Aspect-Oriented Programming

Johan Östlund

johano@dsv.su.se

Why AOP on this Course?

- AOP sets out to manage complexity
    - Modularizing software
- AOP is being accepted/adopted in ever increasing numbers both in academia and industry
- Everyone in the industry is likely to meet aspects at some point (at least in a couple of years)

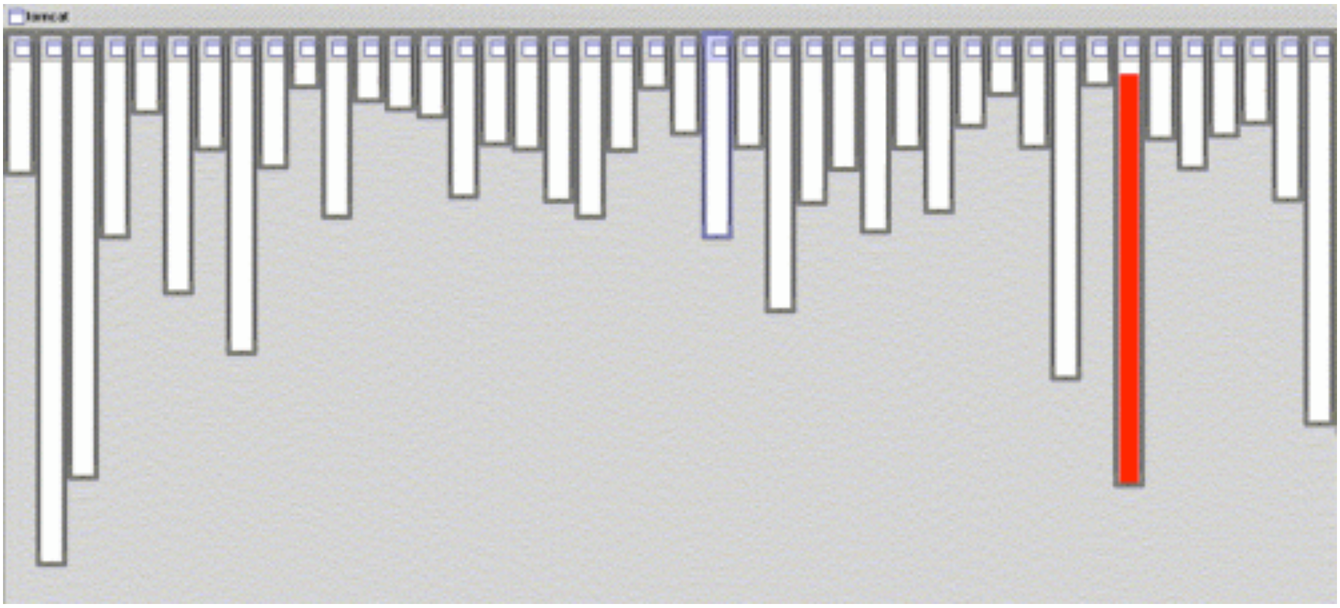
Separation of Concerns

- Subroutines
- Modules
- Object-orientation
  - Classes
  - Inheritance
  - Abstraction / Encapsulation
  - Polymorphism

Separation of Concerns w/OO

- A concern is modeled as an object or a class with slots/attributes and methods
- Well suited for many tasks because many problems are easily modularized into objects
- OO handles “vertical” concerns well

Separation of Concerns w/OO (cont'd)

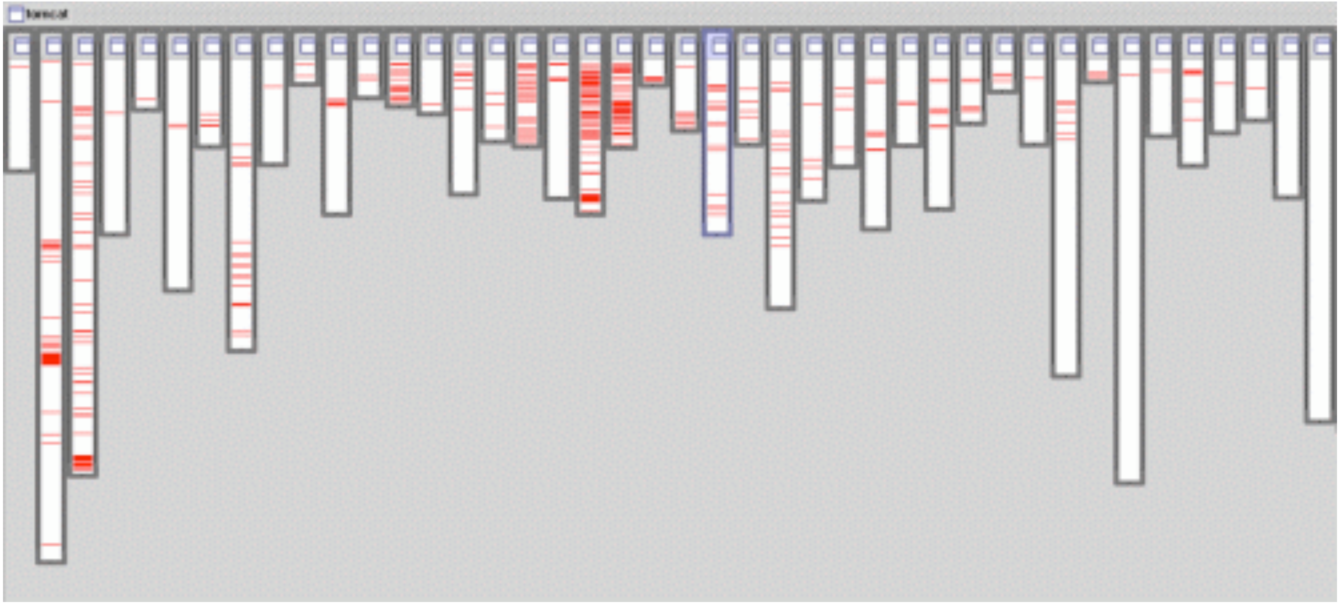


A subset of the classes in the Apache Tomcat Server. Each rectangle represents a class and the red area represents lines of XML-parsing code.

Separation of Concerns w/OO (cont'd)

- OO comes short when concerns are "horizontal", or cross-cutting.

Separation of Concerns w/OO (cont'd)



Logging code in the Apache Tomcat Server. Rectangles are classes and red lines represent lines of logging code.

Cross-cutting Concerns

- A concern that affects several classes or modules
- A concern that is not very well modularized
- Symptoms
  - Code tangling - when a code section or module handles several concerns simultaneously
  - Code scattering - when a concern is spread over several modules and is not very well localized and modularized

Code-tangling

```
def doX
  if session.isValid() and
       session.credentials > level
    actuallyPerformX()
  end
end

def doY
  if session.isValid() and
       session.credentials > level
    actuallyPerformY()
  end
end

def doZ
  if session.isValid() and
       session.credentials > level
    actuallyPerformZ()
  end
end
```

Cross-cutting Concerns break Encapsulation

- Classes handle several concerns
- Decreased cohesion within classes
- Increased coupling between classes

Cross-cutting Concerns decrease Reusability

- Because of the breaking of encapsulation

Code-tangling (cont'd)

```
public class Person
{
  private String name;
  ...
  public void setName(String name)
  {
    Logger.getLogger(...).log(Level.FINEST,
            “Name for “ + toString() +
              “ changed to: “ + name);
    this.name = name;
  }
}
```

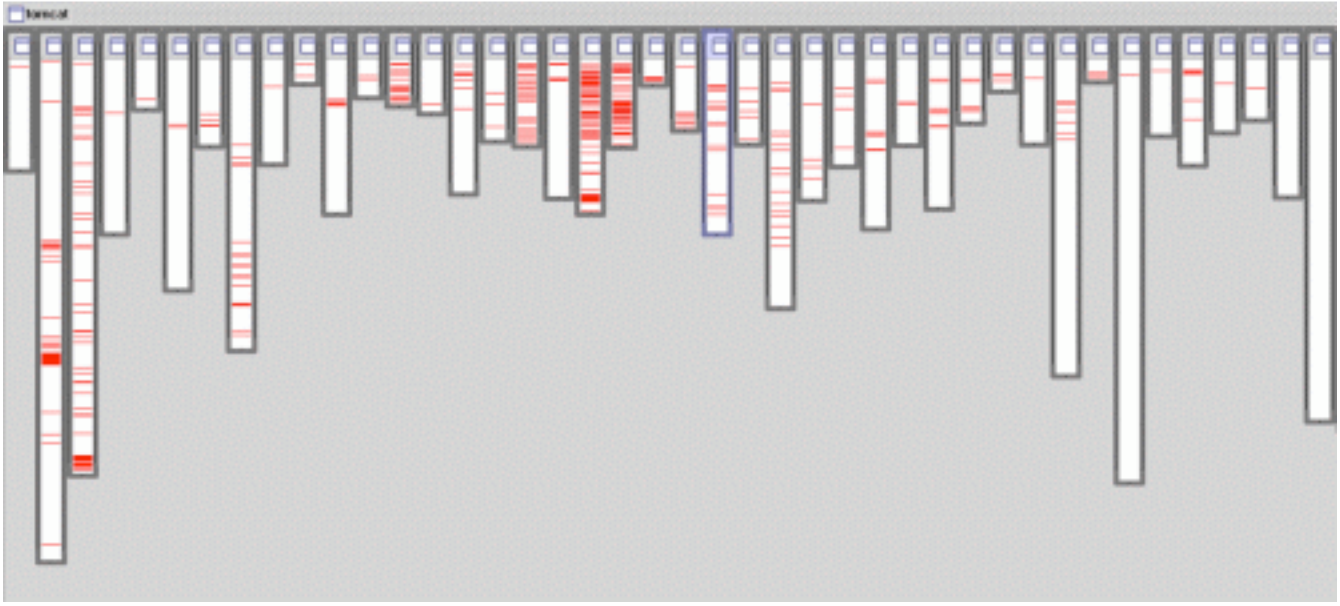
Breaking of Encapsulation

```
public class Person
{
  private String name;
  ...
  public void setName(String name)
  {
    Logger.getLogger(...).log(Level.FINEST,
          “Name for “ + toString() +
            “ changed to: “ + name);
    this.name = name;
  }
}
```

Code-scattering

- When a concern, and thus similar code, is distributed throughout several program modules
- Code duplication

Code-scattering



Logging code in the Apache Tomcat Server. Rectangles are classes and red lines represent lines of logging code.

What Concerns are Cross-cutting?

- Logging/Tracing/Debugging
  - Perhaps the canonical applications of AOP
- Access Control
- Design by Contract (pre-/post-conditions)
- Transaction management
- Thread Synchronization
- Error Handling
- Caching of Data
- etc.

Managing Cross-cutting Concerns

- Conventions - eliminate the bad effects of cross-cutting concerns to the largest extent possible through the use of conventions, etc.

```
public class Person
{
  private String name;
  ...
  public void setName(String name)
  {
    if (Logger.shouldLog(Level.FINEST))
      Logger.getLogger().log(Level.FINEST,
           “Name for “ + toString() +
            “ changed to: “ + name);
    this.name = name;
  }
}
```

Managing Cross-cutting Concerns (cont'd)

- Accepting / Neglecting - simply accepting cross-cutting concerns, and their effects
  - e.g., no code reuse

Managing Cross-cutting Concerns (cont'd)

- Aspect-Oriented Programming

Aspect-Oriented Programming

- AOP allows separate concerns to be separately expressed but nevertheless be automatically unified into working systems

- AOP enables modularization of cross-cutting concerns

Defining AOP

- Each AOPL comes with its own (unambiguous) formal description of what AOP is but there's
- No single definition that is
  - common to all AOPLs and
  - sufficiently distinguishes it from other, long established programming concepts
- There is, though, a common understanding what AOP is good for, namely modularizing cross-cutting concerns

Defining AOP (cont'd)

- The (probably) best known definition AOP is

  aspect-orientation = quantification +
  obliviousness

Obliviousness

- Obliviousness means that a program has no knowledge of which aspects modify it or when
- Some say that obliviousness is what distinguishes AOP from event-driven systems
- Obliviousness as a defining characteristic of AOP has been questioned by some in the AOP community
- Obliviousness comes as a side-effect of quantification

Quantification

- Quantification means that an aspect can affect arbitrarily many different points in a program
- Quantification is widely accepted as a defining characteristic of AOP

Defining AOP (cont'd)

- AOP enables programming statements such as

  In a program P, whenever condition C arises, perform action A.

- As there is no common definition we use the de facto standard, AspectJ, to interpret P, C and A

Defining AOP (cont'd)

- Translated in terms of AspectJ the parts of the formula read
  - P is the execution of a program, which includes the execution of advice
  - C is a set of pointcuts specifying the target elements of the aspect in the program and the context in which they occur (mostly variables, but also stack content)
  - A is a piece of advice (code) that depends on the context captured by C; and
  - the quantification is implicit in the compiler/weaver

Defining AOP (cont'd)

- The sentence "*In programs P, whenever condition C arises perform action A*" captures how an aspect (C, A) affects a given program P,
- but says nothing about P's knowledge of the aspect (C, A), and thus nothing about obliviousness
- However, the context provided to an action A is provided by the aspect (C, A) and not by the program P. Thus the program is oblivious to which program elements an aspect relies on, as opposed to a function call where arguments are explicitly passed to the function

```
public class Person
{
  private String name;
  ...
  public void setName(String name)
  {
    this.name = name;
  }
}

public aspect LoggingAspect
{
  before(Person p, String s) :
    call(void Person.setName(String))
      && target(p)
      && args(s)
  {
    Logger.getLogger(...).log(Level.FINEST,
           “Name for “ + p +
            “ changed to: “ + name);
  }
}
```

```
public class Person
{
  private String name;
  ...
  public void setName(String name)
  {
    this.name = name;
  }
}

public aspect LoggingAspect
{
  before(Object c, Person p, String s) :
    call(void Person.setName(String))
    && this(c)
    && target(p)
    && args(s)
  {
    Logger.getLogger(...).log(Level.FINEST,
        “Name for “ + p +
          “ changed to: “ + s +
           “ (invoked by “ + c + “)”);
   }
}
```

What do we need?

- A programming language
- An aspect language
- A way to intermix the execution of the two (weaving)

Programming Language

- Any programming language
  - Functional (AspectL)
  - Procedural (AspectC)
  - Object-oriented (AspectJ, AspectC++, AspectCocoa, Aspect#)
- Most AOP implementations are for OOPLs due to popularity of OO
- Dynamic languages, e.g. Ruby, typically have extensive reflection and meta programming support which in many cases presents equivalents to AOP features, and thus AOP-support for such languages makes less sense

Aspect Language

- Join points - defined points in the control flow of the base program
    - Method call
    - Constructor call
    - Field access
    - etc.
- Pointcuts - a set of join points
- Advice - code to execute when a pointcut matches

Weaving

- Weaving is the intertwining of aspect code into the base program
- Static weaving
  - Source code weaving
  - Byte or object code weaving
  - Load-time weaving
- Run-time
  - Dynamic weaving (VM-support)

Source Code Weaving

- Basically preprocessing
- Fairly easy to implement
- May require all code to be present at compile time
- Costly for dynamic behavior

Byte Code Weaving

- Separate compilation
- Weaving of third-party classes
- May require all classes to be present at compile time
- May be very time and memory consuming
  - Keep all classes in memory
  - Parse all instructions in the program
- May cause clashes when several instrumenting tools are used
- Reflection calls cannot be handled in a good way
- 64K limit on method byte code (JSP - Servlet)

Dynamic Weaving

- Run-time deployment of different aspects at different times
- Efficient and flexible
- Works with VM-hooks and event-subscribe
- Current implementations have poor support for obliviousness, and the concept of aspect is blurry (JRockit - BEA Systems)
- Steamloom is better, but it’s only a research implementation (IBM’s Jikes Research VM). Uses HotSwap to dynamically recompile methods in run-time

Aspects

- Modularize cross-cutting concerns
- Roughly equivalent to a class or module
- Defined in terms of
    - Join points (not really true)
    - Pointcuts
    - Advice

Join points

- Well defined execution points in the program flow, e.g. method call, constructor call, object instantiation, field access etc.
  - Their textual representation in the program text are referred to as join point shadows, e.g., a method declaration is a shadow of a method execution join point
- The level of granularity or expressiveness is dependent on the base language

```
public class MyClass
{
  private String myField;

  public void setFieldValue(String newValue)
  {
    myField = newValue;
  }
}
```

Pointcuts

- A pointcut is a defined set of join points

```
public class MyClass
{
  private String myField;

  public void setFieldValue(String newValue)
  {
    myField = newValue;
  }
}

public aspect LoggingAspect
{
  before(String s) :
    execution(
      void MyClass.setFieldValue(String))
  {
    System.out.println(“Method called”);
  }
}
```

Advice

- The code to be executed when a pointcut matches
- Roughly equivalent to a method

```
public class MyClass
{
  private String myField;

  public void setFieldValue(String newValue)
  {
    myField = newValue;
  }
}

public aspect LoggingAspect
{
  before(String s) :
    execution(
      void MyClass.setFieldValue(String))
  {
    System.out.println(“Method called”);
  }
}
```

Static Join Points

- Join points which have a static representation

  ```
  call(void MyClass.setFieldValue(String))
  ```

- These places can be found and instrumented at compile-time

Dynamic Join Points

- Join points which don't necessarily have a static representation, or where it's uncertain whether an advice should apply

```
call(* *.*(..))
&& cflow(call(
    void MyClass.setFieldValue(String)))
```

```
call(* *.*(..)) && if (someVar == 0)
```

- May lead to insertion of dynamic checks at, depending on the granularity of the join point model, basically every instruction. Clearly a performance issue.

VM Support for Dynamic Join Points

- Structure preserving compilation
- Run-time (lazy) (re-)compilation of methods to include aspect code, when some dynamic join point has matched.

Examples

Lazy Initialization

- Avoid allocation of resources unless necessary
- Image processing software with thumbnails and lazy loading of actual image files

```
public class Image
{

  private String filename;
  private Thumbnail thumb;
  private ImageBuffer img;

  public Image(String filename)
  {
    this.filename = filename;
    thumb = ImageLoader.getDefault()
                    .loadThumb(filename);
  }
```

```
  public ImageBuffer getImageBuffer()
  {
    if (img == null)
      img = ImageLoader.getDefault()
                       .loadImage(filename);
    return img;
  }

  public void displayImageOn(Canvas c)
  {
    if (img == null)
      img = ImageLoader.getDefault()
                       .loadImage(filename);
    c.drawImage(img);
  }

  public void applyFilter(Filter f)
  {
    if (img == null)
      img = ImageLoader.getDefault()
                       .loadImage(filename);
    // Apply filter on img
  }

} // Image
```

Aspectation

- Extract null check and method call and make it an aspect executing whenever the field *img* is read

```
public aspect LazyAspect
{

  pointcut lazyPointcut(Image i) :
    get(ImageBuffer Image.img)
    && target(i)
    && !within(LazyAspect);

  before(Image i) : lazyPointcut(i)
  {
    if (i.img == null)
      i.img = ImageLoader.getDefault()
                    .loadImage(i.filename);
  }

}
```

```
public class Image
{

  protected String filename;
  private Thumbnail thumb;
  protected ImageBuffer img;

  public Image(String filename)
  {
    this.filename = filename;
    thumb = ImageLoader.getDefault()
                      .loadThumb(filename);
  }
```

```
  public ImageBuffer getImageBuffer()
  {
    return img;
  }

  public void displayImageOn(Canvas c)
  {
    c.drawImage(img);
  }

  public void applyFilter(Filter f)
  {
    // Apply filter on img
  }

} // Image
```

```
public aspect LazyAspect
{

  pointcut lazyPointcut(Image i) :
    get(ImageBuffer Image.img)
    && target(i)
    && !within(LazyAspect);

  before(Image i) : lazyPointcut(i)
  {
    if (i.img == null)
      i.img = ImageLoader.getDefault()
               .loadImageFromDB(i.filename);
  }

}
```

```
public aspect LazyAspect
{

  declare warning :            // or error
    call(ImageBuffer
             ImageLoader.loadImage(String))
    && within(Image);

  pointcut lazyPointcut(Image i) :
    get(ImageBuffer Image.img)
    && target(i)
    && !within(LazyAspect)

  before(Image i) : lazyPointcut(i)
  {
    if (i.img == null)
      i.img = ImageLoader.getDefault()
                    .loadImage(i.filename);
  }

}
```

Default Interface Implementation

- Provide a default implementation of interface methods
- In some sense multiple inheritance, but without conflicts
- Somewhat like traits in Scala or categories in Objective-C

```
public interface Sortable
{

  public int compare(Object other);

  public boolean equalTo(Object other);

  public boolean greaterThan(Object other);

  public boolean lessThan(Object other);

}
```

```
public aspect DefaultSortableAspect
{

  public boolean Sortable.equalTo(
                            Object other)
  {
    return compare(other) == 0;
  }

  public boolean Sortable.greaterThan(
                            Object other)
  {
    return compare(other) > 0;
  }

  public boolean Sortable.lessThan(
                            Object other)
  {
    return compare(other) < 0;
  }
}
```

```
public class AClass implements Sortable
{
  private int number;
  ...

  public int compare(Object other)
  {
    return number - ((AClass) other).number;
  }

  public static void main(String[] args)
  {
    Sortable c1 = new AClass();
    Sortable c2 = new AClass();
    boolean b = c1.greaterThan(c2);
  }

}
```

Listener Control

- We want to make sure that listeners on UI-components are unique and only added once

```
public class Main extends JFrame
  implements ActionListener
{
  private JButton b = new JButton("Button");
  public void actionPerformed(ActionEvent a)
  {
    JOptionPane.showMessageDialog(this,
                                  "Some message");
  }

  public Main()
  {
    JPanel p = new JPanel();
    p.add(b);
    b.addActionListener(this);
    b.addActionListener(this);
    getContentPane().add(p);
    // display the frame
  }

  public static void main(String[] args)
  {
    new Main();
  }
}
```

```
public aspect ListenerAddChecker
{

  private HashSet comps = new HashSet();

  pointcut listenerAdd(Object o) :
    call(void *.add*Listener(..))
    && target(o);

  void around(Object o) : listenerAdd(o)
  {
    if (comps.contains(o) == false)
    {
      comps.add(o);
      proceed(o);
    }
  }
```

```
  pointcut listenerRemove(Object o) :
    call(void *.remove*Listener(..))
    && target(o);

  before(Object o) : listenerRemove(o)
  {
    comps.remove(o);
  }

} // ListenerAddChecker
```

Synchronization

- We want to add synchronization of a system using aspects
- We start with a single-threaded system

```
class Data
{
   public void method()
   {
      // Perform some action
   }
}

public class Main
{
   public static void main(String[] args)
   {
      Data d = new Data();
      d.method();
   }
}
```

Synchronization

- We add our locking scheme with an aspect

```
public interface Shared
{}

public aspect SynchAspect
{
   private LockMap locks = new LockMap();

   declare parents : Data implements Shared;

   after(Object s) :
      execution(Object+.new(..))
      && this(s)
      && if (s instanceof Shared)
      && !within(SynchAspect)
   {
      locks.add(s);
   }
```

```
   pointcut sharedCall(Object s) :
      call(* Object+.*(..))
      && target(s)
      && if (s instanceof Shared)
      && !within(SynchAspect);


   before(Object s) : sharedCall(s)
   {
      locks.acquire(s);
   }

   after(Object s) : sharedCall(s)
   {
      locks.release(s);
   }
}
```

AspectJ,
an aspect-oriented programming language

History

- Developed at XEROX PARC
- Emerged from research on OO, reflection and meta-programming
- In 2002 AspectJ was transferred to an openly-developed eclipse.org project

Pointcuts

- **call**(*MethodPattern*) - captures the call of any method matching *MethodPattern*
- **execution**(*MethodPattern*) - captures the execution of any method matching *MethodPattern*
- **handler**(*TypePattern*) - captures the catching of an exception matching *TypePattern*
- **this**(*Type* | *Identifier*) - captures all join points where the object bound to *this* is an instance of *Type* or has the type of *Identifier*
- **target**(*Type* | *Identifier*) - captures all join points where the target of a method call or field access is an instance of *Type* or has the type of *Identifier*

Pointcuts, cont'd

- **args**([*Types* | *Identifiers*]) - captures all join points where the arguments are instances of *Types* or of the type of *Identifiers*
- **get**(*FieldPattern*) - captures all join points where a field is accessed that has a signature that matches *FieldPattern*
- **set**(*FieldPattern*) - captures all join points where a field is updated that has a signature that matches *FieldPattern*
- **within**(*TypePattern*) - captures all join points where the executing code is defined in a type matched by *TypePattern*
- **cflow**(*Pointcut*) - captures all join points in the control flow of any join point P picked out by *Pointcut*, including P itself

Pointcuts, cont'd

- Pointcuts may be combined with the logical operators && and || and negated by !

```
call(String Object+.toString())
&& within(org.myproject..*)
```

```
call(String Object+.toString())
|| call(boolean Object+.equals(Object))
&& !within(java..*)
```

Pointcut Definition

- pointcut pointcutName(<params>) : pointcuts

```
pointcut allMethodCalls() :
  call(* *.*(..));

pointcut allMethodCalls2(Object o) :
  call(* Object+.*(..)) && target(o);

pointcut captureAllMyClass(MyClass o) :
  call(* MyClass.*(..)) && target(o);

pointcut captureAllListInPackage() :
  call(* List+.add*(..))
  && within(mypackage..*);
```

Advice

- The “methods” of aspects
  - **before** - executes its code before a matching join point
  - **after** - executes its code after a matching join point
  - **around** - wraps a matching join point, depending on an invocation of the special method **proceed**() to proceed to executing the wrapped join point. As the **around** advice runs in place of the join point it operates over (rather than before or after it) it may return a value, which in turn demands that it be declared with a return type.

Advice Declaration

- advice(<*params*>) : pointcuts { stmts }

```
before() : call(* Object+.*toString())
{
  System.out.println(“Logging:”);
}

after() : call(* Object+.*toString())
{
  System.out.println(“Done logging”);
}

boolean around(Object o) :
  call(boolean List+.add*(Object))
  && args(o)
{
  if (o == null) return false;
  else return proceed(o);
}
```

Inter-type Declarations

- Add state or functionality to existing classes

```
public aspect ITDAspect
{
  private String AClass.newField = null;

  public String AClass.getField()
  {
    return newField;
  }

  public void AClass.setField(String s)
  {
    newField = s;
  }

}
```

```
public aspect DeclAspect
{

  declare parents: SomeClass implements
                     Comparable;

  public int SomeClass.compareTo(Object o)
  {
    return 0;
  }

}
```

Exercise

- The exercise is to implement access control in a very simple HRM system using AspectJ
- The system consists of three classes Coder, Manager and Employee, which is the common superclass of Coder and Manager
- There is a JUnit test case; all tests should pass
- No altering of the base code
- http://dsv.su.se/~johano/ericsson/