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]

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

<table>
<thead>
<tr>
<th>ApplicationSession</th>
<th>StandardSession</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ServerSession</th>
<th>StandardSessionManager</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
problems like...

session tracking is not modularized

HTTPRequest
- get_cookies()
- get_request_URI()
- get_session()
- get_requested_session_id()

SessionInterceptor
- request_map(request)
- before_body(req, resp)

HTTPResponse
- get_request()
- set_content_type(content_type)
- get_output_stream()
- set_session_id(id)

Session
- get_attribute(name)
- set_attribute(name, val)
- invalidate()

Servlet
Crosscutting concern

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

- Physical separation: in an aspect
- Pluggable
crosscutting concerns

```
HTTPRequest
getCookies()
getRequestURI()
getSession()
getRequestedSessionId()
...

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

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

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

(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

```java
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
- Object Teams
- UMLAUT
Some examples for Java  [AOSD]

- Caesar
- Java Aspect Components
- JMangler, JOIE, JMuenger
- DJ
- ComposeJ, ConcernJ, JCFF
- Java Layers
- JPiccola
- Pragma
- AspectWerkz
- JBoss-AOP
- Lasagne/J
AspectJ

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

• Download from [AspectJ]
  – Stable: 1.2.1
  – Experimental: 1.5M

• Install
  – Requires JDK 1.3 or 1.4
  – 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

class Hello {
    void greeting()
        System.out.println("Hello!");
    }
    public static void main(String[] args) {
        new Hello().greeting();
    }
}
The first program: an aspect

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
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();
    }
}
```
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

```java
call( void Hello.greeting() )
call( * Hello.*(..) )
call( void Hello.greeting() ) && target(f)
```
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("> ");
    }
}
```
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
- Should be modularized
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

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
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) { ... }
}
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
Composing pointcuts

• Pointcut: pick up joint points
• Composition operators: \&\& | | !
• The result is a pointcut

\texttt{call(void Point.setX(int)) | | call(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

• example-02

```java
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
• Semantical match?
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");
}
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

```java
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

```java
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:

this target args

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)

```java
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
Context in “after returning”

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

• Capturing return value
• Type in returning matches type of method
Context in “after throwing”

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 figureElem.: ");
    try {
        if (System.in.read() == 'Y') proceed();
    } catch (java.io.IOException e) {} 
}
```
Inter-type declarations

- Modify the 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)
Labelling aspect

public aspect Labelling {
    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

```java
declare parents:
    Point implements Clonable;

public int Point.compareTo(Object p) {
    ...
}
declare parents:
    Point implements Comparable;
```
Changing the “extends” relation

public aspect Labelling {
    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

Labelling

Object

Labelled

Point  Line
Problem:

Object

PointBase

Point

LineBase

Line

Labelling

PointBase

Point

LineBase

Line

Labelled

Labelling

Object
Crosscutting labels

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

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

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

```java
public aspect Labelling {
    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. thisJoinPoint
- Bound to an object describing current join point

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

- Expressed separately
- Easy to change

```java
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

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

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 setX(int x): (call(void Point.setX(int)) && args(x))
    || (call(void Point.setXY(int,int)) && args(x,*));
    before(int x): setX(x) {
        if ( x < MIN_X || x > MAX_X )
            throw new IllegalArgumentException( "x is out of bounds.");
    }
    // same for y
}
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

- **withincodem** pointcut

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

- **declare error**, based on static information

```java
static aspect RegistrationProtection {
    pointcut register():
        call(void Registry.register(Figure));
    pointcut canRegister():
        withincode(* Figure.make*(..));
    declare error:
        register() && !canRegister():
            "Illegal call";
}
```
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
Production aspects

- Used in production builds
- Add real functionality to apps
- E.g. the Labeling aspect

- Let’s see some examples
  - Change monitoring
  - Context passing
  - Providing consistent behaviour
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

- Let’s set the color of FigureElements upon creation
- Pass a color (or a color factory) to \texttt{make*}
- This may influence many methods: on the control flow from client to \texttt{make*}
- Non-AOP solution: additional arg to those methods
Passing context with aspect

aspect ColorControl {

  after (ColorControllingClient c)
  returning (FigureElement fe):
    call(* Figure.make*(..)) &&
    cflow( call(* * (..)) && target(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)

```java
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

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);
}
Reusable R/W (1)

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.
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...
A deeper look at the join points

- Describing events
  - Calling a method: `call`
  - Setting a field: `set`

- Describing context
  - Static context: `withincode`
  - Dynamic context: `cflow`, `cflowbelow`
  - Exposing context: `this`, `target`, `args`

- Composing pointcuts (`&&`, `||`, `!`)
Picking up further join points: describing events

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

\texttt{set}(FieldPattern)
\texttt{get}(FieldPattern)

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

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

• Classes: when loading a class

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

- When catching an exception
- Only before advice

`handler(TypePattern)`

`handler(java.sql.SQLException)`
`handler(java.io.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( \textit{MethodPattern} )
call( \textit{ConstructorPattern} )
\texttt{execution( MethodPattern )}
\texttt{execution( ConstructorPattern )}

call( \ast \texttt{Point.setX(\ldots)} )
call( \texttt{Point.new(\ldots)} )
Example

- example-huge
- 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 ... { ... }
```
Construction of objects

• Constructor call
  – not in case of \texttt{this} or \texttt{super} calls
  – returns an object of the class
• Constructor execution
  – object accessible by pointcut \texttt{this}
• Object pre-initialization
• Object initialization
  – object accessible by pointcut \texttt{this}
The process of the construction

- call constructor
- call this(..)  \(\text{(optional: 0, 1 or more)}\)
- call super(..)
- ...
- return from super(..)
- execute initializers  \(\text{(hm. almost...)}\)
- execute rest of this(..) and return  \(\text{(optional: 0, 1 or more)}\)
- execute rest of constructor
- return from constructor
“Constructor call” join point

• 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

- 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

- 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

- 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

- call constructor
- call this(..)
- 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-polymorphic”

```java
class B { void m(){} }
class C extends B { void m(){} }
aspect A {
    after() : call(void B.m()) { ... }
}
(new C()).m()
```
Picking up further join points: describing context

• **Static context**
  - `withinCode` (*MethodPattern* or *ConstructorPattern*)
  - `within` (*TypePattern*)

• **Dynamic context:**
  - `cflow` (*Pointcut*)
  - `cflowBelow` (*Pointcut*)
  - `if` (*boolean-exp*)

• **Exposing context:** `this`, `target`, `args`
Static context

- `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*() )
Dynamic context

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

- Based on run-time value:
  ```java
  if (boolean-expr)
  ```
The `if` pointcut

abstract class Figure {
    boolean isWhite(){ ... }
    ...
}

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

cflow( P && Q )
cflow( P || Q )
cflow( ! P )

cflow(P) && cflow(Q)
cflow(P) || cflow(Q)
! cflow(P)
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**
Examples

\begin{align*}
\text{args}(\text{int}, \text{String}) \\
\text{args}(\text{int}, \text{Object}) \\
\text{pointcut setter}(\text{int v}): \\
\quad \text{call}(\text{void } \ldots \text{set}*(\ldots)) \land \text{args}(v) \\
\text{args}(\text{int,}*) \\
\text{args}(\text{int,} \ldots \text{String})
\end{align*}
Patterns

\textbf{execution}(MethodPattern)
\textbf{execution}(ConstructorPattern)
\textbf{set}(FieldPattern)
\textbf{within}(TypePattern)

• Pattern matching and wildcards
Field patterns

• “Signature” of fields
  – Name
  – Type
• Wildcards
  \texttt{Point.x} \quad \texttt{Point.*} \quad \texttt{Point.*Id}
• Setting a field: \texttt{args}
Type patterns

• Exact type names
  – Qualified names
  – Subtyping: \texttt{call} vs. \texttt{within}

• Composing type patterns:
  \texttt{FigureElement} \&\& !\texttt{(Point || Line)}

• Including subtypes: \texttt{Point+}

• Array types: \texttt{Point[]}

• Wildcards: \* and ..
Wildcards in type patterns

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

```
java.util.*    java.lang.*Error
java.*.List    java..List
java..*        java..Map.*
```
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);
Method and constructor patterns

execution \((\text{MethodPattern})\)

execution \((\text{ConstructorPattern})\)

• Based on method/constructor “signature”
  – modifiers
  – argument (and return) types
  – declared exceptions
  – “qualifying type”
  – (name)
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) )
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 run()) && target(Service)
Negation in throws clause

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

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

void m() throws SQLException, IOException
Matching constructors

call( Foo.new() )

call( Foo+.new() )

call( !public *Handler+.new(int,..)
     throws java.io.IOException  )
Reflective information about join points

- `thisJoinPoint`, `thisJoinPointStaticPart`, `thisEnclosingJoinPointStaticPart`
- Useful for tracing
- Useful for metaprogramming
- Information about
  - signatures, names, types
  - program locations
Reflective programming

private final HashMap constraints = new HashMap(); // java.util
{
    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 + " is out of bounds." );
}
Aspect instances

- Aspects cannot be instantiated with `new`
- By default: singleton
  ```
  aspect Logging { ... }
  aspect Logging issingleton() { ... }
  ```
- Per-object and Per-control-flow aspects
  ```
  aspect A perthis(Pointcut) { ... }
  aspect B pertarget(Pointcut) { ... }
  aspect C percflow(Pointcut) { ... }
  aspect D percflowbelow(Pointcut) { ... }
  ```
Precedence

```java
declare precedence : TypePatternList ;
```

```java
declare precedence: *..*Security*,
                   Logging+,
                   *;
```

- Only for concrete aspects
- Order of advice in aspect
- Subaspects have higher precedence
Aspect as specialization

• Aspect can extend a class / interface / aspect
• 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. Ceasar
Exceptions

• Not allowed to add to the throws clause
  – Only `RuntimeException` can be added

• Advice throwing a checked exception
  ```
  before() throws Exception :
  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 hungs

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

- 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
<aspect>
clone()

HashablePoint
<aspect>
equals(o: Object)
hashCode()

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

ComparablePoint
<aspect>
compareTo(o: Object)
The AspectJ tools

- ajc compiler
- 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
- Compiler API
ajdoc

- Similar to javadoc
- Generates HTML documentation
  - all pointcuts and advice
  - no inter-type declarations yet
- Links to affected methods
- Links to affecting advice
• Editing,
• Compiling
• Weaving
• Navigating
• Running
Ant tasks

• Develop build scripts
• Use Jakarta Ant 1.5.1
• Too complicated for me...
Integrating AspectJ

- Eclipse (AJDT for 2.1, 3.0, 3.1)
- 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 unweaving?
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)


References
(Related technologies - 2)

[MO’05] Mira Mezini, Klaus Ostermann: Untangling Crosscutting Models with Caesar. In [FECA’05].


References
(Some own stuff)


