AspectJ™

how to use aspect-oriented programming to solve common modularity problems in Java™ programs

AspectJ.org is a Xerox PARC project:

Bill Griswold, Erik Hilsdale, Jim Hugunin, Mik Kersten, Gregor Kiczales, Cristina Lopes

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

partially funded by DARPA under contract F30602-97-C0246

© Copyright 2000 Xerox Corporation. All Rights Reserved.

OOP – the language technology

- explicit support for
  - message as goal
  - encapsulation
  - polymorphism
  - inheritance

```
interface FigureElement {
  public void draw(Graphics g);
}

class Point implements FigureElement {
  int _x = 0, _y = 0;
   :
  public void draw(Graphics g) {
    <do own thing>
  }
}

class Line implements FigureElement {
   :
  public void draw(Graphics g) {
    <do own thing>
  }
}
```

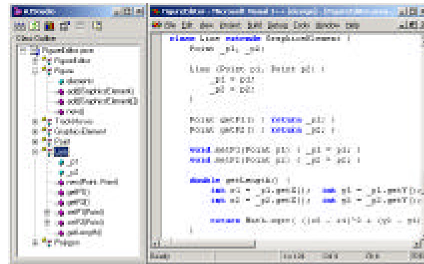
- some variation
  - class- vs. prototype-based
    - Java, C++, Smalltalk, CLOS vs. Self
  - single vs. multiple inheritance
    - Java, Smalltalk, Self vs. C++, CLOS

talk contents

- a programming style
  - aspect-oriented programming
  - "crystallization of style" more than "agglutination of features" [KayHOPLII]
- a language
  - AspectJ
  - integration of AOP into Java
  - 0.7 now aiming for 1.0 this summer
- some tools
  - AJDB, AJDE, AJStudio, ajdoc, AspectFinder™

some tension here!

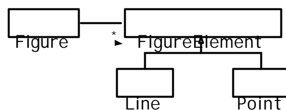
OOP – the tool technology



- tools preserve object abstraction
  - browse classes and methods
  - debug objects and methods

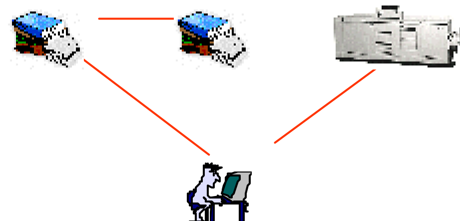
OOP – the programming style

- objects that can operate on themselves
  - encapsulated bundles of code and state
  - "thousands and thousands of computers" [KayHOPLII]
  - messages as "goals"



an example system

a distributed digital library



the code

Document

User

Terminal

Library

Printer

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

locality of user interruption

Document

User

Terminal

Library

Printer

- "tangled code"
- code redundancy
- difficult to reason about
- difficult to change

language support for aspects

Document

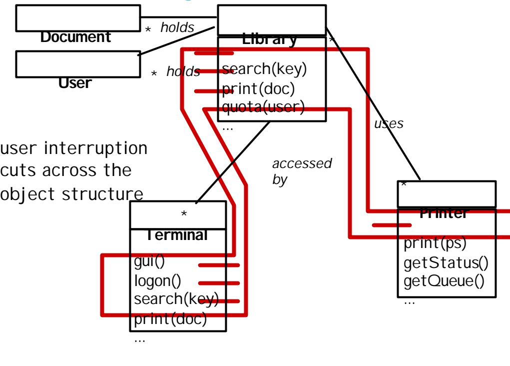
User

Terminal

Library

Printer

a crosscutting concern



user interruption cuts across the object structure

language support for aspects

Document

User

Terminal

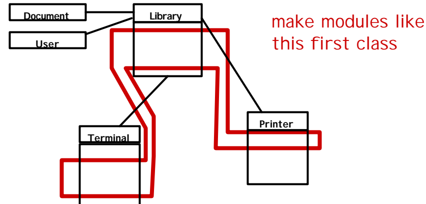
Library

UserInterrupt

Printer

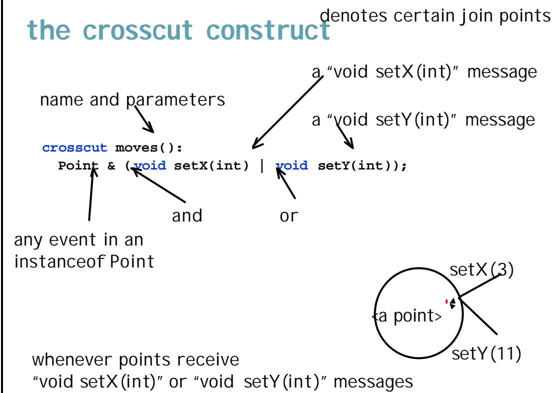
AspectJ

- support for explicitly crosscutting concerns
- smooth integration with Java
- adds 3 new constructs



the crosscut construct

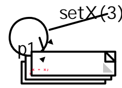
denotes certain join points



semantic framework for Java

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

  Point(int x, int y) { setXY(x,y); }

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

  void setX(int x) { _x = x; }
  void setY(int y) { _y = y; }

  void setXY(int x, int y) {
    _x = x; _y = y;
  }

  void draw(Graphics g) {
    …
  }
}
```

when a "void setX(int)" message* is received by a point, do this



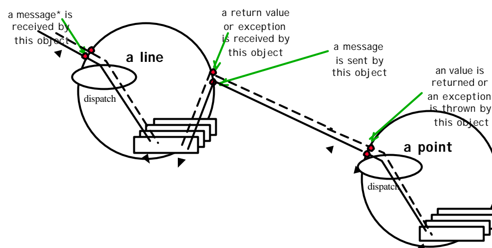
the crosscut construct

can cut across multiple classes

```
crosscut moves():
  (Point & (void setX(int) | void setY(int))) |
  (Line  & (void setP1(Point) | void setP2(Point)));
```

the join points

key events in program execution



the advice construct

additional action to take at crosscut

```
crosscut moves():
  (Point & (void setX(int) | void setY(int))) |
  (Line  & (void setP1(Point) | void setP2(Point)));

static advice(): moves() {
  after {
     <runs after moves>
  }
}
```

a simple aspect

an aspect is a crosscutting concern

```
class TrackMoves {

  static boolean _flag = false;
  static boolean testAndClear() {
    boolean result = _flag;
    _flag = false;
    return result;
  }

  crosscut moves():
    (Point & (void setX(int) | void setY(int))) |
    (Line  & (void setP1(Point) | void setP2(Point)));

  static advice(): moves() {
    after {
       _flag = true;
    }
  }
}
```

advice is

additional action to take at a crosscut

- before — before something else happens
- after returning — when something else exits
- after throwing — when something else throws exception
- after — when something else exits either way
- around — before, with explicit control over when&if something else happens

using context in advice

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

typed variable in place of type name

crosscut signature

```
crosscut moves(Object o):
  o & ((Point & (void setX(int) | void setY(int))) |
       (Line  & (void setP1(Point) | void setP2(Point))));

static advice(Object o): moves(o) {
  after {
    <o is bound to object>
  }
}
```

matches signature

advice parameters

after advice

```
class BoundsChecker {

  crosscut sets(Point p):
    p & (void setX(int) | void setY(int));

  static advice (Point p): sets(p) {
    after {
      assert(p.getX() >= MIN_X);
      assert(p.getX() <= MAX_X);
      assert(p.getY() >= MIN_Y);
      assert(p.getY() <= MAX_Y);
    }
  }

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

using context in advice

```
class TrackMoves {

  static Set _movers = new HashSet();

  static Set getMovers() {
    Set result = _movers;
    _movers = new HashSet();
    return result;
  }

  crosscut moves(Object o):
    o & ((Point & (void setX(int) | void setY(int))) |
         (Line  & (void setP1(Point) | void setP2(Point))));

  static advice(Object o): moves(o) {
    after {
       _movers.add(o);
    }
  }
}
```

before advice

```
class BoundsChecker {

  static advice(Point p, int newX): p & void setX(newX) {
    before {
      assert(newX >= MIN_X);
      assert(newX <= MAX_X);
    }
  }
  static advice(Point p, int newY): p & void setY(newY) {
    before {
      assert(newY >= MIN_Y);
      assert(newY <= MAX_Y);
    }
  }
  static void assert(boolean v) {
    if ( !v )
      throw new RuntimeException(…);
  }
}
```

around advice

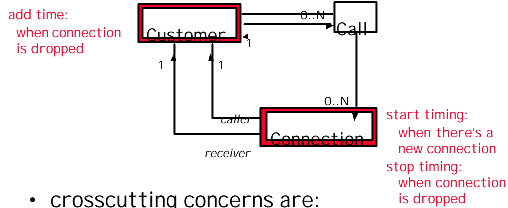
```
class BoundsEnforcer {

  static advice(Point p, int newX) returns void:
         p & void setX(newX) {
    around {
      thisJoinPoint.runNext(p, clip(newX, MIN_X, MAX_X));
    }
  }

  static advice(Point p, int newY) returns void:
         p & void setY(newY) {
    around {
      thisJoinPoint.runNext(p, clip(newY, MIN_X, MAX_X));
    }
  }

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

timing aspect implementation

```
crosscut startTiming(Connection c): c & void complete();
crosscut endTiming(Connection c):   c & void drop();

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

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

an asymmetric multi-object protocol

trivial telecom system



- crosscutting concerns are:
  - timing
  - consistency checks
  - ...

wildcarding in crosscuts

`shapes.util.Point` `shapes..*`	types
`Point & public * *(..)` `shapes..* & !private int *(..)`	messages
`(Point | Line) & new(..)` `shapes..* & new(..)`	instantiations
`* *(..) & handle(RMIException)`	handling exceptions

".." is a multiple-part wild card
"*" is a single-part wild card

timing aspect implementation

```
class Timing {

  private Timer Connection.timer = new Timer();

  private long Customer.totalConnectTime = 0;


  public static long getCustomerTotalConnectTime(Customer c) {
    return c.totalConnectTime;
  }

  :
  :
```

a variable in Connection that only Timing can see

a variable in Customer that only Timing can see

proper accessor for the total connect time

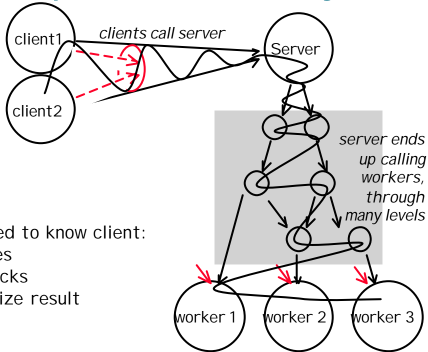
- the aspect state crosscuts the objects
- proper access is through the aspect

property-based crosscuts

```
class LogPublicErrors {

  static Log log = new Log();

  crosscut publicInterface ():
    mypackage..* & ( public * *(..) | public new(..) );

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

- consider another programmer adding public method
  - i.e. extending the public interface
- this code will automatically capture that

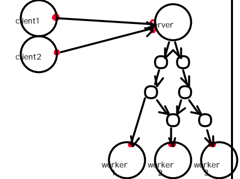
context-dependent functionality



workers need to know client:
- capabilities
- charge backs
- to customize result

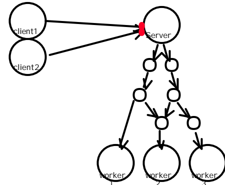
context passing crosscuts

```
crosscut entryPoints():
  Server & (void doService1(Object) |
            void doService2());

crosscut workPoints():
  (ServiceHelper1 & void doWorkItemA()) |
  (ServiceHelper2 & void doWorkItemB()) |
  (ServiceHelper3 & void doWorkItemC());

crosscut invocations(Client c):
  c & calls(entryPoints());
```



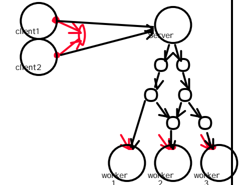
context passing crosscuts

```
crosscut entryPoints():
  Server & (void doService1(Object) |
            void doService2());
```



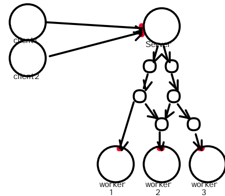
context passing crosscuts

```
crosscut entryPoints():
  Server & (void doService1(Object) |
            void doService2());

crosscut workPoints():
  (ServiceHelper1 & void doWorkItemA()) |
  (ServiceHelper2 & void doWorkItemB()) |
  (ServiceHelper3 & void doWorkItemC());

crosscut invocations(Client c):
  c & calls(entryPoints());

crosscut clientWork(Client c):
  cflow(invocations(c)) & workPoints();
```



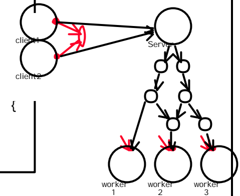
context passing crosscuts

```
crosscut entryPoints():
  Server & (void doService1(Object) |
            void doService2());

crosscut workPoints():
  (ServiceHelper1 & void doWorkItemA()) |
  (ServiceHelper2 & void doWorkItemB()) |
  (ServiceHelper3 & void doWorkItemC());
```



context passing crosscuts

```
class HandleChargebacks {
  crosscut entryPoints():
    Server & (void doService1(Object) |
              void doService2());

  crosscut workPoints():
    (ServiceHelper1 & void doWorkItemA()) |
    (ServiceHelper2 & void doWorkItemB()) |
    (ServiceHelper3 & void doWorkItemC());

  crosscut invocations(Client c):
    c & calls(entryPoints());

  crosscut clientWork(Client c):
    cflow(invocations(c)) & workPoints();

  static advice (Client c): clientWork(c) {
    before {
      c.chargeback();
    }
  }
}
```



IDE demos

- basic support, in emacs and Visual Studio
  - browsing program structure
  - editing
  - compiling
- extension to javadoc
- will expand in future releases
  - debugger
  - more sophisticated browsing
  - more IDEs

a reusable exception handling aspect

```
abstract public class LogRemoteExceptions {

  abstract crosscut msgs();

  static advice msgs() {
    after throwing (RemoteException e) {
      log.println("Remote call failed in: " +
                  thisJoinPoint.methodName +
                  "(" + e + ").");
    }
  }
}
```

abstract

```
public class JWAMRemotExceptionLogger extends LogRemoteExceptions {

  crosscut msgs():
    RegistryServer & * *(..) |
    RMIMessageBrokerImpl & private * *(..);
}
```

AspectJ – libraries

- 2 things this could mean
  - libraries of crosscutting concerns
  - libraries of objects with crosscutting concerns
- language must support
  - inheritance & specialization of aspects
  - composition of aspects (advice & crosscuts)
- key points
  - libraries evolve, come later, are really valuable

common questions

- how do I start?
  - very conservative
    - use AO style
  - somewhat conservative -
    - debugging, tracing, profiling
  - somewhat aggressive
    - use where crosscutting concerns are hurting most
  - more aggressive
    - re-factor all existing code
- back-out is straightforward in all cases
- but how do I find the crosscutting concerns?

inheritance and specialization

- crosscuts can have additional advice
  - in figure editor
  - moves() can have advice from different parts of system
- abstract crosscuts can be specialized

other common aspects

- concurrency control (locking)
- contracts (pre-/post-conditions)
- initialization & cleanup
  - especially of linked structures
- security
- GUI "skins"
- some patterns
- common protocols

common questions

- is this just event based programming? reflection?
- doesn't ______ have a feature like this?

  Visual Age, CLOS, Objective-C

  - crosscutting concerns are an old problem
  - crystallization of style around explicit support for implementation of crosscutting concerns

common questions

- exception handling case study*
- framework w/ 600 classes, 44 KLOC
- reengineered with AspectJ 0.4

	without aspects	with aspects
exception detection	2.1 KLOC pre-conditions .6 KLOC post-conditions	.6 KLOC pre-conditions .3 KLOC post-conditions
exception handling	2.1 KLOC (414 catch statements)	.2 KLOC (31 catch aspects)
% of total LOC	10.9%	2.9%

*** to appear in ICSE'2000**

common questions

- doesn't this violate modularity?
- composition of aspects?
- what if you aren't using Java?
  - C, C++...

related work

in separation of crosscutting concerns

- HyperJ [Ossher, Tarr et. al]
  - multi-*dimensional* separation of concerns
  - generator-based approach
- Composition Filters, Demeter/Java
- OpenC++, OpenJIT, ...
- Intentional Programming
- active community...
  - publishes and holds workshops at: OOPSLA, ECOOP and ICSE

common questions

- does this really help?
- these examples are all simple
- OOP doesn't shine when implementing just Menu OOP shines when doing window system, w/GUI frameworks
  - consider larger systems
  - consider multiple crosscutting aspects
  - consider reading someone else's code

AspectJ status

- **aspectj.org**
  - 250 unique users download each month
  - users list grew from 35 to 379 members since August
- compiler implementation
  - 4 major and 16 minor releases over the last year
    - still needs to support incremental compilation
    - still depends on javac as a back-end
- tutorials, primer, users-guide
  - full-day tutorials grew from 3 in 1998 to 8 in 1999
- tools support
  - initial IDE support for emacs, VisualStudio, *need more*
  - javadoc replacement – ajdoc, *need jdb*

AspectJ future

continue to build language, compiler, tools and user community; community feedback will drive design

- 1.0
  - crosscutting state, type system, tuning existing constructs
  - no longer dependent on javac
  - ajdb; JBuilder and Forte support
- 1.1
  - only minor language changes
  - faster incremental compiler (up to 5k classes), doesn't require source of target classes
  - ???
- 2.0
  - new dynamic crosscut constructs, ...

commercialization decision sometime after 1.0

AOP future

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