Aspects in AspectJ

- Motivation
- Aspect Oriented Programming: a brief introduction to terminology
- Installation
- Experimentation
- AspectJ – some details
- AspectJ – things you should know about but we dont have time to cover in detail
- AspectJ – profiling problem

Motivation

"Ability is what you're capable of doing.
Motivation determines what you do. Attitude
determines how well you do it." **Lou Holtz**

Motivation

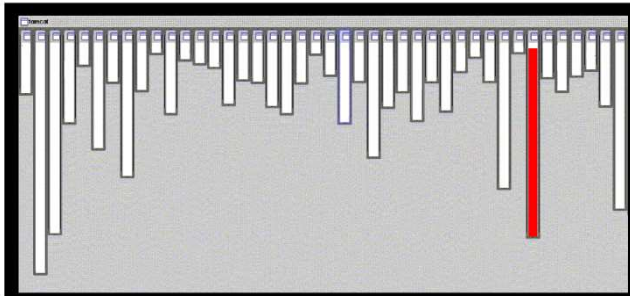Even well designed systems have aspects that cut across a large number of the components

For example, consider Apache Tomcat, where certain aspects of functionality are:

- XML parsing
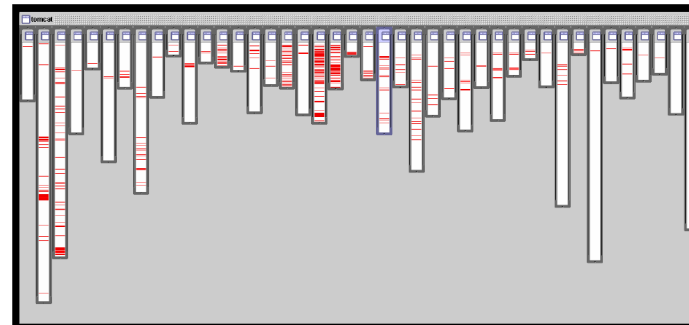- URL pattern matching
- Logging
- Session Management

Question: which of these are well localised?
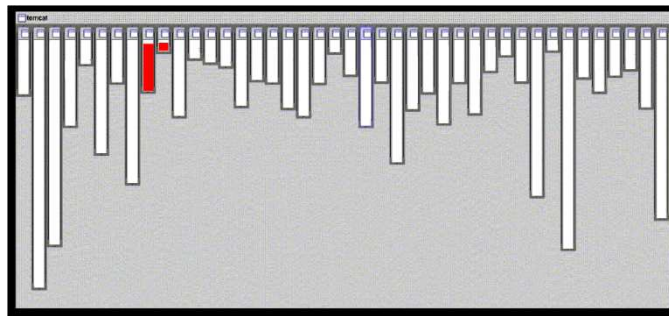
Motivation

XML parsing

URL pattern matching

Logging

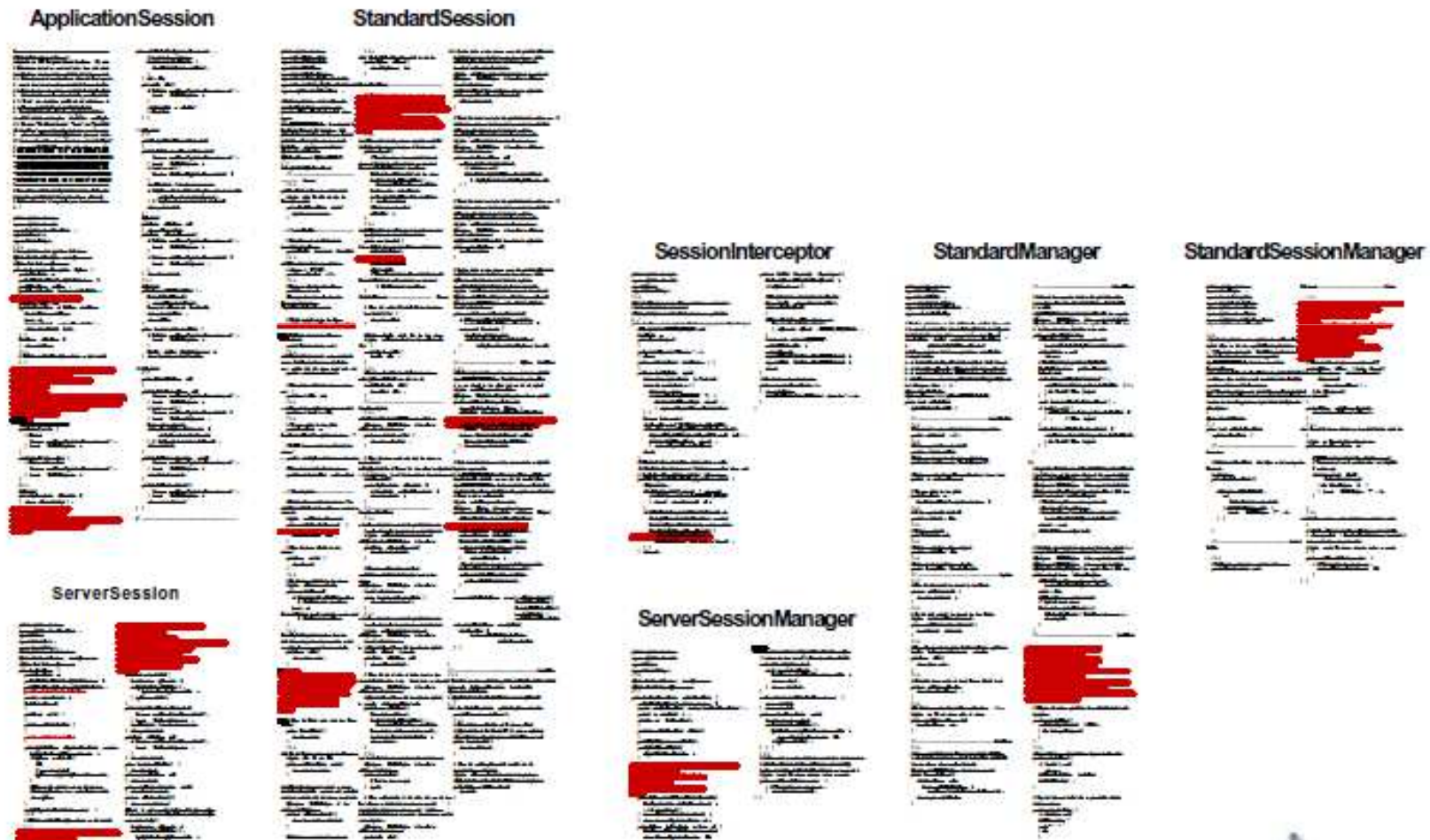Session Management: different types

- Application Session
- Server Session
- Standard Session
- Managers

So the code for each session stage should be well localised?
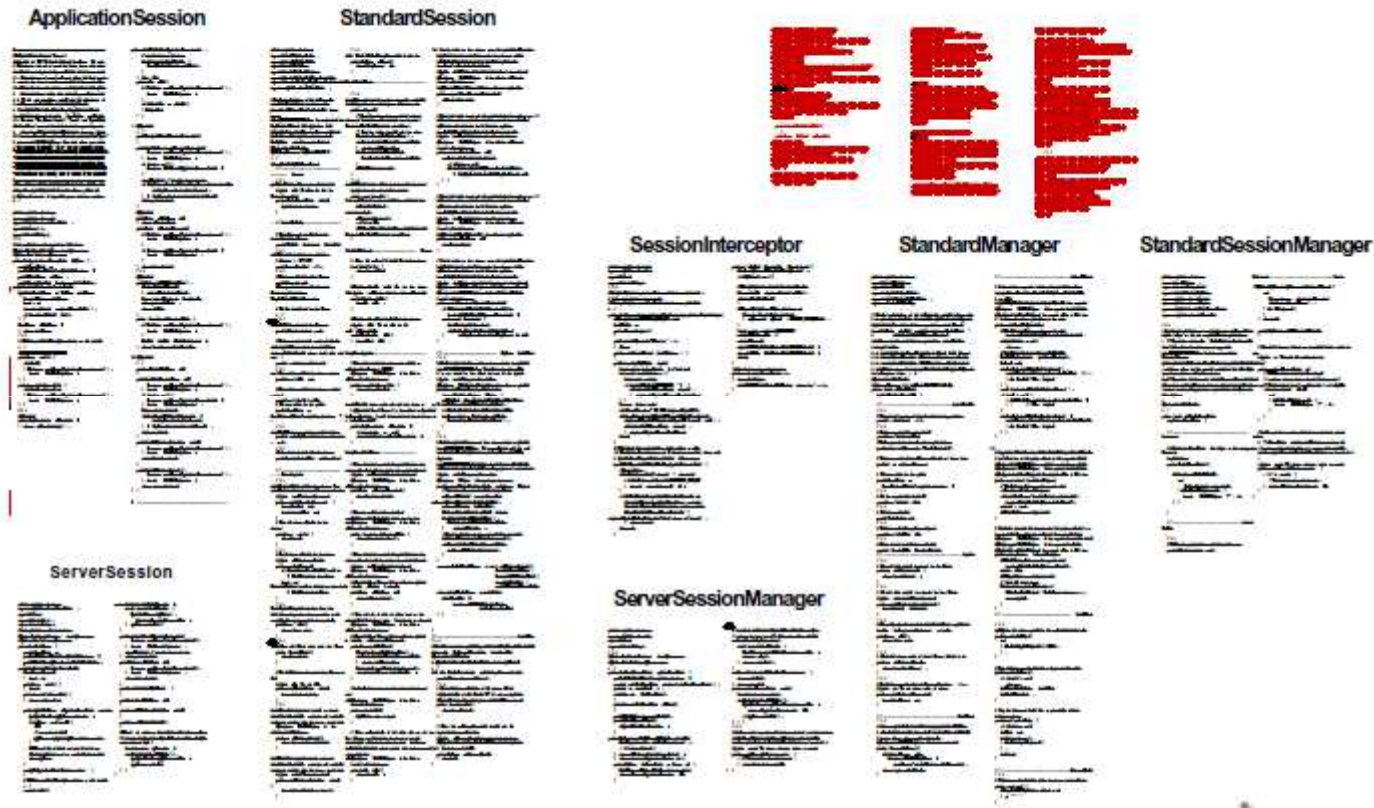
Motivation

Session Expiration: code is distributed through
lots of different classes

Motivation

Session Expiration: a better design?

Unfortunately, if we do this we will have to compromise by redistributing previously localised code.

Motivation

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

D. Parnas. *On the Criteria to Be Used in Decomposing Systems into Modules. Comm. ACM 15, 12 (December 1972), 1053-1058.*

Design principles for decomposition:
- Information-hiding modules
- Identify design decisions that are likely to change.
- Isolate these in separate modules (separation of concerns)

Motivation

Cross-Cutting Concerns

In the motivation,

- XML parsing and URL pattern matching the class hierarchy,
- Logging and Session Mangement 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!

Motivation

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 concens.*

Motivation

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 penalties through indirection
- often leaves some cross-cutting boiler-plate
- can't hope to capture all aspects

Aspect Oriented Programming: a brief introduction

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.

Aspect Oriented Programming: a brief introduction

Don't Forget That Good Design Helps Avoid Cross Cutting: so only use Aspects if they are really needed, and not just because the design is bad!

Example of Good Design – The 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.

The "Law of Demeter" provides a classic solution, see:

> Lieberherr, Karl. J. and Holland, I.
> *Assuring good style for object-oriented programs*
> IEEE Software, September 1989, pp 38-48

Aspect Oriented Programming: a brief introduction

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 . . .*

Implementations of aspects are attached to certain points in the
Implementation, eg:
- 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*

Aspect Oriented Programming: a brief introduction

Code Example: Figure Editor (from http://eclipse.org/aspectj/doc/released/progguide/)
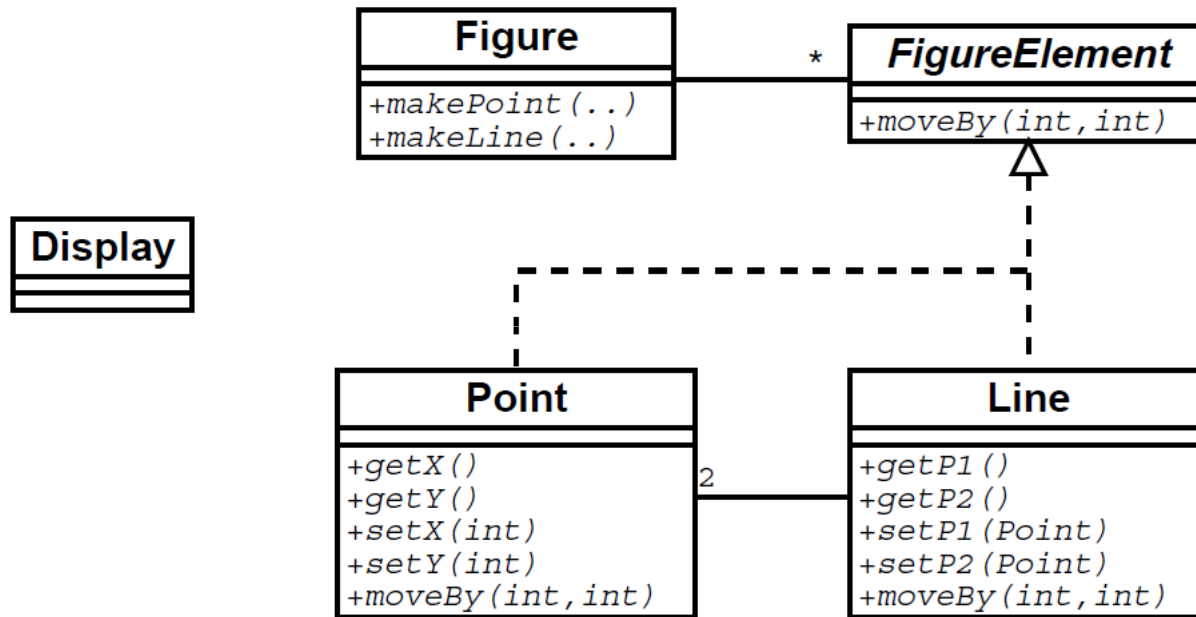
Aspect Oriented Programming: a brief introduction

Code Example: 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) { 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; }
}
```

Aspect Oriented Programming: a brief introduction

Code Example: Figure Editor – some join points

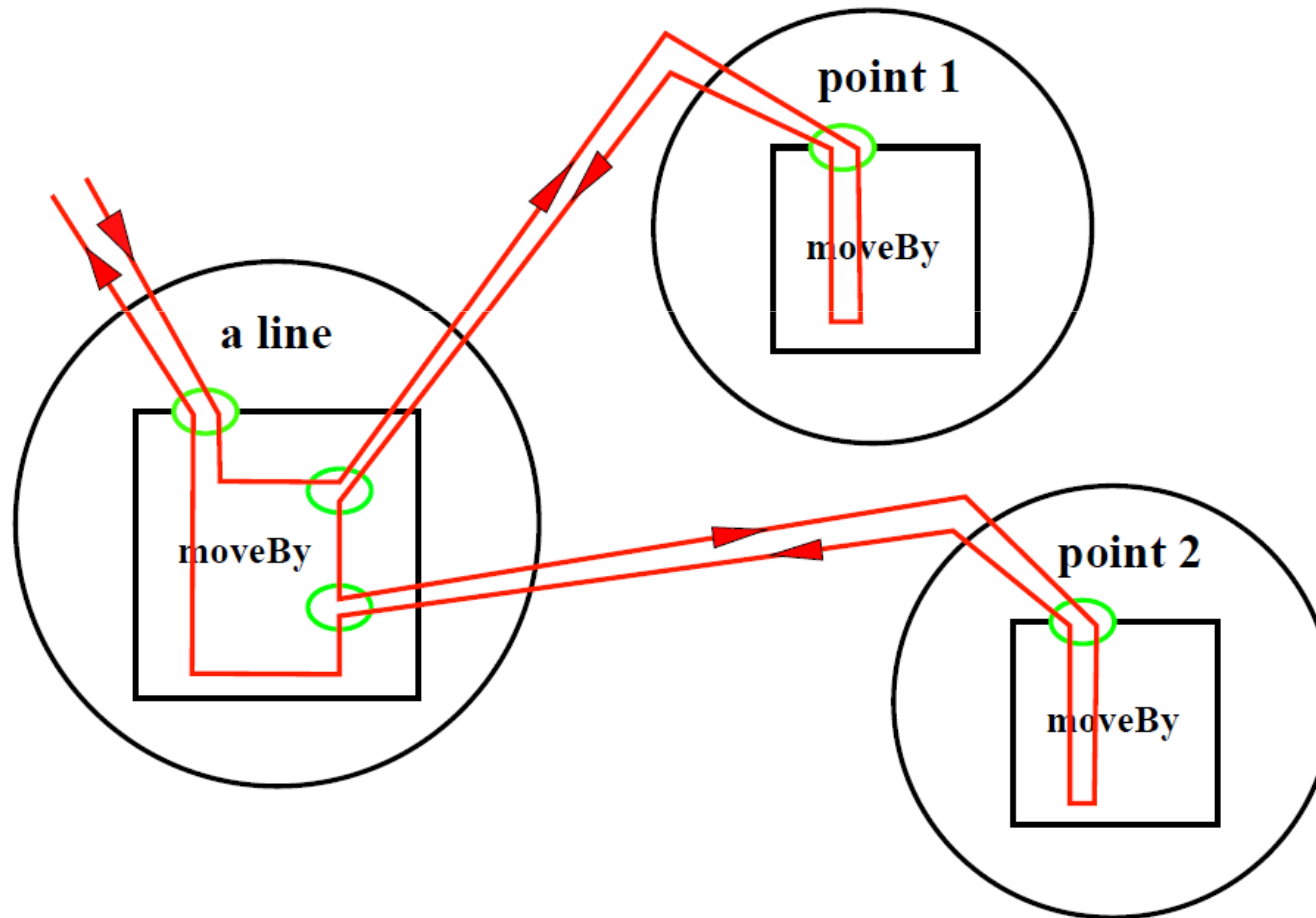**Aspect Oriented Programming: a brief introduction**

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

Aspect Oriented Programming: a brief introduction

Advice

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.

Aspect Oriented Programming: a brief introduction

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.

Aspect Oriented Programming: a brief introduction

Weaving:

A program called an *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)

Aspect Oriented Programming: a brief introduction

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

Installation

Download the Eclipse AJDT plugin from
http://eclipse.org/ajdt/downloads/

AJDT: AspectJ Development Tools

Downloads

Please make sure you pick a version of AJDT which matches the version of Eclipse you are using.

All downloads are provided under the terms and conditions of **the Eclipse Foundation Software User Agreement** unless otherwise specified.

AJDT 2.2 release builds for Eclipse 4.3, 4.2, 3.8, and 3.7

→ AJDT for Eclipse 4.3 *with JDT weaving*
What is **JDT weaving**?

2.2.3 Release Date:	July 2, 2013
AspectJ Version in 2.2.3:	1.7.3
Eclipse 4.3 Update Site URL:	http://download.eclipse.org/tools/ajdt/43/update
AJDT for Eclipse 4.3 Zip file:	ajdt_2.2.3_for_eclipse_4.3.zip

To install from a zip file, download the zip and point your p2 installer to that file. Then proceed as if it were a normal update site. Do *not* unzip the update site into the dropins directory.

→ AJDT for Eclipse 4.2, 3.8, and 3.7 *with JDT weaving*
What is **JDT weaving**?

|	
2.2.3 Release Date:	July 2, 2013
AspectJ Version in 2.2.3:	1.7.3
AspectJ Version in 2.2.2:	1.7.2
Eclipse 3.8 and 4.2 Update Site URL:	http://download.eclipse.org/tools/ajdt/42/update
Eclipse 3.7 Update Site URL:	http://download.eclipse.org/tools/ajdt/37/update
AJDT for Eclipse 4.2 and 3.8	ajdt_2.2.3_for_eclipse_4.2.zip

Installation

If AspectJ is already there then you should not need to add it (but may need to update it)

Installation

Check that you have the most recent version of AspectJ (for your Eclipse distribution)
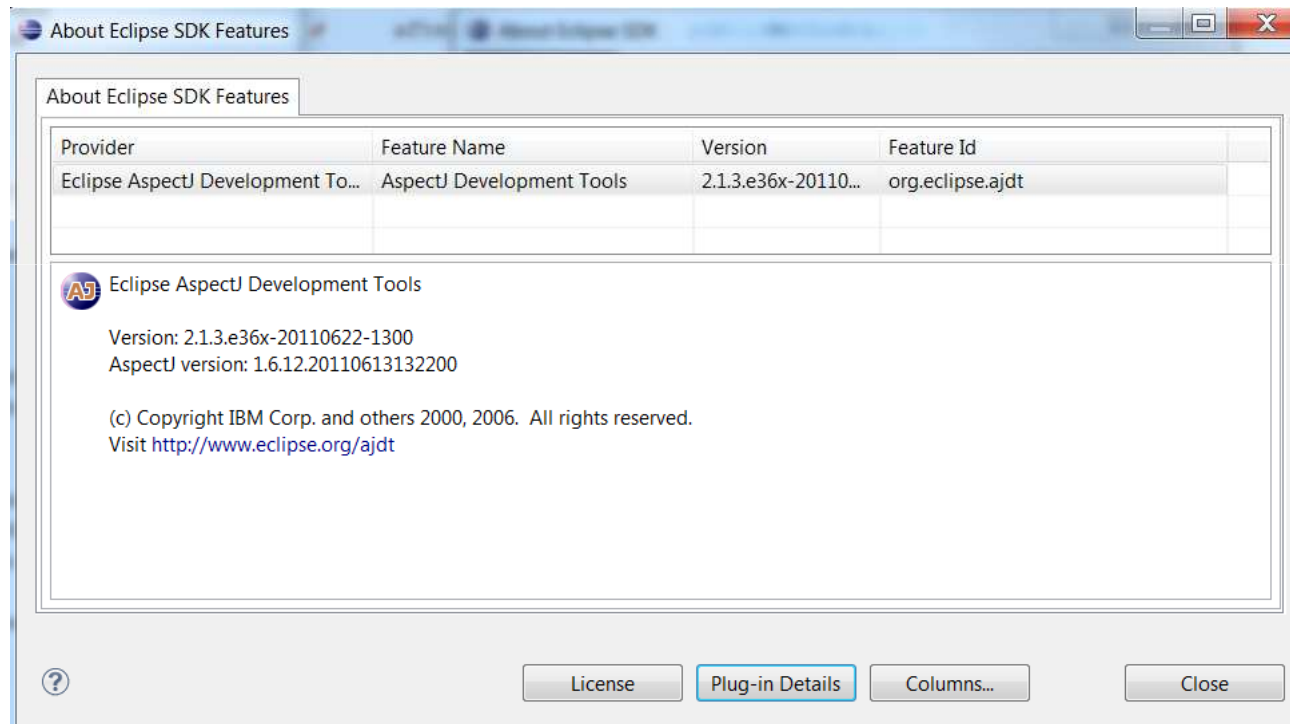
Installation

Check you have all the feature plug-ins that are necessary:
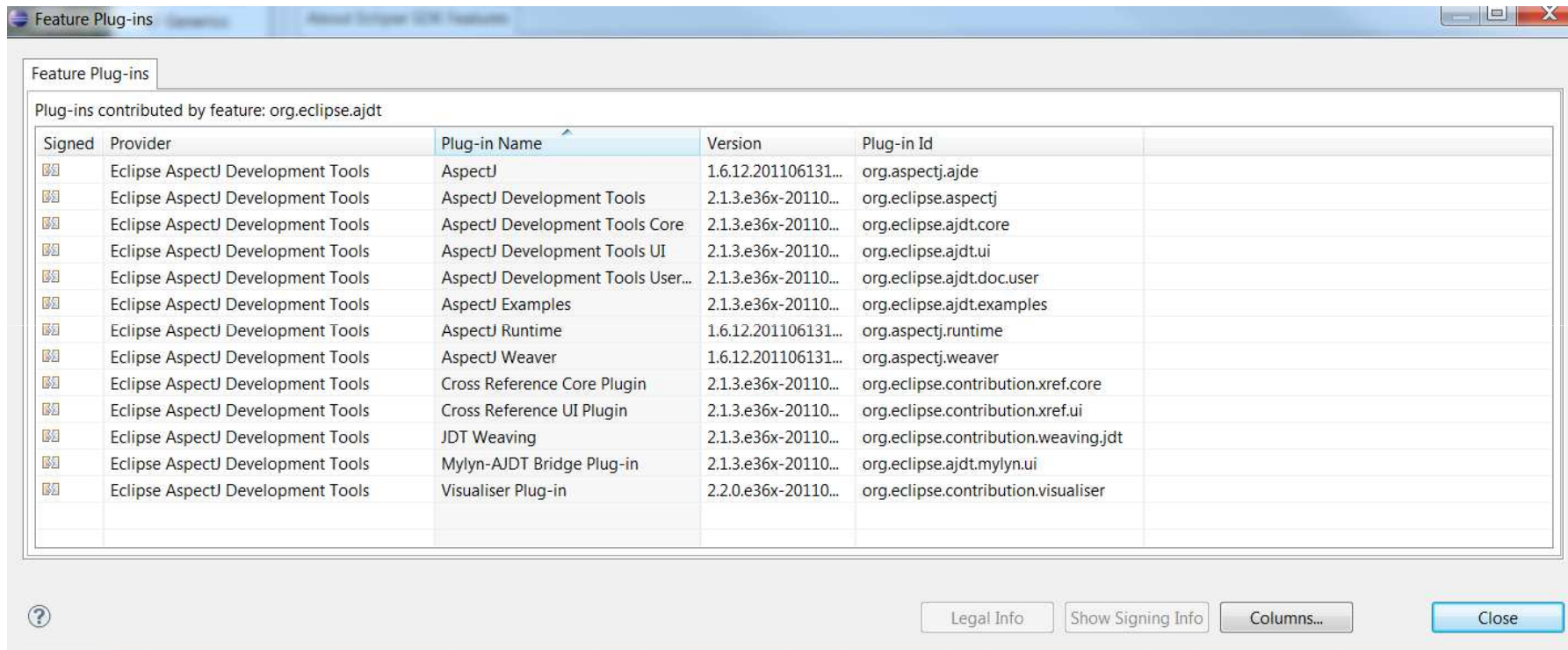
You do not need the optional plug-ins (but be aware that the AspectJ plug-in can interact with others and may require installation of these « glue » options)

Installation

Try Out The HelloWorld Example Demo:

http://www.eclipse.org/ajdt/demos/

Installation

Try Out The HelloWorld Example Demo:

```java
package p_helloworld;

public class Hello {

public static void main(String[] args) {
    sayHello();
}

public static void sayHello() {
    System.out.print("Hello");
}

}
```

Installation

Try Out The HelloWorld Example Demo:

```
package p_helloworld;

public aspect World {

pointcut greeting() : execution(*
Hello.sayHello(..));

    before()  : greeting() {
        System.out.println(" World!");
    }

}
```

Experimentation

Experiment with the Demo: what will/should happen if you add another aspect to the project?

```
package p_helloworld;

public aspect MethodCounter {

int counter =0;

pointcut publicMethodCall() : execution(public * *.*(..));
pointcut afterMain(): execution(public * *.main(..));

after() returning() : publicMethodCall() {
      counter++;
   }

after() returning() : afterMain() {
      System.out.println("Method  call counter = "+counter);
   }
}
```

Experimentation

Experiment with the Demo: can you write an Aspect that will time the execution of each method call?

```
package p_helloworld;

public aspect Profiler{

}
```

This should be easy to complete. Is the on-line documentation good enough?

Experimentation

Experiment with the Demo: can you write an Aspect that will time the execution of each method call?

When you run the code with all 3 aspects woven together, you should get output of the following form (more or less):

```
 World!
Hello
void p_helloworld.Hello.sayHello() took 26042892 nanoseconds

Method  call counter = 2

void p_helloworld.Hello.main(String[]) took 32298490 nanoseconds
```

AspectJ – some details

Reconsider the figure editor
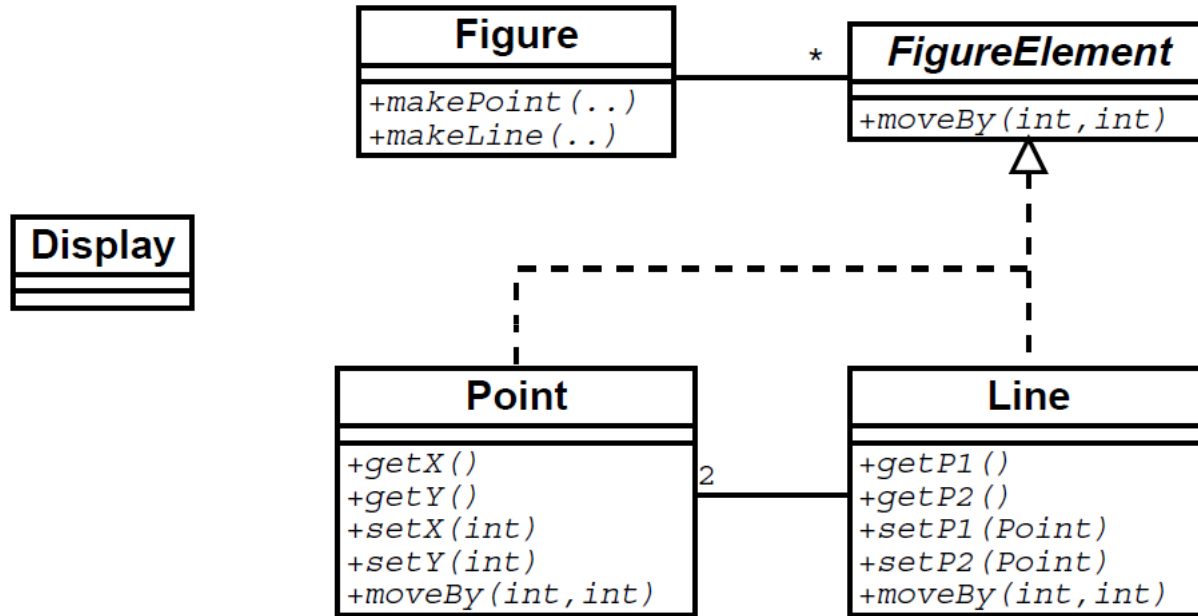
AspectJ – some details

Reconsider the figure editor

Primitive pointcuts:

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

each join point that is a call to a method that has the signature void Point.setX(int)

Also for interface signatures:

```
call(void FigureElement.moveBy(int,int))
```

Each call to the moveBy(int,int) method in a class that implements FigureElement

AspectJ – some details

Reconsider the figure editor

Pointcuts can be joined using boolean operators &&,||,!.

```
call(void Point.setX(int)) ||
call(void Point.setY(int))
```

calls to the setX and setY methods of Point.

Join points from different types possible:

```
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))
```

Any call to a state-changing method in the given FigureElement classes

AspectJ – some details

Reconsider the figure editor

Named Pointcuts

Pointcuts can and should be *declared to give them a name:*

```
pointcut stateChange() :
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));
```

Analogous to method declaration or typedef in C.

After declaration, stateChange() can be used wherever a pointcut is expected.

AspectJ – some details

Reconsider the figure editor

Wildcards

Method signatures can contain wildcards:

```
call(void java.io.PrintStream.println(*))
```

any PrintStream method named println returning void and taking exactly one argument of any type.

```
call(public * Figure.*(..))
```

any public method in Figure.

```
call(public * Line.set*(..))
```

any method in Line with a name starting with set.

AspectJ – some details

Reconsider the figure editor

Example

The pointcut from before, using wildcards:

```
pointcut stateChange() :
call(void FigureElement.moveBy(int,int)) ||
call(* Point.set*(*)) ||
call(* Line.set*(*));
```

AspectJ – some details

Reconsider the figure editor

Advice in AspectJ

Advice can be attached to join points:

```
before(): stateChange() {
System.out.println("about to change state");
}


after() returning: stateChange() {
System.out.println("just successfully changed state");
}
```

AspectJ – some details

//After every state changing call, update the display.

```
public aspect DisplayUpdating {
pointcut stateChange() :
call(void FigureElement.moveBy(int,int)) ||
call(* Point.set*(*)) || call(* Line.set*(*));
after() returning : stateChange() {
Display.update();
}
}
```

QUESTION: Does this look like a good use of Aspects?

AspectJ – some details

Type Patterns

Used to denote a set of types.

- `int`

- `Line`

- `*`

- `java.util.Map*`

- `java..*Factory*`

- `java.io.OutputStream+`

- `Foo+ && !Foo`

- `Foo+ && !(java..Bar*[])`

AspectJ – some details

Method Patterns

Used to denote a set of methods in one or more classes.

Method declaration:

```
public final void write(int x) throws IOException
```

Method pattern:

```
public final void Writer.write(int) throws IOException
```

Wildcards possible everywhere. . .

- `void *.write(int)`

- `void Foo+.set*(int)`

- `* Foo.write(*)`

- `* Foo.write(..)`

- `* Foo.write(..,int)`

- `* *.*(..)`

- `public Foo.*(..)`

- `!public *.*(..)`

- `* *.*(..) throws java.io.IOException`

- `* *.*(..) throws !java.io.IOException`

- `* *.*(..) throws (!java.io.IOException)`

AspectJ – some details

Constructor and Field Patterns

Select a set of constructors:

Like method patterns, with name `new` and no return type:

- `Line.new(int)`

- `public *.new(..)`

- ⋮

Field Patterns:

- `int Line.x`

- `public !final * FigureElement+.*`

- ⋮

AspectJ – some details

Primitive Pointcuts

- `call(MethodPattern)`

- `call(ConstructorPattern)`

- `execution(MethodPattern)`

- `execution(ConstructorPattern)`

- `get(FieldPattern)`

- `set(FieldPattern)`

- `handler(TypePattern)`

AspectJ – some details

Primitive Pointcuts

- `call(MethodPattern)`

- `call(ConstructorPattern)`

- `execution(MethodPattern)`

- `execution(ConstructorPattern)`

- `get(FieldPattern)`

- `set(FieldPattern)`

- `handler(TypePattern)`

AspectJ – some details

Restricting the Scope

Static Scope:

- `within(TypePattern)`

- `withincode(MethodPattern)`

- `withincode(ConstructorPattern)`

Dynamic Scope:

- `cflow(Pointcut)`

- `cflowbelow(Pointcut)`

AspectJ – some details

Pointcuts with Parameters

One often needs information about the context of a join point.

▶ use pointcuts with *parameters*.

Example:

```
pointcut stateChange(FigureElement figElt) :

  target(figElt) &&

    ( call(void FigureElement.moveBy(int,int)) ||

      call(* Point.set*(*))                    ||

      call(* Line.set*(*)) );


after(FigureElement fe) : stateChange(fe) {...}
```

AspectJ – some details

Parameters in pointcut declaration

```
pointcut stateChange(FigureElement figElt) :

  target(figElt) &&

    ( call(void FigureElement.moveBy(int,int)) ||

      call(* Point.set*(*))                    ||

      call(* Line.set*(*)) );
```

- figElt declared in 'header', together with name.

- bound by the target pointcut

target alone matches any non-static call, field access, etc. if the target type matches the declared parameter type.

AspectJ – some details

Example: Diagonal Moves

Define a pointcut for moves with equal dx and dy.

```
pointcut diagHelp(int dx,int dy) :
call(void FigureElement.moveBy(int,int)) &&
args(dx,dy) &&
if(dx==dy);


pointcut diagMove(int dxy) : diagHelp(dxy,int);
```

AspectJ – some details

About Conditionals

AspectJ specification:

> The boolean expression used can only access static members, variables exposed by the enclosing pointcut or advice, (and thisJoinPoint forms).

But still. . . static methods may be called, which may have side effects!

Question: what are the consequences of this?

AspectJ – some details

Before and After

`Before' advice:
```
before(formal parameters) : Pointcut {
. . . advice body. . .
}
```
The advice body gets executed every time just before the program flow
enters a join point matched by the pointcut.
The formal parameters receive values from the pointcut

`After' advice:
```
after(formal parameters) returning : Pointcut {...}
```
The advice body gets executed every time just after the program flow exits a
join point matched by the pointcutby returning normally.

Capture return value:
```
after(...) returning (int ret): Pointcut {...}
```

AspectJ – some details

What about exceptions?

Advice after throwing an exception:
```
after(formal parameters) throwing : Pointcut {...}
```

Capture thrown exception:
```
after(...) throwing (Exception e): Pointcut {...}
```

Match normal and abrupt return:
```
after(...) : Pointcut {...}
```

AspectJ – some details

Around Advice

Run advice *instead of original code:*

```
Type around(. . . ) : . . . {
. . .
proceed(. . . );
. . .
}
```

- run advice body instead of original call, field access, method body, etc.
- use **proceed** to use the original join point, if needed.

Hint: this may help you to write the timer Aspect asked for earlier

AspectJ – some details

thisJoinPoint

For tracing or logging join points, matched methods might have very different signatures.

$\Rightarrow$difficult to pass all useful information through parameters.

Special variable thisJoinPoint available in advice.

- **thisJoinPoint.toString()**
- **thisJoinPoint.getTarget()**
- **thisJoinPoint.getArgs()**

thisJoinPoint object expensive to construct.

thisJoinPointStaticPart is statically constructed.

AspectJ – some details

Aspects and Classes in AspectJ

Aspects are similar to class declarations, but

- Can also contain pointcut and advice declarations,

- System and not user controls instantiation,
    (An aspect gets instantiated the first time, some advice in it
    needs to be executed.)

- Aspects can extend only abstract aspects,
    (They *can extend classes and implement interfaces)*

- Classes cannot extend aspects.

TO DO: Think about these semantic rules. There are a lot of interesting issues here!

AspectJ – some details

Tracing Aspect (first try)

```
package tracing;

public aspect TraceAllCalls {
pointcut pointsToTrace() :
call(* *.*(..)) ;

before() : pointsToTrace() {
System.err.println("Enter " + thisJoinPoint);
}


after() : pointsToTrace() {
System.err.println("Exit " + thisJoinPoint);

}
}
```

QUESTION: Where is the problem?

AspectJ – some details

Tracing Aspect (first try)

```
package tracing;
public aspect TraceAllCalls {
pointcut pointsToTrace() :
call(* *.*(..)) && !within(TraceAllCalls);

before() : pointsToTrace() {
System.err.println("Enter " + thisJoinPoint);
}


after() : pointsToTrace() {
System.err.println("Exit " + thisJoinPoint);
}
}
```

QUESTION: Why do we need the **within**?

AspectJ – things you should know about

Exceptions in Advice

Advice body may throw exceptions:

```
before() : doingIO() {
openOutputFile();
}
```

```
If openOutputFile() throws
java.io.IOException:
before() throws java.io.IOException :
doingIO() {
openOutputFile();
}
```

=> IOException must be declared/handled
at all places where pointcut applies.

AspectJ – things you should know about

Aspects throwing exceptions

Sometimes, an aspect can change the behaviour of methods,
so that new *checked exceptions are thrown:*

•Add synchronization: InterruptedException
•Execute calls remotely: RemoteException

=>Two possibilities:
•catch and handle exception directly in advice. Might not be appropriate.
•pass exception out of advice. Needs lots of declarations.

AspectJ – things you should know about

Aspect Precedence

What happens if several pieces of advice apply at the same join point?

➠ Assign precedence to aspects to control order of advice execution
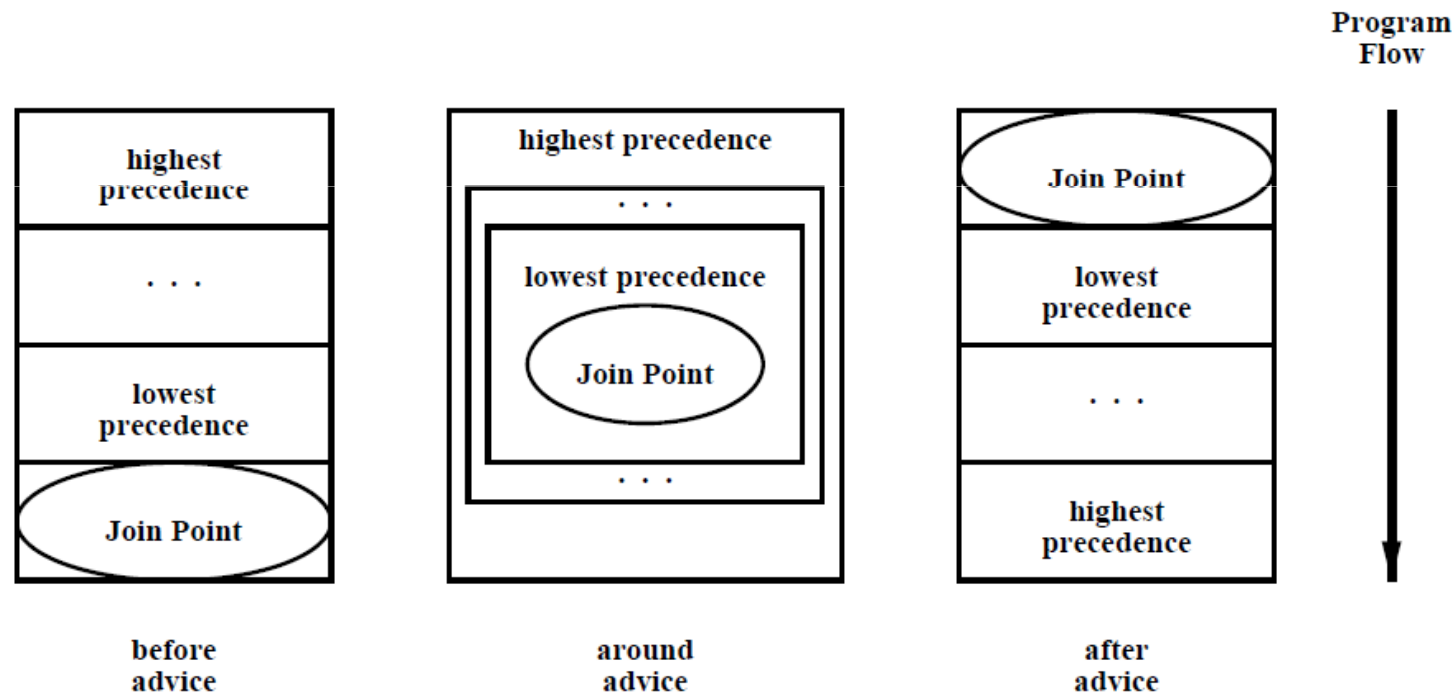
AspectJ – things you should know about

Aspect Precedence (cont.)

Syntax to declare aspect precedence:

```
declare precedence : TypePattern1, TypePattern2, . . . ;
```

May occur in any aspect.
Says that anything matching type pattern 1 has higher precedence
than anything matching type pattern 2, etc.

```
aspect CyclicPrecedence {
declare precedence : AspectA, AspectB;
declare precedence : AspectB, AspectA;
}
```

OK iff aspects share no join points.

AspectJ – things you should know about

Aspect Precedence (cont.)

If not declared, implicit rule for inheritance:

If AspectA extends AspectB, then AspectA has higher priority.

$\Rightarrow$possible to overrule advice from super-aspect.

If still not declared, implicit rule for advice within one aspect:
- If either are after advice, then the one that appears later in the aspect has precedence over the one that appears earlier.
- Otherwise, then the one that appears earlier in the aspect has precedence over the one that appears later.

=> first do something in the same order as they appear in the source

AspectJ – things you should know about

Abstract Aspects

Reminder: aspects can
- extend classes
- extend *abstract aspects*
- implement interfaces

Abstract aspects may contain

- abstract methods, like abstract classes
- abstract pointcut declarations

Abstract aspects are the key to writing *reusable aspects.*

AspectJ – things you should know about

Aspect Instantiation

At runtime, aspects have fields like objects.

When do they get instantiated? Usually:

    Instantiate an aspect once per program execution.

```
aspect Id {...}
aspect Id issingleton {...}
```

Implemented as singleton => static field in aspect class.

NOTE: Things are actually much more complicated when we consider all the different ways in which Java objects (including Aspects) are instantiated

AspectJ – things you should know about

Privileged Aspects

Usually, advice code has no access to private members of advised classes.

(Note that matching in pointcuts *does see private members)*

But the privileged keyword can help:

```
privileged public aspect A {
before(MyFriend f) : this(f) && ... {
System.out.println("My best friends secret: " +
f._privateField);
}
}
```

AspectJ – profiling problem

Return to the profiling problem we looked at in previous class

TO DO: Implement 3 or more interesting profiling aspects and test on different programs

Practical Work For Next Class: Invariant Testing

Write an aspect that tests the invariant of a class everytime a method is executed, and writes to an invariant log file whether the test passes or fails.