Introduction to Aspect-Oriented Programming

Martin Giese

Chalmers University of Technology
Göteborg, Sweden
Motivation
Motivation

XML parsing in Apache Tomcat:

All in one class.
Motivation (cont.)

URL pattern matching in Apache Tomcat:

➡️ All in two classes.
Motivation (cont.)

Logging in Apache Tomcat:

Something in almost every class.
Motivation (cont.)

Session Expiration in Apache Tomcat:
Motivation (cont.)

What we would like:
This Course

In this week, I intend to:

• Explain the underlying problem

• Present the idea of Aspect-Oriented Programming (AOP)

• Give an Overview of some existing approaches

• Present AspectJ in detail

• Show some practical applications

• Get you to do programming exercises

• Explore possible benefits and dangers
Prerequisites

• You *need* to know about Java.

• You *should* know about OO design.

• If you have been involved in the development of a large system, so much the better!

Questions...
Who is this guy?

- Background in logic, formal methods, program verification
- Involved in the development of large software
- *Not* actually used AOP in a real system. (Yet?)
Books

General Ideas of AOP, overview of approaches: Chapter 8 of

Czarnecki, Eisenecker. *Generative Programming*. Addison-Wesley, 2000

AspectJ in particular:


Also AspectJ, slightly out-dated:

Web Sites

Aspect-Oriented Analysis and Design Portal:

http://www.aosd.net/

Current AspectJ development:

http://eclipse.org/aspectj/

Origin of AspectJ at Xerox PARC:

Introduction
Concerns

- XML parsing
- UML pattern matching
- Logging
- Session management

are concerns of a software system.

Separation of Concerns

is a time-honored principle of Software design.
Separation of Concerns


Design principles for decomposition:

- Information-hiding modules
- Identify design decisions that are likely to change.
- Isolate these in separate modules (separation of concerns)
Cross-Cutting Concerns

In the motivation,

- XML parsing and URL pattern matching fit the class hierarchy,
- Logging and Session Management do not.

A **cross-cutting concern** is one that needs to be addressed in more than one of the modules in the hierarchical structure of the software.

Cross-cutting concerns are also called **aspects**.

What is an aspect depends on the chosen decomposition!
Problems with Cross-Cutting Concerns

Cross-cutting concerns pose problems for standard, e.g. OO programming techniques:

- hard and error-prone to introduce in an existing system
- hard to change afterwards
- hard to understand/explain to newcomers

Cross-cutting implementation of cross-cutting concerns does *not* provide *separation of concerns*. 
Solutions

Possible treatment of cross-cutting concerns:

➤ Refactor them away.

Change the module hierarchy so that the aspect becomes modular, often through application of adequate design patterns.

But:

• often performance penalty through indirection

• often leaves some cross-cutting boiler-plate

• can’t hope to capture all aspects
Aspect-Oriented Programming

A programming methodology is called *Aspect-Oriented* if it provides possibilities to cleanly separate concerns that would otherwise be cross-cutting.

There are various Aspect-Oriented methods. They differ in the kinds of aspects they can address and in the ways aspects and their relation to the chosen hierarchical decomposition are expressed.
Example: Demeter

Law of Demeter:

An object should only call methods on this, instance variables, method arguments.

no this.getWife().getMother().getMaidenName() chains.

Prevents dependency on too many other classes.

Forward information and control through a maze of little methods.

Still a lot to change when the class structure changes.
Solution in ‘Demeter’ (Karl Lieberherr):

Specify *traversal strategies* which say in which classes we are interested in, not how to navigate to get there.

► *structure-shy behaviour specification*

Demeter handles the aspect of navigating through the class structure.
Join Points

Analyse commonly occurring aspects.

Cross-cutting implementations can often be formulated in terms like

- Before ... is called, always check for ...
- If any of ... throws an exception, ...
- Everytime ... gets changed, notify ...
- ...
Join Points (cont.)

Implementations of aspects are attached to certain points in the implementation:

- method calls
- constructor calls
- field access (read/write)
- exceptions

These correspond to point in the dynamic execution of the program.

Such points are called *join points*
Code Example: Figure Editor

Class diagram:

- **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)

- Display

AOP Course 2003 – p.24/36
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) { p1.moveBy(dx,dy); p2.moveBy(dx,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) { x += dx; y += dy; }
}
Some Join Points

![Diagram showing join points and moveBy actions.]

- point 1
- point 2
Pointcuts

A pointcut designates a set of join points for any program execution.

At execution, any join point may or may not be selected by a pointcut.

Examples:

- all calls to public methods of the Point class
- every execution of a constructor with one int argument
- every write access to a public field

Membership of a join point can be determined at runtime.
Advice is code that should be inserted before, after or even instead of existing code at some set of join points.

‘Mainstream’ AOP:

- Designate sets of join points using some specification language for pointcuts.
- Declare advice to be executed before/after/instead of the calls/method executions/field accesses etc. selected by the pointcut.
Example

Display updating in the Figure Editor:

“After every change in a FigureElement’s state, update the display“.

AOP implementation:

➤ pointcut to select every state-changing method in FigureElement classes.

➤ ‘after’-advice which calls display update code.
Implementation of AOP

Original idea:

A program called *aspect weaver* is used to *weave* the advice code into the main program code.

Often, join point membership in a pointcut can be decided statically.

► no need to insert code at every possible join point.

Modern systems:

- some do weaving and compilation in one step
- some can do weaving at runtime (e.g. on Java byte-code)
Static Cross-Cutting

Pointcuts and advice modify the dynamic behaviour of software.

Possible modifications of static structure:

• Every sub-class of FigureElement should implement the Drawable interface.

• Every FigureElement should have a ‘modified’ flag.

Note that this might be necessary to specify dynamic behaviour.

‘Introduction’ of members in existing classes often possible in AOP systems.
AspectJ

- Extensions of the Java language for
  - pointcuts
  - attaching advice
  - static cross-cutting
- Originally developed at Xerox PARC
- First versions in Spring of 2000
- Hosted by eclipse.org since December 2002
- We’ll hear a lot about it later this week.
Spin-Offs

• AspectC++ for C++

• AspectC# for C#

• AspectC for C

See www.aosd.net for more
AspectWerkz

• Also for Java, but without language extensions.

• Specify advice and pointcuts through Java API.

• Specify some weaving details and static cross-cutting in external XML files.

• Allows aspect weaving at runtime by changing the ClassLoader.

Won’t talk about it here, because a specialized language makes the concepts clearer.
Other library-based approaches

- Nanning Aspects, also for Java
- AspectS for Smalltalk
- Aspect.pm for Perl

Note the predominance of interpreted, ‘dynamic’ languages

Again, there are more examples on www.aosd.net
Conclusion

You should now know about . . .

• the motivation for AOP
• what a concern is
• what an aspect is
• join points, pointcuts, and advice
• static vs. dynamic cross-cutting
• the existence of various different AOP systems

Tomorrow you will learn how to write aspects in AspectJ