

# Aspect-Oriented Programming - The propaedeutics

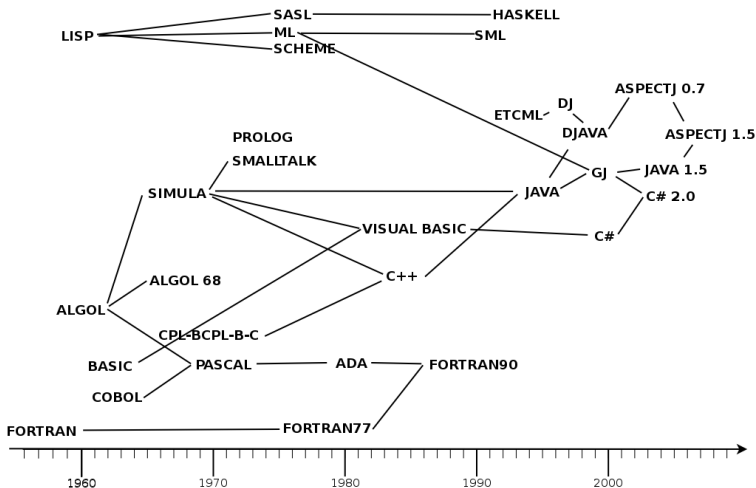
Łukasz Szala  
lukasz.szala@e-informatyka.pl

March 21, 2006

# Agenda

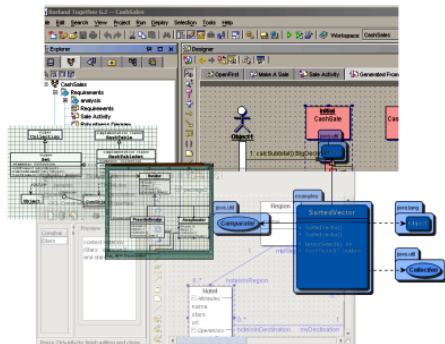
- 1 AOP
  - The history
  - Object-Oriented Paradigm
  - Crosscutting concerns
  - Aspect-Oriented Paradigm
  - Theme Approach
- 2 AspectJ
  - Syntax
  - AJDT
  - Examples
- 3 Project2006
- 4 Summary

# The evolution of programming languages.



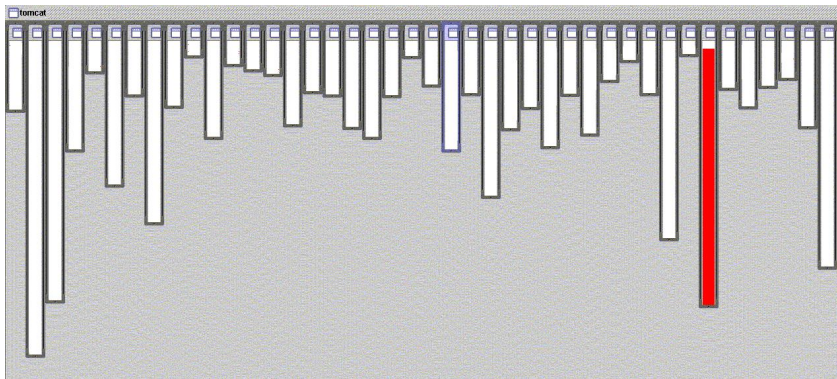
# Is OOP sufficient?

- Basic concepts:
  - Abstraction
  - Encapsulation
  - Polymorphism
  - Inheritance
- Well-designed IDE suites;
- Mature technology;
- Widespread documentation;
- Commonly used;



# Is OOP sufficient?

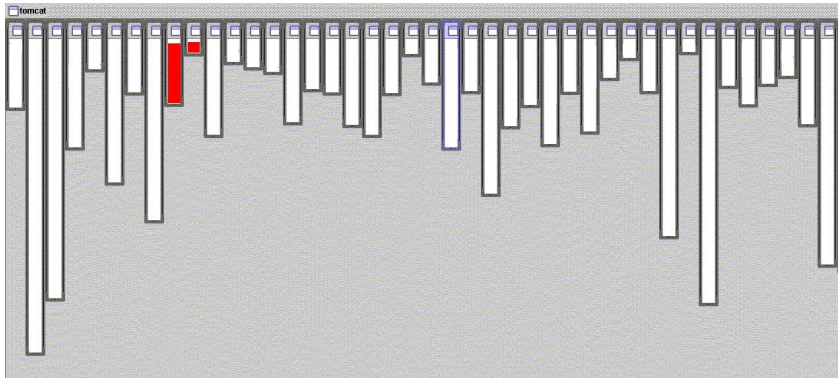
XML parsing in org.apache.tomcat



from: aspectj.org

# Is OOP sufficient?

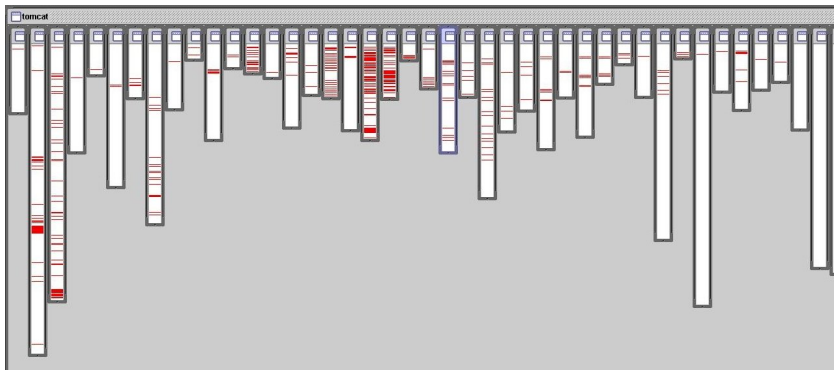
Matching URL paths in org.apache.tomcat



from: [aspectj.org](http://aspectj.org)

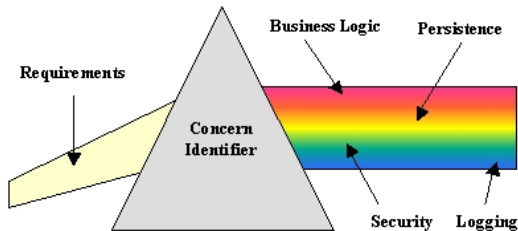
# Is OOP sufficient?

Logging in org.apache.tomcat



from: aspectj.org

# User's requirements are prone to be crosscutting

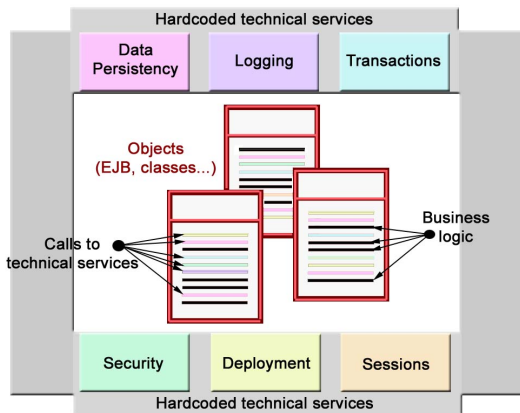


from: javaworld.com

# User's requirements are prone to be crosscutting

Common requirements:

- performance,
- persistence,
- logging,
- debugging,
- authentication,
- security,
- multi-threading,
- error checking ...



An application server container

from: JAC

# Drawbacks of OOP

## Problems

- **Code Tangling** is when two or more concerns are implemented in the same body of code or component, making it more difficult to understand.
- **Code Scattering** is when similar code is distributed throughout many program modules.

## Consequences

Changes to one implementation may cause unintended changes to other tangled and scattered concerns.

## Crosscutting concerns

### Siobhàn Clarke

*A concern triggered in multiple situations or whose structure and behaviour are **scattered** across the code base and **tangled** with code related to other concerns.*

### Gregor Kiczales

*A crosscutting concern is still crosscutting, even if it has been implemented in a modular way (no S&T) using AOP.*

# Aspect-Orientation

## Informal definition

The aim of Aspect Oriented Programming (AOP) is the production of code that is easier to understand and evolve, thanks to the separation of the crosscutting concerns from the principal decomposition

# The basics

## Gregor Kiczales

*AOP isn't explicit metaprogramming. Instead, it provides a direct semantics for designing and coding crosscutting concerns.*

## Robert Filman

*It is characterised by both:*

- **quantification:** *the ability to do something when a specific property occurs to be true;*
- **obliviousness:** *programmers are more or less unaware of what can be triggered by their code;*

# Aspects (1)

## An aspect (Wikipedia)

An aspect is a part of a program that cross-cuts its core concerns, therefore violating its separation of concerns.

## An aspect (AspectJ)

Aspects are how developers encapsulate concerns that cut across classes, the natural unit of modularity in Java.

## An aspect

A modular representation of crosscutting concerns indicating when the appropriate behaviour should be triggered.

## Aspects (2)

Aspects are responsible for:

- grouping behaviour regarding a crosscutting concern,
- describing rules of triggering this behaviour in an application,
- indicating circumstances of triggering and evoking this concerns,
- separating crosscutting concerns from a core (base) code.

# Join Points

## Join Point (AspectJ)

A **join point** is a well-defined point in the execution of a program. Not every execution point is a join point: only those points that can be used in a disciplined and principled manner are.

## Join Point (Wikipedia)

A **join point** is a point in the flow of a program. A join point is where the main program and the aspect meets.

## Join Point

All possible execution points in an application's control flow that can be furthermore reached from an aspect.

# Pointcuts

## Pointcut (AspectJ)

A **pointcut** is a program element that picks out join points and exposes data from the execution context of those join points.

## Pointcut (Wikipedia)

A **pointcut** is a set of join points.

## Pointcut (Siobhàn Clarke)

A predicate that can determine, for a given join point, whether it is matched by the predicate.

# Advices

## Advice (AspectJ)

**Advice** defines crosscutting behaviour. It is defined in terms of pointcuts. The code of a piece of advice runs at every join point picked out by its pointcut.

## Advice (Wikipedia)

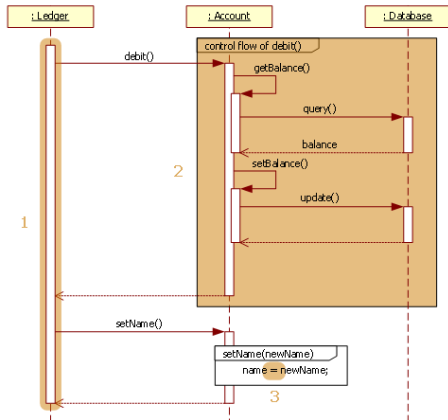
An **advice** is a piece of code executed when a join point is reached, that is associated with the pointcut of the advice.

## Advice (Siobhàn Clarke)

A triggered behaviour.

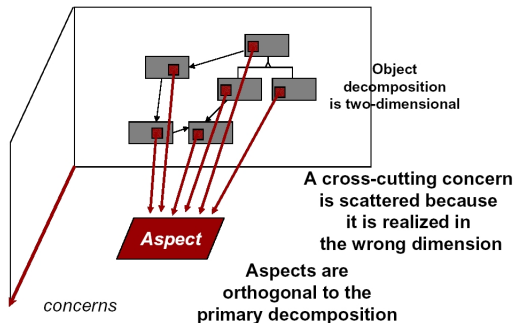
# Pointcuts, Join Points, Aspects, Advices

```
public aspect FooAspect{  
  
    //define a pointcut  
    pointcut fooPointcut() :  
        call(public * Ledger.*(..)) ;  
  
    //define an advice  
    before() : fooPointcut() {  
        System.out.println(" Hello" );  
    }  
}
```



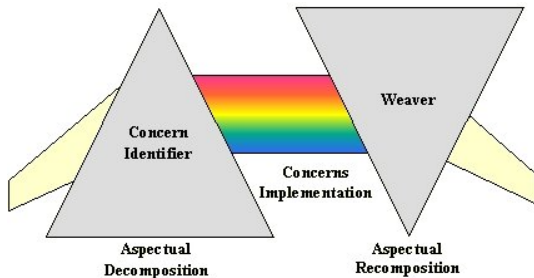
from: AOP@Work

## 3<sup>rd</sup> dimension



from: Frank Sauer (Technical Resource Connection Inc)

# What is a Weaver?



from: javaworld.com

## Weaving process (1)

```
// HelloWorld.java
public class HelloWorld {

    public static void say(String message){
        System.out.println(message);
    }

    public static void sayToPerson
    (String message, String name) {
        System.out.println(name+" , "+message);
    }
}
```

```
// MannersAspect.aj
public aspect MannersAspect {
    pointcut callSayMessage() :
        call(public static void
            HelloWorld.say*(..));

    before() : callSayMessage() {
        System.out.println("Good day!");
    }

    after() : callSayMessage() {
        System.out.println("Thank you!");
    }
}
```

## Weaving process (2)

```
public class HelloWorld {  
  
    public static void say(String message) {  
        System.out.println("Good day!");  
        System.out.println(message);  
        System.out.println("Thank you!");  
    }  
  
    public static void sayToPerson(String message, String name) {  
        System.out.println("Good day!");  
        System.out.println(name + ", " + message);  
        System.out.println("Thank you!");  
    }  
}
```

# The asymmetric philosophy

- Relies on the notion that there is a base body of code that is then augmented with aspects.
- There is a conceptual difference between the base and the aspects so that an aspect can not serve as the base of another composition.
- Most asymmetric approaches use language extensions to declare aspects as first class entities.
- Example: AspectJ

from Juri's Memmert wordpress page

# The symmetric philosophy

- Is based on the notion that all concerns in a system are created equal and, in the asymmetric terminology, can serve as both aspect and base in different compositions.
- Most symmetric approaches use programming languages as-is, although there are approaches that rely on language extensions as well.
- Example: Hyper/J

from Juri's Memmert wordpress page

# What Is a Theme?

- Themes are not Aspects! Themes are more general.
- A Theme is an encapsulation of a concern.
- Themes may be related to each other.
  - Concept Sharing - different themes have design elements that represent the same core concept in the domain.
  - Crosscutting - the behaviour in one theme is triggered by behaviour in other themes.
- Themes are extracted from requirements.

# Theme Approach

## Theme/Doc

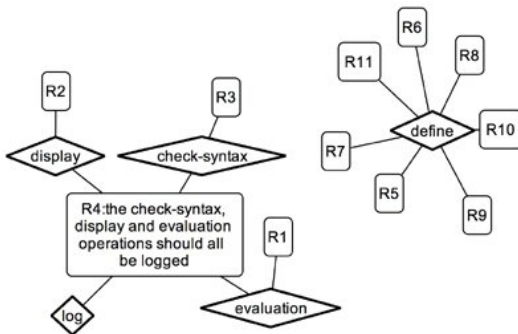
A set of heuristics for analysis of software requirements documentation.

## Theme/UML

A way to write themes (both aspects and base) as UML.

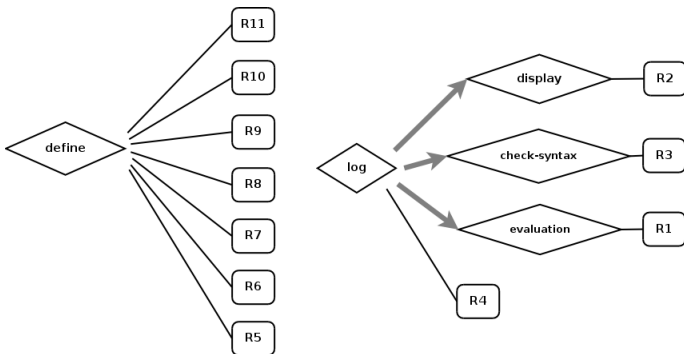
# Theme/Doc views

Theme-relationship view.



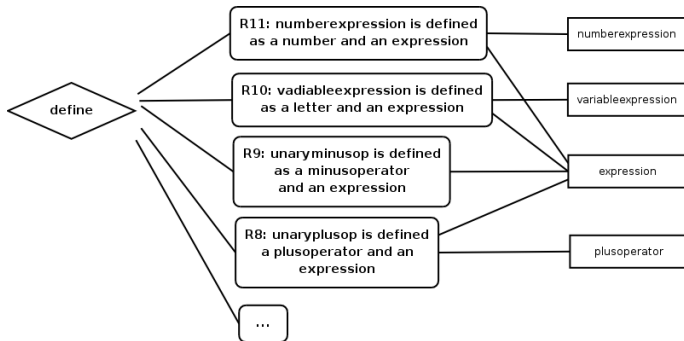
# Theme/Doc views

Crosscutting-relationship view.



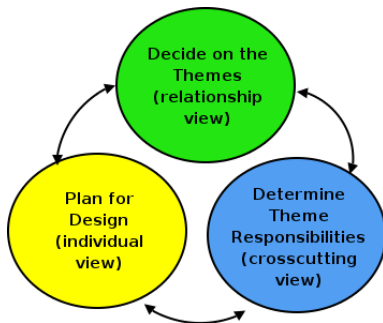
## Theme/Doc views

Individual Theme view.



## Theme/Doc views

General view of the analysis process.

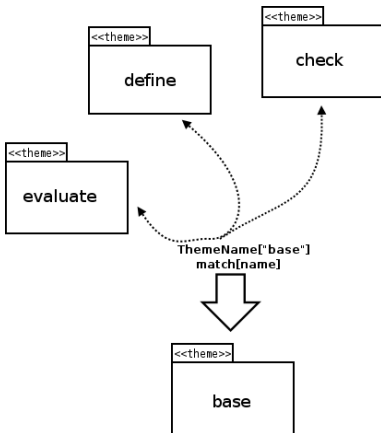


## Aspect identification

- Can the requirement be split up to isolate themes?
- Is one theme dominant in the requirement?
- Is behaviour of the dominant theme triggered by the other themes mentioned in the requirement?
- Is the dominant theme triggered in multiple situations?

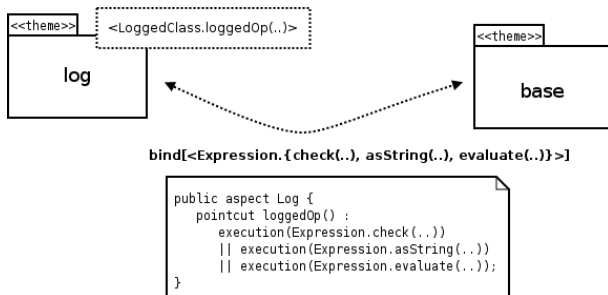
# Theme/UML view

Composition relationship: shared concepts.



# Theme/UML view

Composition relationship: crosscutting.



# AspectJ - Overview

- Is based on over ten years of research at Xerox Palo Alto Research Center.
- *AspectJ<sup>TM</sup>* is an extension to the *Java<sup>tm</sup>* programming language that adds to Java aspect-oriented programming capabilities.
- The AspectJ technologies include a compiler ([ajc](#)), a debugger ([ajdb](#)), a documentation generator ([ajdoc](#)), a program structure browser ([ajbrowser](#)) and the Ant task ([iajc](#)).

# Join Points

- Method call, Method execution, Constructor call, Constructor execution;
- Static initializer execution, Object pre-initialization, Object initialization;
- Field reference, Field set;
- Handler execution;
- Advice execution;

# Pointcuts (1)

- **call(MethodPattern)** method call join point whose signature matches *MethodPattern*;
- **execution(MethodPattern)** method execution join point whose signature matches *MethodPattern*;
- **get(FieldPattern)** field reference join point whose signature matches *FieldPattern*;
- **set(FieldPattern)** field set join point whose signature matches *FieldPattern*;

## Pointcuts (2)

- **call(ConstructorPattern)** constructor call join point whose signature matches *ConstructorPattern*;
- **execution(ConstructorPattern)** constructor execution join point whose signature matches *ConstructorPattern*;
- **initialization(ConstructorPattern)** object initialization join point whose signature matches *ConstructorPattern*;

## Pointcuts (3)

- **handler(**TypePattern**)** exception handler join point whose signature matches *TypePattern*;
- **adviceexecution()** all advice execution join points;
- **within(**TypePattern**)** join point where the executing code is defined in a type matched by *TypePattern*;
- **withincode(**MethodOrConstructorPattern**)** join point where the executing code is defined in a method whose signature matches *MethodOrConstructorPattern*;

## Pointcuts (4)

- **cflow(Pointcut)** join point in the control flow of any join point P picked out by Pointcut, including P itself;
- **cflowbelow(Pointcut)** join point in the control flow of any join point P picked out by Pointcut, but not P itself;
- **this(Type)** join point where the currently executing object is an instance of Type;
- **target(Type)** join point where the target object is an instance of Type;
- **args(Type)** join point where the arguments are instances of the appropriate Type;

## Pointcuts (5) examples

- `call(public void MyClass.myMethod(String))`  
Call to `myMethod()` in `MyClass` taking a `String` argument, returning `void`, and with public access;
- `execution(* MyClass.myMethod*(String,..))`  
Execution of any method with name starting in "myMethod" in `MyClass` and the first argument is of `String` type;
- `set(int MyClass.x)`  
Execution of write-access to field `x` of type `int` in `MyClass`;
- `cflow(call(* MyClass.myMethod(..))`  
All the joinpoints in control flow of call to any `myMethod()` in `MyClass` including call to the specified method itself;

# Advices (1)

Each piece of advice is of the form:

```
AdviceSpec [ throws TypeList ] : Pointcut { Body }
```

where AdviceSpec is one of

- before( Formals )
- after( Formals ) returning [ ( Formal ) ]
- after( Formals ) throwing [ ( Formal ) ]
- after( Formals )
- Type around( Formals )

## Advices (2) Example

```
public aspect CheckCreditAspect {  
    pointcut checkCredit(float f, BankAccount ba) :  
        call(* BankAccount.credit(float)) && args(f) && target(ba) ;  
  
    //advice  
    void around(float sum, BankAccount ba) : checkCredit(sum, ba) {  
        System.out.println("credit: "+sum+", account: "+ba);  
        if(ba.getSum() > sum)  
            proceed(f, ba);  
    }  
}
```

# Static Crosscutting (1)

AspectJ allows the declaration of members by aspects that are associated with other types.

- Inter-type member declarations
- Extension and Implementation
- Warnings and Errors
- Softened exceptions
- Advice Precedence

## Static Crosscutting (2) Example

```
interface I {}

aspect A {
    pointcut uglyFunction() : call(* void *.sayABadWord(..)) ;

    declare error: uglyFunction(): "Don't do this!!";
    declare parents: SomeClass implements Runnable;
    declare precedence: A, *;

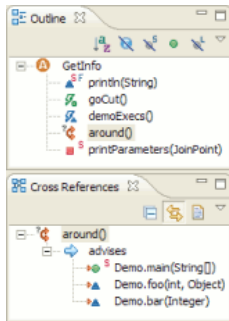
    public void SomeClass.run() {System.out.println("I'm a new thread!");}

    private void I.m() {System.err.println("I'm a private method on an interface");}

    void worksOn(I iface) {
        // calling a private method on an interface
        iface.m();
    }
}
```

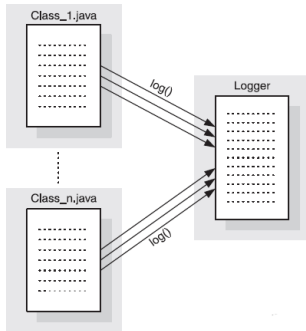
# AspectJ Development Tools

**AJDT** provides Eclipse IDE integration for **AspectJ**, and includes the **AJDE** (AspectJ Development Environment) libraries from the AspectJ project as part of its packaging. Development of the AspectJ compiler and AJDE takes place under the AspectJ project.

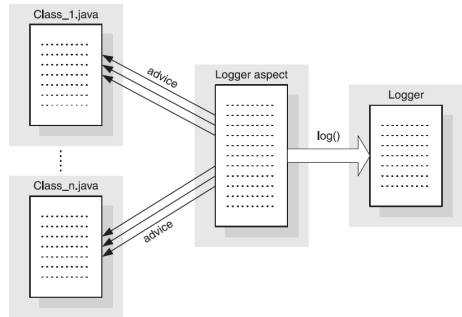


# Examples (1) Logging Aspect

## OOP solution

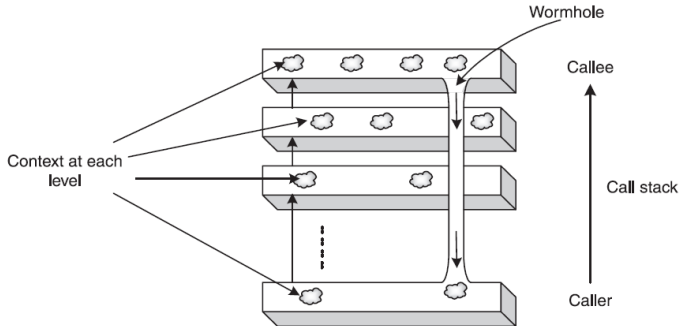


## AOP solution



from: AspectJ in Action

# Examples (2) Wormhole Pattern



from: AspectJ in Action

## Examples (2) Wormhole Pattern

```
public aspect WormholeAspect {
    pointcut callerSpace("<caller context>") : "<caller pointcut>";

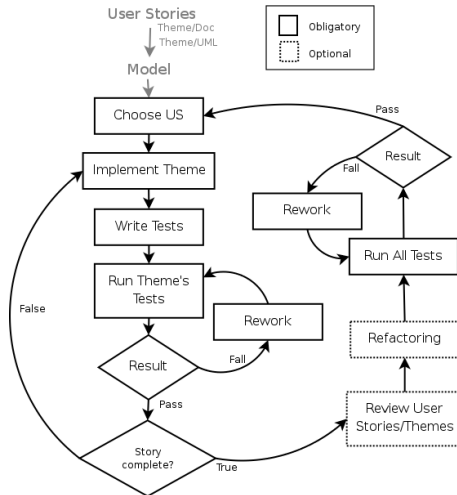
    pointcut calleeSpace("<callee context>") : "<callee pointcut>";

    pointcut wormhole("<caller context>", "<callee context>") :
        cflow(callerSpace("<caller context>"))
        && calleeSpace("<callee context>");

    // advices to wormhole
    around("<caller context>", "<callee context>") :
        wormhole("<caller context>", "<callee context>") {
        ... advice body
    }
}
```

from: AspectJ in Action

## AOP and TLD





# Exercises

- Exercise1 Open Classes
- Exercise2 Application to OO design patterns
- Exercise3 Advices and Pointcuts (1)
- Exercise4 Advices and Pointcuts (2)
- Exercise5 Around
- Exercise6 Worm Hole Pattern
- Exercise7 Aspect instantiation

# Advantages

- **Modularised implementation of crosscutting concerns:** modularised implementations even in the presence of crosscutting concerns.
- **Easier-to-evolve systems:** it's easy to add newer functionality by creating new aspects.
- **Late binding of design decisions:** an architect can delay making design decisions for future requirements
- **More code reuse:** each individual module is more loosely coupled

# Disadvantages

- **Learning curve:** the concept of an aspect is new and requires some additional up front work.
- **Obliviousness:** we can no longer reason about a class just by looking at the code for it.
- **Most Domain Models are Aspect Free:** are they?

from: slashdot.org discussion on AOP

*AOP can be used for great evil, but it can also be used for great good. When I first read up on it, my first reaction was "wow! This must be how God programs!" :)*

# Books (AOP)



Aspect-Oriented Analysis and Design (The Theme Approach) by Siobhàn Clarke



Aspect-Oriented Software Development by Robert E. Filman



Aspect-Oriented Software Development with Use Cases by Ivar Jacobson

# Books (AspectJ)



AspectJ in Action: Practical Aspect-Oriented Programming by Ramnivas Laddad



AspectJ Cookbook by Russell Miles



Eclipse AspectJ : Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools

# Articles



## I want my AOP!, Part 1-3

<http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect-p3.html>



## The AOP@Work series

[http://www-128.ibm.com/developerworks/views/java/libraryview.jsp?search\\_by=AOP@work:](http://www-128.ibm.com/developerworks/views/java/libraryview.jsp?search_by=AOP@work)



## AOSD Europe

<http://www.aosd-europe.net/>



## Aspect-Oriented Software Engineering Special Interest Group at Lancaster University

<http://www.comp.lancs.ac.uk/computing/aop/>



## Aspect Programmer

<http://aspectprogrammer.org/>

# Q&A

