **what you would expect**
  - update calls are tangled through the code
  - “what is going on” is less explicit

pointcuts

can cut across multiple classes

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


pointcuts

can use interface signatures

```
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));
```

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();
  }
}
```

aspectj.org

using values at join points

demonstrate first, explain in detail afterwards

- pointcut can explicitly expose certain values
- advice can use value

```
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) {
  <fe is bound to the figure element>
}
```

parameter mechanism being used

aspectj.org

explaining parameters…

of pointcut designators

- **in this, target & args pointcut designators**
  - typed variable pulls corresponding value out of join points
- **variable bound in user-defined pointcut designator**
  - makes value accessible on pointcut

pointcut parameters

typed variable

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

```
after(Line line): move(line) {
  <line is bound to the line>
}
```

aspectj.org

explaining parameters…

of advice

- **variable bound in advice**
- **variable in place of type name in pointcut designator**
  - pulls corresponding value out of join points
  - makes value accessible within advice

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

advice parameters

typed variable

```
after(Line line): move(line) {
  <line is bound to the line>
}
```

aspectj.org

explaining parameters…

- **value is 'pulled'**
  - right to left across ':' left side : right side
  - from pointcut designators to user-defined pointcut designators
  - from pointcut to advice

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

```
after(Line line): move(line) {
   <line is bound to the line>
}
```


target

primitive pointcut designator

`target(<type name>)`

- **any join point at which**
- **target object is an instance of type (or class) name**

- `target(Point)`
- `target(Line)`
- `target(FigureElement)`

- **"any join point" means it matches join points of all kinds**
  - method & constructor call join points
  - method & constructor execution join points
  - field get & set join points
  - exception handler execution join points
  - static & dynamic initialization join points

an idiom for…

getting target object in a polymorphic pointcut

```
target(<supertype name>)  &&
```

- does not further restrict the join points
- does pick up the target object

```
pointcut move(FigureElement figElt):
  target(figElt)  &&
  (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) {
  <fe is bound to the figure element>
}
```


context & multiple classes

DisplayUpdating v3

```
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);
  }
}
```

without 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;

  }
}
```


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;

  }
}
```

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();
  }
}
```


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);
  }
}
```

- **non-modular "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);
  }
}
```

- **modular “display updating” aspect**
  - all changes in single aspect
  - evolution is modular

advice is

additional action to take at join points

- **before** — before proceeding at join point
- **after returning** — a value to join point
- **after throwing** — a throwable to join point
- **after** — returning to join point either way
- **around** — on arrival at join point gets explicit control over when&if program proceeds

contract checking

simple example of before/after/around

- **pre-conditions**
  - check whether parameter is valid
- **post-conditions**
  - check whether values were set
- **condition enforcement**
  - force parameters to be valid

pre-condition

using before advice

```
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);
  }

  private void assert(boolean v) {
    if ( !v )
      throw new RuntimeException();
  }
}
```

what follows the ':' is always a pointcut – primitive or user-defined

aspectj.org

post-condition

using after advice

```
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);
  }

  private void assert(boolean v) {
    if ( !v )
      throw new RuntimeException();
  }
}
```

aspectj.org

condition enforcement

using around advice

```
aspect PointBoundsEnforcement {

  void around(Point p, int newX):
      call(void Point.setX(int)) && target(p) && args(newX) {
    proceed(p, clip(newX, MIN_X, MAX_X));
  }

  void around(Point p, int newY):
      call(void Point.setY(int)) && target(p) && args(newY) {
    proceed(p, clip(newY, MIN_Y, MAX_Y));
  }

  private int clip(int val, int min, int max) {
    return Math.max(min, Math.min(max, val));
  }
}
```


special static method

```
<result type> proceed(arg1, arg2, …)
```

available only in around advice

means “run what would have run if this around advice had not been defined”

other primitive pointcuts

```
this(<type name>)
within(<type name>)
withincode(<method/constructor signature>)
```

any join point at which
- currently executing object is an instance of type or class name
- currently executing code is contained within class name
- currently executing code is specified method or constructor

```
get(int Point.x)
set(int Point.x)
```

field reference or assignment join points

fine-grained protection

a runtime error

```
class Figure {
  public Point makeLine(Line p1, Line p2) { new Line...  }
  public Point makePoint(int x, int y)    { new Point... }
  ...
}
```

want to ensure that figure elements are only constructed using the factory methods

more on wildcards in 3 slides

```
aspect FactoryEnforcement {
  pointcut illegalNewFigElt():
         (call(Point.new(..)) || call(Line.new(..))) &&
        !withincode(* Figure.make*(..));

  before(): illegalNewFigElt() {
    throw new Error("Use factory method instead.");
  }
}
```

fine-grained protection

a compile-time error

```
class Figure {
  public Point makeLine(Line p1, Line p2) { new Line...  }
  public Point makePoint(int x, int y)    { new Point... }
  ...
}
```

want to ensure that figure elements are only constructed using the factory methods

```
aspect FactoryEnforcement {
  pointcut illegalNewFigElt():
         (call(Point.new(..)) || call(Line.new(..))) &&
        !withincode(* Figure.make*(..));

  declare error: illegalNewFigElt()
    "Use factory method instead.");
  }
}
```

must be a static pointcut (more on this later)

aspectj.org

fine-grained protection

as a static inner class

```
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) { ... }

  static aspect SetterEnforcement {
    declare error: set(Point Line.*) &&
                   !withincode(void Line.setP*(Point))
      "Use setter method, even inside Line class.";
}
```

common idiom when scope of enforcement aspect is a single class, and you never want to unplug it (more on unplugging aspects later)

aspectj.org

wildcarding in pointcuts

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

```
target(Point)
target(graphics.geom.Point)
target(graphics.geom.*)
target(graphics..*)
```

any type in graphics.geom
any type in any sub-package of graphics

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

any public method on Point
any public method on any type

```
call(void Point.getX())
call(void Point.getY())
call(void Point.get*())
call(void get*())
```

any getter

```
call(Point.new(int, int))
call(new(..))
```

any constructor

aspectj.org

property-based crosscutting

```
package com.xerox.pri
public class C1 {
  …
  public void foo() {
    A.doSomething(…);
    …
  }
  …
}
```

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

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

- **crosscuts of 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

aspectj.org

property-based crosscutting

```
aspect PublicErrorLogging {

  Log log = new Log();

  pointcut publicCall():
    call(public * com.xerox..*.*(..));

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

means we are at a public call to com.xerox.. 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

special value

reflective* access to the join point

```
thisJoinPoint.
      Signature   getSignature()
      Object[]    getArgs()
      ...
```

available in any advice

thisJoinPoint is abbreviated to 'tjp' in these slides to save slide space

* introspective subset of reflection consistent with Java

using thisJoinPoint

in highly polymorphic advice

```
aspect PointCoordinateTracing {

  before(Point p, int newVal): set(int Point.*) &&
                               target(p) &&
                               args(newVal) {
    System.out.println("At " +
                       tjp.getSignature() +
                       " field is set to " +
                       newVal +
                       ".");
  }
}
```

using thisJoinPoint makes it possible for advice to recover information about where it is running

other primitive pointcuts

`execution(void Point.setX(int))`
method/constructor execution join points (actual running method)

`initialization(Point)`
object initialization join points

`staticinitialization(Point)`
class initialization join points (as the class is loaded)

other primitive pointcuts

`cflow(`*`pointcut designator`*`)`
all join points within the dynamic control flow of any join point in *pointcut designator*

`cflowbelow(`*`pointcut designator`*`)`
all join points within the dynamic control flow below any join point in *pointcut designator*

context sensitive aspects

what happens when moveBy is called on a Line?

```
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);
  }
}
```

context sensitive aspects

DisplayUpdating v4

```
aspect DisplayUpdating {

  pointcut move(FigureElement figElt):
    target(figElt) &&
    (call(void FigureElem
     call(void Line.setP1
     call(void Line.setP2
     call(void Point.setX(int))    ||
     call(void Point.setY(int)));

  pointcut topLevelMove(FigureElement figElt):
    move(figElt) && !cflowbelow(move(FigureElement));

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

the non-reentrant calls idiom:

```
ptc && !cflowbelow(ptc)
```


intra-type declarations[1]

DisplayUpdating v5

```
aspect DisplayUpdating {

  private Display FigureElement.display;

  static void setDisplay(FigureElement fe, Display d) {
    fe.display = d;
  }

  pointcut move(FigureElement figElt):
    <as before>;

  after(FigureElement fe): move(fe) {
    fe.display.update(fe);
  }
}
```

adds members to target interface/class

1. recently termed “open classes”

field/getter/setter idiom

```
aspect DisplayUpdating {

  private Display FigureElement.display;

  static void setDisplay(FigureElement fe, Display d) {
    fe.display = d;
  }

  pointcut
    <as bef

  after(Fig
    fe.display.update(fe);
  }
}
```

private with respect to enclosing aspect declaration

the display field

- is a field of FigureElement, but
- belongs to DisplayUpdating aspect
- so DisplayUpdating aspect should provide getter/setter

a "GOF pattern"

```
aspect DisplayUpdating {

  private List FigureElement.observers = new LinkedList();

  static void addDisplay(FigureElement fe, Display d) {
    fe.observers.add(d);
  }
  static void removeDisplay(FigureElement fe, Display d) {
    fe.observers.remove(d);
  }

  pointcut move(FigureElement figElt):
    <as before>;

  after(FigureElement fe): move(fe) {
    Iterator iter = fe.observers.iterator();
    ...
  }
}
```

another pattern

```
aspect XXXReaderWriterSynchronization {

  pointcut readers(Object o): …;
  pointcut writers(Object o): …;

  before(Object o): readers(o) { ... }
  after (Object o): readers(o) { ... }
  before(Object o): writers(o) { ... }
  after (Object o): readers(o) { ... }

}
```

from Doug Lea's concurrent programming in Java

- **a common synchronization pattern**

another pattern

```
aspect RegistryReaderWriterSynchronization {

  pointcut readers(Object o):
      call(Iterator Display.iterator()) ||
      call(FigureElement getElementNear(int, int));

  pointcut writers(Object o):
      call(void Display.addElement(FigureElement)) ||
      call(void Display.removeElement(FigureElement));

  before(Object o): readers(o) { ... }
  after (Object o): readers(o) { ... }
  before(Object o): writers(o) { ... }
  after (Object o): readers(o) { ... }

}
```

inheritance & specialization

- **pointcuts can have additional advice**
  - aspect with
    - concrete pointcut
    - perhaps no advice on the pointcut
  - in figure editor
    - `move()` can have advice from multiple aspects
  - module can expose certain well-defined pointcuts
- **abstract pointcuts can be specialized**
  - aspect with
    - abstract pointcut
    - concrete advice on the abstract pointcut

a shared pointcut

```
public class FigureEditor {
   ...
   pointcut move(FigureElement figElt):
     target(figElt) && ...;
   ...
}


aspect DisplayUpdating {
   ...
   after(FigureElement fe) returning:
       FigureEditor.move(fe) {
     ...
   }
   ...
}
```

a reusable aspect

```
abstract public aspect RemoteExceptionLogging {

  abstract pointcut logPoint();

  after() throwing (RemoteException e): logPoint() {
    log.println("Remote call failed in: " +
                thisJoinPoint.toString()  +
                "(" + e + ").");
  }
}
```

abstract

```
public aspect MyRMILogging extends RemoteExceptionLogging {
  pointcut logPoint():
    call(* RegistryServer.*.*(..)) ||
    call(private * RMIMessageBrokerImpl.*.*(..));
}
```

aspectj.org

an aspect as a reusable pattern

```
aspect RWSynchronization {

  abstract pointcut readers(Object o);
  abstract pointcut writers(Object o);

  before(Object o): readers(o) { ... }
  after (Object o): readers(o) { ... }
  before(Object o): writers(o) { ... }
  after (Object o): readers(o) { ... }
}
```

```
aspect MumbleRWSynchronization extends RWSynchronization {

  pointcut readers(Object o): …;
  pointcut writers(Object o): …;

}
```

aspectj.org

aspect instances

```
aspect PublicErrorLogging
    pertarget(PublicErrorLogging.publicInterface()) {

  Log log = new Log();

  pointcut publicInterface():
    call(public * com.xerox..*.*(..));

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

one instance of the aspect for each object that ever executes at these points

looking up aspect instances

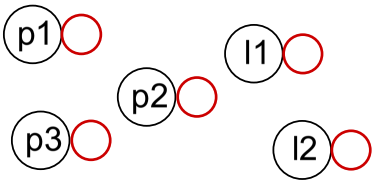
```
    :

 static Log getLog(Object obj) {
    return (PublicErrorLogging.aspectOf(obj)).log;
  }
}
```

- **static method of aspects**
  - for default aspects takes no argument
  - for aspects of pertarget/perthis takes an Object
  - for aspects of percflow takes no arguments
- **returns aspect instance**

aspect relations

`pertarget(<pointcut>)`
`perthis(<pointcut>)`
one aspect instance for each object that is ever "this" at the join points

`percflow(<pointcut>)`
`percflowbelow(<pointcut>)`
one aspect instance for each join point in pointcut, is available at all joinpoints in cflow or cflowbelow



aspectj.org

context sensitive aspects

DisplayUpdating v4

```
aspect DisplayUpdating {
  List movers = new LinkedList();
  List movees = new LinkedList();
  // …

  pointcut moveCall(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) returning:
      moveCall(mover, movee) {
    movers.add(mover);
    movees.add(movee);
  }
}
```


aspectj.org

summary

join points

- method & constructor
  - call
  - execution
- field
  - get
  - set
- exception handler
  - execution
- initialization

aspects

- crosscutting type
  - pertarget
  - perthis
  - percflow
  - percflowbelow

pointcuts

-primitive-

- call execution
- handler
- get set
- initialization
- this target args
- within withincode
- cflow cflowbelow

-user-defined-

- pointcut declaration
- abstract
- overriding

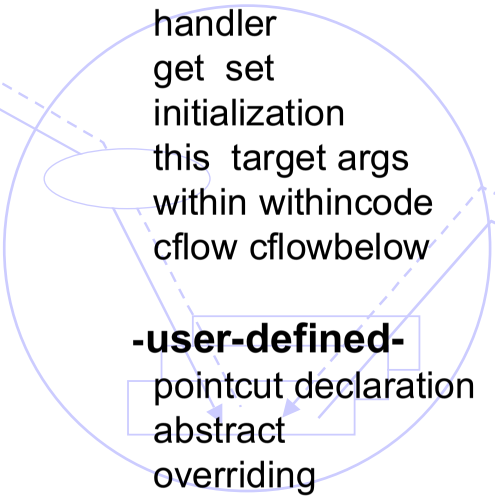
advice

- before
- after
- around

declare

- error

intra-type decls



where we have been…

… and where we are going

problem structure

↓

Part IV:

crosscutting in the design, and how to use AspectJ to capture that

↑

AspectJ mechanisms

Part II:

crosscutting in the code mechanisms AspectJ provides

Part III

AspectJ IDE support

aspectj.org

programming environment

- **AJDE support for**
  - emacs, JBuilder, Forte
- **debugger**
  - jdb style (ajdb)
  - window-based
- **ajdoc variant of javadoc**

- **show examples of**
  - navigating AspectJ code
  - compiling, tracking errors
  - debugging
  - using ajdoc

aspectj.org

Part IV

Using Aspects

aspectj.org

goals of this chapter

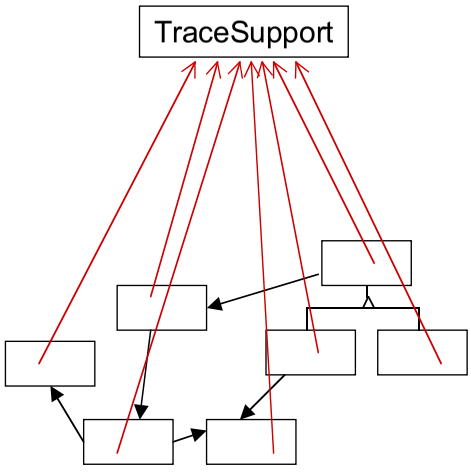
- **present examples of aspects in design**
  - intuitions for identifying aspects
- **present implementations in AspectJ**
  - how language support helps
- **work on implementations in AspectJ**
  - putting AspectJ into practice
- **raise some style issues**
  - objects vs. aspects
  - when are aspects appropriate?

aspectj.org

example – plug&play tracing

- **simple tracing**
  - exposes join points and uses very simple advice
- **an unpluggable aspect**
  - core program functionality is unaffected by the aspect

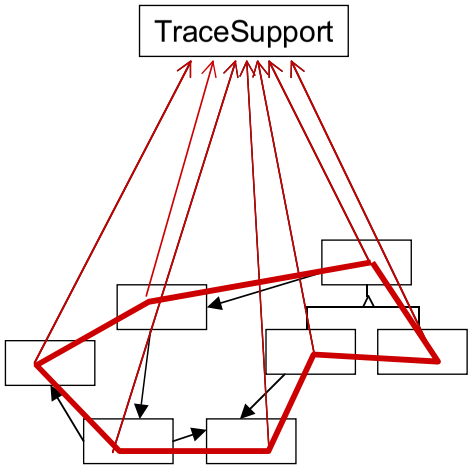
tracing without AspectJ




```
class TraceSupport {
  static int TRACELEVEL = 0;
  static protected PrintStream stream = null;
  static protected int callDepth = -1;

  static void init(PrintStream _s) {stream=_s;}

  static void traceEntry(String str) {
    if (TRACELEVEL == 0) return;
    callDepth++;
    printEntering(str);
  }
  static void traceExit(String str) {
    if (TRACELEVEL == 0) return;
    callDepth--;
    printExiting(str);
  }
}
```

```
class Point {
  void set(int x, int y) {
    TraceSupport.traceEntry("Point.set");
    _x = x; _y = y;
    TraceSupport.traceExit("Point.set");
  }
}
```

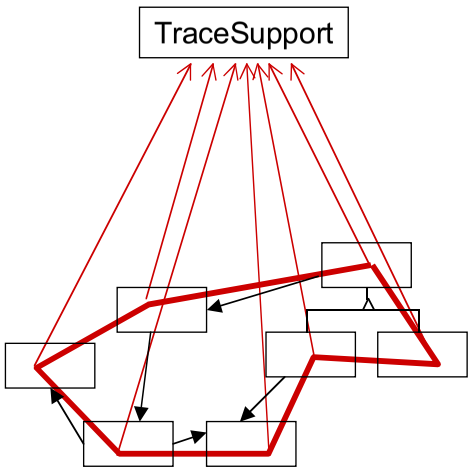
a clear crosscutting structure



all modules of the system use the trace facility in a consistent way: entering the methods and exiting the methods

this line is about interacting with the trace facility

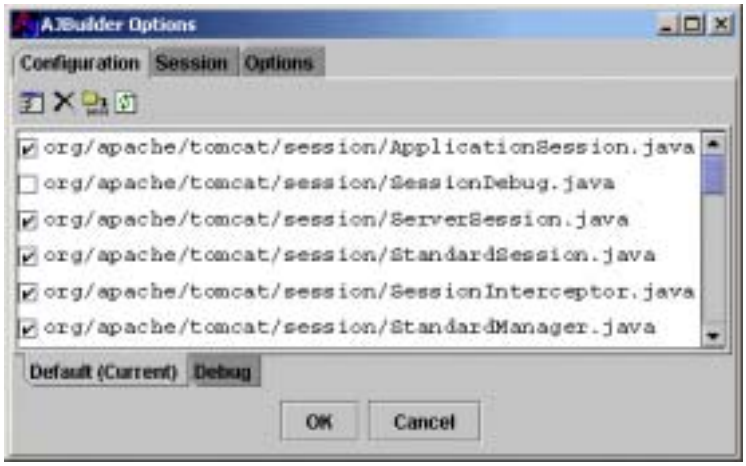
tracing as an aspect




```
aspect MyTracing {

  pointcut trace():
       within(com.bigboxco.boxes.*) &&
       execution(* *(..));

  before(): trace() {
    TraceSupport.traceEntry(
      thisJoinPoint.getSignature());
  }
  after(): trace() {
    TraceSupport.traceExit(
      thisJoinPoint.getSignature());
  }
}
```

plug and debug

- **plug in:** `ajc Point.java Line.java TraceSupport.java MyTracing.java`
- **unplug:** `ajc Point.java Line.java`
- **or…**



aspectj.org

plug and debug

//From ContextManager

```
public void service( Request rrequest, Response rresponse ) {
 // log( "New  request " + rrequest );
 try {
     //     System.out.print("A");
     rrequest.setContextManager( this );
     rrequest.setResponse(rresponse);
     rresponse.setRequest(rrequest);

     // wront 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;
}
```

```
// log( "New  request " + rrequest );

// System.out.print("A");

// System.out.print("B");

if(debug>0)
    log("Error closing request " + ex);

// log("Done with request " + rrequest);

// System.out.print("C");
```


aspectj.org

plug and debug

- **turn debugging on/off**
  - without editing classes
  - with no runtime cost
- **can save debugging code between uses**
- **can be used for profiling, logging**
- **easy to be sure it is off**
  - simply compile without aspects

aspects in the design

have these benefits

- **objects are not responsible for using TraceSupport**
  - XXXTracings aspect encapsulates that responsibility
    - MyTracing, DisplayTracing etc.
- **objects have no knowledge of TraceSupport**
  - are shielded from changes to TraceSupport interface
  - only XXXTracing aspect is affected
- **removing tracing from the design is trivial**
  - just remove the XXXTracing aspects and TraceSupport class

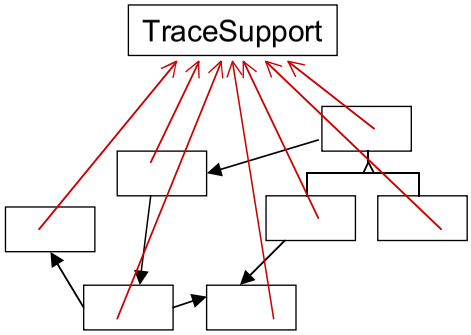
aspects in the code

have these benefits

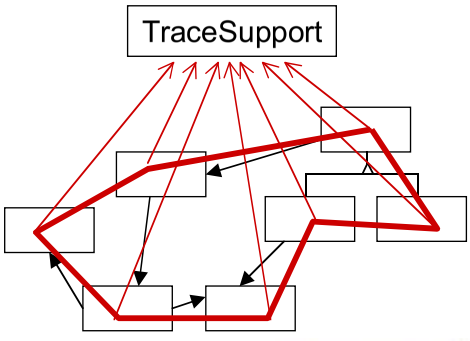
- **classes contain no calls to trace methods**
  - XXXTracing aspects encapsulate those calls
    - MyTracing, DisplayTracing etc.
- **if the TraceSupport interface changes, there is no need to modify the object classes**
  - only the trace aspect class needs to be modified
- **removing tracing from the application is trivial**
  - compile without the trace aspect class

tracing: object vs. aspect

- **using an object captures tracing support, but does not capture its consistent usage by other objects**



- **using an aspect captures the consistent usage of the tracing support by the objects**



a Tracing library aspect

Tracing exercise 1

- **refactor MyTracing into**
  - a reusable (library) aspect Tracing and
  - an extension equivalent to MyTracing

exercise

refactor MyTracing into a reusable (library) aspect and an extension equivalent to MyTracing

```
abstract aspect Tracing {
  // what goes here?




}
```

```
aspect MyTracing extends Tracing {
  // what goes here?




}
```

exercise

define one advice for normal return, a second for abrupt return

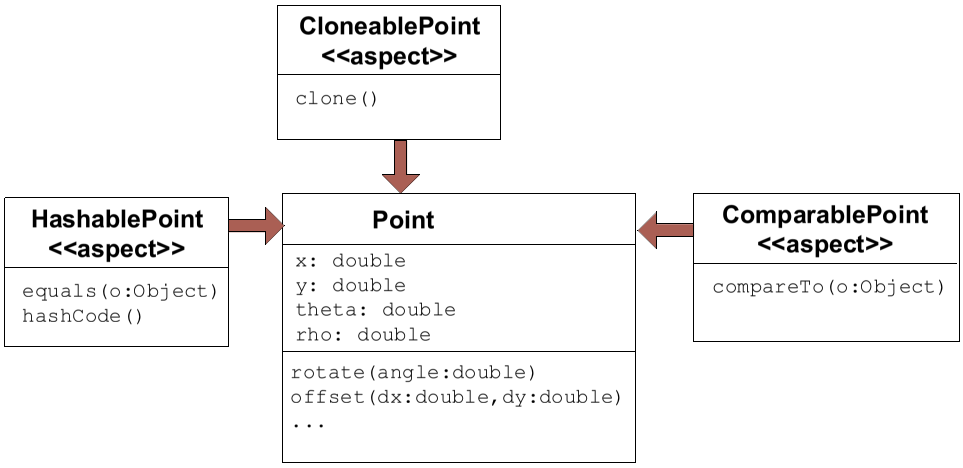
```
abstract aspect Tracing {
  // what goes here?




}
```

```
aspect MyTracing extends Tracing {
  // what goes here?




}
```


discussion

- what, at the design level, does each of these aspects implement?
- did you retain TraceSupport class?
- what do you think of the names Tracing and MyTracing for these aspects? what did you name the abstract pointcut
- can you combine the techniques of OO frameworks with these aspects to make the Tracing aspect even more flexible? is it a good idea to do so?
- …

example – roles/views



CloneablePoint

```
aspect CloneablePoint {

    declare parents: Point implements Cloneable;

    public Object Point.clone() throws CloneNotSupportedException {
      // we choose to bring all fields up to date before cloning
      makeRectangular();      // defined in class Point
      makePolar();            // defined in class Point
      return super.clone();
    }
}
```

roles/views

exercise/discussion

- Write the HashablePoint and ComparablePoint aspects.
- Consider a more complex system. Would you want the HashablePoint aspect associated with the Point class, or with other HashableX objects, or both?

example – counting bytes

```
interface OutputStream {
    public void write(byte   b);
    public void write(byte[] b);
}

/**
 * This SIMPLE aspect keeps a global count of all
 * the bytes ever written to any OutputStream.
 */
aspect ByteCounting {

    int count = 0;
    int getCount() { return count; }

    //                  //
    // what goes here? //
    //                  //
}
```

exercise

complete the code for `ByteCounting`

```
/**
 * This SIMPLE aspect keeps a global count of all
 * all the bytes ever written to an OutputStream.
 */
aspect ByteCounting {

    int count = 0;
    int getCount() { return count; }









}
```


counting bytes v1

a first attempt

```
aspect ByteCounting {

    int count = 0;
    int getCount() { return count; }

    after() returning:
            call(void OutputStream.write(byte)) {
       count = count + 1;
    }

    after(byte[] bytes) returning:
            call(void OutputStream.write(bytes)) {
       count = count + bytes.length;
    }
}
```

counting bytes

some stream implementations

```
class SimpleOutputStream implements OutputStream {
    public void write(byte b) { … }

    public void write(byte[] b) {
       for (int i = 0; i < b.length; i++) write(b[i]);
    }
}


class OneOutputStream implements OutputStream {
    public void write(byte b) { … }

    public void write(byte[] b) { … }
}
```


counting bytes

another implementation

```
class OtherOutputStream implements OutputStream {
    public void write(byte b) {
       byte[] bs = new byte[1] { b };
       write(bs);
    }

    public void write(byte[] b) { … }
}
```

exercise

make a more robust implementation

```
aspect ByteCounting {

    int count = 0;
    int getCount() { return count; }

    pointcut write(): call(void OutputStream.write(byte)) ||
                      call(void OutputStream.write(byte[]));



    after() returning:
            call(void OutputStream .write(byte))                    {
        count++;
    }

    after(byte[] bytes) returning:
            call(void OutputStream .write(bytes))                   {
        count = count + bytes.length;
    }
}
```


counting bytes v2

using cflowbelow for more robust counting

```
aspect ByteCounting {

    int count = 0;
    int getCount() { return count; }

    pointcut write(): call(void OutputStream.write(byte)) ||
                      call(void OutputStream.write(byte[]));

    pointcut writeCflow(): cflowbelow(write());

    after() returning:
            !writeCflow() && call(void OutputStream .write(byte)) {
        count++;
    }

    after(byte[] bytes) returning:
            !writeCflow() && call(void OutputStream .write(bytes)) {
        count = count + bytes.length;
    }
}
```

exercise

per-stream counting

```
aspect ByteCounting {

  /* put declarations here that provide per-stream count state */









  pointcut write(): call(void OutputStream.write(byte)) ||
                    call(void OutputStream.write(byte[]));


  /* ... and show how to properly increment the count state */

  ... count = count + 1;

  ... count = count + bytes.length;

}
```

aspectj.org

counting bytes v3

per-stream counting

```
aspect ByteCounting {

  /* put declarations here that provide per-stream count state */

  private int OutputStream.count = 0;



  static public int getCount(OutputStream os) { return os.count; }
  public int OutputStream.getCount() { return count; }


  pointcut write(): call(void Output
                    call(void Output


  /* ... and show how to properly in

  ... s.count = s.count + 1;

  ... s.count = s.count + bytes.length;

}
```

third party perspective:

```
ByteCounting.getCount(s);

s.getCount();
```

aspectj.org

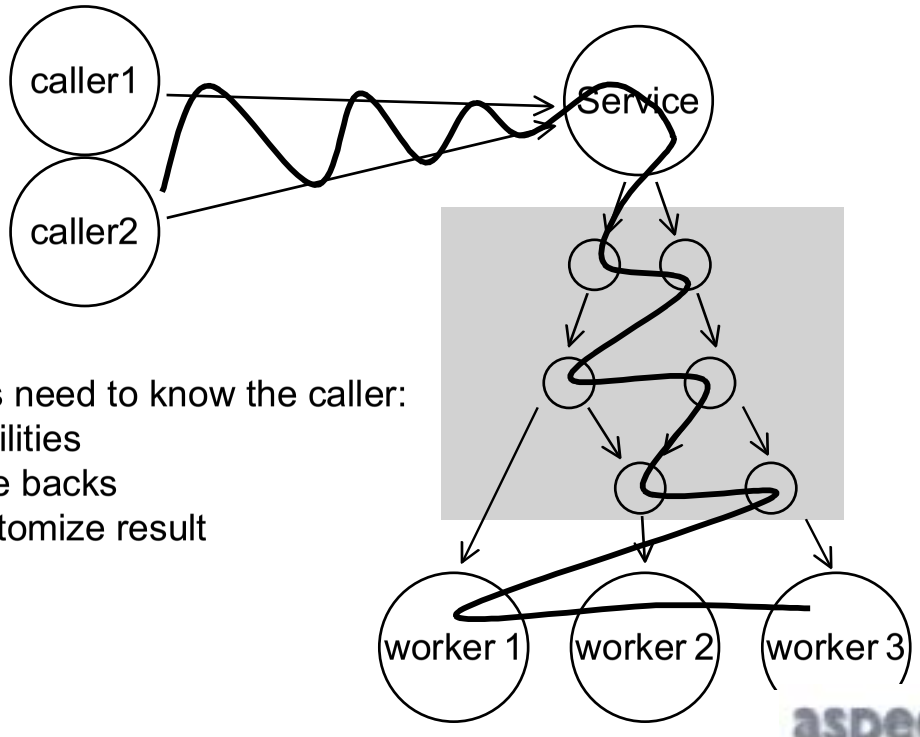
counting bytes

exercises

- How do the aspects change if the method `void write(Collection c)` is added to the OutputStream interface?
- How would you change v2 to handle byte generators:

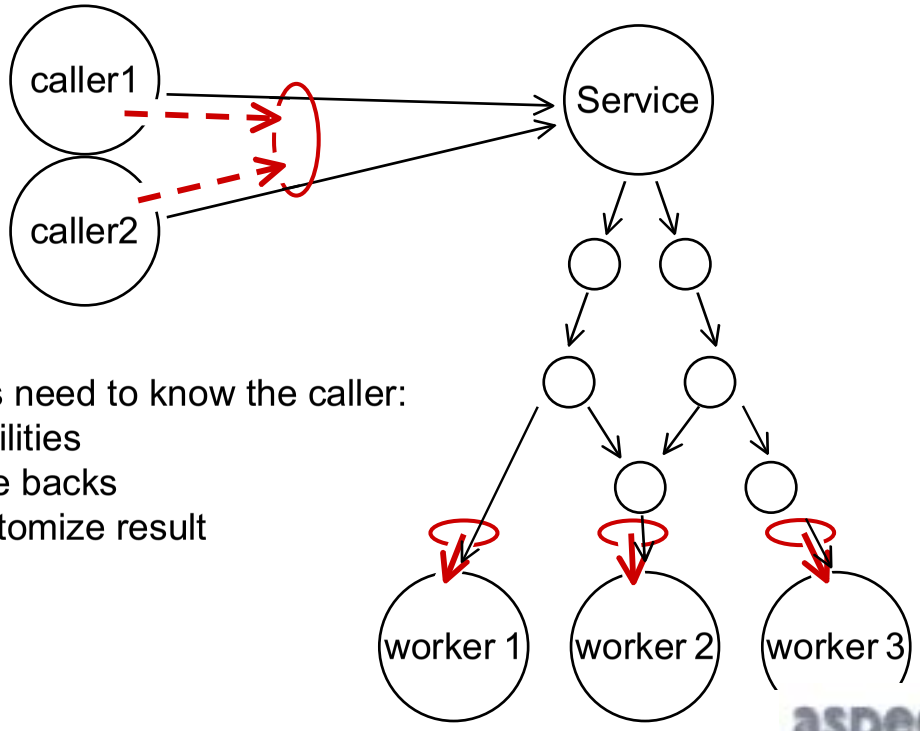
```
interface ByteGenerator {
  int getLength();
  void generateTo(OutputStream s);
}
```


example – context passing



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

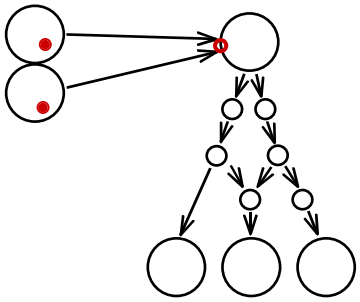
context-passing aspects



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

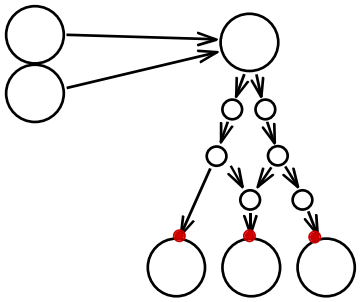
context-passing aspects

```
pointcut invocations(Caller c):
  this(c) && call(void Service.doService(String));
```



context-passing aspects

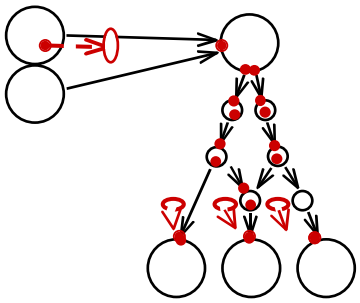
```
pointcut invocations(Caller c):
  this(c) && call(void Service.doService(String));

pointcut workPoints(Worker w):
  target(w) && call(void Worker.doTask(Task));
```



aspectj.org

context-passing aspects

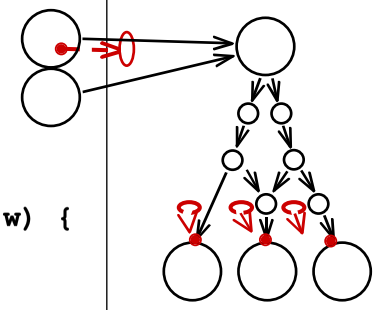
```
pointcut invocations(Caller c):
  this(c) && call(void Service.doService(String));

pointcut workPoints(Worker w):
  target(w) && call(void Worker.doTask(Task));

pointcut perCallerWork(Caller c, Worker w):
  cflow(invocations(c)) && workPoints(w);
```



aspectj.org

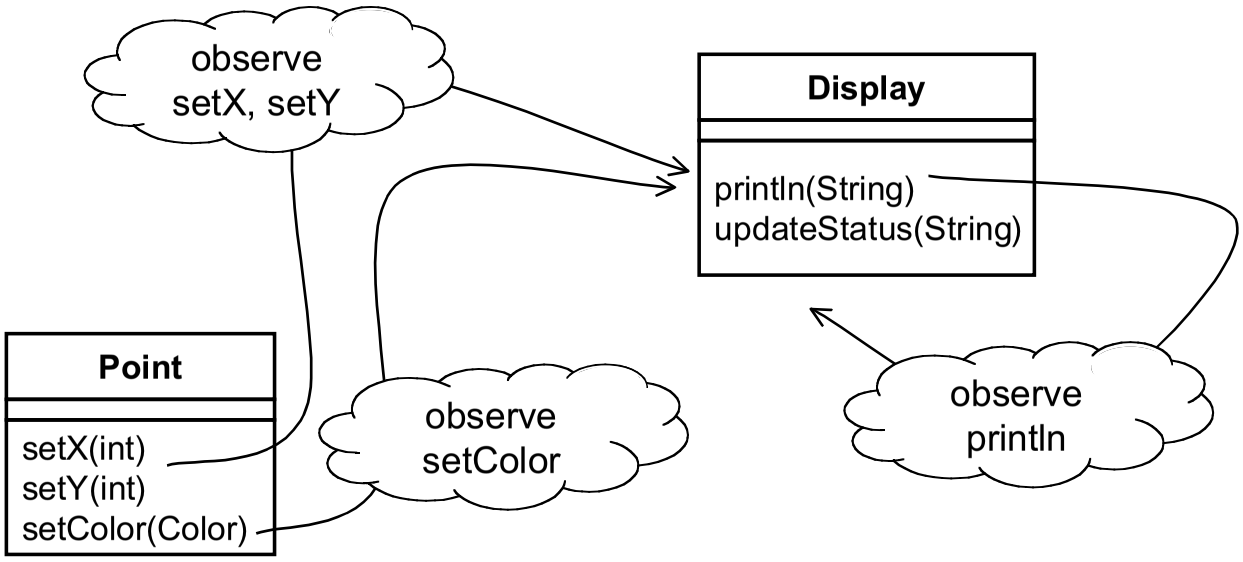
context-passing aspects

```
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);
  }
}
```

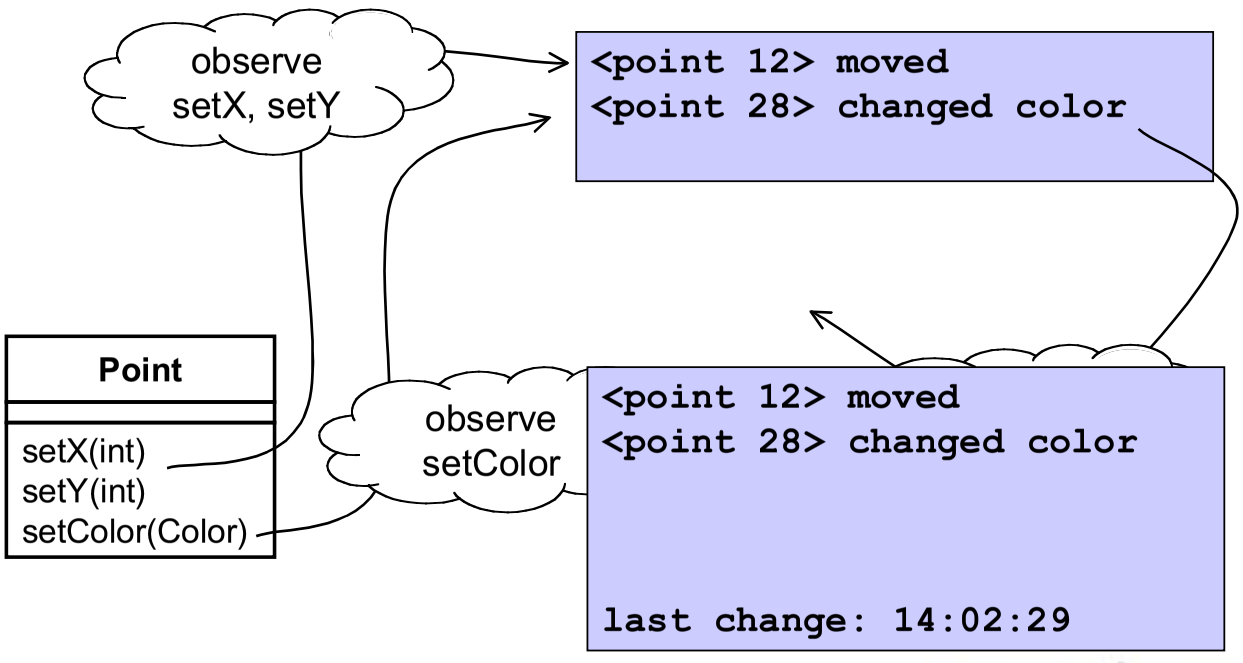


example n1

- **points**
  - with location and color
- **a screen**
  - that monitors location changes
  - that monitors color changes
  - that monitors screen changes



aspectj.org



aspectj.org

exercise

complete the code

```
aspect CoordinateObserving {
  HashMap perPointObservers = new WeakHashMap();
  static void addObserver(Point p, Display d) { ... }
  static void removeObserver(Point p, Display d) { ... }
  Iterator getObservers(Point p) { ... }

  pointcut change(): (call(void Point.setX(int))  ||
                      call(void Point.setY(int))) &&
                     target(p);

  after(Point p) returning: change() {
    // iterate for each display in observers…
    update(p, d);
  }
  void update(Point p, Display d) {
     d.println("Point " + p + "has moved" …);
  }
}
```

aspectj.org

exercise

complete the code

```
aspect CoordinateObserving {
  HashMap perPointObservers = new WeakHashMap();
  static void addObserver(Point p, Display d) { ... }
  static void removeObserver(Point p, Display d) { ... }
  Iterator getObservers(Point p) { ... }

  pointcut change(): (call(void Point.setX(int))  ||
                      call(void Point.setY(int))) &&
                     target(p);

  after(Point p) returning: change() {
    // iterate for each display in observers…
    update(p, d);
  }
  void update(Point p, Display d) {
     d.println("Point " + p + "has moved" …);
  }
}
```

aspectj.org

exercise

complete the code

```
aspect Observing {
  HashMap perPointObservers = new WeakHashMap();
  static void addObserver(Object p, Object d) { ... }
  static void removeObserver(Point p, Display d) { ... }
  Iterator getObservers(Point p) { ... }

  abstract pointcut change();

  after(Point p) returning: change() {
    // iterate for each display in observers…
    update(p, d);
  }
  abstract void update(Point p, Display d);
  }
}
```


exercise

complete the code

```
aspect ColorObserving {

  Display d; // assume this is set up for you









}
```

exercise

complete the code for `DisplayXXX`

```
aspect DisplayObserving {

  Display d; // assume this is set up for you









}
```


exercise

abstract these into a reusable aspect

```
aspect Observing {









}
```

exercise

abstract these into a reusable aspect

```
abstract aspect Observing {

    private WeakHashMap perSubjectObservers = new WeakHashMap();

   private Collection getObservers(Subject s) {
       Collection observers = (Collection)perSubjectObservers.get(s);
       if ( observers == null ) {
           observers = new LinkedList();
           perSubjectObservers.put(s, observers);
       }
       return observers;
   }

    protected interface Subject  { }
    protected interface Observer { }

    public void    addObserver(Subject s, Observer o) { getObservers(s).add(o);    }
    public void removeObserver(Subject s, Observer o) { getObservers(s).remove(o); }

    abstract pointcut change();

    after(Subject s): change() {
        Iterator iter = getObservers(s).iterator();
        while ( iter.hasNext() ) {
            updateObserver(s, ((Observer)iter.next()));
        }
    }

    abstract void updateObserver(Subject s, Observer o);
}
```


exercise

abstract these into a reusable aspect

```
public aspect ColorObserving extends Observing {

  declare parents: Point  implements Subject;
  declare parents: Screen implements Observer;

  pointcut changes(Subject s): call(void Point.setColor(Color)) && target(s);



  void updateObserver(Subject s, Observer o) {
      ((Screen)o).display("Screen updated because color changed.");
  }

}
```

exercise

abstract these into a reusable aspect

```
public aspect CoordinateObserving extends Observing {

  declare parents: Point  implements Subject;
  declare parents: Screen implements Observer;

  pointcut changes(Subject s): (call(void Point.setX(int)) ||
                                call(void Point.setY(int)) ) &&
                               target(s);

  void updateObserver(Subject s, Observer o) {
      ((Screen)o).display("Screen updated because coordinates changed.");
  }

}
```


exercise

abstract these into a reusable aspect

```
public aspect ScreenObserving extends Observing {

  declare parents: Screen implements Subject;
  declare parents: Screen implements Observer;

  pointcut changes(Subject s): call(void Screen.display(String)) &&
                               target(s);


  void updateObserver(Subject s, Observer o) {
      ((Screen)o).display("Screen updated because screen displayed.");
  }

}
```

example – properties of interfaces

```
interface Forest {
    int howManyTrees();
    int howManyBirds();
    ...
}

pointcut forestCall():
    call(* Forest.*(..));

before(): forestCall():  {
}
```


aspects on interfaces

a first attempt

```
aspect Forestry {
  pointcut forestCall():
    call(* Forest.*(..));

  before(): forestCall() {
    System.out.println(tjp.getSignature() +
                       " is a Forest-Method.");
  }
}
```

aspects on interfaces

an implementation

```
class ForestImpl implements Forest {
    public static void main(String[] args) {
        Forest f1 = new ForestImpl();

        f1.toString();
        f1.howManyTrees();
        f1.howManyTrees();
    }
    public int howManyTrees() { return 100; }
    public int howManyBirds() { return 200; }
}
```

- interface `Forest` includes methods from `Object`, such as `toString()`

aspects on interfaces

adding constraints

```
aspect Forestry {
  pointcut forestCall():
    call(* Forest.*(..)) &&
    !call(* Object.*(..));

  before(): forestCall() {
    System.out.println(thisJoinPoint.methodName +
                       " is a Forest-method.");
  }
}
```

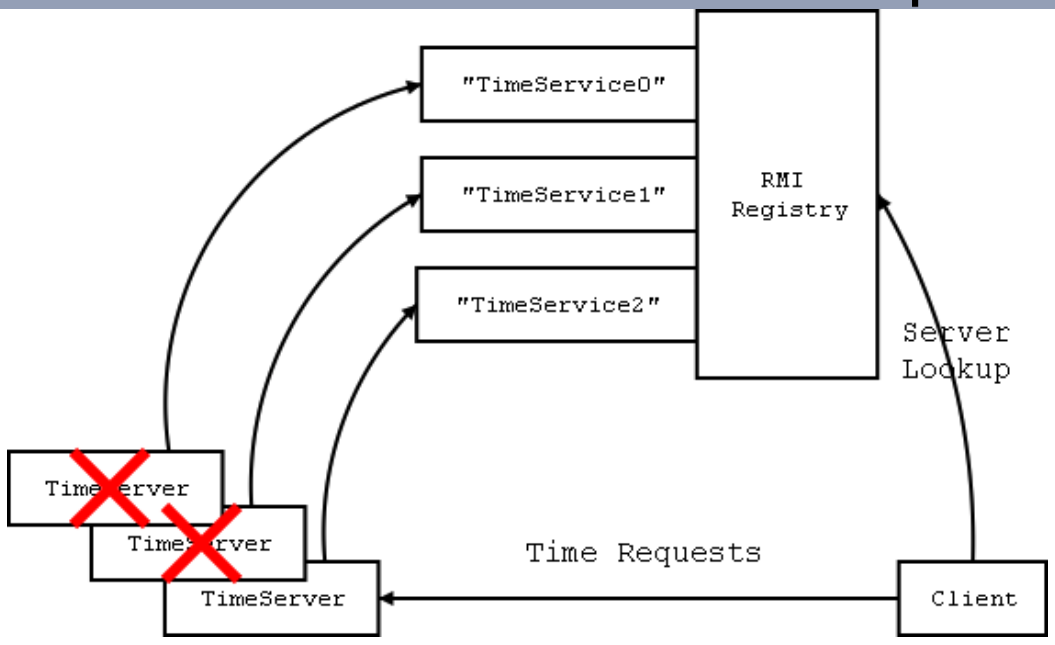
aspects on interfaces

exercises

- In this example you needed to constrain a pointcut because of undesired inheritance. Think of an example where you would want to capture methods in a super-interface.
- Constraining a pointcut in this way can be seen as an aspect *idiom*. What other idioms have you seen in this tutorial?

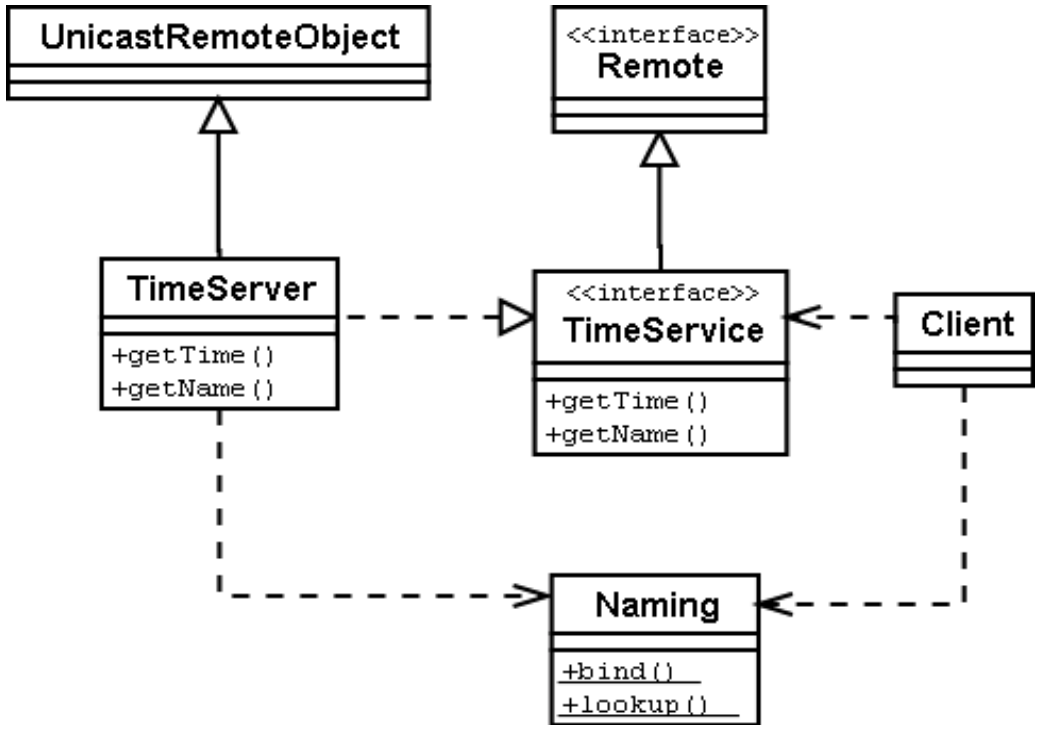
example 6

RMI exception aspects



client reactions to failures:
- abort
- try another server

a TimeServer design



the TimeService

```
public interface TimeService extends Remote {

  /**
   * What's the time?
   */
  public Date getTime() throws RemoteException;

  /**
   * Get the name of the server
   */
  public String getName() throws RemoteException;

  /**
   * Exported base name for the service
   */
  public static final String nameBase = "TimeService";
}
```

the TimeServer

```
public class TimeServer extends UnicastRemoteObject
                        implements TimeService {
  /**
   * The remotely accessible methods
   */
  public Date   getTime() throws RemoteException {return new Date();}
  public String getName() throws RemoteException {return toString();}
  /**
    * Make a new server object and register it
    */
  public static void main(String[] args) {
    TimeServer ts = new TimeServer();
    Naming.bind(TimeService.nameBase, ts);
  }
  /**
   * Exception pointcuts. Code is not complete without advice on them.
   */
  pointcut create():
    within(TimeServer) && call(TimeServer.new());

  pointcut bind(): within(TimeServer) && call(void Naming.bind(String,..));
  pointcut bindName(String name): args(name, ..) && bind();
}
```

no exception catching here, but notice

AbortMyServer

```
aspect AbortMyServer {
  TimeServer around(): TimeServer.create() {
    TimeServer result = null;
    try {
      result = proceed();
    } catch (RemoteException e){
      System.out.println("TimeServer err: " + e.getMessage());
      System.exit(2);
    }
    return result;
  }
  declare soft: RemoteException: TimeServer.create();

  void around(String name): TimeServer.bindName(name) {
    try {
      proceed(name);
      System.out.println("TimeServer: bound name.");
    } catch (Exception e) {
      System.err.println("TimeServer: error " + e);
      System.exit(1);
    }
  }
  declare soft: Exception: TimeServer.bind();
}
```

RetryMyServer

```
aspect RetryMyServer {
  TimeServer around(): TimeServer.create() {
    TimeServer result = null;
    try { result = proceed(); }
    catch (RemoteException e){
      System.out.println("TimeServer error."); e.printStackTrace();
    }
    return result;
  }
  declare soft: RemoteException: TimeServer.create();

  void around(String name): TimeServer.bindName(name) {
    for (int tries = 0; tries < 3; tries++) {
      try {
        proceed(name + tries);
        System.out.println("TimeServer: Name bound in registry.");
        return;
      } catch (AlreadyBoundException e) {
        System.err.println("TimeServer: name already bound");
    }
    System.err.println("TimeServer: Giving up."); System.exit(1);
  }
  declare soft: Exception: TimeServer.bind();
}
```

aspectj.org

the Client

```
public class Client {
  TimeService server = null;
  /**
   * Get a server and ask it the time occasionally
   */
  void run() {
    server = (TimeService)Naming.lookup(TimeService.nameBase);
    System.out.println("\nRemote Server=" + server.getName() + "\n\n");
    while (true) {
      System.out.println("Time: " + server.getTime());
      pause();
    }
  }
  /**
   * Exception pointcuts. Code is not complete without advice on them.
   */
  pointcut setup(): call(Remote Naming.lookup(..));
  pointcut setupClient(Client c): this(c) && setup();

  pointcut serve(): call(* TimeService.*(..));
  pointcut serveClient(Client c, TimeService ts):
    this(c) && target(ts) && serve();

  … other methods …
}
```

again, no exception catching here

aspectj.org

AbortMyClient

```
aspect AbortMyClient {
  Remote around(Client c): Client.setupClient(c) {
    Remote result = null;
    try {
      result = proceed(c);
    } catch (Exception e) {
      System.out.println("Client: No server. Aborting.");
      System.exit(0);
    }
    return result;
  }
  declare soft: Exception: Client.setup();

  Object around(Client c, TimeService ts): Client.serveClient(c, ts) {
    Object result = null;
    try {
      result = proceed(c, ts);
    } catch (RemoteException e) {
      System.out.println("Client: Remote Exception. Aborting.");
      System.exit(0);
    }
    return result;
  }
  declare soft: RemoteException: Client.serve();
}
```


RetryMyClient

```
aspect RetryMyClient {

  Remote around(Client c): Client.setupClient(c) {
    Remote result = null;
    try { result = proceed(c);}
    catch (NotBoundException e) {
      System.out.println("Client: Trying alternative name...");
      result = findNewServer(TimeService.nameBase, c.server, 3);
      if (result == null) System.exit(1);  /* No server found */
    } catch (Exception e2) { System.exit(2); }
    return result;
  }
  declare soft: Exception: Client.setup();

  Object around(Client c, TimeService ts): Client.serveClient(c,ts) {
    try { return proceed(c,ts); }
    catch (RemoteException e) {
      c.server = findNewServer(TimeService.nameBase, c.server, 3);
      if (c.server == null) System.exit(1);  /* No server found */
      try { return proceed(c, c.server); }
      catch (RemoteException e2) { System.exit(2); }
      return null;
      }
  }
  declare soft: RemoteException: Client.serve();

  static TimeService findNewServer(String baseName,
    Object currentServer, int nservers) { … }
}
```

building the client

- **abort mode:**

```
ajc Client.java TimeServer_Stub.java AbortMyClient.java
```

- **retry mode:**

```
ajc Client.java TimeServer_Stub.java RetryMyClient.java
```

- **switch to different failure handling modes without editing**
- **no need for subclassing or delegation**
- **reusable failure handlers**

RMI exception handling

exercises

- **Write another exception handler that, on exceptions, gives up the remote mode and instantiates a local TimeServer.**
- **How would this client look like if the exception handling were not designed with aspects? Can you come up with a flexible OO design for easily switching between exception handlers?**
- **Compare the design of exception handlers with aspects vs. with your OO design**

exercise

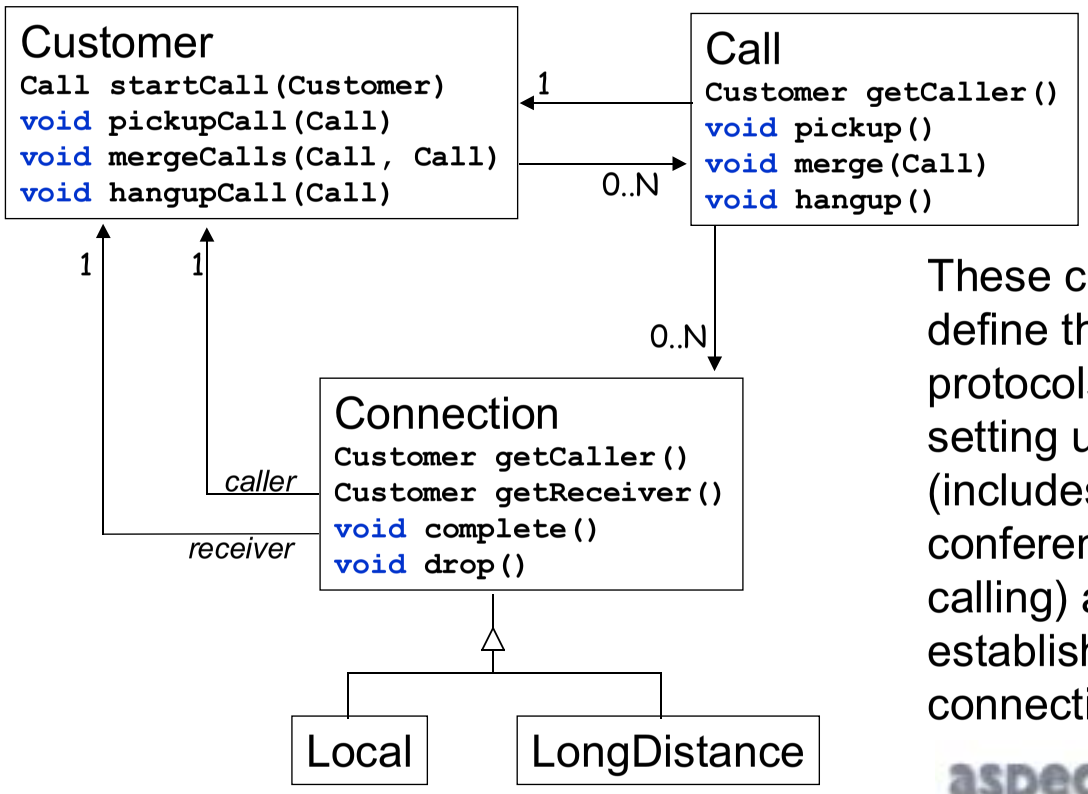
```
aspect UseLocalClient {









   Object around(Client c, TimeService ts): Client.serveClient(c, ts) {
    Object result = null;
    try {
      result = proceed(c, ts);
    } catch (RemoteException e) {
      c.s = new TimeServer();
      proceed(c, s);
    }
    return result;
  }
  declare soft: RemoteException: Client.serve();
}
```


example 7

layers of functionality

- **given a basic telecom operation, with customers, calls, connections**
- **model/design/implement utilities such as**
  - timing
  - consistency checks
  - …

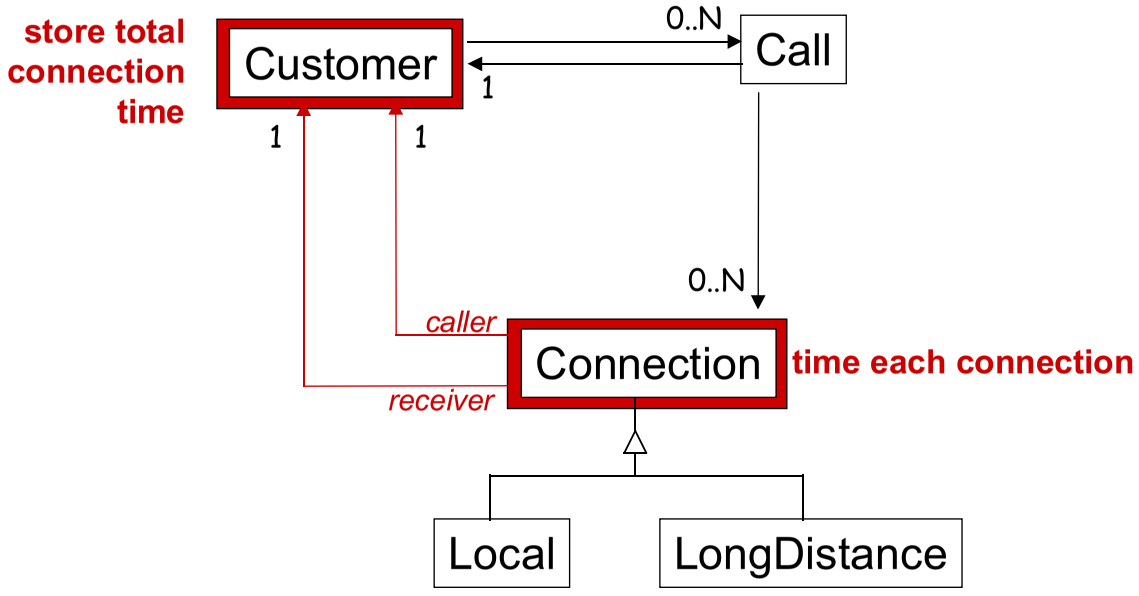
telecom basic design



These classes define the protocols for setting up calls (includes conference calling) and establishing connections

timing

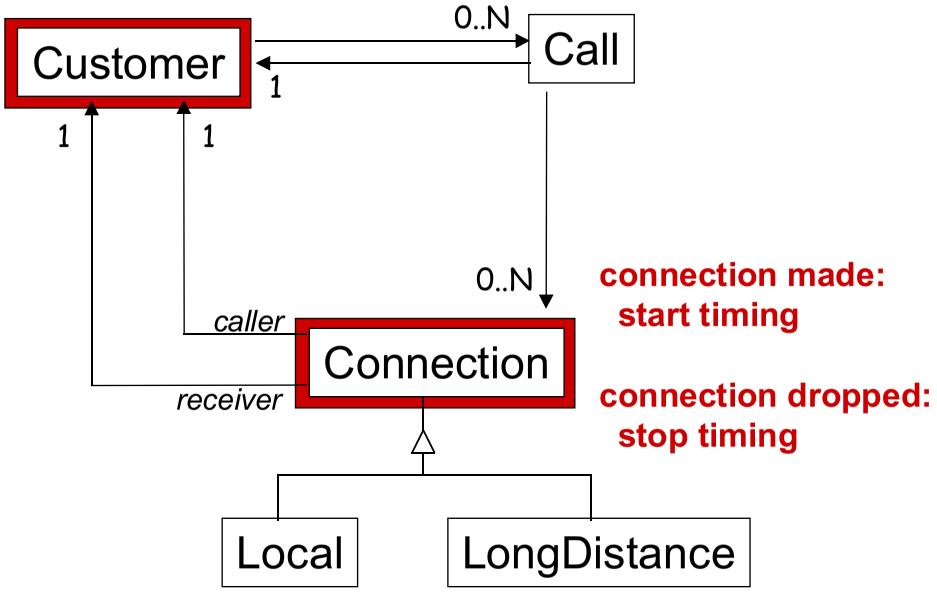
entities



timing

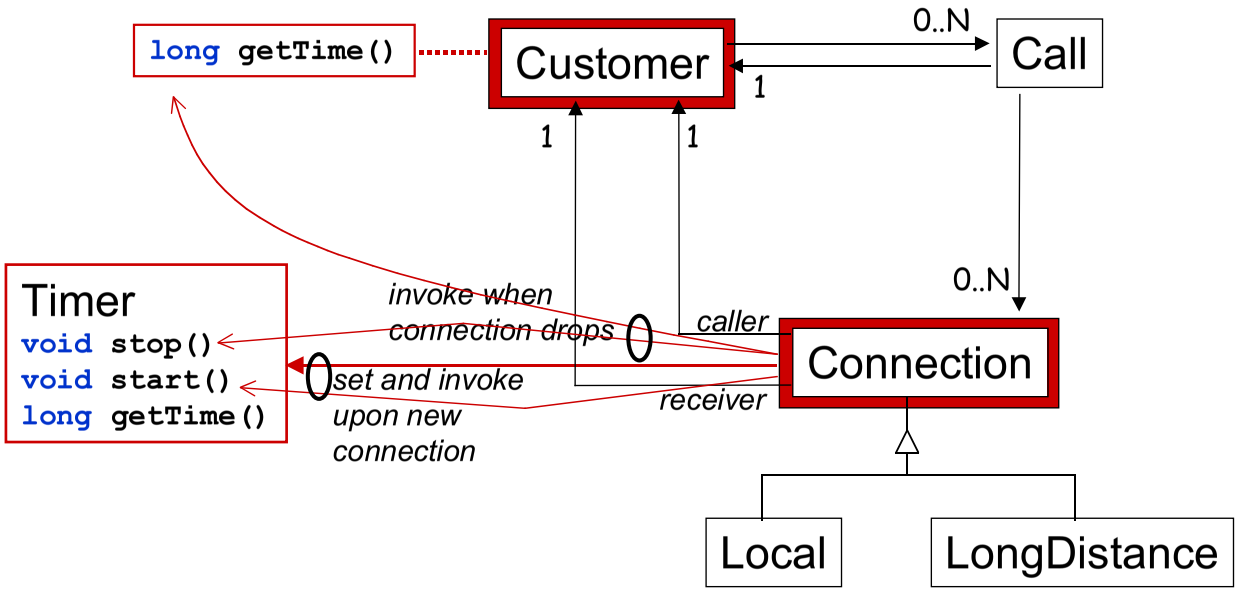
some actions

connection dropped:
add time



timing

additional design elements



timing

exercise

- **Write an aspect representing the timing protocol.**

timing

what is the nature of the crosscutting?

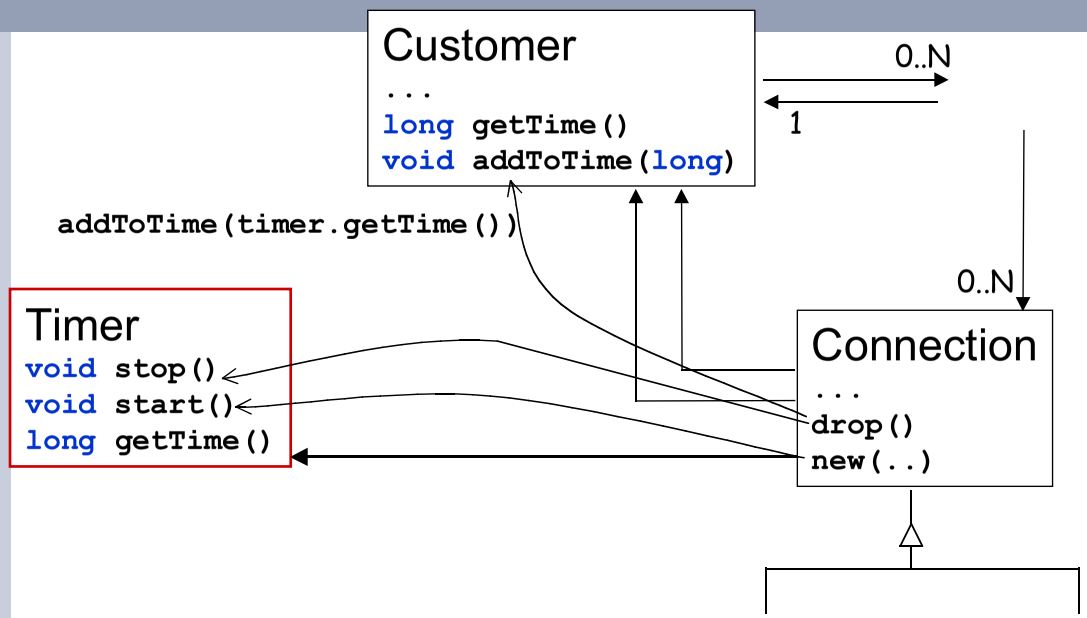
- **connections and calls are involved**
- **well defined protocols among them**
- **pieces of the timing protocol must be triggered by the execution of certain basic operations. e.g.**
  - when connection is completed, set and start a timer
  - when connection drops, stop the timer and add time to customers' connection time

timing

an aspect implementation

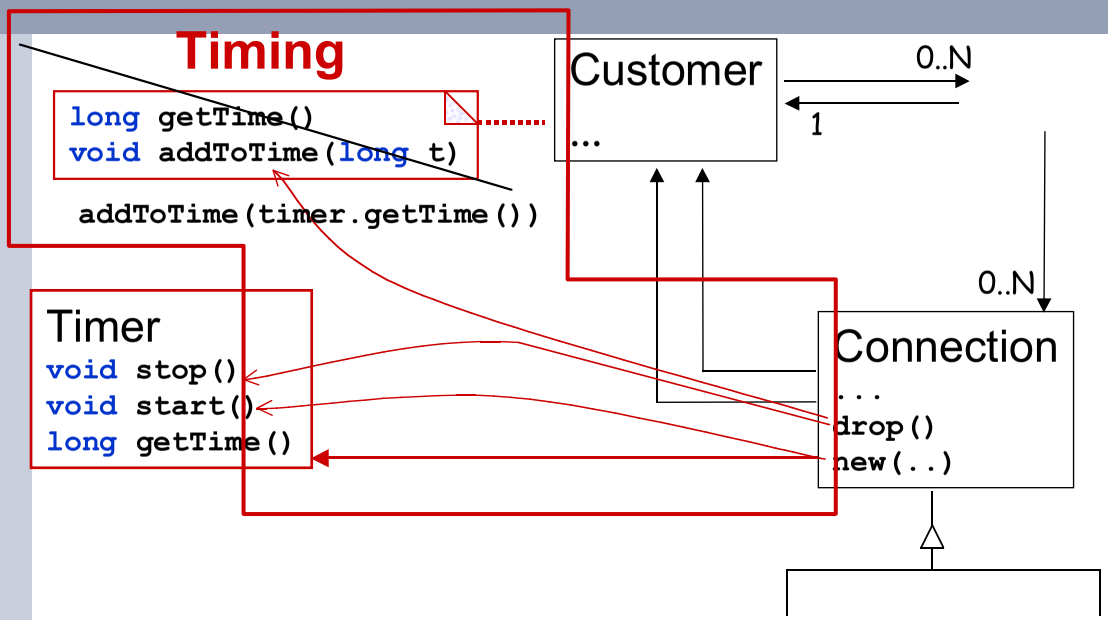
```
aspect Timing {
  private Timer Connection.timer = new Timer();

  private long Customer.totalConnectTime = 0;
  public static long getTotalConnectTime(Customer c) {
    return c.totalConnectTime;
  }

  pointcut startTiming(Connection c): target(c) && call(void c.complete());
  pointcut   endTiming(Connection c): target(c) && call(void c.drop());

  after(Connection c): startTiming(c) {
    c.timer.start();
  }

 after(Connection c): endTiming(c) {
    Timer timer = c.timer;
    timer.stop();
    long currTime = timer.getTime();
    c.getCaller().totalConnectTime += currTime;
    c.getReceiver().totalConnectTime += currTime;
  }
}
```


timing as an object



timing as an object captures timing support, but does not capture the protocols involved in implementing the timing feature

timing as an aspect



timing as an aspect captures the protocols involved in implementing the timing feature

timing as an aspect

has these benefits

- **basic objects are not responsible for using the timing facility**
  - timing aspect encapsulates that responsibility, for appropriate objects
- **if requirements for timing facility change, that change is shielded from the objects**
  - only the timing aspect is affected
- **removing timing from the design is trivial**
  - just remove the timing aspect

timing with AspectJ

has these benefits

- **object code contains no calls to timing functions**
  - timing aspect code encapsulates those calls, for appropriate objects
- **if requirements for timing facility change, there is no need to modify the object classes**
  - only the timing aspect class and auxiliary classes needs to be modified
- **removing timing from the application is trivial**
  - compile without the timing aspect class

timing

exercises

- **How would you change your program if the interface to Timer objects changed to**

```
Timer
void start()
long stopAndGetTime()
```

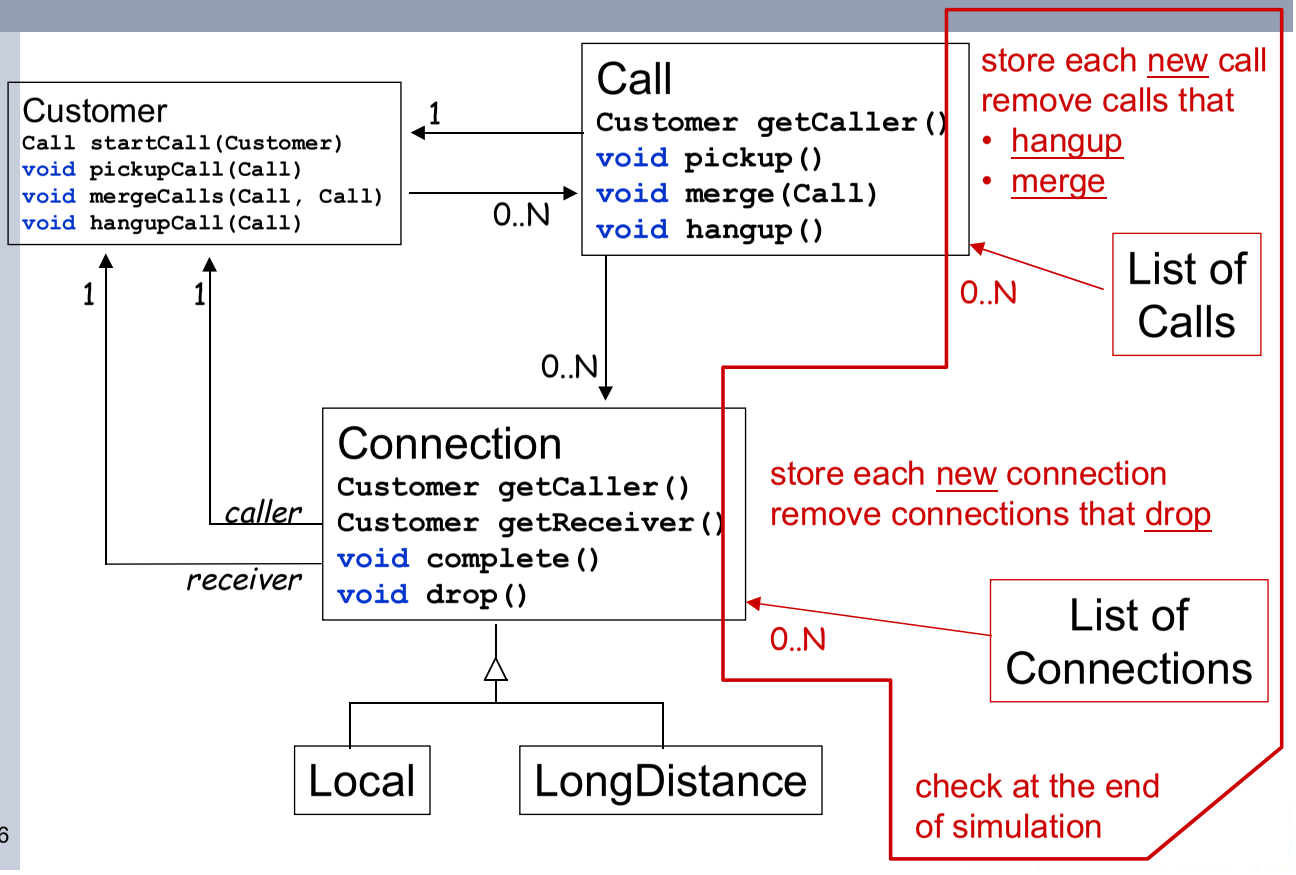
- **What changes would be necessary without the aspect abstraction?**

telecom, continued

layers of functionality: consistency

- **ensure that all calls and connections are being shut down in the simulation**

consistency checking



consistency checking

```
aspect ConsistencyChecker {
  Vector calls = new Vector(), connections = new Vector();
  /* The lifecycle of calls */
  after(Call c): target(c) && call(Call.new(..)) {
    calls.addElement(c);
  }
  after(Call c): target(c) && call(* Call.hangup(..)) {
    calls.removeElement(c);
  }
  after(Call other): args(other) && (void Call.merge(Call)) {
    calls.removeElement(other);
  }

  /* The lifecycle of connections */
  after(Connection c): target(c) && call(Connection.new(..)) {
    connections.addElement(c);
  }
  after(Connection c): target(c) && call(* Connection.drop(..)) {
    connections.removeElement(c);
  }
  after(): within(TelecomDemo) && executions(void main(..)) {
    if (calls.size() != 0) println("ERROR on calls clean up.");
    if (connections.size()!=0) println("ERROR on connections clean up.");
  }
}
```


summary so far

- **presented examples of aspects in design**
  - intuitions for identifying aspects
- **presented implementations in AspectJ**
  - how the language support can help
- **raised some style issues**
  - objects vs. aspects

when are aspects appropriate?

- **is there a concern that:**
  - crosscuts the structure of several objects or operations
  - is beneficial to separate out

… crosscutting

- **a design concern that involves several objects or operations**
- **implemented without AOP would lead to distant places in the code that**
  - do the same thing
    - e.g. traceEntry("Point.set")
    - try grep to find these [Griswold]
  - do a coordinated single thing
    - e.g. timing, observer pattern
    - harder to find these

… beneficial to separate out

- **does it improve the code in real ways?**
  - separation of concerns
    - e.g . think about service without timing
  - clarifies interactions, reduces tangling
    - e.g. all the traceEntry are really the same
  - easier to modify / extend
    - e.g. change the implementation of tracing
    - e.g. abstract aspect re-use
  - plug and play
    - tracing aspects unplugged but not deleted

good designs

summary

- **capture "the story" well**
- **may lead to good implementations, measured by**
  - code size
  - tangling
  - coupling
  - etc.

learned through experience, influenced by taste and style

expected benefits of using AOP

- **good modularity, even in the presence of crosscutting concerns**
  - less tangled code, more natural code, smaller code
  - easier maintenance and evolution
    - easier to reason about, debug, change
  - more reusable
    - more possibilities for plug and play
    - abstract aspects

Part V

References, Related Work

AOP and AspectJ on the web

- **aosd.net**
  - collection of wide range of work on AO software development
- **aspectj.org**
  - the AspectJ web site
    - downloads
    - documentation
    - examples
    - articles…

aspectj.org

workshops

- **2001**
  - ICSE, ECOOP, OOPSLA, Nantes, Lancaster, Brussels…
- **2000**
  - ICSE, ECOOP, OOPSLA
- **1999**
  - ECOOP, OOPSLA
- **1998**
  - ICSE, ECOOP
- **1997**
  - ECOOP
- **1996**
  - AOP friends meeting (at PARC)
- **Reflection workshops**
  - ECOOP –
  - OOPSLA –

aspectj.org

conference

- **First Int'l Conference on AOSD**
  - technical papers, tutorials, workshops
- **in cooperation w/ ACM SIGPLAN and SIGSOFT**
- **sponsored by AITO (ECOOP organization)**
- **see aosd.net for link to conference site**

AOP future – idea, language, tools

- **objects are**
  - code and state
  - "little computers"
  - message as goal
  - hierarchical structure
- **languages support**
  - encapsulation
  - polymorphism
  - inheritance
- **tools**
  - browser, editor, debuggers
    - preserve object abstraction

- **aspects are**
  -
  -
  -
  -
  + crosscutting structure
- **languages support**
  -
  -
  -
  + crosscutting
- **tools**
  -
    -
    + preserve aspect abstraction

AOP future

- **language design**
  - more dynamic crosscuts, type system …
- **tools**
  - more IDE support, aspect discovery, re-factoring, re-cutting…
- **software engineering**
  - finding aspects, modularity principles, …
- **metrics**
  - measurable benefits, areas for improvement
- **theory**
  - type system for crosscutting, fast compilation, advanced crosscut constructs

AspectJ & the Java platform

- **AspectJ is a small extension to the Java programming language**
  - all valid programs written in the Java programming language are also valid programs in the AspectJ programming language
- **AspectJ has its own compiler, ajc**
  - ajc runs on Java 2 platform
  - ajc is available under Open Source license
  - ajc produces Java platform compatible .class files

AspectJ status

- **release status**
  - 3 major, ~18 minor releases over last year (1.0b1 is current)
  - tools
    - IDE extensions: Emacs, JBuilder 3.5, JBuilder 4, Forte4J
    - ajdoc to parallel javadoc
    - debugger: command line, GUI, & IDE
  - license
    - compiler, runtime and tools are free for any use
    - compiler and tools are Open Source
- **aspectj.org**
  - May 1999: 90 downloads/mo, 20 members on users list
  - Feb 2001: 600 downloads/mo, 600 members on users list

AspectJ future

continue building language, compiler & tools

- **1.0**
  - language semantics stable
  - compilation direct to bytecodes
- **1.1**
  - incremental compiler
- **1.2**
  - source of target classes not required?
- **2.0**
  - new dynamic crosscut constructs

credits

AspectJ.org is a Xerox PARC project

slides, compiler, tools & documentation are available at aspectj.org

partially funded by DARPA under contract F30602-97-C0246

aspectj.org