Aspect-Oriented Programming and the AspectJ

Tamás Kozsik
(kto@elte.hu, http://kto.web.elte.hu/)
Dept. Programming Languages and Compilers
Eötvös Loránd University, Budapest (Hungary)
Contents

• Need for a novel paradigm
• Aspect-Oriented Software Development
  – Aspect-Oriented Design
  – Aspect-Oriented Programming
    • AspectJ (language and tools)
    • Other languages and tools
  – Applications, examples
• Related technologies
Schedule

• Need for a novel paradigm
Separation of concerns
Aspect-Oriented Programming

• AspectJ

• Aspect-Oriented Software Development
Related technologies
Popular paradigms

• High-level languages (5*-6*)
• Structured programming (6*-7*)
• Modular programming (7*-8*)
• Object-oriented programming (8*-9*)
• Component-oriented programming (9*-0*)
• ?
Further important paradigms

• Declarative programming (e.g. functional, logic)
• Generic programming
• Metaprogramming
• Application of design patterns
Nowadays

- Aspect-Oriented Programming
- Generative programming
- Intentional programming
- Multi-paradigm programming
- Service-oriented programming
Managing complexity

• Driving force
  – Methodologies
  – Programming languages

• Abstraction and modularity
Increase reusability

- Nice dream
- Only partially achieved
- Modularity and abstraction
<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Abstraction</th>
<th>Modularization</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-level languages</td>
<td>machine independence</td>
<td>instruction</td>
</tr>
<tr>
<td>Structured programming</td>
<td>control flow</td>
<td>control structure</td>
</tr>
<tr>
<td>Procedural programming</td>
<td>computations</td>
<td>procedure</td>
</tr>
<tr>
<td>Modular programming</td>
<td>libraries, data types</td>
<td>module</td>
</tr>
<tr>
<td>Object-oriented programming</td>
<td>data types</td>
<td>class/object</td>
</tr>
<tr>
<td>Component-O. programming</td>
<td>scenarios</td>
<td>component</td>
</tr>
</tbody>
</table>
Modularity

• Methodology
  – decomposition
  – hierarchy
  – cohesion within components
  – narrow interface between components

• Language
  – encapsulation, information hiding
  – subprograms, classes, packages
  – compilation units, libraries
Decomposition

• Separation of concerns (Dijkstra)

• Based on
  – functionality (features)
  – data structures (objects)
  – control flow (concurrency)
  – services, technologies (framework)
  – …
Example by [OT’99]

```
a := b + 5

syntax correct

Check

ea = 16

Display

Eval
```

Figure 1. Tools and Shared AST in the Expression SEE.
“Tools” are implemented as methods on each AST class.

Figure 3. Partial UML Design for the Expression SEE
Problems with this design

- Consider e.g. the concern “display”
- Scattering and tangling
- Bad decomposition? Need another one?

- No!
Tyranny of the Dominant Decomposition [OT’99]

• Arbitrariness of the decomposition hierarchy [MO’05]
• Current methodologies/languages have DD
• Need to overthrow the tyranny
• Need for decomposition in multiple dimensions simultaneously
• Several approaches…
Multi-Dimensional Separation of Concerns

- IBM Research, Ossher, Tarr
- General solution
- For every artifact in software development
- A realization: Hyperslices
- An implementation: Hyper/J
- later…
Concerns

• A dimension of concerns
  – Decomposition is based on ~
  – Functionality, data structures, control flow, etc.

• Concern
  – Obtained by decomposition
  – An element of a dimension of concerns
  – A program feature, a class, a process
  – BinaryOp, Plus, Display, Evaluation
Decompose simultaneously
Dimensions

- Artifacts
  - Specification, Design docs, Code, Test suite
- Functionality (features)
  - Evaluation, Display, Persistence
- Data structures (classes)
  - Expression, UnaryOp, Plus
- Variants
  - For different configurations (e.g. style checks)
- Units of change
The overall system

• Can be viewed and modified in different dimensions of concerns
  – Different developers
  – Same developer at different times

• The dimensions and the decompositions
  – are coequal
  – can evolve
Aspect-Oriented Software Development

• A less general solution
• Base functionality + crosscutting concerns
• Simple and powerful
• Became popular and wide-spread
• Many approaches, many implementations
• Aspect-Oriented Programming
• Most famous: AspectJ
good modularity

XML parsing

- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box
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)
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
problems like...
session expiration is not modularized
problems like…

session tracking is not modularized

```
HttpRequest
getCookies()
getRequestURI()
getSession()
getSessionId()
...

SessionInterceptor
requestMap(request)
beforeBody(req, resp)
...

HttpResponse
getRequest()
setContentType(contentType)
getOutputStream()
setSessionId(id)
...

Session
getAttribute(name)
setAttribute(name, val)
invalidate()
...
```

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
Crosscutting concern

- A concern that appears at many different places in the program
- Scattering
- Tangling

- Physical separation: in an aspect
- Pluggable
crosscutting concerns

[Diagram showing class relationships and methods involving HTTPRequest, HttpResponse, SessionInterceptor, Session, and Servlet classes.]

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
language support to...
Examples of crosscutting

- Tracing and profiling
- Logging
- Configuration management
- Exception handling
- Security
- Synchronization
- Verifying correctness
- Visualization
Example of aspect

import java.io.*;
aspect SimpleLogging {
    pointcut loggedCalls():
        call(public * *.*(..));

    after() throwing (IOException e): loggedCalls()
    {
        System.err.println(e);
    }
}
AOP terminology

- Crosscut: affect more modules
- Aspects: modules implementing crosscutting behaviour
- Obliviousness: base code not referring to aspects
- Join points: where composition happens
- Quantification: reference to more join points
- Aspect-weaver: composes aspects with base code
“Traditional” techniques

- Application of design patterns
- Tool support (e.g. profilers)
- Macros and pre-compilers
- Frameworks
Macros and pre-compilers

- Tracing (debugging), configuration management
- Certain aspects become pluggable
- Crosscutting concerns are not separated
Frameworks

- CORBA, J2EE etc.
- Certain concerns of components handled by framework
  - Services
  - Security, transaction management, persistence, remote access, multithreading, life cycle, pooling, clustering, ...
- Provided in specific languages
  - J2EE: deployment descriptor
- Fixed concerns with fixed possibilities
- No obliviousness
Development of AOP languages

• “Special purpose” approach
  – Frameworks
    (like deployment descriptors of J2EE)
  – Concern-specific AOP languages

• “General purpose” approach
  – E.g. AspectJ
My story of AOP

- (Multi-agent) simulations
- Concerns
  - Modelling
  - Observation
- OOP: tangling and scattering
  - Conceptual separation
  - Publishing models
  - Multiple observers
- Multi-Agent Modelling Language (MAML), 1999. [GK’99]
The beginning of AOP  [Lopes’05]

- Much related research from early 90’
- Special purpose AOP languages  [K…’97]
  - Concurrency: D framework  [Lopes’97]
    - Cool: coordination
    - Ridl: remote access
  - Performance
    - RG: image processing  [MKL’97]
    - AML: sparse matrix manipulation  [I…’97]
- DJ, DJava, AspectJ
- AspectJ (general purpose)  [LK’98]
- Further general purpose AOP languages and technologies
General purpose AOP language

• Not concern-specific
• Different concerns expressed with same language constructs
• Basic constructs for
  – modularization
  – composition
• Usually built over a host language
Some examples  [AOSD]

- AspectC++
- AspectC
- Aspect#, AspectDNG, LOOM.NET, AspectC#, EOS
- Aspect-Oriented Perl, Aspect.pm
- Aspects, Pythius (Python)
- AspectR (Ruby)
- AspectS, Apostle, MetaclassTalk (Squeak/Smalltalk)
- AspectXML
- AOPHP, AspectPHP
- UMLAUT
Some examples for Java [AOSD]

- Spring AOP
- JBoss-AOP
- AspectWerkz
- Object Teams
- Caesar
- Java Aspect Components
- JMangler, JOIE, JMunger
- DJ
- ComposeJ, ConcernJ, JCFF
- Java Layers
- JPiccola
- Pragma
- Lasagne/J
Reminder – I

• Managing complexity: abstraction and modularization
• Better separation of concerns is needed
• Problems with DD: scattering and tangling
• Popular solution: AOSD, AOP
• Crosscutting concerns turned into aspects
• Obliviousness, join points, aspect-weaver
• AspectJ, among others…
AspectJ

- How does it work
- The language
- Tool support
- Integration
- Why AOP/AspectJ
How AspectJ works

- Download from [AspectJ]
  - AspectJ 5 Development Kit v1.5.3
- Install
  - Requires at least JDK 1.3
  - Best to have at least Java 5
  - Tools (e.g. ajc) and docs
- Command line, PATH and CLASSPATH
The AspectJ language

- Join points (concept)
- Pointcuts
- Advice (constructs)
- Inter-type declarations
- Aspects
The first program: a class

public class Hello {

    void greeting(){
        System.out.println("Hello!");
    }

    public static void main( String[] args ){
        new Hello().greeting();
    }

}
The first program: an aspect

```java
public aspect With {
    before() : call( void Hello.greeting() )
    {
        System.out.print("> ");
    }
}
```
The first program: compile and run

- `example-01`
- Source file for aspects: `.java` or `.aj`
- `PATH` includes `<aspectj>/bin`
- `CLASSPATH` includes
  - `<aspectj>/lib/aspectjrt.jar`

`ajc Hello.java With.aj`
`java Hello`
ajc

- Aspect weaver
- Compiles Java and AspectJ
- Produces efficient code
- Incremental compilation
- Accepts bytecode
- Java5 and AspectJ5 features:
  
  a j c  - 1 . 5  ...
The first program: after weaving (Simplified view!!)

```java
public class Hello {
    void greeting(){ System.out.println("Hello!"); }

    public static void main( String[] args ){
        Hello dummy = new Hello();
        System.out.print("> ");
        dummy.greeting();
    }
}
```
Alternatives for compilation

$ ajc Hello.java With.aj

$ cat sources.txt
Hello.java
With.aj
$ ajc @sources.txt

$ ajc -sourceroots .
Join points

- New concept
- Well-defined points in the program flow
  - call of a method or constructor
  - execution of a method or constructor
  - execution of a `catch`
  - getting/setting a field
  - initialization of a class, object or aspect
  - execution of advice
Pointcut

• A language construct
• Picks out certain join points (and values): quantification
• Composition

call( void Hello.greeting() )
call( * Hello.*(..) )
call( void Hello.greeting() ) && target(f)
call(void Point.setX(int)) ||
    call (void Point.setXY(int,int))
Advice

• A language construct
• Code to be executed at certain join points
  - before, after or around

```java
before() : call( void
Hello.greeting() ) {
   System.out.print("> ");
}
```
Inter-type declaration

• A language construct
• Modify the static structure of a program
  – introduce new members
  – change relationship between classes
Aspect

• A language construct
• The unit of modularity for crosscutting concerns
• May contain pointcuts, advice and inter-type declarations

```java
public aspect With {
    before() : call( void Hello.greeting() ) {
        System.out.print("> ");
    }
}
```
a simple figure editor

Display

Figure

- makePoint(....)
- makeLine(....)

* FigureElement

- moveBy(int, int)

Point

- getX()
- getY()
- setX(int)
- setY(int)
- moveBy(int, int)

Line

- getP1()
- getP2()
- setP1(Point)
- setP2(Point)
- moveBy(int, int)

factory methods

operations that move elements
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) { ... }
}
Composing pointcuts

• Pointcut: pick up joint points
• Composition operators:  
• The result is a pointcut

\[
\text{call(} \text{void Point.setX(int))} \; || \; \text{call(} \text{void Point.setY(int))}
\]
(Name-based) crosscutting

call(void FigureElement.moveBy(int,int))
|| call(void Point.setX(int))
|| call(void Point.setY(int))
|| call(void Line.setP1(Point))
|| call(void Line.setP2(Point))

- Affects multiple types
Named pointcuts

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

• Declares a pointcut named move
• May be used many times
Using named pointcuts

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

before() : move()
    { System.out.println("moving something"); }
Property-based crosscutting

call( public * Figure.* (..) )
call( * Figure.make*(..) )

• Wildcards in the signature
• Not only syntactical, but also lexical match
Dynamic joint point model

- Joint point model – classification of AOP languages
- AspectJ: dynamic j.p.m.
- Run-time events
- Containment
  - Dynamic context of a joint point
Dynamic context of a joint point

cflow( move() )
cflowbelow( move() )

• Join point selection based on dynamic semantics

before() : move() && (!
cflowbelow(move())) {
    System.out.println("moving something");
    ++counter;
}
Exercise

Give a pointcut expression

• for calling a public method of any class returning an int

• for calling a setter method in the control flow of a make* in Figure
Solution of exercise

call( public int *.*(..) )

cflow( call(* Figure.make*(..)) )
&& call( void *.set*(..) )
Advice

• Provide code to execute at a join point

• before

• after
  – if succeeds
  – if fails

• around
before advice

• Right before the join point

before() : call( * *.set*(..) ) {
    System.out.println("about to set");
}

• Before method call
• After the arguments are evaluated
after advice

• Right after the join point

```java
after() : call( * *.set*(..) ) {
    System.out.println("after setting");
}
```

• Variants for testing success
After success

```java
after() returning : call(* *.set*(..)) {
    System.out.println("setting OK");
}
```

- When method exited normally
After failure

```
after() throwing : call( * *.set*(..) ) {
    System.out.println("setting failed");
}
```

- When method exited with an exception
around advice

- Instead of, or around, join points

```java
void around() : call(void Figure.moveBy(..)) {
    System.out.print("Press Y to really move figure: ");
    try {
        if (System.in.read() == 'Y')   proceed();
    } catch (java.io.IOException e) {} 
}
```
Parametrized advice

- Formal parameter list in advice
- Bound by the pointcut

```java
before( Figure f ) :
call(* Figure.moveBy(..)) && target(f) {
    System.out.println("before move: " + f);
}
```
Exposing context in pointcuts

- With three primitive pointcuts:
  
  ```java
  this target args
  ```

  ```java
  before(Figure f, FigureElement fe, int x, int y):
  call(void FigureElement.moveBy(int,int))
  && this(f) && target(fe) && args(x,y)
  {
      ...
  }
  ```
With named pointcuts (1)

- Formal parameter list in pointcut

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

pointcut moveFE(FigureElement fe):
    move() && target(fe);
```
With named pointcuts (2)

pointcut moveFE(FigureElement fe):
    move() && target(fe);

before(FigureElement fe): moveFE(fe) {
    System.out.println("moving " + fe);
}
Pointcuts with or without parameters

pointcut setByLine() :
  call(* Point.set*(..)) && this(Line)

pointcut setByLine(Line line) :
  call(* Point.set*(..)) && this(line)

• The same join points selected
Ignoring pointcut arguments

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

pointcut setCoordNoArg(): setCoord(Point);

  No overloading...
Context in “after returning”

```java
after() returning (int n):
call(public int Point.get*()) {
    System.out.println(n);
}
```

- Capturing return value
- Type in `returning` matches type of method
import java.io.*;

aspect SimpleLogging {
    pointcut loggedCalls():
        call(public * *.*(..));

    after() throwing (IOException e):
        loggedCalls() {
            System.err.println(e);
        }
}
Exercise

• Print which FigureElement and with which vector

```java
void around() : call(void
    FigureElement.moveBy(..)) {
    System.out.print("Press Y to really move FE: ");
    try {
        if (System.in.read() == 'Y')
            proceed();
    } catch (java.io.IOException e) {} }```

Solution of exercise

```java
void around(FigureElement fe, int dx, int dy) :
  target(fe) && args(dx,dy) && call(void FigureElement.moveBy(..))
  {
    System.out.print("About to move "+fe+" with "+dx+", "+dy);
    System.out.print(". Press Y to really move figureElement: ");
    try {
      if (System.in.read() == 'Y') proceed();
    } catch (java.io.IOException e) {}
  }
```
Reminder – I

- Managing complexity: abstraction and modularization
- Better separation of concerns is needed
- Problems with DD: scattering and tangling
- Popular solution: AOSD, AOP
- Crosscutting concerns turned into aspects
- Obliviousness, join points, aspect-weaver
- AspectJ, among others…
Reminder – II

• AspectJ: pointcut, advice, aspect
• Inter-type declarations?
• a j c (get it and run it) and JVM byte code
• Pointcuts: composition of ~, named ~
  – call, cflow, cflowbelow
  – Context exposure (e.g. this, target, args)
• Named-based and property based crosscutting
• Advice:
  before, after (+returning, +throwing), around
Inter-type declarations

- Modify the static structure of the program
- Compile-time effect
- Addition of fields, methods or constructors to a class
- … or to multiple classes (crosscutting)
- Change the inheritance hierarchy
Introducing line labels

• Point, Line: geometrical properties of FigureElements
  – translate, rotate, reflect, etc.

• Labels for lines
  – relevant for displaying lines
  – another aspect (part of displaying aspect)
Labeling aspect

```java
public aspect Labeling {
    private String Line.label;
    public void Line.setLabel(String s) {
        label = s;
    }
    public String Line.getLabel() {
        return label;
    }
    ...
}
```
Inter-type scope

- Public, package-visible or private
- No protected
- Private: relative to aspect
- Package-visible: relative to aspect’s package
Changing the “implements” relation

declare parents: Point implements Clonable;

class Point implements Comparable {
    public int compareTo(Object p) {
        ...  
    }

declare parents: Point implements Comparable;
Changing the “extends” relation

```java
public aspect Labeling {
    public static class Labelled {
        private String label;
        public void setLabel(String s) {
            ...
        }
        public String getLabel() {...
    }
}

declare parents:
    (Point || Line) extends Labelled;
```
Mixin-like technique

Object

Point

Line

Labeling

Object

Labelled

Point

Line
Problem:

- Object
  - PointBase
    - Point
  - LineBase
    - Line

- LineBase
  - Line

- PointBase
  - Point

- Labelled
  - Point
  - Line

- Labeling

Diagram indicating the relationship between Point, Line, Object, PointBase, LineBase, and Labelled.
From interface to superclass

```java
public abstract class FigureElement {
  ...
}

public class Point extends FigureElement {
  ...
}

public class Line extends FigureElement {
  ...
}
```
Changing the "extends" relation

```java
public aspect Labeling {
    public static class Labelled extends FigureElement {
        private String label;
        public void setLabel(String s){...}
        public String getLabel(){...}
    }
}

declare parents:
    (Point || Line) extends Labelled;
```
Again, similar to mixins

```
FigureElement
  Point
  Line

Labeling

FigureElement
  Labelled
    Point
    Line
```
Problem again:
Crosscutting labels: default implementation

```java
public interface FigureElement { ... }

public aspect Labeling {
    private String FigureElement.label;
    public void FigureElement.setLabel(String label) {
        this.label = label;
    }
    public String FigureElement.getLabel() {
        return label;
    }
    ...
}
```
Let’s create another problem!

class FigureElement {...}
class Point extends FigureElement {...}
class Line extends FigureElement {...}
class CompoundObject {...}
class Figure extends CompoundObject {...}

• Add labels to Point, Line and Figure!
Marker interface

public aspect Labeling {
    interface Labelled {}
    private String Labelled.label;
    public void Labelled.setLabel(String s)
        { label = s; }
    public String Labelled.getLabel()
        { return label; }

    declare parents: (Point||Line||Figure)
    implements Labelled;
}
Aspects

• Unit of modularity
• Implementation of a crosscutting concern
• Pointcuts, advice, inter-type declarations
• Own methods, fields, initializers
• Default: singleton
Local state of an aspect

aspect Logging {
    OutputStream logStream = System.err;
    before(): move() {
        logStream.println("about to move");
    }
}

• A single instance of Logging
• A single logStream variable
Special variables

- E.g. \texttt{thisJoinPoint}
- Bound to an object describing current join point

```java
aspect SimpleTracing {
    pointcut traced(): call(*
    *.set*(..));
    before(): traced() {
        System.out.println("Entering: " +
        \texttt{thisJoinPoint});
    }
}
```
Concerns to implement as aspects

- Development aspects
  - Tracing, Profiling, Logging
  - Pre- and postconditions, Contract enforcement
  - Configuration management

- Production aspects
  - Change monitoring
  - Context passing
  - Providing consistent behavior
  - Collaboration-based design

- Reusable aspects
Tracing

- Info at certain events
- Code discovery, debugging, profiling
- During program development/maintenance
- Should be separated
- Should be easily modified (e.g. evolution)
- Should be pluggable
- Should be modularized
Modularized tracing

• Expressed separately
• Easy to change

aspect SimpleTracing {
    pointcut traced(): call(*
        *.set*(..));
    before(): traced() {
        System.out.println("Entering: " +
            thisJoinPoint);
    }
}
Weaving tracing

• Turning tracing on or off
• Can be removed from production built
• Alternative tracing modules
Profiling

- Flexible: programmatically

```java
aspect SetsInRotateCounting {
    int rotateCount = 0;
    int setCount = 0;
    before(): call(void Line.rotate(double))
    { rotateCount++; }
    before(): call(void Point.set*(int)) &&
        cflow(call(void Line.rotate(double)))
    { setCount++; }
}
```
Pre- and postconditions

- Design by contract

```java
aspect PointBoundsChecking {
    before(int x):
        call(void Point.setX(int)) && args(x) {
            if ( x < MIN_X || x > MAX_X )
                throw new IllegalArgumentException("x is out of bounds.");
        }
    // same for y
}
```
public class Point {
    private int x = 0, y = 0;
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    public void setXY(int x, int y) {
        this.x = x; this.y = y;
    }
    ...
}
setX and setXY handled

aspect PointBoundsChecking {

    pointcut settingX(int x):
        (call(void Point.setX(..)) && args(x)) ||
            (call(void Point.setXY(..)) &&
            args(x,*));

    before(int x): settingX(x) {
        if ( x < MIN_X || x > MAX_X )
            throw new IllegalArgumentException(
                "x is out of bounds.");
    }
}
Handle non-setter methods

- E.g. moveBy
- Using the `set` pointcut

```java
aspect PointBoundsChecking {
  before(int x): set(Point.x) && args(x) {
    if (x < MIN_X || x > MAX_X)
      throw new IllegalArgumentException("x is out of bounds.");
  }
  // same for y
}
```
Contract enforcement

• **withincode** pointcut

```java
aspect RegistrationProtection {
    pointcut register(): call(void Registry.register(FigureElement));
    pointcut canRegister():
        withincode(* Figure.make*(..));
    before(): register() && !canRegister() {
        throw new IllegalStateException(...);
    }
}
```
Static contract enforcement

- declare error, based on static information

aspect RegistrationProtection {
  pointcut register(): call(void Registry.register(FigureElement));
  pointcut canRegister():
      within code(* Figure.make*(. .));
  declare error:
      register() && ! canRegister();
}
Concept checking

- declare error
- declare warning

- Have the compiler issue (programmed) compilation errors/warnings
- Extend the compiler with additional grammatical and static semantical rules
Configuration management

- Configuration-specific aspects of code
- Each configuration implemented in an aspect definition
- Add one of them to the make file at compile time
- Or even at load time
Reminder – I

• Managing complexity: abstraction and modularization
• Better separation of concerns is needed
• Problems with DD: scattering and tangling
• Popular solution: AOSD, AOP
• Crosscutting concerns turned into aspects
• Obliviousness, join points, aspect-weaver
• AspectJ, among others…
Reminder – II

• AspectJ: pointcut, advice, aspect
• Inter-type declarations?
• ajc (get it and run it) and JVM byte code
• Pointcuts: composition of ~, named ~
  – call, cflow, cflowbelow
  – Context exposure (e.g. this, target, args)
• Named-based and property based crosscutting
• Advice:
  before, after (+returning, +throwing), around
Reminder – III

- Inter-type declarations
  - Insertion of methods and attributes
  - Changing the inheritance relation
- Mixins and multiple code inheritance
- Development aspects
  - Tracing, logging and profiling
  - Design by contract, Contract enforcement
  - Configuration management
- Pointcuts: withincode, set
- Extending the compiler with errors and warnings
- Aspects: visibility modifiers, instantiation of ~
Production aspects

- Used in production builds
- Add real functionality to applications
- E.g. the Labeling aspect
- Further examples
  - Change monitoring
  - Context passing
  - Providing consistent behaviour
  - Security
  - Resource pooling, caching
  - Synchronization of threads
Change monitoring

- Indicate whether any of the FigureElement has moved since last display
- Dirty flag introduced
- Setting the dirty flag at each method that moves a figure element
Implementation in an aspect

aspect MoveTracking {
    private static boolean dirty = false;
    public static boolean testAndClear() {
        boolean result = dirty; dirty = false;
        return result;
    }
    ...
    after() returning: move() { dirty = true; }
}
Advantages

• The structure of the concern is made explicit
• Evolution is easier [ECOOP 01]
• Plug in or out
• More stable implementation
Context passing

• Set the color of FigureElements upon creation
• Pass a color (or a color factory) to `make*`
• This may influence many methods: on the control flow from client to `make*`
• Non-AOP solution: additional arg to those methods
• AOP solution: pass information between far-away code fragments
Passing context with aspect

aspect ColorControl {

    after (ColorControllingClient c) returning (FigureElement fe):
    call(* Figure.make*(..)) &&
    cflow( call(* * (..)) && this(c) )
    { fe.setColor(c.colorFor(fe)); } 

}
A fragment of ajc

aspect ContextFilling {
    pointcut parse( JavaParser jp ):
    call(* JavaParser.parse*(..)) && target(jp)
    && !call(Stmt parseVarDec(boolean));
    around(JavaParser jp) returns ASTObject: parse(jp) {
        Token beginToken = jp.peekToken();
        ASTObject ret = proceed(jp);
        if (ret != null) jp.addContext(ret, beginToken);
        return ret;
    }
}
Consistent behaviour

- Advising 35 methods
  - `parseMethodDec`, `parseThrows`, `parseExpr` etc.
- Explicit exclusion of `parseVarDec`
- Clear expression of a clean crosscutting modularity
- Java → AspectJ refactoring revealed two bugs
Synchronization

• Not thread-safe

```java
public class Account {
    ...
    public int getBalance(){...}
    public void deposit( int amount )
    {
    }
    public void withdraw( int amount )
    {
    }
}
```
Mutual exclusion

```
aspect Mutex {
    Object around(Account a):
    call(* Account.*(..)) && target(a) {
        synchronized(a){
            return proceed(a);
        }
    }
}
```

- **Object** has special meaning in this case!
Readers/writers (1)

private int Account.readers = 0;
private boolean Account.writing = false;

pointcut reader(Account a):
    target(a) && call(int getBalance());

pointcut writer(Account a):
    target(a) &&
    ( call(void deposit(int)) ||
    call(void withdraw(int)) );
Readers/writers (2)

```java
before(Account a): reader(a) {
    boolean canGo = false;
    while( !canGo )
        synchronized(a){
            canGo = !a.writing;
            if( canGo ){ a.readers++; }
            else {  try { a.wait(); } }
            catch (Exception e){}
        }
    }
}
```
Evaluation of Mutex and R/W

- Separation of concerns
- Decreased redundancy
- Aspect-private variables
- Easier to improve

- Tied to Account
Reusable aspects

- Implementation of “reusable” concerns
- Binding to base code
- Interference with other aspects
- E.g. mutual exclusion, readers/writers
Abstract pointcut/aspect

abstract aspect Mutex {

  abstract pointcut callToSync(Object o);

  Object around(Object o): callToSync(o){
    synchronized(o){
      return proceed(o);
    }
  }

}
Extending an aspect

aspect MutexForAccount extends Mutex {

  pointcut callToSync(Object o):
    call(* Account.*(..)) && target(o);

}
Monitor versus Critical section

- So far: monitor-like mutex
  - 1 semaphore for the methods of 1 object

- Critical section
  - More control over the semaphores
  - E.g. mutex for all methods of all accounts
More reusable Mutex

abstract aspect Mutex {
    interface Sync{ Object getSemaphore(); } 

    abstract pointcut callToSync(Sync s); 

    Object around(Sync s): callToSync(s){
        synchronized(s.getSemaphore()){ 
            return proceed(s); 
        }
    }
}
Critical section for Account

```java
aspect CritSecForAccount extends Mutex {
    private static Object s = new Object();
    public Object Account.getSemaphore() {
        return s;
    }

declare parents: Account implements Sync;

    pointcut callToSync(Sync a):
        call(* Account.*(..)) && target(a);
}
```
Monitor for Account

aspect MonitorForAccount extends Mutex {
  public Object Account.getSemaphore() {
    return this;
  }
  declare parents:
    Account implements Sync;
  pointcut callToSync(Sync a):
    call(* Account.*(..)) && target(a);
}
abstract aspect RW {

    interface Sync {}

    private int Sync.readers = 0;
    private boolean Sync.writing = false;

    abstract pointcut reader(Sync s);
    abstract pointcut writer(Sync s);

    ...

Reusable R/W (2)

...  
before(Sync s): reader(s)  
  boolean canGo = false;  
    while( !canGo )  
      synchronized(s){  
        canGo = !s.writing;  
        if( canGo ) s.readers++;  
        else try { s.wait(); }  
          catch (Exception e){}  
      }  
}
Exercise

- Define more reusable R/W implementation
  - More control over object to synchronize on
- Define normal (per account) rw synch.
- Define per Account class rw synch.
The Subject/Observer design pattern

```java
public abstract aspect ObserverProtocol {
    protected interface Observer {} 
    protected interface Subject {} 
    private List<Observer> Subject.observers = new LinkedList<>(); 
    public void Subject.addObserver(Observer o) {
        observers.add(o); 
    }
    public void Subject.removeObserver(Observer o) {
        observers.remove(o); 
    }
    protected abstract void notifyObserver(Observer o, Subject s);
    protected abstract pointcut observedEvent(Subject s);
    after(Subject s) returning: observedEvent(s) {
        for (Observer o: s.observers) notifyObserver(o, s); 
    }
}
```

May 21 - June 1, 2007.  Aspect-Oriented Programming and the AspectJ (Tamás Kozsik)  143/300
aspect Binding extends ObserverProtocol {
    declare parents: Point implements Subject;
    declare parents: Point implements Observer;

    protected pointcut observedEvent(Subject s):
        set(int Point.x) && target(s);

    protected void notifyObserver( Observer o,
        Subject s ){
        ((Point)o).x = ((Point)s).x;
    }
}
Problems to solve

• With ObserverProtocol
  – If a class is a Subject in two independent bindings...

• With Binding
  – If two different bindings interfere (notification in one of them triggers notification in the other one and vice versa)
AspectJ 5

• **Supports Java5 features**
  – Generics
  – Annotations
  – Covariant return types
  – Auto-(un)boxing
  – Varargs
  – Enumeration types

• **Generics + aspects = ?**

• **Annotations + aspects = ?**
Parent/child relationship (1)

```java
public abstract aspect ParentChildRelationship <Parent, Child> {

    public interface ParentHasChildren <C extends ChildHasParent> {
        Set<C> getChildren();
        void addChild(C child);
        void removeChild(C child);
    }

    public interface ChildHasParent <P extends ParentHasChildren> {
        P getParent();
        void setParent(P parent);
    }

    ...
}
```
Parent/child relationship (2)

```java
public abstract aspect ParentChildRelationship <Parent,Child> {
    public interface ParentHasChildren <C extends ChildHasParent>
    { ... }

    public interface ChildHasParent <P extends ParentHasChildren>
    { ... }

    declare parents:
    Parent implements ParentHasChildren<Child>;

    declare parents:
    Child implements ChildHasParent<Parent>;

    ...
}
```
Parent/child relationship (3)

```java
private Set<C> ParentHasChildren<C>.children = new HashSet<C>();

private P ChildHasParent<P>.parent;

public Set<C> ParentHasChildren<C>.getChildren() {
    return Collections.unmodifiableSet(children);
}

public P ChildHasParent<P>.getParent() {
    return parent;
}
```
Parent/child relationship (4)

```java
public void ParentHasChildren<C>.addChild(C child) {
    if (child.parent != null)
        child.parent.removeChild(child);
    children.add(child);
    child.parent = this;
}

class ParentHasChildren<T> {
    // Additional methods...
}

class ChildHasParent<T> {
    public void setParent(T parent) {
        parent.addChild(this);
    }
    // Additional methods...
}
```
Pointcut designators – so far...

- Describing events
  - Calling a method: call
  - Setting a field: set
- Describing context
  - Static context: withincode
  - Dynamic context: cflow, cflowbelow
  - Exposing context: this, target, args, returning, throwing
- Composing pointcuts (&&, ||, !)
Picking out further join points:

kinded pointcuts describe events

call (MethodPattern)
call (ConstructorPattern)
execution (MethodPattern)
execution (ConstructorPattern)
set (FieldPattern)
get (FieldPattern)
initialization (ConstructorPattern)
preinitialization (ConstructorPattern)
staticinitialization (TypePattern)
handler (TypePattern)
adviceexecution ()
Accessing fields

\[ \text{set (} FieldPattern \text{)} \]
\[ \text{get (} FieldPattern \text{)} \]

\[ \text{set ( } \text{int Point.x } \text{)} \]
\[ \text{get ( } * \text{ Point.* } \text{)} \]
Initializing objects and classes

• Objects
  – constructor call/execution
  – (pre)initialization

• Classes: when loading a class

\[
\text{static initialization}( \text{TypePattern} ) \\
\text{static initialization}( \text{java.util.*} )
\]
Exception handling

- When catching an exception
- Only before advice

handler (TypePattern)

handler( java.sql.SQLException )
handler( IOException && !EOFException )
handler( java.lang.*Error )
Call versus Execution (1)

- Call: in the caller
- Execution: in the callee

- Additional pointcut clauses can make a difference
- Call+restrictions: not every execution of a method

```java
execution( * Point.get*() )
call( * Point.get*() ) && this(Line)
```
Method versus Constructor

call (MethodPattern)
call (ConstructorPattern)
execution (MethodPattern)
execution (ConstructorPattern)

call ( * Point.setX(..) )
call ( Point.new(..) )
Example

- example-weak
- A Huge object stores a byte array
- First aspect: add IllegalArgumentException
- Second aspect: optimize representation
Privileged aspects

- Can access members that are otherwise not visible
- In aspect `OptimizeHuge`
  field data of `Huge` is accessed

privileged aspect ... { ... }

May 21 - June 1, 2007. Aspect-Oriented Programming and the AspectJ (Tamás Kozsik) 159/300
Reminder

• Introduction to, and justification of, AOP
• AspectJ
  – The ajc aspect-weaver / compiler
  – Join points
  – Pointcuts, advice, inter-type declarations, aspect
• Development, production and reusable aspects
• AspectJ keywords:
  – call, execution, get, set, preinitialization, initialization, staticinitialization, handler, adviceexecution
  – within, withincode, cflow, cflowbelow, if, @within, @withincode
  – this, target, args, @this, @target, @args, @annotation
  – && | ! * .. +
  – before, after (throwing, returning), around (proceed)
  – pointcut, aspect (privileged, perthis…)
  – declare parents, error/warning, precedence, soft, @type…
  – thisJoinPoint, thisJoinPointStaticPart…
Kinded pointcut designators

- call
- execution
- get
- set
- preinitialization
- initialization
- staticinitialization
- handler
- adviceexecution
Construction of objects

- **Constructor call**
  - not in case of `this` or `super` calls
  - returns an object of the class
- **Constructor execution**
  - object accessible by pointcut `this`
- **Object pre-initialization**
- **Object initialization**
  - object accessible by pointcut `this`
The process of the construction

- allocation and default initialization
- call constructor
- call this(..)  
  *(optional: 0, 1 or more)*
- call super(..)
- ...
- return from super(..)
- execute initializers
- execute rest of this(..) and return  
  *(optional: 0, 1 or more)*
- execute rest of constructor
- return from constructor
“Constructor call” join point

- allocation and default initialization
- call constructor
- call this(..)
- call super(..)
- ...
- return from super(..)
- execute initializers
- execute rest of this(..) and return
- execute rest of constructor
- return from constructor
“Object pre-initialization” join point

- allocation and default initialization
- call constructor
- `call this(..)`
- call super(..)
- ...
- return from super(..)
- execute initializers
- execute rest of this(..) and return
- execute rest of constructor
- return from constructor
“Object initialization” join point

- allocation and default initialization
- call constructor
- call this(..)
- call super(..)
- …
- return from super(..)
- execute initializers
- execute rest of this(..) and return
- execute rest of constructor
- return from constructor
“Constructor execution” join point

- allocation and default initialization
- call constructor
- call this(..)
- call super(..)
- ...
- return from super(..)
- execute initializers
- execute rest of this(..) and return (if there is at least one)
- execute rest of constructor
- return from constructor
“Constructor execution” join point

- allocation and default initialization
- call constructor

- call super(..)
- ...
- return from super(..)
- execute initializers

- execute rest of constructor
- return from constructor
Call versus Execution (2)

- Constructors: when `super` or `this` is called
  - there is no `constructor call` join point
  - there is a `constructor execution` join point

- Methods: when called on `super`
  - there is no `method call` join point
  - there is a `method execution` join point
Join points and subtyping

- Join points are subtype-aware

```java
class B { void m(){} void n(){} }
class C extends B { void m(){} }
aspect A {
    after() : call(void B.m()) { ... }
    after() : call(void B.n()) { ... }
    after() : call(void C.m()) { ... }
    after() : call(void C.n()) { ... }
}
(new C()).m() (new C()).n()
```
Call versus Execution (3)

class B { void m(){} void n(){} }
class C extends B { void m(){} }
aspect A {
    after() : call(void B.m()) { ... }
    after() : call(void B.n()) { ... }
    after() : call(void C.m()) { ... }
    after() : call(void C.n()) { ... }
    after() : execution(void B.m()) { ... }
    after() : execution(void B.n()) { ... }
    after() : execution(void C.m()) { ... }
    after() : execution(void C.n()) { ... }
}

(new C()).m() (new C()).n()
Context-related pointcuts

- Exposing context and matching dynamic type: this, target, args, (returning, throwing)

- Lexical structure (static context)
  - `withinCode(MethodPattern or ConstructorPattern)`
  - `within(TypePattern)`

- Runtime structure (dynamic context)
  - `cflow(Pointcut)`
  - `cflowBelow(Pointcut)`
  - `if(boolean expr)`
Exposing context

• Primitive pointcuts
  - `this (Type or Id)`: currently executing object
  - `target (Type or Id)`: whose member is accessed
  - `args (Type or Id or ".." or "*", ...)`: arguments of method call / field set / exc. handler

• after returning/throwing, around
Static type and dynamic type

- Kinded pointcut designators: static types
- this, target, args: dynamic type

call( boolean String.equals(Object) )
call( boolean *.equals(Object) ) &&
target(String)

... "hello".equals("hi") ...

Object o = "hello"; ... o.equals("hi") ...
Examples

args(int,String)

pointcut setter(int v):
    call(void *.*set*(..)) && args(v);

args(int,*)

args(int,..,String)

args(int, Object)

pointcut setter(Object v): // Object has special
    call(void *.*set*(..)) && args(v); // meaning here
Generics

- Parametrized types are not allowed in `this` and `target` pointcut designators
  - Type erasure in Java 5
  - No dynamic type information on type parameters
- Parametrized types are allowed in `args`
- Might result in an `unchecked` warning (can be suppressed…)
- Requiring static guarantees help avoid the warning
aspect A {
    static class C<T> {
        void m( C<T> p ){};
        void n( C<T> p ){ m(p); } 
    }
    before() : call( void m(..) )
    && this(C/*<String>*/ &&
    target(C/*<String>*/
    && args(C<String>)
    {
        System.out.println(thisJoinPoint);
    }
    public static void main( String[] args ){
        new C<String>().n(new C<String>());
        new C<Integer>().n(new C<Integer>());
    }
}
import java.util.*;

class C {
    void foo(List<String> p){}
    void bar(List<Double> p){}
    void goo(List<? extends Number> p){}
}

aspect B {
    before(List<Double> list)
        : execution( void C.*(..) ) && args(list) {
            for( Double d: list ){ /* ... */ }
        }

    public static void main( String[] args ){
        C c = new C();
        c.foo(new ArrayList<String>());
        c.bar(new ArrayList<Double>());
        c.goo(new ArrayList<Double>());
        c.goo(new ArrayList<Integer>());
    }
}
import java.util.*;

class C {
    void foo(List<String> p){}
    void bar(List<Double> p){}
    void goo(List<? extends Number> p){}
}

aspect B {
    before(List<Double> list)
    : execution( void C.*(List<Double>)) &&
    args(list) {
        for( Double d: list ) { /* ... */ }
    }
}

public static void main( String[] args ){
    C c = new C();
    c.foo(new ArrayList<String>());
    c.bar(new ArrayList<Double>());
    c.goo(new ArrayList<Double>());
    c.goo(new ArrayList<Integer>());
}

May 21 - June 1, 2007. Aspect-Oriented Programming and the AspectJ (Tamás Kozsik) 179/300
aspect A {
    static class C<T> {
        void m( C<T> p ){}
        void n( C<T> p ){ m(p); }
    }
    before() : call( void m(C<String>) )
                && this(C/*<String>*/) &&
                target(C/*<String>*/) &&
                args(C<String>)
    {
        System.out.println(thisJoinPoint);
    }
    public static void main( String[] args ){
        new C<String>().n(new C<String>());
        new C<Integer>().n(new C<Integer>());
    }
}
<table>
<thead>
<tr>
<th>Join Point</th>
<th>Current Object</th>
<th>Target Object</th>
<th>Arguments</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Call</td>
<td>executing object*</td>
<td>target object**</td>
<td>method arguments</td>
<td>method result</td>
</tr>
<tr>
<td>Method Execution</td>
<td>executing object*</td>
<td>executing object*</td>
<td>method arguments</td>
<td>None</td>
</tr>
<tr>
<td>Constructor Call</td>
<td>executing object*</td>
<td>None</td>
<td>constructor arguments</td>
<td>constructed object</td>
</tr>
<tr>
<td>Constructor Execution</td>
<td>executing object</td>
<td>executing object</td>
<td>constructor arguments</td>
<td>None</td>
</tr>
<tr>
<td>Static initializer execution</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Object pre-initialization</td>
<td>None</td>
<td>None</td>
<td>constructor arguments</td>
<td>None</td>
</tr>
<tr>
<td>Object initialization</td>
<td>executing object</td>
<td>executing object</td>
<td>constructor arguments</td>
<td>None</td>
</tr>
<tr>
<td>Field reference</td>
<td>executing object*</td>
<td>target object**</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Field assignment Handler</td>
<td>executing object*</td>
<td>executing object*</td>
<td>assigned value</td>
<td>None</td>
</tr>
<tr>
<td>Advice execution</td>
<td>executing aspect</td>
<td>executing aspect</td>
<td>advice arguments</td>
<td>None</td>
</tr>
</tbody>
</table>
Lexical structure

• withincode (*MethodPattern*)
• withincode (*ConstructorPattern*)
• within (*TypePattern*), e.g. within (Point)
• When the executing code belongs to Point

```java
pointcut jdbcCall():
  call(* java.sql.*(..))
  || call(* javax.sql.*(..));

pointcut inDataLayer():
  within(com.xyz.persistence..*);

declare warning: jdbcCall() && !inDataLayer() :
  "Illegal JDBC call";
```
within versus sender

within( Point ) && call( * *.get*() )
• Call of a getter in the Point class

this( Point ) && call( * *.get*() )
• Call of a getter by a Point object
Exercise

!within(Point) &&
this(Point) &&
call(**.get*() )

within(Point) &&
!this(Point) &&
call(**.get*() )
Runtime structure

• Based on control flow:
  - cflow()
  - cflowbelow()

• Based on run-time value:
  \[ \text{if} \ ( \text{boolean-expr} ) \]
The \texttt{if} pointcut designator

\begin{verbatim}
abstract class Figure {
    boolean isWhite(){ ... }
...
}

aspect ChessTracer {
    pointcut whiteSteps(Figure f):
        call(void Figure.moveTo(int,int)) &&
        target(f) && if(f.isWhite());
...
}
\end{verbatim}
Exercise

cflow(P && Q)        cflow(P) && cflow(Q)
cflow(P || Q)        cflow(P) || cflow(Q)
cflow(! P)           ! cflow(P)
Kinded pointcut designators

• Matching is based on the kind of the join points
• Pointcuts contain patterns
  – Signature
  – Modifiers
• Multiple signatures, one set of modifiers
• Matching signatures on static types

• call, execution, get, set, preinitialization, initialization, staticinitialization, handler, adviceexecution
Multiple signatures

• Method call, method execution, field get, field set
• Others have only one…

... "hello".hashCode() ...

call(int String.hashCode())
call(int Object.hashCode())
call(* hash*(..))
set and get

class P { String s; }
class S extends P { String s; }
class T extends S {

(new T()).s

String S.s
String T.s
The signatures of a call join point

- **call** join point for \( t.m("hello") \)
- if the static type of \( t \) is \( T \)
- and \( A \) is a supertype of \( T \)
- and \( R \ m(String) \) is defined for type \( A \)
- then \( R \ A.m(String) \) is a signature
Covariant return type (Java5)

- **call** join point for \( t.m("hello") \)

- if the static type of \( t \) is \( T \)
- and \( A \) is a supertype of \( T \)
- and \( R \) \( m(String) \) is defined for \( A \)
- (maybe \( R' \) \( m(String) \) is defined for \( T \),
  where \( R \) is a supertype of \( R' \))

- **then** \( R A.m(String) \) is a signature
Call versus Execution (4)

- **execution** join point for \( t.m("hello") \)

- If the static type of \( t \) is \( T \)
- And \( A \) is a supertype of \( T \)
- And \( R.m(String) \) is declared in type \( A \)

- Then \( R.A.m(String) \) is a signature
class B { void m(){} void n(){} }
class C extends B { void m(){} }
aspect A {
    after() : call(void C.n()) { ... }
    after() : execution(void C.n()) { ... }
}

(new C()).n()
Examples

... "hello".equals("hi") ... 

call(boolean
  String.equals(String)) // no

call(boolean
  String.equals(Object)) // yes

call(boolean
  Object.equals(Object)) // yes
Static type and dynamic type - revisited

- Kinded pointcut designators: static types
- this, target, args: dynamic type

```java
call( boolean String.equals(Object) )
call( boolean *.equals(Object) ) &&
    target(String)

... "hello".equals("hi") ...

Object o = "hello"; ... o.equals("hi") ...
```
Examples

class B { B copy(){ ... } }
class C extends B { C copy(){ ... } }

... (new C()).copy() ...

call(C C.copy()) // yes
call(B B.copy()) // yes

call(B C.copy()) // no
call(C B.copy()) // no
Examples

class B { B copy() { ... } }
class C extends B { C copy() { ... } }

B b = new C(); ... b.copy() ...

call(C C.copy()) // no
call(B B.copy()) // yes

call(B C.copy()) // no
call(C B.copy()) // no
The number of signatures

- **call**: the number of types for which the method is defined
- **execution**: the number of types which (re-)defines the method
Reminder

• Context exposure and dynamic type matching
• Lexical structure
• Run-time structure
• Kinded pointcut designators

• Matching join points on patterns in pointcuts
  – Signatures (maybe multiple)
  – Modifiers
Examples

interface Q
{
    Object m(String s);
}
class P implements Q
{
    Object m(String s);
    ...;
}
class S extends P
{
    String m(String s);
    ...;
}
class T extends S
{
}
class U extends T
{
    String m(String s);
    ...;
}

call new U().m("hello") execution

Object Q.m(String) Object Q.m(String)
Object P.m(String) Object P.m(String)
String S.m(String) String S.m(String)
Example revisited

• **call** join point for \( t.m("hello") \)

• if the static type of \( t \) is \( T \)
• and \( A \) is a supertype of \( T \)
• and \( R \ m(String) \) is defined for \( A \)
• (maybe \( R' \ m(String) \) is defined for \( T \), where \( R \) is a supertype of \( R' \))

• **then** \( R \ A.m(String) \) is a signature
Static dispatch

- **call** join point for `t.m("hello")`

- if the static type of `t` is `T`
- and `A` is a supertype of `T`
- and `R m(String)` is not defined for `T`
- and `R m(Object)` is defined for `A`
- (maybe `R' m(Object)` is defined for `T`, where `R` is a supertype of `R'`)
- **then** `R A.m(Object)` is a signature
Generics

• Signatures may contain type variables
  – Generic methods
  – Generic constructors
  – Members (methods and fields) and constructors of generic types

• Signatures may contain parametrized types
  – Types of fields
  – Argument types of constructors and methods
  – Return types of methods
Signatures containing type variables

class Seq<T> {
    T value;
    Seq<T> next;

    Seq(T Value){ this.value = value; }
    Seq(Seq<T> fst, Seq<T> snd ){ /* concat */ }
    <S extends T> Seq(Seq<S> s){ /* upcast */ }

    static <T> Seq<T> concat(Seq<T> fst, Seq<T> snd ){...}
    static <T, S extends T> Seq<T> upcast(Seq<S> s){...}

    T head(){ return value; }
    Seq<T> copy(){...}
    void append(Seq<T> s){...}
}
Join point signatures: by erasure

- Type erasure:
  No run-time info about type parameters in Java 5
- Signatures of members and constructors that contain type variables are matched by their erasure

```java
class Seq<T> {
    T value
    ...
    void append(Seq<T> s) {...} void Seq.append(Seq)
}
```
Signatures containing parametrized types

class Boss extends Employee {
    Set<Employee> slaves = ...  
    List<Integer> bonuses = ...  
    int computeBonuses( List<Boolean> received ) ...  
    void addSubordinates( Set<? extends Employee> subs ) ...
}

Set<Employee> Boss.slaves
List<Integer> Boss.bonuses
int Boss.computeBonuses(List<Boolean>)
void Boss.addSubordinates(List<? extends Employee>)
Modifiers

- public/protected/private, static, abstract...
- throws clause
- annotations
Modifier set of a join point

public class X {
    @Foo protected void doIt() {...}
}

public class Y extends X {
    public void doIt() {...}
}

(new X()).doIt() → {@Foo, protected}
(new Y()).doIt() → {public}
Patterns

execution(\textit{MethodPattern})

execution(\textit{ConstructorPattern})

set(\textit{FieldPattern})

within(\textit{TypePattern})

• Pattern matching and wildcards
Matching pointcuts and join points

• A join point has
  – (Possibly multiple) signature(s)
  – A set of modifiers

• Patterns in pointcuts refer to
  – Signatures
  – Modifiers

• A pattern matches a join point, if
  – It matches (one of) its signature(s)
  – It matches its modifiers
Type patterns

• Exact type names
  – Qualified names, import statements
  – Subtyping: call vs. within

• Composing type patterns:
  FigureElement && !(Point || Line)

• Including subtypes: Point+

• Array types: Point[]

• Wildcards: * and ..
Wildcards in type patterns

- Type names consist of name components (packaging and nesting)

```java
java.util.*  java.lang.*Error
java.*.List   java..List
java..*        java..Map.*
```
Refering to subtypes

call(*Handler+.new(..))

call( (Foo+ && !Foo).new(..) )
Only from subtypes

pointcut callToUndefinedMethod():
    call(* Point+.*(..)) &&
    !call(* Point.*(..));

pointcut executionOfUndefinedMethod():
    execution(* *(..)) &&
    within(Point+) &&
    !within(Point);
Excluding subtypes

pointcut callNotOnSubtype(Point p):
call(* *(..)) && target(p) &&
if(p.getClass() == Point.class);
Generics and type patterns

• A raw type (e.g. `List`) in a pattern matches
  – The raw type (e.g. `List`)
  – The generic type (e.g. `List<E>`)  
  – The parametrized types (e.g. `List<String>, List<?> >, List<?> extends Number> etc.)

• A parametrized type (e.g. `List<String>`) in a pattern matches
  – Parametrized types with appropriate parameters

• No type variables in type patterns
Examples

<table>
<thead>
<tr>
<th>types</th>
<th>patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>ArrayList</td>
</tr>
<tr>
<td>Set&lt;Employee&gt;</td>
<td>Set&lt;Employee&gt;,</td>
</tr>
<tr>
<td>List&lt;? extends Employee&gt;</td>
<td>List&lt;? extends Employee&gt;,</td>
</tr>
<tr>
<td></td>
<td>List&lt;&gt;,</td>
</tr>
<tr>
<td>Seq&lt;T&gt;</td>
<td>Seq&lt;&gt;,</td>
</tr>
</tbody>
</table>

May 21 - June 1, 2007. Aspect-Oriented Programming and the AspectJ (Tamás Kozsik) 218/300
### Generic wildcards and pattern wildcards

<table>
<thead>
<tr>
<th>Types</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Set&lt;Boss&gt;</code></td>
<td><code>Set&lt;Boss&gt;,</code></td>
</tr>
<tr>
<td></td>
<td><code>Set</code></td>
</tr>
<tr>
<td></td>
<td><code>*</code></td>
</tr>
<tr>
<td></td>
<td><code>Set&lt;*&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>Set&lt;Employee&gt;</code>,<code>&gt;</code></td>
</tr>
<tr>
<td><code>List&lt;? extends Boss&gt;</code></td>
<td><code>List&lt;? extends Boss&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>List</code></td>
</tr>
<tr>
<td></td>
<td><code>*</code></td>
</tr>
<tr>
<td></td>
<td><code>List&lt;*&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>List&lt;Employee&gt;</code></td>
</tr>
<tr>
<td><code>+&gt;</code></td>
<td><code>List&lt;? extends Employee&gt;</code></td>
</tr>
</tbody>
</table>
Field patterns

• Signature of field access join points
  – Name
  – Type
  – Qualifying type

• Wildcards
  int Point.x
  * Point.*
  int *.*Id

• Setting a field: args
Examples

class Boss extends Employee {
    List<Integer> bonuses = ...
}

List<Integer> Boss.bonuses
List Boss.bonuses
* Boss.bonuses
* *.bonuses
* *.bonus*
List<*> Boss.bonuses
List<Number> Boss.bonuses

class Seq<T> {
    T value;
    Seq<T> next;
}

Seq Seq.next
* Seq.next
* *.xt
Method and constructor patterns

\texttt{execution}(\textit{MethodPattern})

\texttt{execution}(\textit{ConstructorPattern})

• Based on the join point signatures and set of modifiers
  – modifiers
  – argument (and return) types
  – declared exceptions
  – qualifying type
  – name
  – annotations
Pattern matching method signatures

static class G<T> {
    List<T> foo( List<Integer> list ) { return null; }
}

new G<String>().foo(null);

call( List ) foo(List) )
call( List ) foo(List<Integer>) )
call( * ) foo(List<*>) )
call( List<*>) foo(List<Integer>) )
call( * ) foo(List<Number+>) )
call( List<Number+>) foo(List<Number+>) )
call( List ) G.foo(List<Number+>) )
call( List ) /*.o*(List<Object+>) )
Declaring type

class Service implements Runnable {
  public void run() { ... }
}

Runnable[] threads = new Runnable[10];
threads[0] = new Service();
threads[0].run();

call(void Service.run())
call(void Runnable.run()) && target(Service)
Matching on “modifiers”

- Modifiers
  - public, protected, private, static, abstract, ...
- Throws clauses
- Annotations (Java5, AspectJ5)

- A join point has one set of “modifiers”
- A pattern might
  - require
  - disallow
  - ignore
  the presence of each modifier
Example: matching modifiers

- Select all public getters
  \[
  \text{call}( \text{public } \ast \text{get}() )
  \]

- Select all non-public getters
  \[
  \text{call}( \neg \text{public } \ast \text{get}() )
  \]

- Select all getters (no matter whether public or not)
  \[
  \text{call}( \ast \text{get}() )
  \]
Matching methods

call( public final void C.foo() 
   throws java.io.IOException )

call( !public !final * *.*(..) 
   throws java.io.IOException )

call( * create*(..,int) throws 
   (Exception && !SecurityException) )

call( !public !protected !private * *(..) )

May 21 - June 1, 2007. Aspect-Oriented Programming and the AspectJ (Tamás Kozsik) 227/300
Not package-visible setter method

call(!public !protected !private *(..) )

call((public  *(..)))
  || (protected *(..))
  || (private *(..))
  )
Negation in throws clause

call( * *(..) throws !IOException ) // N

call( * *(..) throws (!IOException) ) // Y

void m() throws SQLException, IOException
Matching constructors

call( Foo.new() )

call( Foo+.new() )

call( !public *Handler+.new(int,..) throws java.io.IOException )
Auto-(un)boxing

Call to

    public void foo( Integer i )

is not matched by

    call( void foo(int) )

is matched by

    call( void foo(…) && args(int) )

\[
\text{args(Integer)} \equiv \text{args(int)}
\]
Auto-(un)boxing

```java
public void foo( Integer I )

pointcut foo(int i) :
    call (* *(..)) && args(i)

before(Integer i) : foo(i) { ... }
```
Static method dispatch

- Without autoboxing, autounboxing, vararg
- With auto-(un)boxing, without vararg
- With all of them

```java
class C {
    void m( long i )
    void m( Integer i )
}
... m(3) ...
```
Varargs

withincode( * hu.elte.inf.kto..*(int, String...) )

initialization ( hu.elte.inf.kto..new((Foo|| Goo)...)) )

void foo(String...) call(*(String...))

void bar(String[]) execution(*(String[]))

args(String[])

before(String[] s) : call(*(String...)) &&
    args(s) {...}

before(String[] s) : call(*(String[])) &&
    args(s) {...}

before(String[] s) : call(*(String[])) &&
    args(s) {...}
Summary of pointcuts

• Selecting sets of join points (quantification)
• Primitive pointcut designators + composition
• Kinded p. d., lexical/run-time struct., ctx. exposure
• Named pointcuts
  – In aspects and in classes
  – Might be abstract (in abstract aspects) or final
  – Might expose context for advice bodies
  – No overloading
  – Scope: enclosing aspect/class definition
• Pattern matching
Scope of pointcut declarations

• The whole enclosing aspect/class definition

aspect A percflow( mOfFoo() ) {
    pointcut mOfFoo(): call(* Foo.m(..));
    ...
}

May 21 - June 1, 2007.   Aspect-Oriented Programming and the AspectJ (Tamás Kozsik)   236/300
Reminder

- Introduction to, and justification of, AOP
- AspectJ
  - ajc compiler/aspect-weaver, other tools
  - AspectJ by examples (join points, pointcuts, advice, inter-type declarations, aspects)
  - Development/production/reusable aspects
  - Thorough coverage of pointcuts
  - More details on declare statements and aspects
  - Annotations
- Other AOP(-like) solutions
Reflective information about join points

- `thisJoinPoint`,
- `thisJoinPointStaticPart`,
- `thisEnclosingJoinPointStaticPart`

- `org.aspectj.lang.reflect`

- Useful for tracing
- Useful for metaprogramming
- Information about
  - signatures, names, types
  - program locations
Reflective programming

```java
private final HashMap constraints = new HashMap();
{
    constraints.put( "x", new int[]{1,640} );
    constraints.put( "y", new int[]{1,480} );
}

before (int n) : set(int Point.*) && args(n) {
    String attr =

    thisJoinPointStaticPart.getSignature().getName();
    int[] bounds = (int[]) constraints.get( attr );
    if ( n < bounds[0] || n > bounds[1] )
        throw new IllegalArgumentException( attr +
```
Preconditions revisited

• Checking for null arguments
• Solution in JML with ESC/Java2
• Solution by aspects
• Pre-condition $\rightarrow$ exception
• Reusable aspects?
Checking for null arguments: JML

```java
package dblayer;
import java.sql.*;
public class DB {
  /*@ requires name != null */
  public static void insertEmployee
    ( String name, boolean permanent ) throws SQLException {
    Connection c = DriverManager.getConnection
      ( "jdbc:mysql://localhost/hrdb" );
    try {
      Statement s = c.createStatement();
      try {
        s.executeUpdate("INSERT INTO employee VALUES
          ( ",
            + name + "," + permanent + 
          ")" );
      } finally { s.close(); }
    } finally { c.close(); }
  }
}
```
Insertion of run-time checks
[Kis’02]

public aspect NullChecker {
    pointcut dbcalls(): call(* dblayer.*.*(..));
    before() : dbcalls() {
        Object[] args = thisJoinPoint.getArgs();
        for (Object arg : args) {
            if (null == arg) {
                throw new IllegalArgumentException
                    ("null values are not allowed!");
            }
        }
    }
}
Aspect instances

- Aspects cannot be instantiated with \texttt{new}
- By default: singleton

\begin{verbatim}
aspect Logging { ... }
aspect Logging issingleton() { ... }
\end{verbatim}

- Per-object, Per-control-flow and Per-type aspects

\begin{verbatim}
aspect A perthis(Pointcut) { ... }
aspect B pertarget(Pointcut) { ... }
aspect C percflow(Pointcut) { ... }
aspect D percflowbelow(Pointcut) { ... }
aspect D pertypewithin(TypePattern) { ... }
\end{verbatim}
Precedence

declare precedence : TypePatternList ;

declare precedence: *..*Security*, Logging+, *

- Only for concrete aspects
- Order of advice in aspect
- Subaspects have higher precedence
Idiom for independent aspects

```
declare precedence: A, B;
declare precedence: B, A;
```

```
declare precedence: A, B, A;  // illegal
```
Aspect as specialization

- Aspect can extend a class / interface / aspect
  - Only abstract aspects can be extended
- Aspect can be abstract
- Aspect cannot be extended by class
- Inheritance hierarchy
Nested aspects

• Nested in class/interface/aspect
• Must be static
• No “virtuality”
• Cf. Caesar
Exceptions

• Not allowed to add to the throws clause
  – Only RuntimeException can be added
• Advice throwing a checked exception
  before() throws SomeException :
  execution(void Point.set*()) { ... }
  – If the original code can throw it...
• Soften an exception
  declare soft: Exception:
  execution(void Point.set*());
  – Wrap it in the (unchecked)
    org.aspectj.lang.SoftException
Infinite loops

aspect A {
    before(): call(* *(..)) {
        System.out.println("before");
    }
}

• Or aspects applied to each other...
• Possibly not causing stack overflow
No stack trace, just hangs

aspect A {
    before(): call(* *(..)) {
        System.out.println("before");
    }

    after(): call(* *(..)) {
        System.out.println("after");
    }
}
StackOverflowException with stack trace

aspect A {
    before(): call(* *(..)) {
        System.out.println("before");
    }

    after() returning: call(* *(..)) {
        System.out.println("after");
    }
}
Remove the loop

aspect A {
    before(): call(* *(..)) && !
        within(A) {
            System.out.println("before");
        }

    after(): call(* *(..)) && !
        within(A) {
            System.out.println("after");
        }
}
“Advice execution”

aspect A {
    before():
    call(* *(..)) &&
    !cflow(adviceexecution())
    {
        System.out.println("before");
    }
}

Examples in AspectJ docs

- JavaBean with bound properties
- Roles with inter-type declarations
- Abstract aspects/pointcuts: tracing
- Reusable aspects: tracing revisited
- Subject/observer (and other design patterns)
- Telecom simulation
- Spacewar game
- EJB with BMP (a,b)
Inter-type declarations: roles

CloneablePoint

equals(o:Object)
hashCode()

ComposePoint

x: double
y: double
theta: double
rho: double
rotate(angle:double)
offset(dx:double,dy:double)
...

examples/introduction
Annotations

- Modifier set of join points ↔ pointcut patterns
- Annotate aspects
- Instructing ajc
- Describing aspects
Annotate aspects

- Aspects
- Fields and methods
- Types nested in aspects
- ITD fields and methods
- Advice

- *Not pointcuts!*
Join point matching

@interface Marked {}

aspect A {
    before(): call( @Marked * *.*(..) ){
        System.out.println(thisJoinPoint);
    }
}

class C {
    @Marked static void m(){}
    static void n(){}
    public static void main( String[] args ){
        m(); n();
    }
}
Annotation patterns

@Immutable

!@Persistent

@Foo @Goo

@(Foo || Goo)

@(hu.elte.inf.kto.annotations..*)
Type patterns

(@Immutable *)

(!@Persistent *)

(@Foo @Goo Bar+)

(@(@Inherited *) *)
Signature patterns

@Transaction * (@Persistent *).*(..)

@Foo (@Goo *) (@Hoo *).*

* *.**( @Immutable *, ..)
Run-time type matching and context exposure

- Pointcut designators: @this, @target, @args, @within, @withincode, @annotation
Annotation values

pointcut txRequiredMethod( Tx txAnnotation ):
    execution( * *(..) ) && @this(txAnnotation) &&
    if( txAnnotation.policy() == TxPolicy.REQUIRED )
Declare annotations

declare @type : org.xyz.model..* : 
   @BusinessDomain

declare @method : public * BankAccount 
   +.*(..) : 
      : 
      @Secured(role="supervisor")

declare @constructor : public BankAccount 
   +.new(..) : 
      : 
      @Secured(role="supervisor")

declare @field : * DAO+.* : @Persisted
@AspectJ

@Aspect
public class Foo {
    @Pointcut("call(* *.*(..))")
    void anyCall() {}

    @Before("anyCall() && target(Foo)")
    public void callOnFoo() {
        System.out.println("Call on a Foo");
    }

    public static void main( String[] args ) {
        System.out.println(
            new Foo().toString()
        );
    }
}
More tricky examples

```java
pointcut anyCall(ArrayList list)
: call(* *(java.util.ArrayList)) && args(list);

@Pointcut
("call(* *(java.util.ArrayList)) && args(list)"
void someCall(ArrayList list) {

pointcut someCallWithIfTest(int i)
: call(* *(int)) && args(i) && if(i > 0);

@Pointcut("call(* *(int)) && args(i) && if()"
public static boolean someCallWithIfTest(int i)
{
    return i > 0;
}
```
public aspect MoodIndicator {
    public interface Moody {};
    private Mood Moody.mood = Mood.HAPPY;
    public Mood Moody.getMood() { return mood; }
    declare parents : people..* implements Moody;
    before(Moody m) : execution(* *.*(..)) && this(m) {
        System.out.println("I'm feeling " + m.getMood());
    }
}

@Aspect public class MoodIndicator {
    public interface Moody { Mood getMood(); }
    public static class MoodyImpl implements Moody {
        private Mood mood = Mood.HAPPY;
        public Mood getMood() { return mood; }
    }

    @DeclareParents(value="people..*",defaultImpl=MoodyImpl.class)
    private Moody implementedInterface;

    @Before("execution(* *.*(..)) && this(m)")
    void feelingMoody(Moody m) {
        System.out.println("I'm feeling " + m.getMood());
    }
}
package people;
public class Boy {
    public void test() {
        Mood mood = ((Moody) this).getMood();
    }
}

@Aspect public class MoodIndicator {
    public interface Moody {
        Mood getMood();
    }
    public static class MoodyImpl implements Moody {
        private Mood mood = Mood.HAPPY;
        public Mood getMood() { return mood; }
    }

    @DeclareParents(value = "people..*", defaultImpl = MoodyImpl.class)
    private Moody implementedInterface;

    @Before("execution(* *.*(..)) && this(m)")
    void feelingMoody(Moody m) {
        System.out.println("I'm feeling " + m.getMood());
    }
}
The AspectJ tools

- ajc compiler
- aj load-time weaver
- ajdoc documentation generator
- AspectJ browser
- Ant tasks
ajc

- Aspect weaver
- Compiles Java and AspectJ
  - For aspects: .java or .aj
- Produces efficient code
- Incremental compilation
- Accepts bytecode
  - E.g. in jars
  - Both for java and aspect code
  - The first version worked on source code
- Compiler API
Suppress warnings

• Warnings like “adviceDidNotMatch”
• Lint messages configurable from error to void
  – Default: warning
• @SuppressAjWarnings
  – @SuppressAjWarnings
  – @SuppressAjWarnings("uncheckedArgument")
• Similar to @SuppressWarnings of Java
Aspect weaving

- **Static-time**
  - Source code (compile-time weaving)
  - Target code (post-compile or binary weaving)
  - (During linking)

- **Dynamic-time**
  - Load-time
  - Run-time
    - In AspectJ: idioms for dynamically enable/disable certain advice
Load-time weaving

- **Agents**
  - Java 5 with JVMTI
  - Java 1.3/1.4 with BEA JRocket
- **Scripts: aj and aj5**
  - aj runs a Java VM with a special system class-loader
- **Support for custom class loader**
- **Configuration: aop.xml**
aj

- Compile all Java sources with `javac`!
- Compile all AspectJ sources with `ajc`!
- Set `ASPECTPATH` to the `ajc` target code
- Run `aj` on the `javac` target code
ajdoc

• Similar to javadoc
• Generates HTML documentation
  – all pointcuts and advice
  – no inter-type declarations yet
• Links to affected methods
• Links to affecting advice
ajbrowser

- Editing
- Compiling
- Weaving
- Navigating
- Running
Ant tasks

• Develop build scripts
• Use Jakarta Ant 1.5.1
Integrating AspectJ

- Eclipse  (AJDT for 2.1, 3.0, 3.1, 3.2, 3.3)
- JBuilder   (AJDE for 4-7, 9)
- NetBeans  (for Forte 3+, NetBeans 3.3, 3.4, 3.5)
- JDeveloper (for 10.1.2 - 9.0.5.1)
- Emacs
- Vim???
AspectJ and Eclipse

- See [AJDT]
- Syntax highlight, AspectJ projects, Aspect Wizard
- Integrated ajc, ajdoc
- Navigation
  - Outline view
  - Visualizer perspective
Why AOP/AspectJ?

- Simple
- Small steps
- Matured concepts and tools
- Popular, well supported
  - Literature available
  - Foundations
Criticism of AspectJ

- More general approach?
- Why distinguish base code and aspects?
- More join points? (e.g. each instruction)
- More pointcuts?
- More support for reuse/composability? (aspectual polymorphism)
- More support on precedence?
- Advice based on history of prg. execution
- Simple programming model?
- Dynamic weaving and unwrapping?
AOP language design

- Symmetry
- Joint point model
- Composition mechanism
- Quantification
- Encapsulation
- Type safety
- Obliviousness
- Domain specificity
- Reuse and parametrization
- Conflict resolution
- Legacy relationships
- Run-time aspect dynamics
- Analyzability
- Debugability
- Testability
- Software process
- Implementation
- Run-time environment
History of AspectJ

- XEROX Palo Alto Research Center
- Gregor Kiczales
- 1997

- Eclipse project: 2002
Other AOP solutions (1)

- Hyperspace [OT’99, Hyper] (Hyper/J [Hyper/J])
- Subject-Oriented Programming [SOP]
- Composition filters [CF]
  (Sina, Sina/st, CFIST, C++/CF, Compose*.NET, ComposeJ, ConcernJ, JCFF)
- Adaptive Programming [AP] (DJ)
- Collaboration Interfaces [MO’05] (Caesar)
Other AOP solutions (2) [FECA’05]

- Mixin layers (Java Layers)
- First-class namespaces (JPiccola)
- Reflection (MetaclassTalk)
- Program transformation (JMangler, JOIE)
- Object Infrastructure Framework (Pragma)
- Java Aspect Components
Composition Filters [CF]

- University of Twente, NL
- Filters modify messages sent and received
- Tied to OOP
- A set of filters may implement an aspect
  - concern, filtermodule, superimposition
- Also for delegation and dynamic inheritance
- Java, C++, .NET, Sina, Smalltalk
Adaptive Programming  [AP]

- Karl Lieberherr
- Northeastern University, Boston, U.S.A.
- Concern-shy programming
- Insensitive to small changes in interface
- Traversals over the object graph
- DJ, DAJ
Subject-Oriented Programming
[SOP]

- IBM Research
- Based on OOP
- Subject: solve a subproblem
  - collection of classes and class fragments
- Compare to Point/Clonable/Comparable/...
- Roles, collaboration-based design
- Impl.: Extension to IBM Visual Age for C++
Multi-Dimensional Separation of Concerns  [OT’99]

• IBM Research
• Symmetrical approach
• All artifacts of software development
• “Having divided to conquer, we must reunite to rule”
• Hyperspaces, Hyper/J  [Hyper, Hyper/J]
Collaboration Interfaces  [MO’05]

- Caesar
- Hierarchy in aspect structure
- More reusable
- Collaboration-based decomposition
- Aspect abstractions
- Aspectual polymorphism
- Virtual types
- Family polymorphism
And many more...
References

• Aspect-Oriented Software Development
• AspectJ
• Related technologies
• Own stuff
References
(Aspect-Oriented Software Development)


References
(Aspect-Oriented Software Development)


References
(AspectJ)


[Col’05] Adrian Colyer: “AspectJ”. In [FECA’05]


[Lopes’05] Cristina Videira Lopes: “AOP: A Historical Perspective (What’s in a Name?)”. In [FECA’05]

References
(Related technologies - 1)


[SOP] *Subject-oriented programming.* http://www.research.ibm.com/sop/

[CF] *Composition Filters.* http://trese.cs.utwente.nl/composition_filters/

References
(Related technologies - 2)

<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors/Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>[MO’05]</td>
<td>Mira Mezini, Klaus Ostermann: <em>Untangling Crosscutting Models with Caesar</em>. In [FECA’05].</td>
</tr>
</tbody>
</table>
References
(Some own stuff)


Assignments

• Null arguments: reusable aspects (1 point)
• Rewrite the reusable R/W aspect in @AspectJ! (1 point)
• Reusable aspect for starvation-free R/W (2 points)
• Write a heatbugs simulation, and implement the display functionality in a separate aspect! (3 points)