Aspect-Oriented Programming

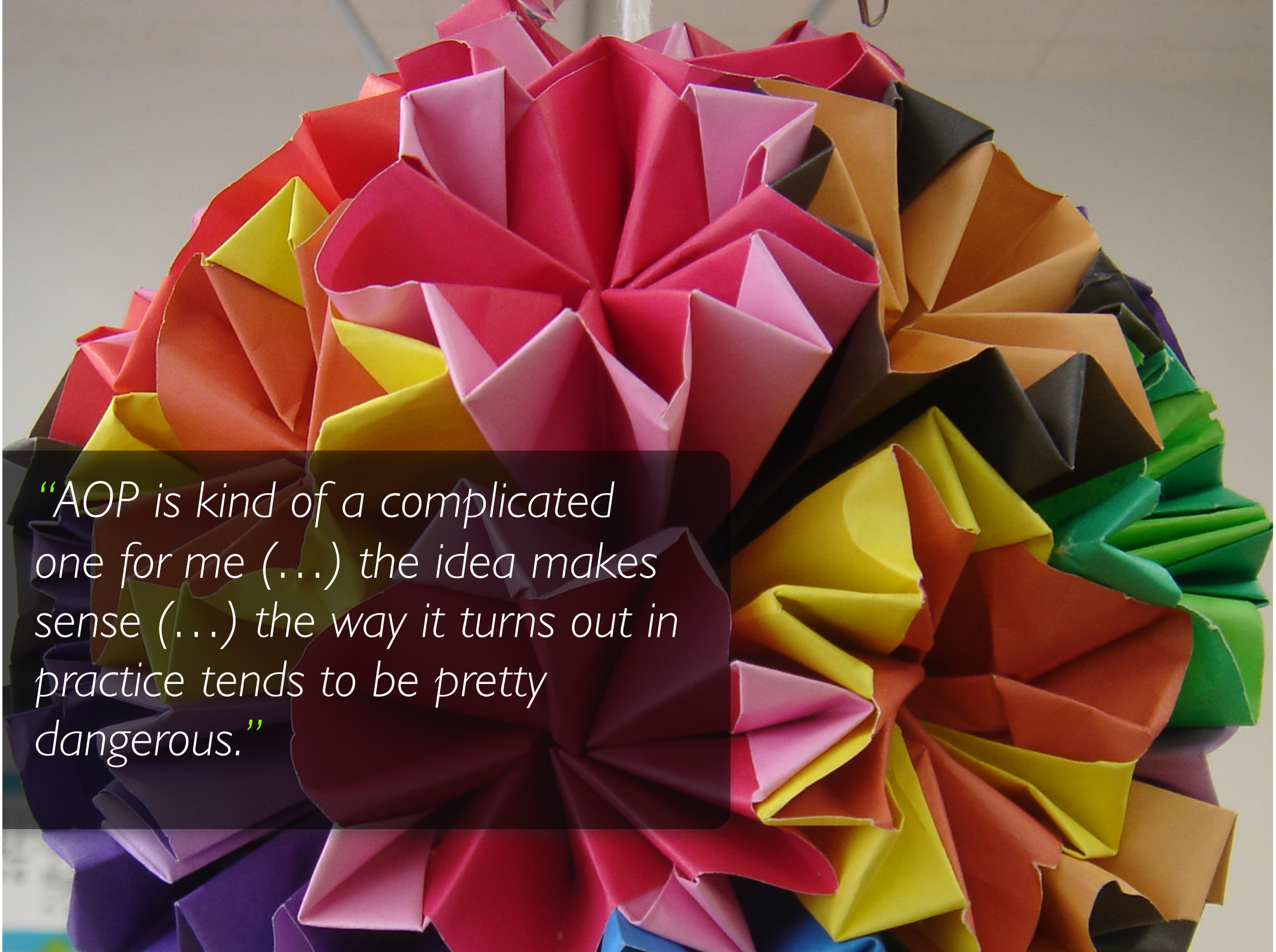
Based on the Example of AspectJ

Prof. Harald Gall

University of Zurich, Switzerland

Universität Zürich[UZH]

s.e.a.l.
software evolution & architecture lab



"AOP is kind of a complicated one for me (…) the idea makes sense (…) the way it turns out in practice tends to be pretty dangerous."

Background

Procedural programming
- Executing a set of commands in a given sequence
- **Fortran, C, Cobol**

Functional programming
- Evaluating a function defined in terms of other functions
- **Lisp, ML, Scheme**

Logic programming
- Proving a theorem by finding values for the free variables
- **Prolog**

Object-oriented programming (OOP)
- Organizing a set of objects, each with its own responsibilities
- **Smalltalk, Java, C++ (to some extent)**

Aspect-oriented programming (AOP)
- Executing code whenever a program shows certain behaviors
- **AspectJ (a Java extension)**

Does not *replace* OO programming, but rather *complements* it

AOP Basics - Functional Concerns

- Applications are a collection of concerns

  (more or less: features)

- Most concerns are functional, and OOP works well for them

  Billing, Ordering, generating some statistics, etc.

AOP Basics - Crosscutting Concerns

- Most applications also have cross-cutting concerns, which show up at many places at the same time

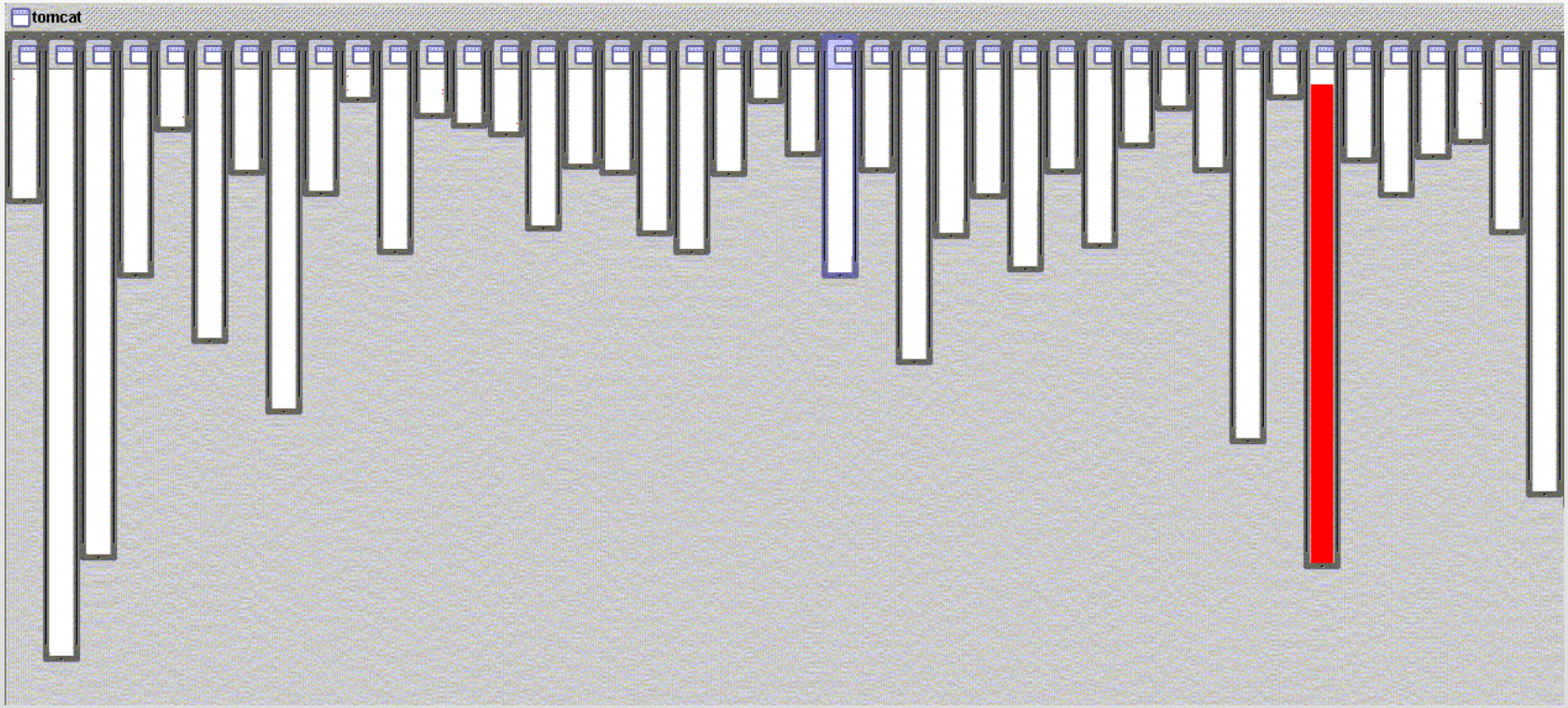
  Logging, access control, generating CEP events, …

- OOP does not work well at all for those and leads to problems of modularity

Modularity

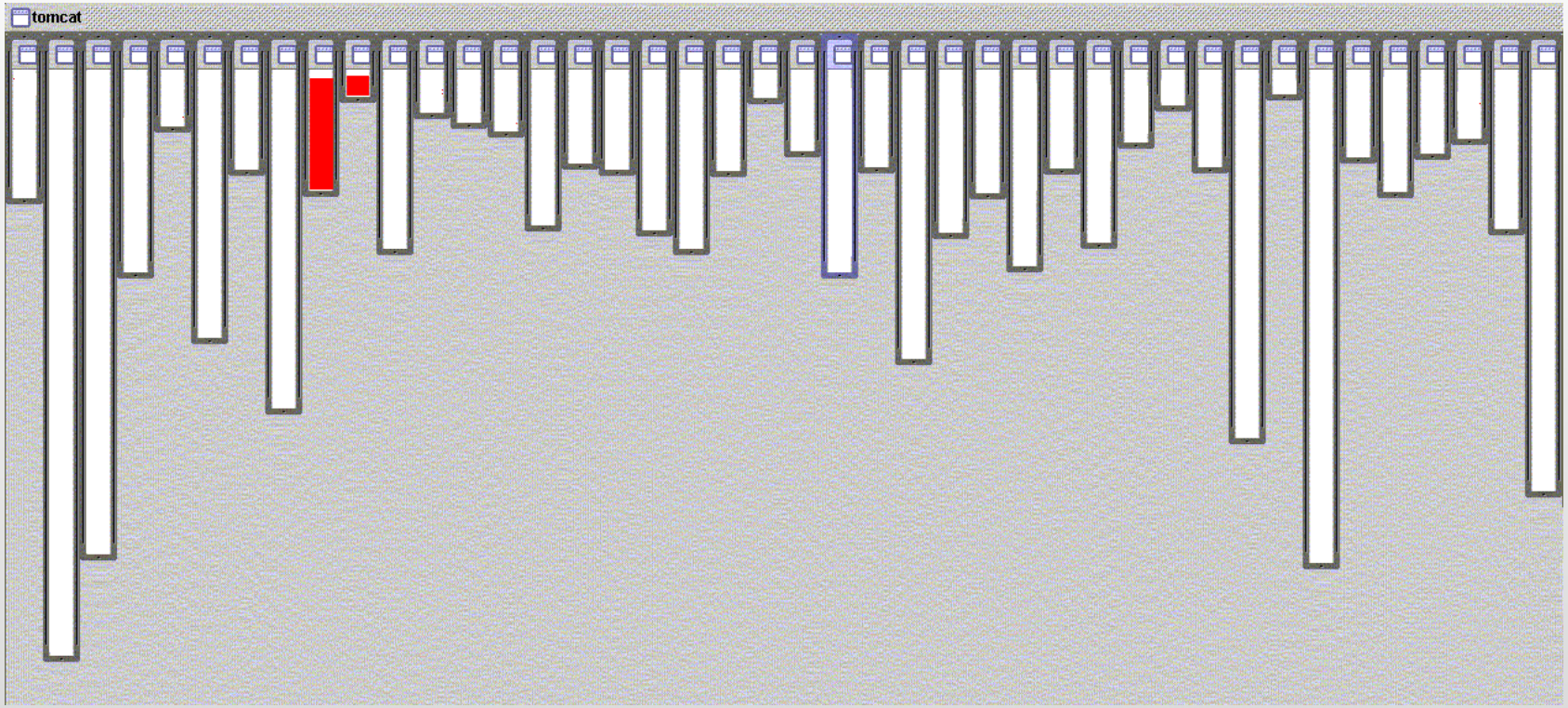
Good Modularity

XML parsing in *org.apache.tomcat*



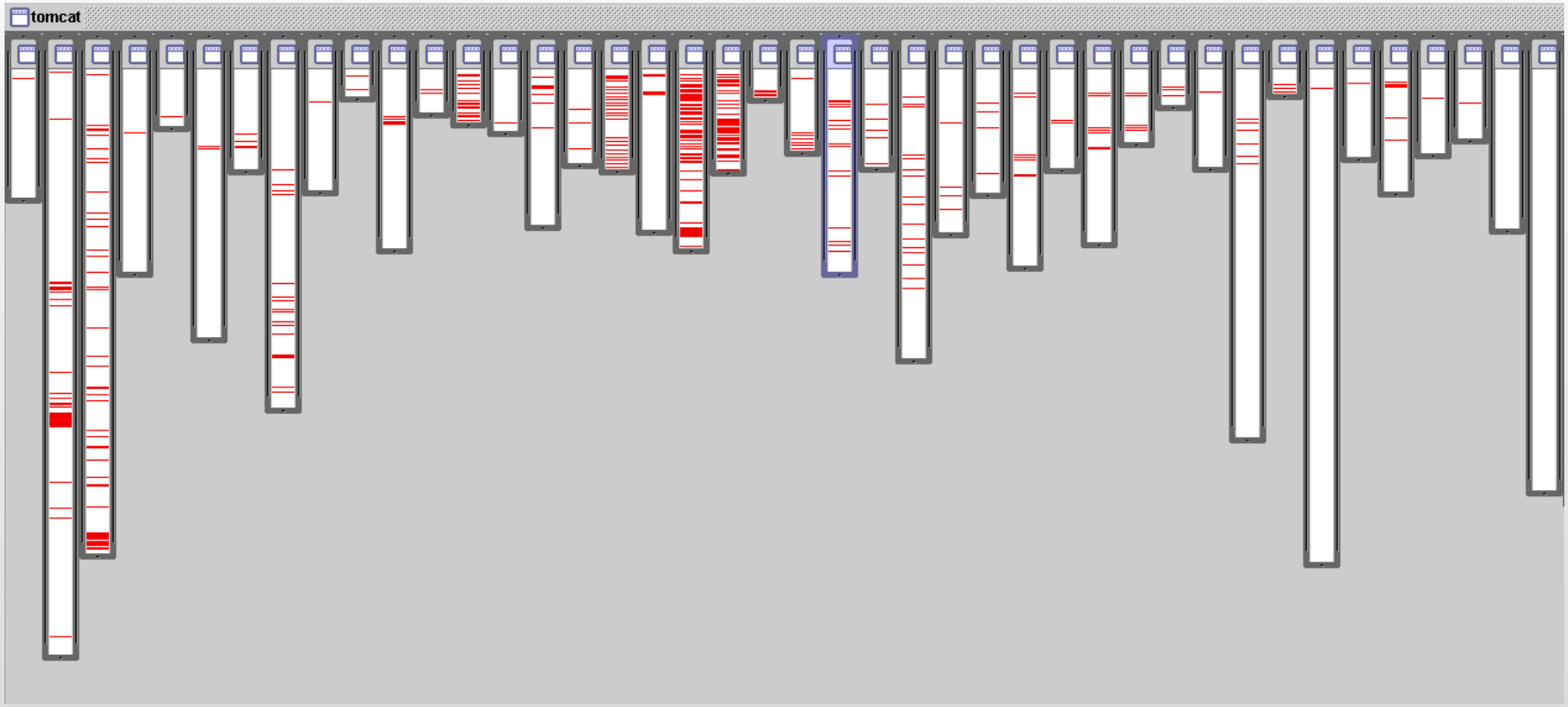
(pretty) Good Modularity

URL pattern matching in *org.apache.tomcat*



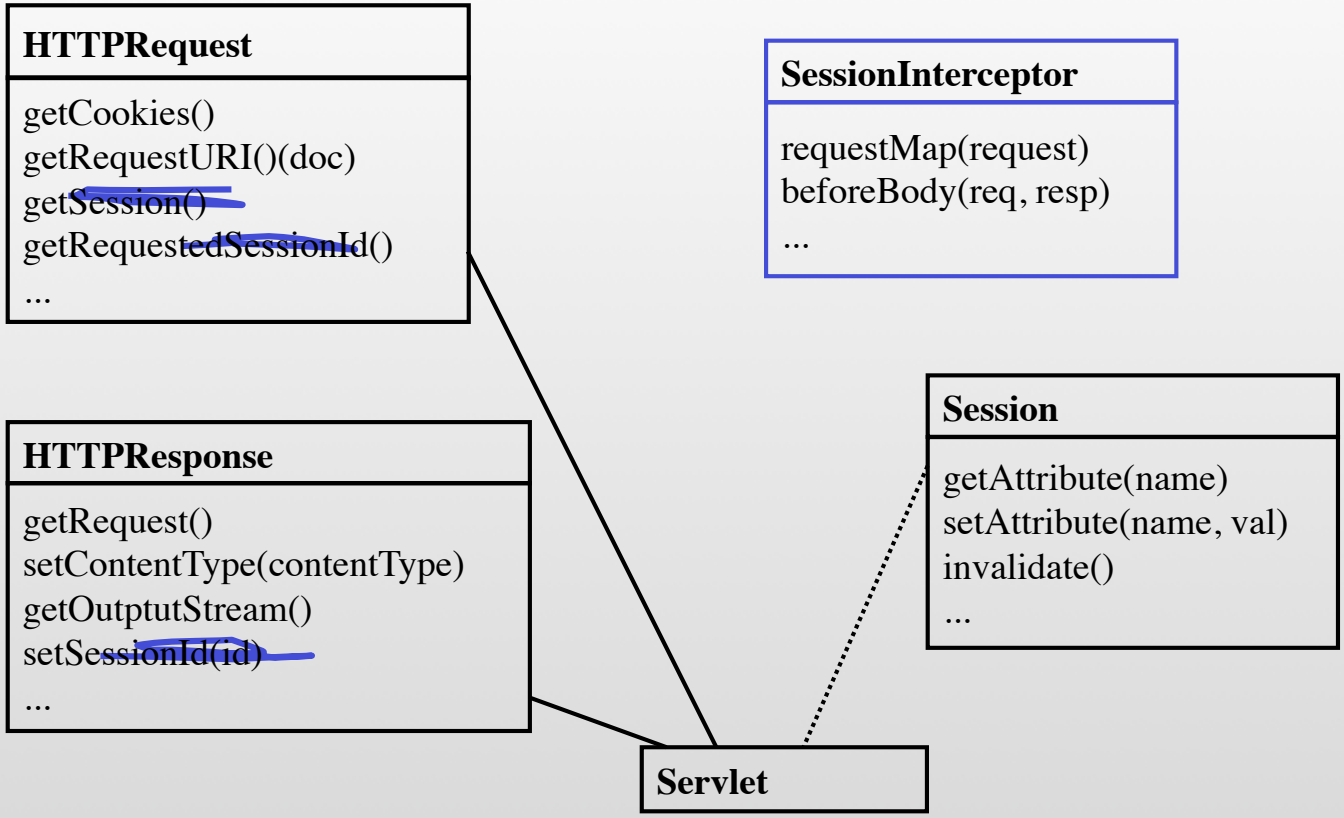
Bad Modularity

Logging in *org.apache.tomcat*



Bad Modularity

Session tracking in *org.apache.tomcat*



AOP Basics - Tangled Code (1)

- Critical aspects of large systems often do not fit into modules

Tangled Code

- Tangled Code has a cost
  - Difficult to understand
  - Difficult to change
  - Increases with system size
  - **—> Large maintenance costs**

AOP Basics - Tangled Code (2)

```
import com.foo.Bar;
// Import log4j classes.
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class MyApp {
        // Define a static logger variable so that it references the
        // Logger instance named "MyApp".
        static Logger logger = Logger.getLogger(MyApp.class);

        public static void main(String[] args) {
                // Set up a simple configuration that logs on the console.
                BasicConfigurator.configure();

                logger.setLevel(Level.DEBUG); // optional if log4j.properties not used
                // Possible levels: TRACE, DEBUG, INFO, WARN, ERROR, and FATAL

                logger.info("Entering application.");
                Bar bar = new Bar();
                bar.doIt();
                logger.info("Exiting application.");
        }
}
```

AOP Idea (1)

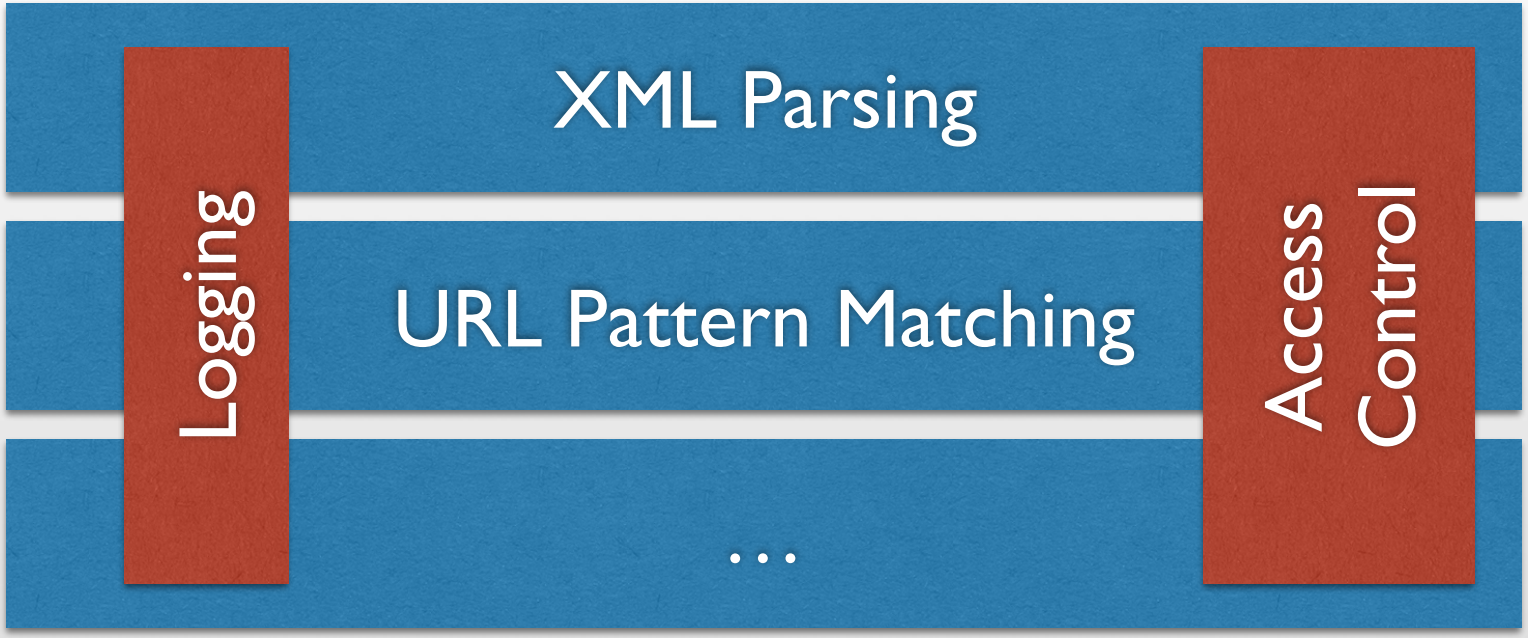
- **Crosscutting is inherent in real systems**

- Crosscutting concerns
  - … have a clear purpose
  - … have a natural structure

AOP Idea (2)

- Instead of trying to get rid of crosscutting concerns, let's
  - Use well-defined semantics to describe them
  - Capture them in a modular way
- Clearly, those modules are then orthogonal to the modules implementing functional constraints

Orthogonal Modules



AOP Idea (4)

Aspects are

Well-modularized crosscutting concerns

(along with a language to define them and
and tools to actually build them)

Aspect-Oriented Software Development

In AOSD, we …

- … build functional concerns as usual
- … but build all tangling code as **completely independent software modules**.

These crosscutting modules are then *weaved into the functional concerns as appropriate*

- Essentially: build the crosscutting concern independently, but tell the AOP compiler which parts of the functional modules it should be applied to.

AspectJ

AspectJ is a small, well-integrated extension to Java

Based on the 1997 PhD thesis by Christina Lopes, *A Language Framework for Distributed Programming*

AspectJ modularizes crosscutting concerns

That is, code for one *aspect* of the program (such as tracing) is collected together in one place

The AspectJ compiler is free and open source

Best online writeup:

http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

(Parts of this lecture were taken from the above link)

Logging Example (1)

```
public class LoggingAspect {

        public void logMethodFinished(JoinPoint jp) {

                String className = jp.getSignature().getDeclaringTypeName();
                Logger logger = Logger.getLogger(className);

                String methodName = jp.getSignature().getName();
                log.exiting(className, methodName);

        }
}
```

Logging Example (2)

```
@Aspect
public class LoggingAspect {

        @AfterReturning("execution(* (@my.package.Logged *).*(..))")
        public void logMethodFinished(JoinPoint jp) {

                String className = jp.getSignature().getDeclaringTypeName();
                Logger logger = Logger.getLogger(className);

                String methodName = jp.getSignature().getName();
                log.exiting(className, methodName);

        }
}
```

Basic Notions (1)

- Concerns form the functionality of the program
- A join point is a defined point in the program flow
- A pointcut is a group of join points + context
- An advice is code that is executed at a pointcut
- Introduction modifies the members of a class and the relationships between classes

Basic Notions (2)

- An aspect is a module for handling crosscutting concerns
  - Aspects are defined in terms of pointcuts, advice, and introduction
  - Aspects are reusable and inheritable

- Weaving is the process of modifying the functional concerns and adding the crosscutting concerns
  - Typically done as part of the build process

Join Points

- A join point is a well-defined point in the program flow
  - We want to execute some code ("advice") each time a join point is reached
  - We do *not* want to clutter up the code with explicit indicators saying *"This is a join point"*
  - AspectJ provides a syntax for indicating these join points "from outside" the actual code

- A join point is a point in the program flow "where something happens"
  - Examples:
    - When a method is called
    - When an exception is thrown
    - When a variable is accessed

Pointcuts

- A pointcut is used to select one or more concrete joinpoints
- Additionally: expose context

Context

```
call(void setX(XObject)) && within(YObject) && target(obj)
```

Pointcut Designators (1)

When a method is called:

```
call(void Point.setX(int))
execution(void Point.setX(int))
```

When a field is accessed:

```
set(!private * *.*)
get(!private * *.*)
```

When an exception handler executes:

```
handler(ArrayOutOfBoundsException)
```

Pointcut Designators (2)

When an object is executing (i.e., this is of a given type):

this(MyClass)

Class Name

When an object is target of an execution or field access:

target(MyOtherClass)

… these are often used for context exposition

Instance Variable Name

target(obj)

Pointcut Designators (3)

Static initialization:

```
staticinitialization(MyClass)
```

When an advice is executing:

```
adviceexecution()
```

Wildcards may be used instead of all names:

```
set(!private * *.*)
```

Annotations are designated somewhat specially with ‘@‘:

```
execution(* (@Logged *.*(*))
```

Pointcut Composition

Pointcuts compose through the operations **or** ("||"), **and** ("&&") and **not** ("!")

target(Point) && call(int *())

Chooses any call to an int method with no arguments on an instance of Point, regardless of its name

call(* *(..)) && (within(Line) || within(Point))

Chooses any call to any method where the call is made from the code in Point's or Line's type declaration

within(*) && execution(*.new(int))

Chooses the execution of any constructor taking exactly one int argument, regardless of where the call is made from

!this(Point) && call(int *(..))

Chooses any method call to an int method when the executing object is any type except Point

Notable Restriction

Joinpoint matching is type-level only

You cannot match a specific instance of a type

(at weaving time, no instances exist)

Kinds of Advices

In addition, different kinds of advice can be used:

`@Pointcut` (to **define** a pointcut, but not directly use it in an advice, typically used on an empty dummy method)

```
@Pointcut("call(void Point.setX(int))")
```

`@Before` (weave advice code in **before** the pointcut)

```
@Before("call(void Point.setX(int))")
```

`@After` (weave advice code in **after** the pointcut)

Variants: `@AfterReturning` and `@AfterThrowing`

```
@After("call(void Point.setX(int))")
```

`@Around` (**replace** pointcut with code)

```
@Around("call(void Point.setX(int))")
```

Replacing Pointcuts

In `@Around` advices, a special method parameter of type `ProceedingJoinPoint` is used to give access to the original implementation

```
@Around("get(public static final Long TestClass.TIMEOUT)")
public Long replaceTimeoutGetter(ProceedingJoinPoint pjp)
   throws Throwable {
   // increase timeout * 1000
   return ((Long)pjp.proceed()) * 1000;
}
```

We are not required to ever invoke the original implementation!

```
@Around("get(public static final Long TestClass.TIMEOUT)")
public Long replaceTimeoutGetter() {
   // return fixed timeout of 60000
   return 60 * 1000;
}
```

Call vs. Execution Pointcuts (1)

- Notorious source of error: confusing `call` with `execution` join points for methods
  - `Call` matches before or after a method is called (i.e., still in the scope of the caller)
    - E.g., the `call` join point is the last thing that happens **before** the method is actually invoked
  - `Execution` matches when the method starts to execute (i.e. already in the scope of the callee)
    - E.g., the `execution` join point is the first thing that happens during method invocation

```
@Pointcut("call(void Point.setX(int))")
@Pointcut("execution(void Point.setX(int))")
```

Call vs. Execution Pointcuts (2)

- `Call`
  - … **this** refers to the caller of the matched method
  - … **target** refers to the callee
  - … does not match reflective calls (unless you are also weaving system libraries)
  - … can be used to replace the result of constructor calls (e.g., for building object factories)
  - … can often be used to decorate the behavior of classes you cannot directly weave

Call vs. Execution Pointcuts (3)

- `Execution`
  - … **this** refers to the callee of the matched method
  - … **target** refers to the callee as well (!)
  - … also matches reflective calls
  - … cannot be used to replace the result of constructor calls

Weaving

The process of merging aspects into the program code is called weaving

Three ways of weaving:

- **Compile-Time Weaving** (weave as part of source-to-binary compilation)
- **Binary Weaving** (compile normally, then merge binaries in a post-compilation step)
- **Load-Time Weaving** (like binary weaving, but done when the class is loaded by the classloader – implemented via Java agent mechanism)

Compile-Time Weaving

- **Advantages:**
  - No startup performance degradation
  - Allows you to store and see the produced source code (good for debugging, and error tracking)
- **Disadvantages:**
  - Can’t weave third-party code (e.g., used libraries)
  - Requires usage of special compiler (`ajc`), which may produce worse regular Java code than your Sun or openjdk compiler

Binary Weaving

- **Advantages:**
  - No startup performance degradation
  - Allows you to store and see the produced binary (eases debugging)
  - Can be used with any compiler
  - Can weave third-party libraries, as long as you can live with permanently modifying them
- **Disadvantages:**
  - For the above reason, you really only want to do this with libraries that only this application is using (i.e., not system libraries, or things in your local Maven repository)

Load-Time Weaving

- **Advantages:**
  - Can be used with any compiler
  - Can sensefully weave pretty much anything, even system libraries
- **Disadvantages:**
  - Startup can become very slow
  - Hard to debug and understand, as the running code exists nowhere outside the classloader

Static Crosscutting

- Everything discussed so far was what is called dynamic crosscutting in AspectJ
  - Note that so far we have only changed implementations, never interfaces of components
- In rare cases we also want to change the interfaces of components

? Static Crosscutting / Introduction

Static Crosscutting

- One use case for static crosscutting:
  - Implementing **mixins** (classes that inherit from multiple superclasses at the same time)
  - Essentially a way to implement something that looks like multiple inheritance in Java
    - Technically: combination of adding an interface to the class definition and aggregation

Example: Introducing an ID Field Into All Classes

```
public interface IIdentifiable {
  UUID getPlatformId();
  void setPlatformId(UUID id);
}
```

Example: Introducing an ID Field Into All Classes

```
public interface IIdentifiable {
  UUID getPlatformId();
  void setPlatformId(UUID id);
}

public class IdentifiableMixin implements IIdentifiable {

  private UUID id;

  public UUID getPlatformId() { return id; }
  public void setPlatformId(UUID id) { this.id = id; }
}
```

Example: Introducing an ID Field Into All Classes

```
public interface IIdentifiable {
  UUID getPlatformId();
  void setPlatformId(UUID id);
}

public class IdentifiableMixin implements IIdentifiable {

  private UUID id;

  public UUID getPlatformId() { return id; }
  public void setPlatformId(UUID id) { this.id = id; }
}

@Aspect
public class MyAspect {
  @DeclareMixin("!is(InterfaceType) && !is(EnumType)")
  public static IIdentifiable createIIdentifiable() {
     return new IdentifiableMixin();
  }
}
```

Example: Introducing an ID Field Into All Classes

```
public class MyObject {
  private String aa;
   public void myMethod() { // … }
}
```

Example: Introducing an ID Field Into All Classes

```
public class MyObject {
  private String aa;
   public void myMethod() { // … }
}
```

```
MyObject object = new MyObject();

// note that this works after weaving!
((IIdentifiable) object).setPlatformId(UUID.randomUUID());

// …

UUID id = ((IIdentifiable) object).getPlatformId();
```

Conclusions

Aspect-oriented programming (AOP) is a different paradigm - a different way to think about programming

AspectJ is not itself a complete programming language, but an adjunct to Java

AspectJ does not add new capabilities to what Java can do, but adds new ways of modularizing the code

The core entities in AOSD are crosscutting concerns, joinpoints, pointcuts, advises, and weaving

Usually, AOP only changes the implementation of types. If we are also changing the interface, we call this static crosscutting or introduction.