SPA 2005
AspectJ Tutorial

Dan Haywood

Eoin Woods

Introductions

- Dan Haywood
  - Haywood Associates
  - dan@haywood-associates.co.uk
- Eóin Woods
  - Zuhlke Engineering Ltd
  - ewo@zuhlke.com

Objectives

- Introduce AOP concepts
- Show how AOP concepts are implemented in Java using AspectJ
- Provide practical exercises to allow learning via experimentation
- Start thinking about applicability and viability

Caveats

- The material is all AspectJ specific
  - Other AOP systems may vary
- This is an introductory workshop
  - Some advanced features will be omitted
  - More sophisticated options exist for examples
- We're pretty new to this ourselves
  - Learning together
  - Don't know all the answers

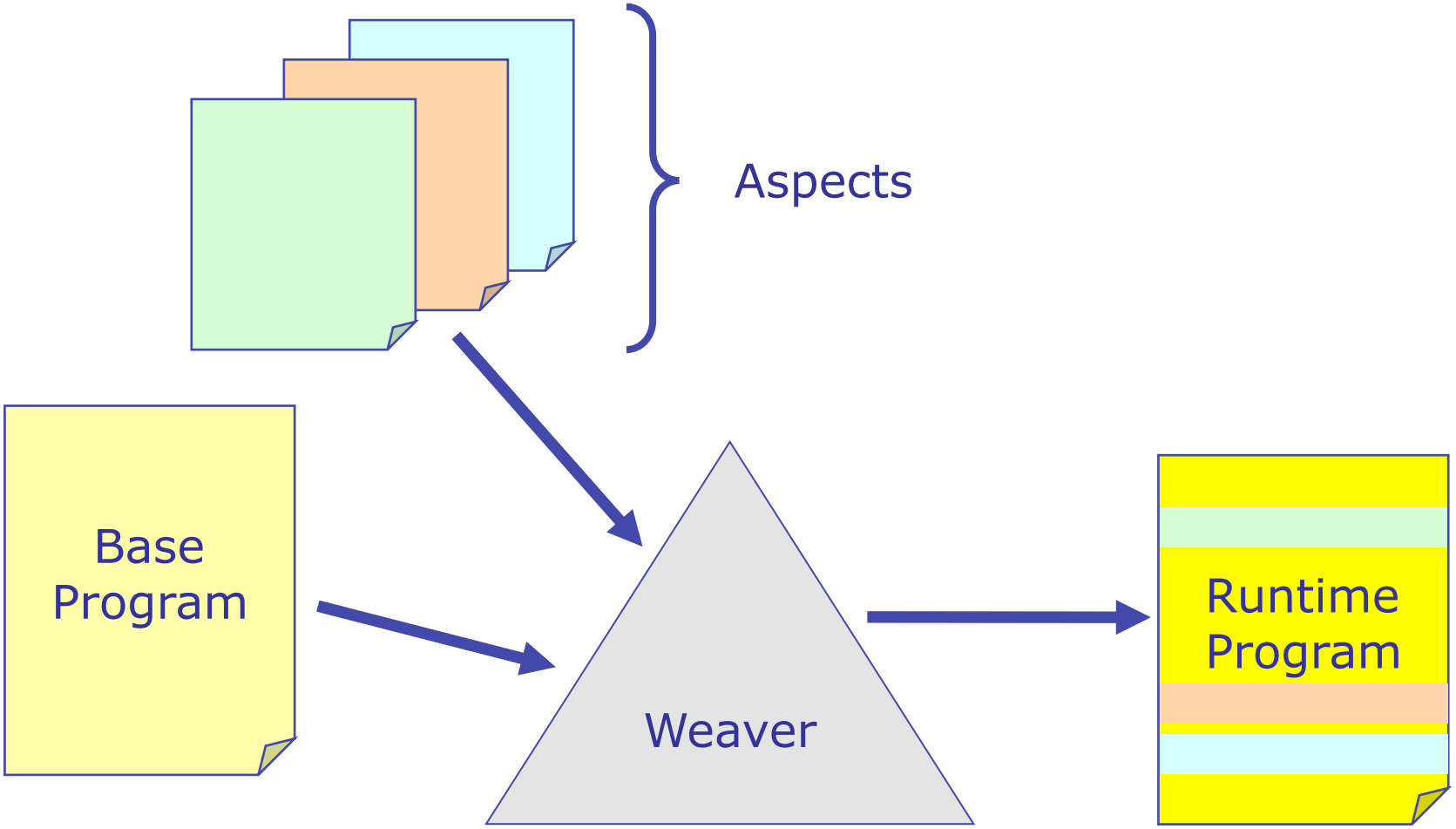
AOP Overview

- Aspect Oriented Programming
  - Xerox PARC, late 1990s (Lopes & Kiczales et al)
- Addresses complexity of cross cutting concerns
  - Security, logging, coordination, availability, …
- Handling many concerns simultaneously makes code complex and causes errors

AOP Overview

- AOP is a novel approach to *program composition*
- Start with a "*basic functionality program*"
  - i.e. the "business logic"
- Develop different cross-cutting "aspects" of the program separately
  - allowing focus on each independently
- Automatically "weave" the aspects together with the basic functionality program to form a complete program
  - runtime program = business logic + aspects

AOP Overview
Weaving



Aspects

- The basic functionality program …
  - … is just a plain OO program in a supported language
- The aspects
  - Similar to classes, but can't be used in isolation
    - aspects can inherit from each other
    - cannot instantiate directly
    - (compiled to classes behind the scenes)
  - Contain code implementing the functionality required
    - Call a logging library in a logging aspect
    - Perform security checks in a security aspect
  - Contain a specification of where to apply the code to the base program
    - Before public methods? After private methods? …

AOP Overview
Jargon

- AOP comes with its own jargon ...

- Join Point
  - a location in a base program that an aspect could be applied to
- Pointcut
  - the specification of a set of join points
- Advice
  - the code to apply to a join point
- Aspect
  - pointcut + advice
- Weaving
  - the process of applying aspects to base program

Join Points

- Locations in the base program
- Most commonly used:
  - Call to/execution of a method
  - Call to/execution of a constructor
  - Accessing or mutating a field
  - Execution of an exception handler
- A few others exist too:
  - Execution of a static initialisation block
  - Object initialisation, object pre-initialisation, advice execution

Join Point Examples

```
public class ExampleClass
{
      static int staticValue ;
      static { ExampleClass.staticValue = 1; }

      private String strVal
      private int intVal ;

      public ExampleClass(int i)
      {
            this.intVal = i ;
            this.strVal = Integer.toString(i) ;
      }
... continued ...
```

static field assignment

static initialiser

execution of constructor

member field assignment

member field assignment

call to (static) method

Join Point Examples (ctd)

```
... continued ...

     public int getValue()
      {
            return this.intValue ;
      }

      public int getRandomLarger()
      {
            Random r = new Random() ;
            return this.intValue + r.nextInt();
      }
}
```

method execution

member field access

method execution

call to constructor

member field access

call to method

Pointcuts

- A pointcut represents a place(s) in the base program where something of interest occurs
  - that is, a pointcut captures a set of join points
- For example:

```
pointcut fbpubmethods() :
        call(public void com.foobar..*(..)) ;
```

Wildcard

Translation:

  - *"all public methods with void return type, in classes in the com.foobar package hierarchy"*
- Specify everything that matters …
  - … wildcard everything that doesn't

Defining a Pointcut

- Pointcuts can be categorized into
  - *kinded* pointcuts
    - based on the kind of joinpoint that's being captured
  - *context collecting* pointcuts
    - obtain information about the joinpoint
      - information available depends on the kind
    - sometimes called execution object pointcuts
  - *control flow* pointcuts
    - restrict to specific thread of control
  - *lexical structure* pointcuts
    - restrict to specific class (.java file)

Defining a Pointcut

Kinded Pointcuts

- Kinded pointcut definitions consist of:
  - The kind of pointcut
  - Signature of code to match
- Kind indicates the nature of the join point being captured
  - Methods/Constructors: “call” and “execution”
  - Fields: “get” and “set”
  - Exception handlers: “handler”
- Signature indicates the places in the code
  - uses wildcards with Java syntax
  - use annotations
  - signature varies with kind

Pointcut Examples

- Some more examples using AspectJ syntax:
  - Writing to any public integer field in class Foo:

```
set(public int com.foobar.Foo.*)
```

  - Calling any private method, in any class, with "key" in its name, returning String

```
call(private String *.*key*(..))
```

  - The execution of a NullPointerException handler

```
handler(NullPointerException)
```

Defining a Pointcut

Context Collection Pointcuts

- Used most often with *call* or *execution* joinpoints
- Restrict the scope based on
  - the type of the object calling the method
    - this(...)
    - n/a for execution pointcut
  - the type of the object on which the method is being called
    - target(...) if using "call"
    - this(...) if using "execution"
  - the type of the arguments
    - args(...)
- Extended syntax allows *this*, *target* or *args* to be named as parameters
  - applied advice can then use these parameters, potentially modifying them
- Also called *Execution Object* pointcuts

Defining a Pointcut

Control Flow & Lexical Structure

- *Control flow* pointcuts define a scope based on execution path
  - e.g. "all join points executed as part of executing a public method in the Foo class that returns an integer"
  - *cflow(…)* and *cflowbelow(…)*
- *Lexical structure* pointcuts define a scope based on the structure of the code
  - e.g. "all join points within the com.foo classes"
  - e.g. "all join points within this class"
  - e.g. "all join points within this aspect"
    - aspects are compiled down to classes
  - *within(…)* and *withincode(…)*

Defining an Aspect

- Name
- Pointcuts
  - places of interest in the base program
- Advice
  - what to do at a pointcut
  - c.f. methods in a class
- Declarations
  - instance variables
    - represent the state of the aspect instance
  - local methods
    - functional decomposition, just as in classes
- Modifiers
  - public/private
  - privileged
    - can manipulate private fields in base program
- Inheritance
  - of abstract aspect
- In AspectJ, aspects:
  - reside in an .aj file
  - compiled into a .class file
  - can be packaged into libraries

Defining an Aspect
Advice

- *Advice* (in an aspect) is the code that will get woven into the base program
- Definition consists of three parts:
  - Which pointcut to apply it to
  - How to apply it to the pointcut
    - before / after / around
  - The code to weave into the join point

c.f. method signature

c.f. method implementation

- Can be parameterized
  - Context collection pointcuts provide the actual values for the parameters

Defining an Aspect
Advice Types

- *Before Advice* is executed before the join point
- *After Advice* is executed after the join point
  - can optionally be scoped based on result
    - after throwing, after returning
- *Around Advice* provides total control
  - allows code before and after the join point and the execution of the join point to be controlled
  - use `proceed(…)` pseudo method call to execute the captured join point
  - return value is same type as the join point being executed

Defining an Aspect
Aspect Example

- Prints names of methods called by code within com.foobar package or subpackage

```
public aspect SimpleAspect {

  pointcut allFooBarCalls() :
           call(* *.*(..)) &&
                cflowbelow(call(* com.foobar..*.*(..))) ;

  before() : allFooBarCalls()
  {
    System.out.println("Entering: " +
            thisJoinPoint.getSignature().getName());
  }
}
```

one or more pointcuts

control flow

one or more pieces of advice

c.f. this

c.f. Reflection API

Defining an Aspect
Anonymous Pointcuts

- Pointcuts can be named
  - as shown above
- Pointcuts can also be introduced implicitly
  - that is, without a name:

```
public aspect SimpleAnonPointCutAspect {
  // Apply this advice to all public methods in FooBar
  before() : call(public * FooBar.*(..))()
  {
      //...
  }
}
```

- We prefer to use names
  - it describes the semantics of the pointcut to the reader

Simple AOP Examples

- Some possibilities with the features already introduced:
  - Logging
  - Tracing
  - Pooling

Simple AOP Examples

Logging

```
// Log a message before the execution of any public method or
// constructor in the class "MyClass"
public aspect LogAspect {

   pointcut logPoint() : !within(LogAspect) &&
        (execution(public * MyClass.*(..)) ||
        execution(public MyClass.new(..)));

   before() : logPoint() {
      System.out.println(thisJoinPoint.getTarget() + ', ' +
                         thisJoinPoint.getThis() + ', " +
                         thisJoinPoint.getSignature());
   }
}
```

Could log parameter values too … see later

Simple AOP Examples

Logging

```
// How the class would look after weaving
public class MyClass {

   //...
   public MyClass(int value) {
      System.out.println(...) ;
      this.internalInit(value) ;
   }
   // ...
   private void internalInit(int value) {
      // ... no woven code as no joinpoint ...
      this.value = ... // some change to "value" parameter
   }
   // ...
   public int getValue() {
      System.out.println(...) ;
      return this.value ;
   }
}
```

new woven code

new woven code

Simple AOP Examples

Tracing

```
// Log a message before and after any method call, indenting
// the messages as the call stack grows
public aspect TraceAspect
{
   pointcut tracePt(): !within(Tracer) &&
                                execution(* *.*(..));

   before() : tracePt()
   {
        logEntry(thisJoinPoint) ; // Call internal method
   }
   after() : tracePt()
   {
      logExit(thisJoinPoint) ; // Call internal method
   }
   // Map of current call stack depths by thread
   private static HashMap depths = new HashMap() ;

... continued ...
```

Simple AOP Examples

Tracing (ctd)

```
... continued ...

   // Local methods to log entry and exit
   private void logEntry(JoinPoint jp) {
        Integer d = (Integer)depths.get(Thread.currentThread()) ;
        if (d == null)
            d = new Integer(0);
       else
            d = new Integer(d.intValue()+1) ;
       depths.put(Thread.currentThread(), d) ;
        printSpaces(d.intValue()) ; // print prefix spaces
        System.out.println(jp.getSignature()) ;
   }
   private void logExit(JoinPoint jp) {
        // Print suitable msg & decrement depth value for thread
   }
   private void printSpaces(int depth) {
       // print "depth" spaces to System.out
   } ;
}
```

Simple AOP Examples

Pooling

```
// Aspect to wrap around standard JDBC connection handling
// and use a ConnectionPool class to manage connection objects
public aspect PoolConnectionsAspect
{
   pointcut poolGet() :
        call(static Connection DriverManager.getConnection(..));
   pointcut poolPut() : call(void Connection.close()) ;

   Connection around() : poolGet()
   {
      return ConnectionPool.nextConnection() ;
   }
   void around() : poolPut()
   {
      ConnectionPool.return(thisJoinPoint.getTarget()) ;
   }
}
```

Installation of Software

- CD with a single ZIP and PDF
- Extract ZIP to C:\ (or equiv)

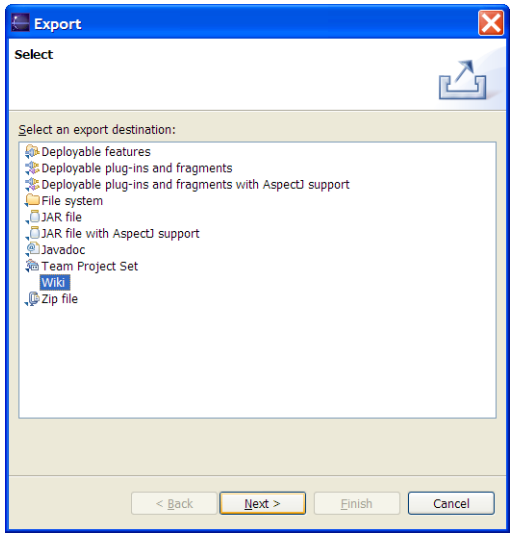
```
eclipse\
    Eclipse 3.1 SDK
    extensions\
      AJDT\
      GEF, EMF, SDO, XSD, VEP\
      Wiki\
      Spring IDE\
      Oxygen XML\
    workspace\
      exercises + (some) solutions
    Installation.PDF
```

- Remaining installation described in Installation.PDF
  - 2 minutes for extensions
  - 5 minutes for Ant support

- The extensions
  - AJDT is development tools for Eclipse + AspectJ itself
  - GEF et al. provide a plugin infrastructure, used by some advanced plugins
  - Spring IDE understands semantics of Spring XML configuration files
  - Oxygen provides DTD-aware editing of Spring XML configuration files
  - Wiki
    - described next page

A note about the Exercises

- Each exercise is in a separate Eclipse project
  - some projects also have solutions (suffixed "Solution")
  - up on Sourceforge under CVS
    http://sourceforge.net/projects/aspectjworkshop/
- The exercises to complete are described in .wiki files, in a wiki/ subfolder
  - the Wiki extension allows this to be edited (use twiki syntax)
  - you can edit as you wish and make your own notes
  - Can export entire project using File → Export
- There is also a separate "Wiki" project
  - general notes and reference
  - feel free to expand



Exercises

- Install software
- Review the three examples presented previously, and get working in Eclipse
  - the aspects have already been coded for you
  - each aspect is in its own project
  - open up appropriate project in the Eclipse workspace, and follow the notes in *wiki/HomePage.wiki*
- The Eclipse projects are
  - Logging
  - Tracing
  - Pooling
- Don't forget the reference material in the "Wiki" project

Further Exercises

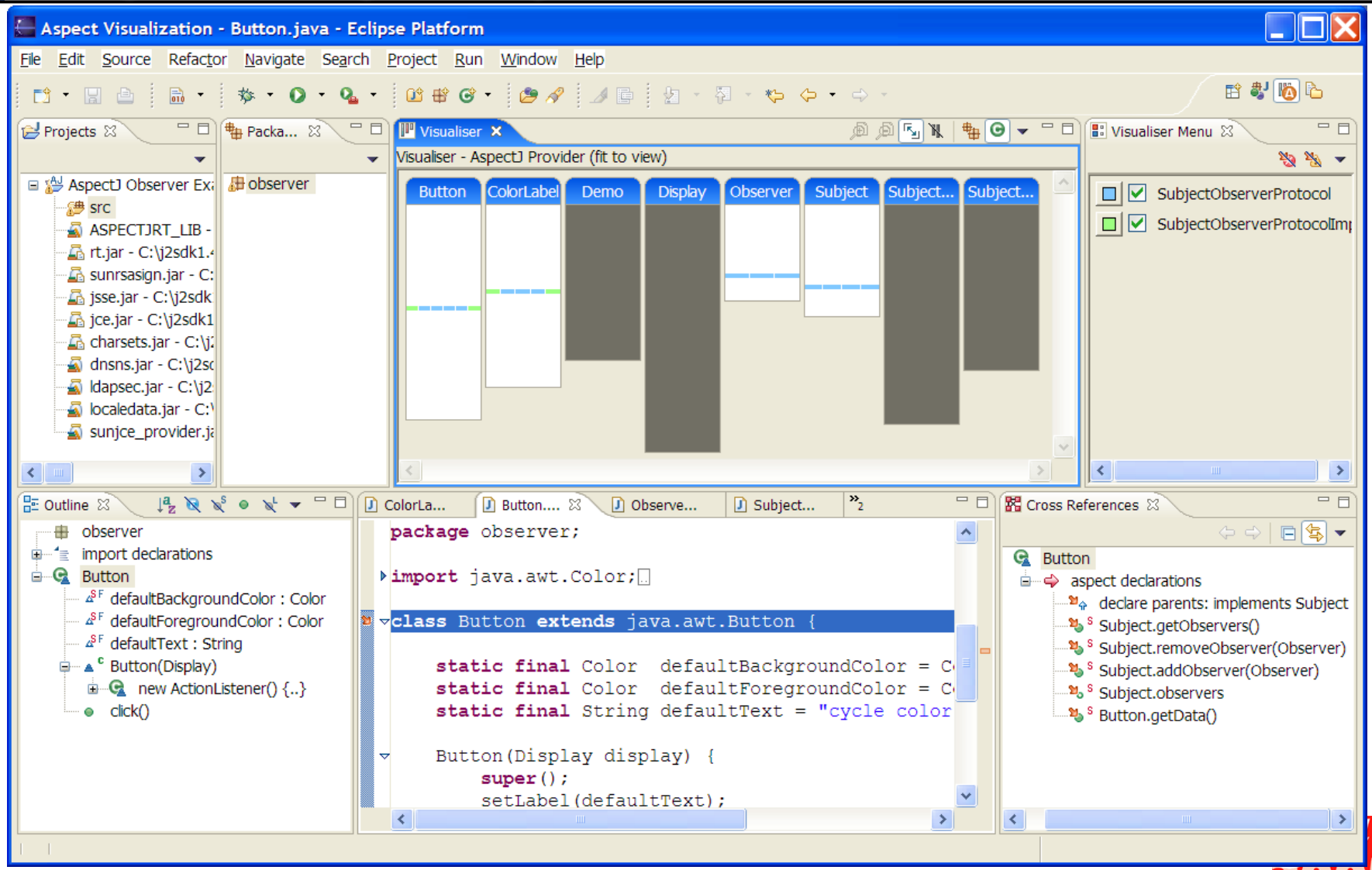
- Modify the logging example so that logging is performed whenever an exception handler fires
  - look at the handler join point.
- Modify the tracing example to factor out indenting into a separate aspect.
- Modify the tracing example to perform profiling
  - Use a pointcut to capture all method invocations outside the aspect itself
  - In the *before()* advice
    - obtain the method signature, capture the time, store in a hash
      - the hash will be keyed on signature, and could contain a list of pairs of before/after times
  - In the *after()* advice
    - obtain the method signature, work out time taken, update hash
  - Provide a method to dump / analyze the results of hash
  - Is your aspect threadsafe?

Eclipse AJDT Project

- AspectJ Development Tools
  - AspectJ weaver integrated with Java compiler
  - Enhancements to Java debugger, aware
- Can use Java editor
  - turns off "spellchecker"
  - aspects stored in .aj files
  - markers
- Provides own builder
- AspectJ perspective + views
  - Visualizer
  - Cross-referencing view
  - Outline for Aspects
  - Problems view (AspectJ compile errors)
- Defines AspectJ environmental settings
- A few limitations, but more than capable
  - e.g. code completion does not yet understand introductions

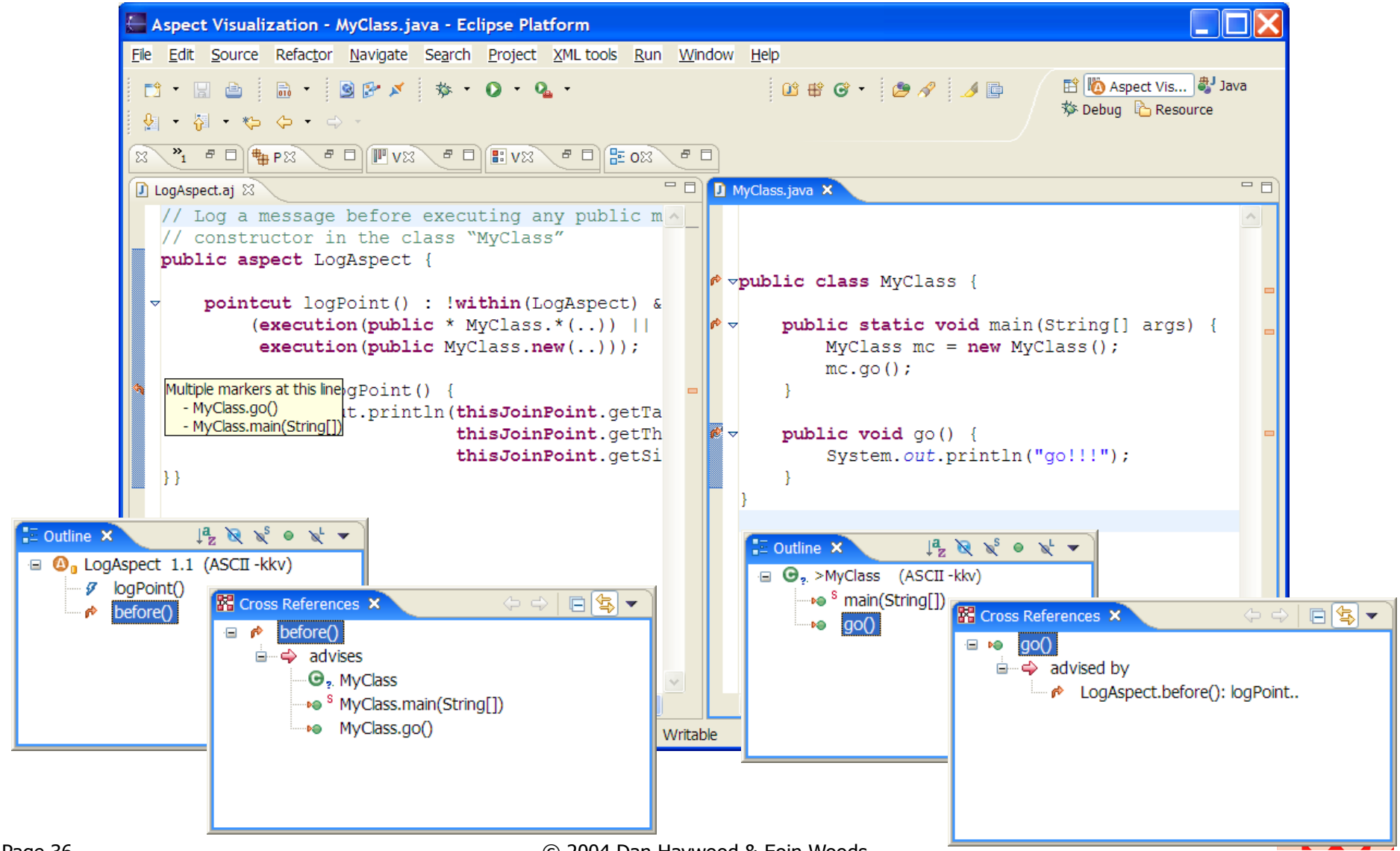
AJDT Features

AspectJ perspective



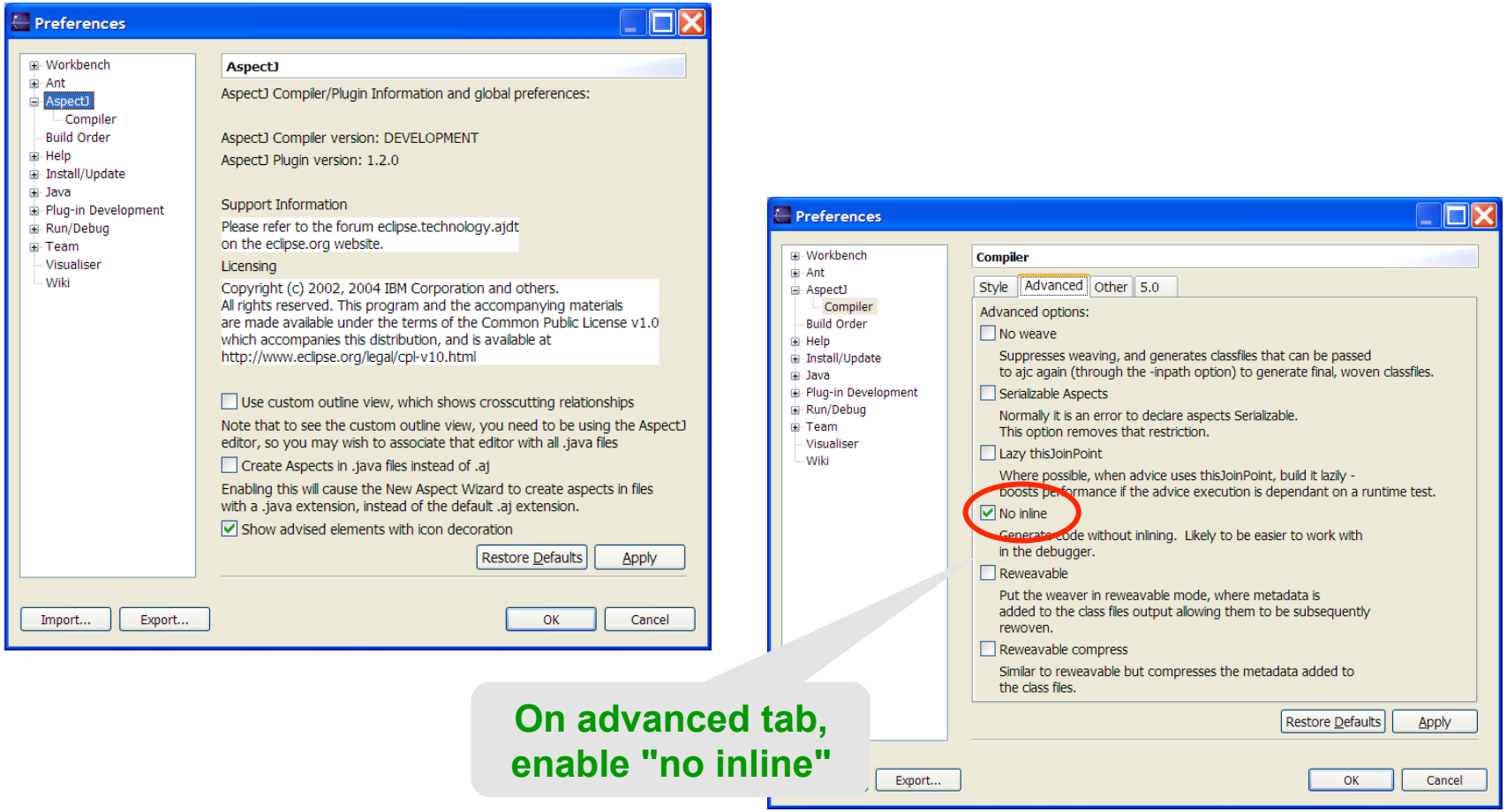
AJDT Features

Outline & Cross-References



AJDT Features

Preferences

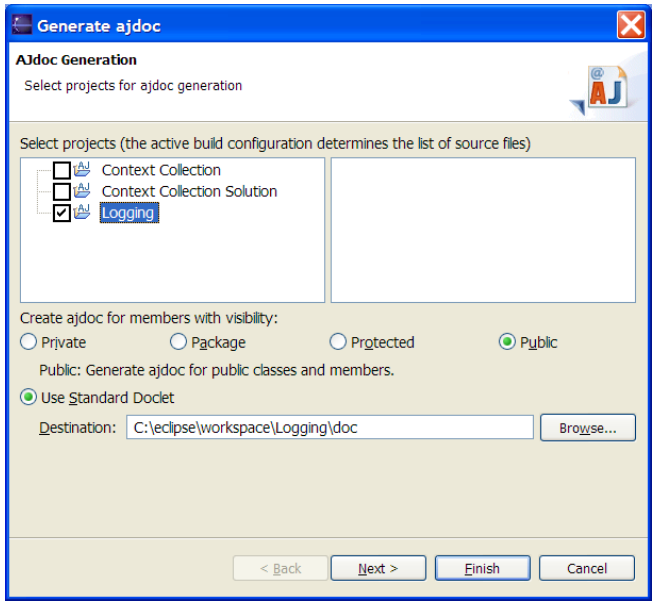


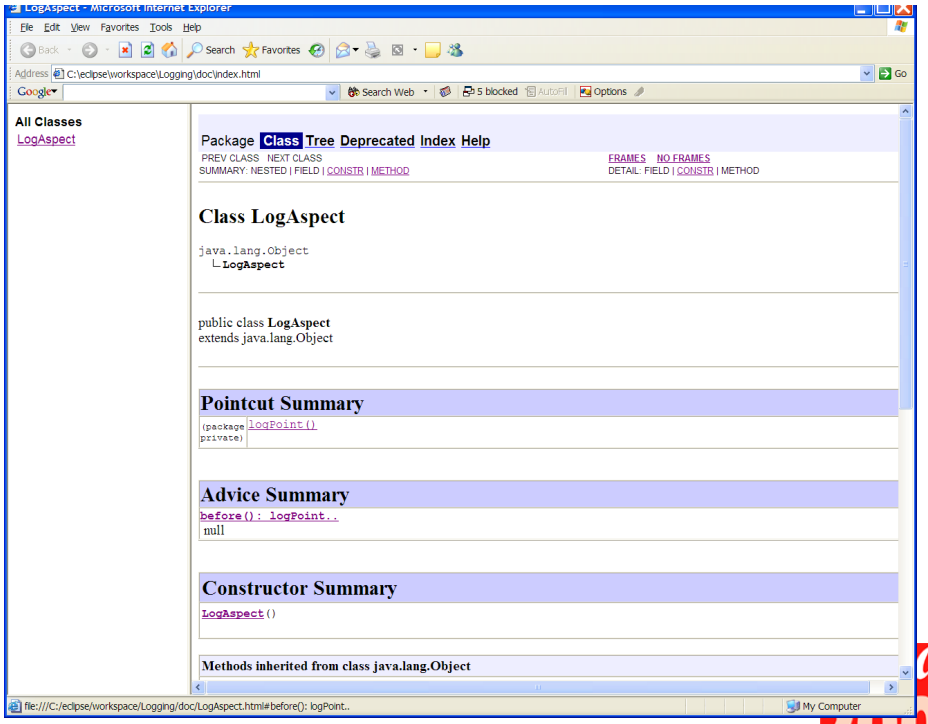
- "No inline" enables breakpoints in around() advices

AJDT Features

ajdoc

- Creates Javadoc-like HTML output
  - extended to document aspects
- Project → Generate ajdoc





Exercise

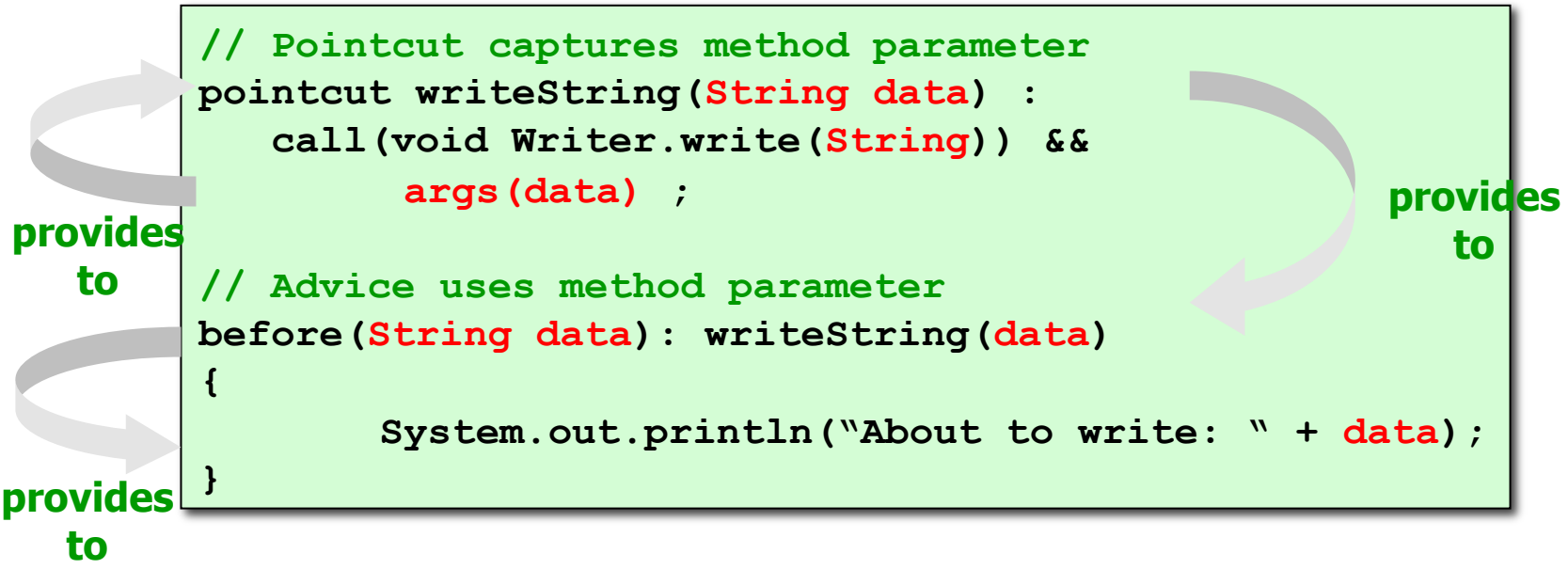
- Explore the relationship between the outline and cross-references views
- Use the debugger to step into advice
- Run ajdoc
- Optional: view the AJDT Demos
  - http://www.aspectprogrammer.org/ajdt/demos/

Context Collection

- Aspects developed so far have a major limitation
  - no access to the runtime context
- Useful aspects often need access to
  - Parameter values
  - Current object & object being invoked
  - Ability to modify method return values
- AspectJ provides all of these facilities
  - *args()* pointcuts
  - *this()* and *target()* pointcuts
  - *around()* advice
- Typically used in conjunction with *call()* or *execution()*
  - can be used with elsewhere, eg *handler()*

Capturing parameter values

- The **`args()`** pointcut captures and names the actual values in the executing join point
  - For example:



Context Collection

Multiple Parameter values

- The **`args()`** pointcut *must* match the signature of method
  - thus all parameters are available
  - if don't need an argument value, just specify the type and don't name

```
// Pointcut captures first two parameters of
// StringBuffer.append(char[] chr, int offset, int len)
pointcut appendChars(char[] data, int start) :
   call(void StringBuffer.append(char[],int,int)) &&
        args(data, start, int)
```

without this, the pointcut won't match the method.

Context Collection

Capturing the target

- Capturing target and parameter values
  - use `target()` in the pointcut definition ***or***
  - use `thisJoinPoint.getTarget()` in the advice

```
// Pointcut captures method parameter & target
pointcut writeString(Writer w, String data) :
    call(void Writer.write(String)) &&
         args(data) && target(w);

// Advice uses method parameter and target
before(Writer w, String data): writeString(w, data)
{
    System.out.println(
      "About to write: " + data + " to " + w) ;
}
```

Context Collection

Capturing the current object

- Capturing the current object
  - when combined with call(), it is the calling object
  - when combined with execution(), it is the called object
- Syntax
  - use `this()` in the pointcut definition ***or***
  - use `thisJoinPoint.getThis()` in the advice

```
// Pointcut captures calling object
pointcut setPassword(Object o) :
    call(void SecMgr.setPassword(String, String)) &&
         this(o);

// Advice uses calling object
before(Object o) : setPassword(o) {
  assert o == thisJoinPoint.getThis() ;
       System.out.println("Password set from object " + o);
}
```

Context Collection

Conditional execution

- Which (OO) design pattern is being implemented here?

```
// Pointcut captures calls to retrieve an object
pointcut obtainConnectionPool():
 call(ConnectionPool.new(..));

// Advice checks if Pool needs to be created or not
ConnectionPool around(): obtainConnectionPool() {
  if existingInstance == null {
     existingInstance = proceed() ;
  }
  return existingInstance ;
}
```

```
// get hold of our pool…
ConnectionPool myPool = new ConnectionPool();
// … and use
Connection connection = pool.getConnection();
```

Context Collection
Examples

- Examples showing the use of context collection:
  - Caching
  - Buffering
  - Encode & Decode, Encryption
- Many of these use context collection point cuts with around() advice – a powerful combination
  - *around()* advice can capture and change parameters
  - *around()* advice can capture and change return values
  - *around()* advice can even decide whether or not to perform the current pointcut

Context Collection Examples
Caching

- Use around(...) with target(...) and args(...)
  - Caching and reusing "Holiday Calendar" objects

```
public aspect CachingAspect {
  pointcut getHolidays(Country c, String y) :
    call(HolidayCalendar Country.getHolidays(String)) &&
    args(y) && target(c) ;

  private HashMap holidayCache = new HashMap() ;

  HolidayCalendar around(Country c, String y): getHolidays(c, y) {
    String key = c + ":" + y ;
    HolidayCalendar cal = (HolidayCalendar)holidayCache.get(key) ;
    if (cal == null) {
      cal = proceed(c, y) ;
      holidayCache.put(key, cal) ;
    }
    return cal ;
  }
}
```

Context Collection Examples

Buffering

- Use around(...) with target(...) and args(...)

```
public aspect BufferingAspect {
  pointcut fileWrite(FileWriter f, char[] data):
    execute(void FileWriter.write(char[])) && target(f) && args(data);
  pointcut fileClose(FileWriter f):
    execute(void FileWriter.close()) && target(f) ;

  private static ThreadBuffer buf = new ThreadBuffer() ;

  void around(FileWriter f, char[] data): fileWrite(f, data) {
    buf.append(f, data) ;
    if (buf.isFull(f)) {
      proceed(buf.getData(f)) ;
      buf.clear(f) ;
    }
  }
  void before(FileWriter f) throws IOException: fileClose(f) {
    f.write(buf.getData(f)) ;
     buf.remove(f) ;
  }
}
```

ThreadBuffer is a ThreadLocal array of chars

Context Collection Examples

Base64 Encoding

- Use around() with args()

```
// Pointcut captures method parameter
pointcut dataWrite(char[] data) :
       call(void Connection.send(char[])) && args(data);

// Advice encodes parameter before calling joinpoint
// Throws runtime exception if encoding fails
void around(char[] data): dataWrite(data)
   throws IllegalArgumentException {
  String encoded = Base64.encode(data) ;
  if (encoded == null)
     throw new IllegalArgumentException("...") ;
      proceed(encoded.toCharArray()) ;
}
```

encode data

Context Collection Examples

Base 64 Decoding

- Use around() to change result

```
// Pointcut captures calls to receive data
pointcut dataRead(): call(char[] Connection.receive());

// Advice decodes return value before proceeding
// Throws runtime exception if decoding fails
char[] around(): dataRead() throws IllegalStateException {
  char[] ret = proceed() ;
       String decoded = Base64.decode(ret);
  if (decoded == null)
     throw new IllegalStateException("...") ;
       return decoded ;
}
```

decode data

Context Collection Examples

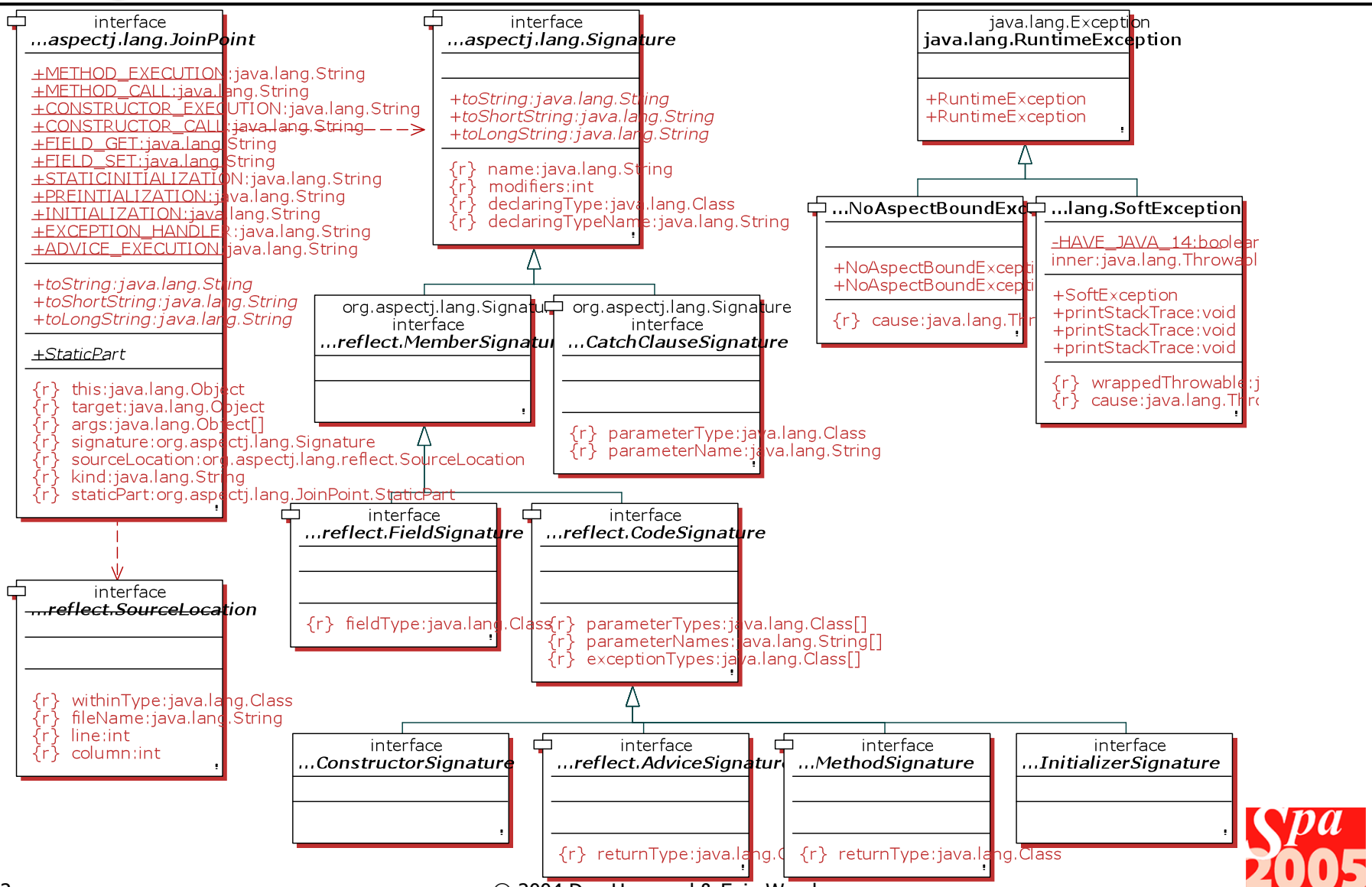
Encryption

- Use around(...) with args(...)

```
public aspect PasswordControlAspect {
  pointcut passwdChange(String pass) :
    call(void SecMgr.setPassword(String)) && args(pass) ;

  void around(String pass)
       throws IllegalArgumentException: passwdChange(pass) {
     proceed(CryptoMgr.oneWayEncrypt(pass));
   }
}
```

AspectJ Runtime API



Exercise

- Write a cache for
  - factorials
  - calculating squares
  - is there any repetition – if so, what might that mean?
- If code is run outside the non EventQueue thread, then run it transparently within the non EventQueue
  - hint: the pointcut *if (!EventQueue.isDispatchThread())* should help.
- Exploring
  - Show the equivalence of context collecting
    - this(..) and ThisJointPoint.getThis()
    - target(..) and ThisJointPoint.getTarget()
  - Show what happens when use this(…) or thisJoinPoint.getThis() to match a static method.
  - If apply an around and an after, which has precedence?

Introductions

- Pointcuts used thus far are dynamic
  - evaluated at runtime & intercept program execution
- Pointcuts can also be static
  - make changes to the static structure of the code
- Common static pointcuts include
  - introduce a new field to a class
  - introduce a new method to a class
  - change a class' inheritance relationships
    - typically to implement an interface
  - soften an exception

look at these in this next segment

- These modifications are often combined
  - a class can implementing an interface, and introduce its implementation

Introductions

Introducing a New Member

- Add a qualified member to the aspect

```
// Introduce a private field to MyClass
private int MyClass.pubCallCount = 0 ;

// Introduce a public method to MyClass
public int MyClass.getCallCounter() {
  return this.pubCallCount ;
}

// Use it from the related advice
before() : somePointcut() {
   MyClass c = (MyClass)thisPointCut.getTarget();
   c.pubCallCount++ ;
}
```

```
// Use from a regular class
MyClass c = new MyClass() ;
// call public methods on c ...
int count = c.getCallCounter();
```

A Java IDE wouldn't know this is valid (but AJDT should...)

Altering the Class Hierarchy

- Implement an interface
  - Make a class Comparable

```
public aspect ComparableAspect {

  declare parents MyClass:
    implements Comparable;

  public int
    MyClass.compareTo(Object o) {
    MyClass other = (MyClass)o;
    return other.getKey().
            compareTo(this.getKey());
  }
}
```

Assuming we can compare using MyClass.getKey()

- Define a new superclass
  - Make a Bundle into a Collection

Should be subclass of original superclass

```
public aspect MakeCollection

  declare parents Bundle:
    extends AbstractCollection;

  public Iterator Bundle.iterator() {
    // Call existing Bundle methods
    // to create an iterator
  }

  public int Bundle.size() {
    // Call existing Bundle methods
    // to work out its size
  }
}
```

Adverbs & Adjectives

- Oversimplifying somewhat ...
  - classes are nouns
  - methods are verbs
- ... and aspects are adverbs / adjectives
- For example
  - a secure Transaction
  - a persistent Customer
  - a trackable Order
- The *Director* AO pattern gives us a way to apply this
  - we'll cover this in a slide or two

Introductions

Adjectives are like Mixins

```
public class Order {
  private Money total;
  public Money getTotal() {
    return total;
  }
}
```

Here is the noun

```
public aspect TrackabilityAspect {
  public interface Trackable {};
  private int Trackable.trackingId;
  public int Trackable.getTrackingId() {
    return trackingId;
  }
}
```

Here is the adjective

Here's what binds 'em together

```
public aspect TrackableAccountAspect {
  declare parents :
    Order implements TrackabilityAspect.Trackable;
}
```

Director AO Pattern

- We can formalize the concept of using mixins
- The *Director* aspect
  - defines roles
  - describes interactions between participants solely in terms of those roles

Also called the Participant pattern

- A separate aspect (or sub-aspect) binds the roles to concrete classes
  - Director aspect will be abstract if some details of the implementation are specific to the binding.
- The Director aspect represents an implementation of a particular (OO) design pattern
  - e.g. publish/subscribe

Director AO Pattern

A quick aside: Abstract Aspects

- Aspects can extend abstract aspects
  - not from concrete aspects
- Abstract aspects can have
  - abstract pointcuts
  - abstract methods
- Abstract pointcuts allow the subaspect to identify the particular joinpoints to pick out
- AspectJ only applies advice from concrete aspects

```
public abstract aspect NavelGazerAspect {
  abstract pointcut changeTo(Navel n);
  after(Navel n): changeTo(Navel n) {
    aNavelHasChanged(n);
  }
  protected abstract void aNavelHasChanged(Navel n);
}
```

```
public aspect MyNavelGazerAspect extends NavelGazerAspect {
  pointcut changeTo(Navel n):
    call(* Navel.set*(..)) && target(n);
  protected void aNavelHasChanged(Navel n) { … }
}
```

Director AO Pattern

Example

```
public abstract aspect ObserverPattern {
  protected interface Model {
    void addListener(Listener l);
    void removeListener(Listener l); }
  protected interface Listener {
    void modelChanged(Model m);      }

  private List Model.listeners = new LinkedList();
  public void Model.addListener(Listener l) { … }
  public void Model.removeListener(Listener l) { … }
  private void Model.notifyListeners() { … }
  after(Model m): modelChange(m) {m.notifyListeners(); }

  protected abstract pointcut modelChange(Model m);
}
```

Define roles

Describe interactions

Delegate downwards

```
public aspect CustomerListener extends ObserverPattern {
  declare parents: Customer implements Model;
  protected pointcut modelChange(Model m): {
    call(* Customer.set*(..)) & target(m)

  declare parents: CustomerView implements Listener;
  public void CustomerView.modelChanged(Model m) { … }
}
```

Bind model

Bind listener

Exercise

- Review & run the AspectJ Introductions Example
  - Comparable, "Hashable", Cloneable
- Refactor the ComparablePoint aspect
  - split into an abstract aspect that defines the comparison, and concrete sub-aspect that binds to the Point class
- Refactor the HashablePoint aspect
  - define a Hashable interface in an abstract aspect
  - make aspect abstract and be defined in terms of Hashable
  - create concrete sub-aspect that binds to the Point class
- Reuse your abstract aspects in a different context

Further Exercises

- Generalize your previous caching aspect
  - define a Cacheable interface
  - encapsulate the caching logic within an abstract aspect
  - rewrite your solutions as concrete subaspects
- Explore and extend some pre-written aspects that use the Director pattern
  - Persistence, using Hibernate
  - Transactions
    - undo/redo in memory
  - Publish changes onto a bus
  - Optimistic locking
  - Security, using JAAS

Exception Handling

- An aspect might cause an exception to be thrown
  - Such an exception cannot be propagated
- Two solutions
  - simple one is AspectJ's support for Soft exceptions
    - wrap the aspect's exception in a runtime exception
  - more involved is a design pattern
    - expose aspect exception in terms understood by client
- As a by-the-by, soft exceptions can also be applied to regular Java classes
  - banish checked exceptions forever???

Softening an Exception

- Allows a checked exception to be treated as unchecked
- Applies at specified join points
  - Checked exception wrapped in a `SoftException`

```
// Declare soft exception for StringWriter.close()
declare soft: java.io.IOException:
    call(* java.io.StringWriter.close());
```

  - If the softened exception does occur:

```
Exception in thread "main" org.aspectj.lang.SoftException
    at TestClass.main(TestClass.java:24)
```

Send Email on Exception

- A more realistic example: mail on exception

```
public aspect MailExceptionsAspect {
  public interface MailMyExceptions {}

  declare soft: javax.mail.MessagingException:
    within(MailExceptionsAspect) && call(* javax.mail.*(..));

  pointcut captureExceptions(Exception ex) :
    within(MailMyExceptions+) && handler(Exception) && args(ex);

  after(Exception ex): captureExceptions(ex) {
    sendMail(ex);
  }
  private void sendMail(Exception ex) {
    // calls to javax.mail that might throw an exception
  }
}
```

```
public aspect ApplyMail {
  declare parents:
    SomeFlakyClass implements MailMyExceptions;
}
```

We cover within(…) in more detail soon

No more Checked Exceptions?

- Soften InterruptedException

```
public aspect SoftenSleepInterruptAspect {
   declare soft: InterruptedException :
       call(void Thread.sleep(..));
}
```

No need to wrap in try/catch

```
public class Sleeper {
  public static void main(String[] args) {
    System.out.println("Sleeping ...") ;
    Thread.sleep(1000);
    System.out.println("    ... Woken") ;
  }
}
```

Exercise

- Adapt a logging aspect to write exceptions to a file
  - use exception softening to hide any IO errors
- Apply exception softening to hide HibernateExceptions
- Investigate the Exception Introduction pattern
  - provides a way of wrapping AspectJ (system) exceptions within domain (business) exceptions
  - AspectJ in Action, chapter 8

Aspects for Constraints

- Aspects so far have *changed* behaviour
- Aspects can also *enforce* constraints
  - Code structure (e.g. layering)
  - Deprecated / dangerous methods
  - Check context for certain operations
- Constraints enforced at compile time
  - Behave like a compiler extension
  - Can't use runtime context (*target()*, *args()*, *if(…)* etc.)

Constraints

Errors and Warnings

- Can declare that certain pointcuts are either error or warning conditions
  - AspectJ compiler will check this for you
  - defining additional application-specific semantics
- Warn about unimplemented method

```
declare warning: call(void Thread.destroy()) :
                 "Thread.destroy() is not implemented";
```

- Prevent call to dangerous methods

```
declare error: call(void Thread.stop()) ||
  call(void Thread.suspend()) ||
  call(void Thread.resume()) :
    "Call to deprecated Thread method not allowed";
```

Constraints
Examples

- Code in a package called from wrong location
- Public fields in a class
- Constructors called outside specified classes
- Non thread safe code called from known dangerous context
- Calling System.out/err instead of logging package
- Use of deprecated methods or classes

Constraint Examples

Checking the Calling Class

- Use *within()* to check what's allowed in a particular type
  - a static check so can be used in constraints

```
// Warn about call to AWT in subclasses of an
// abstract base class
declare warning:
  call(* java.awt.*(..)) &&
  within(com.foo.FrameworkAbstractClass+) :
  "AWT access not allowed in framework classes";

// Check for illegal use of JDBC
declare error:
   call(* java.sql.*(..)) &&
  !within(com.foo.db.*) :
  "JDBC not to be used outside database code";
```

- Useful type modifier is "+" (type or subtype)

Constraint Examples

Checking the Calling Method

- Use *withincode()* to check what's allowed in a particular method

```
// Disallow state update from within getter methods
declare error:
  withincode(* com.foo.*.get*()) &&
  (set(* com.foo.*.*) || call(* com.foo.*.set*(..))):
  "Get methods cannot have side effects";

// Don't fiddle with threads inside constructors
declare warning:
  call(* Thread.*(..)) &&
  withincode(com.foo.*.new(..)) :
  "Thread state should not be changed in constructor";
```

Constraint Examples

Checking the Called Class

- Check for usage of System.out and System.err

```
public aspect CheckOutputStreamAspect {
  // Pointcut to identify potentially problematic code
  pointcut useOfSystemStreams() :
    get(* System.out) || get(* System.err);

  pointcut utilityCode(): within(com.foo.util..*);

  // Warning for access to system streams
  declare warning: useOfSystemStreams() :
    "Avoid using System.out and System.err - " +
    "use logging instead";

  // Error if used in the utility package
  declare error: useOfSystemStreams() && utilityCode():
    "System.out and System.err may not be used " +
    "in utility package";
}
```

This is, admittedly, slightly round-about

Constraint Examples
Structure

- All classes implementing Stateless should *be* stateless

```
public aspect EnforceStateless {
  pointcut isStateless() : within(com.foo.Stateless+) ;

  pointcut accessingState() :
    set(* com.foo..*.*) || get(* com.foo..*.*);

  declare error: isStateless() && accessingState() :
    "Cannot access state in a stateless object";
}
```

- This is not perfect:
  - the error is reported in the caller
  - (should be where error is, in Stateless class)

Constraint Examples

Pointcut Equivalence

- We prefer generalized pointcuts
- Check equivalence of explicit and general pointcuts

```
public aspect PointcutEquivalenceExample {
  // Capture public methods by enumeration
  pointcut allPublicMethodsExplicit():
    execution(* MyClass.method1(int)) ||
    execution(* MyClass.method2()) ||
    execution(* MyClass.method3(int)) ;

  // Capture public methods by pattern match
  pointcut allPublicMethodsImplicit():
    execution(public * MyClass.*(..));

  // See if the pointcuts are equivalent
  declare error:
    ( allPublicMethodsExplicit() && !allPublicMethodsImplicit()) ||
    (!allPublicMethodsExplicit() &&  allPublicMethodsImplicit()):
       "The two public method pointcuts are not equivalent";
}
```

Exercise

- Try out some of the examples from the slides
- What sort of tests you might apply at your own work
  - have a go at implementing them…

(More) Advanced Topics

- Runtime Constraints
- Inner Aspects
- AspectJ Idioms
- percthis(), perctarget(), percflow()
- Annotations (Aspect 5 + Java 5)
- Aspect Precedence

Runtime Constraints

- c.f. OCL, we have
  - preconditions
  - postconditions
  - class invariants
- Can check preconditions using before()
- Can check postconditions using extended version of after()
  - after returning
  - after throwing
- Can check class invariants by checking after any modification

Preconditions

- Use before(...) with args(...)

```
public aspect CheckForNullArgumentsAspect {
  pointcut invokeStringSetter(String value) :
    call(* ..model.*.set*(String)) && args(value) ;

  before (String value) throws IllegalArgumentException:
      invokeStringSetter(pass) && if (value == null) {
    throw new IllegalArgumentException(
                              "Value must not be null");
  }
}
```

```
public aspect CheckRangeAspect {
  pointcut invokeSetMark(int mark):
    call(* ..model.Exam.setMark(int)) && args(mark) ;

  before(int mark) throws IllegalArgumentException:
      invokeSetMark(mark) && if (mark < 0 || mark > 100) {
    throw new IllegalArgumentException(
                  "Mark must be in range [0,100]");
  }
}
```

Postconditions

- Use after(...) returning / after(...) throwing to check
  - Checking system services are return something

```
public aspect CheckSystemServicesResultAspect {
  pointcut invokeSystemServiceFacade(Object caller, Service service):
    call(* systemServices..*.*(..)) && this(caller) && target(service);

  after(Object caller, Service service)
      returning (Object result)
    : invokeSystemServiceFacade(caller, service) {
    if (result == null) {
      throw new RuntimeException("Service '" + service + " failing");
    }
  }
  after(Object caller, Service service)
      throwing (ServiceNotInitializedException ex)
    : invokeSystemServiceFacade(caller, service) {
    throw new RuntimeException(
        "Caller '" + caller + "' didn't initialize service");
  }
}
```

Can't use the if joinpoint on result with returning(...)

Advanced Topics: Runtime Constraints

Check Invariant

```
public class Exam {
  private int score;
  public int getScore(){…}
  public void setScore(int score){…}
  public static void
      main(String[] args) {
    Exam e = new Exam();
    e.setScore(10);
    e.setScore(-1);
  }
}
```

```
public aspect CheckAllIsWellAspect {
  public interface Constrained {
    public boolean checkInvariant();
  }
  pointcut changeInState(Constrained c):
    set(* *.*) && target(c);
  after(Constrained c): changeInState(c) {
    if (!c.checkInvariant()) {
      throw new RuntimeException(
        "Invariant violated after " +
        thisJoinPoint.getSignature());
    }
  }
}
```

To access private fields

```
public privileged aspect BindCheckAllIsWellToExamAspect {
    declare parents:
      Exam implements CheckAllIsWellAspect.Constrained;

    public boolean Exam.checkInvariant() {
        return this.score >= 0 && this.score <= 100;
    }
}
```

Advanced Topics

Inner Aspects

- cf inner classes
  - define aspect within an outer class
- Eg: Local binding

```
public abstract aspect CheckAllIsWellAspect {
  public interface Constrained {
    public boolean checkInvariant(); }
  pointcut abstract changeInState(
                                  Constrained c);
  after(Constrained c): changeInState(c) {
    … as before …
  }
}
```

```
public class Exam {
  private int score;
  public int getScore(){…}
  public void setScore(int score){…}
  public void bumpScore(int percent) {…}

  private static aspect CheckAllIsWellHere
        extends CheckAllIsWellAspect {
    declare parents: Exam implements Constrained;

    public boolean Exam.checkInvariant() {
        return this.score >= 0 && this.score <= 100;
    }
    pointcut changeInState(Constrained c):
      call(* set*(..)) && target(c) ||
      call(* bump*(..)) && target(c);
  }
}
```

Idioms

- Avoiding infinite recursion

```
public aspect Tracing {
  before(): call(* *.*(..) &&
            !within(Tracing) {
    System.out.println(
      thisJointPointStaticPart);
  }
}
```

- Nullifying advice

```
public aspect Tracing {
  before(): somePointcut() &&
            !if(false) {
  … rest of implementation …
  }
}
```

- Providing empty pointcut definitions

```
public abstract aspect
                  CheckArgs {
  abstract pointcut
    checkIntArg(int i);
 abstract pointcut
    checkStringArg(String s);
}
```

```
public aspect CheckArgs {
  pointcut
    checkIntArg(int i);
  pointcut
    checkStringArg(String s):
    call(* set*(String));
  … rest of implementation …
}
```

Advanced Topics:
Aspect precedence

- In the ideal world, all aspects would be orthogonal
  - however, they aren't.
- Use *declare precedence* to determine order

```
public aspect SystemPrecedences {
  declare precedence:
    AuthenticationAspect, Authorization, *;
  declare TransactionAspect, PublishChangeAspect;
  declare *, TracingAspect;
}
```

- Neat trick: use to check aspects don't overlap

```
public aspect CheckXAndYDontOverlap {
  declare precedence: AspectX, AspectY;
  declare precedence: AspectY, AspectX;
}
```

Aspect Instantiation Models

- By default, aspects are singletons
  - obtain reference using aspectOf()

```
public aspect ProfilingAspect {
  …
  public void dumpResults() { … }
}
```

```
ProfilingAspect pa =
  ProfilingAspect.aspectOf();
pa.dumpResults();
```

- Can also instantiate implicitly
  - specify a pointcut expression to scope

- per this
  - one aspect instance per object bound to this where pointcut matched
- per target
  - one aspect instance per object bound to target where pointcut matched
- per cflow & per cflowbelow
  - one aspect instance per thread at / below place where pointcut matched

- The instantiation model is specified as a modifier

perthis & pertarget

- Aspect's state and behaviour is an "addendum" to the state to objects on which pointcut matches

```
public aspect CountAcctUpdates
  perthis(updateAcct(Account)) {

  pointcut updateAcct(Account acct):
  (execution(* Account.credit(..)) ||
   execution(* Account.debit(..) )   )
   && this(acct);
  before(Account acct): updateAcct(acct) {
      accountUpdates++;
  }
  private int accountUpdates;
  public int getAccountUpdates() { … }
}
```

```
public aspect CountAcctUpdates
  pertarget(updateAcct(Account)) {

  pointcut updateAcct(Account acct):
  (call(* Account.credit(..)) ||
   call(* Account.debit(..) )   )
   && target(acct);
   before(Account acct): updateAcct(acct) {
      accountUpdates++;
  }
  private int accountUpdates;
  public int getAccountUpdates() { … }
}
```

```
CountUpdates cu =
 CountUpdates.aspectOf(acct1);
cu.getAccountUpdates();
```

- Use *perthis/pertarget* or introductions?
  - former is scoped to those matching pointcut
  - latter affects every instance
- Use *perthis* or *pertarget* ?
  - depends on whether code is available to weave into

percflow & percflowbelow

- Aspect's state and behaviour is scoped by thread
  - Akin to using a singleton aspect and storing its state in a ThreadLocal
  - Much more straightforward though
- Archetypal uses
  - transaction management
  - session management
- To obtain, just use aspectOf()
  - call within transaction boundary

```
OverflowAspect oa =
 OverflowAspect.aspectOf();
cu.getCounter();
```

```
public abstract aspect OverflowAspect
  percflow(dubiousBoundary()) {

  pointcut abstract dubiousCode();
  pointcut dubiousBoundary():
    dubiousCode() &&
    !cflowBelow(dubiousCode());

  Object around(): dubiousCode() {
    try {
      if (counter++>30)
        throw new RuntimeException("Doh!");
      return proceed();
    } finally { counter--; }
  }

  private int counter=0;
  public int getCounter() { … }
}
```

Annotations

- Specify explicitly where aspects should apply
  - Rather than using naming conventions or types

```
package com.aspectsrus.security;
import org.aspectj.lang.*;
public aspect SecurityAspect {

  pointcut securedMethods():
    execution( @Secured * *..*(..) );

  Object around(): securedMethods() {
    Signature sig = thisJoinPointStaticPart.getSignature();
    if (!permitted(sig)) {
      throw new SecurityException(
         "Invoking " + sig.toShortString() + " not permitted");
    }
    return proceed();
  }

  private boolean permitted(
            Signature signature) {
    … some implementation …
  }
}
```

```
package com.aspectsrus.security;
import java.lang.annotation.*;

@Target({ElementType.METHOD})
public @interface Secured { }
```

```
public class Customer {

    @Secured
    public Order placeOrder() { … }

    @Secured
    public void rejectOrder(Order o) { … }

    … and so on …
}
```

Exercise

- Use inner aspects to enforce a check invariant constraint for two different classes
- Disable aspects by nullifying an advice
- Make earlier profiling exercise threadsafe using percflow(...)
- Create aspects that use annotations
  - to enforce immutability per an @Immutable annotation
  - to implement (simulate) transactionality per a @Transacted annotation
- We've seen the use of AspectJ for preconditions, postconditions and invariants
  - Should AspectJ be used for other invariants, e.g. derived attributes?
  - If so, are there any implications for analysis / design?

Further Exercises

- Review, understand, and implement other AO Patterns
  - from *AspectJ in Action*, chapter 8
    - worker object creation pattern
    - wormhole pattern
    - (exception introduction pattern, was mentioned in previous exercise)
    - (participant pattern is same as director pattern, already covered)
  - from *AspectJ Cookbook*, chapter 23
    - cuckoo's egg pattern
      - e.g. use to implement AbstractFactory OO pattern
    - (director pattern already covered)
    - (border control pattern covered – we didn't name it as such)
    - (policy pattern covered – we didn't name it as such)

Thinking about Aspects

- We're going to wrap up the session with a bit of group reflection
- A few of our thoughts
- A few things you might want to discuss
- Your group discussion
- We'll try to aggregate our thoughts

Implications on Process

Concept : Implementation Ratio

- We strive for a 1:1 concept:implementation ratio ...
  - ... but we don't get it
- Instead we have 1:n
  - Same concept appears all over
  - Code scattering
- And we have also have n:1
  - Multiple concepts appear in single block
  - Code tangling
- How does this relate to notion of abstraction?
- How does this relate to MDA?

Implications on Process

Naïve Coding

- So that programmers *can* code naively
  - which means can focus on base program functionality
    - business problem at hand
- Aspects then apply system concerns
  - such as performance
    - eg use caching
  - enforce system rules
    - modifying UI outside the UI thread
  - Aspects can check if too naïve
    - constraints
- Who should code up aspects?

Implications on Architecture ?
Viewpoints

These are just a crib list for a discussion centred on architecture (we're not going to describe them)

- Functional
  - identifies and names system's runtime functional elements, their interfaces, responsibilities and interactions
- Information
  - analyses how data is stored, manipulated, managed and distributed
- Concurrency
  - maps functional units onto concurrency units
  - describes how concurrency is controlled and co-ordinated
- Development
  - describes architectural constraints relevant to the development process
- Deployment
  - describes the runtime environment
  - identifies dependencies from system upon its runtime
- Operational
  - How system will be operated and administer when in production

Implications on Architecture ?

Perspectives

Crib list continued (we're not going to describe them)

- Availability & Resilience
  - ability to be fully/partly operational (eg 24x7)
  - handling of failures (eg disk or network failure)
- Evolution
  - support for diagnosis of problems
  - malleability (allowing changes to functionality/architecture)
- Performance & Scalability
  - predictably execute within performance profile (eg xactns per second)
- Security
  - control, monitor and auditing of resources within system
- Accessibility
  - support for people with disabilities
- Development Resource
  - ability to design, build and deploy within known resource constraints
- Internationalisation
  - support for multiple locales, language or culture
- Location
  - how absolute distance affects the system
- Regulation
  - conformance to legislation, company policies and other standards
- Usability
  - ease with which people can interact with the system
  - may also address usability when runtime resources are limited (can the system run on a 600Mhz CPU?)

Implications on Design

- Keep an OO view of world?
  - Design by contract
  - Aspects provide a simple implementation of getter-based dependency injection
  - Should we convert OO design patterns to AO?
- Or adopt an AO view of world?
  - Should we design new systems in terms of aspects?
  - (is this really an architectural discussion again?)
- Will AO design patterns play an role in popularizing AO?
  - Should we look for new AO patterns?
- Should we refactor using AO?

Implications on Design

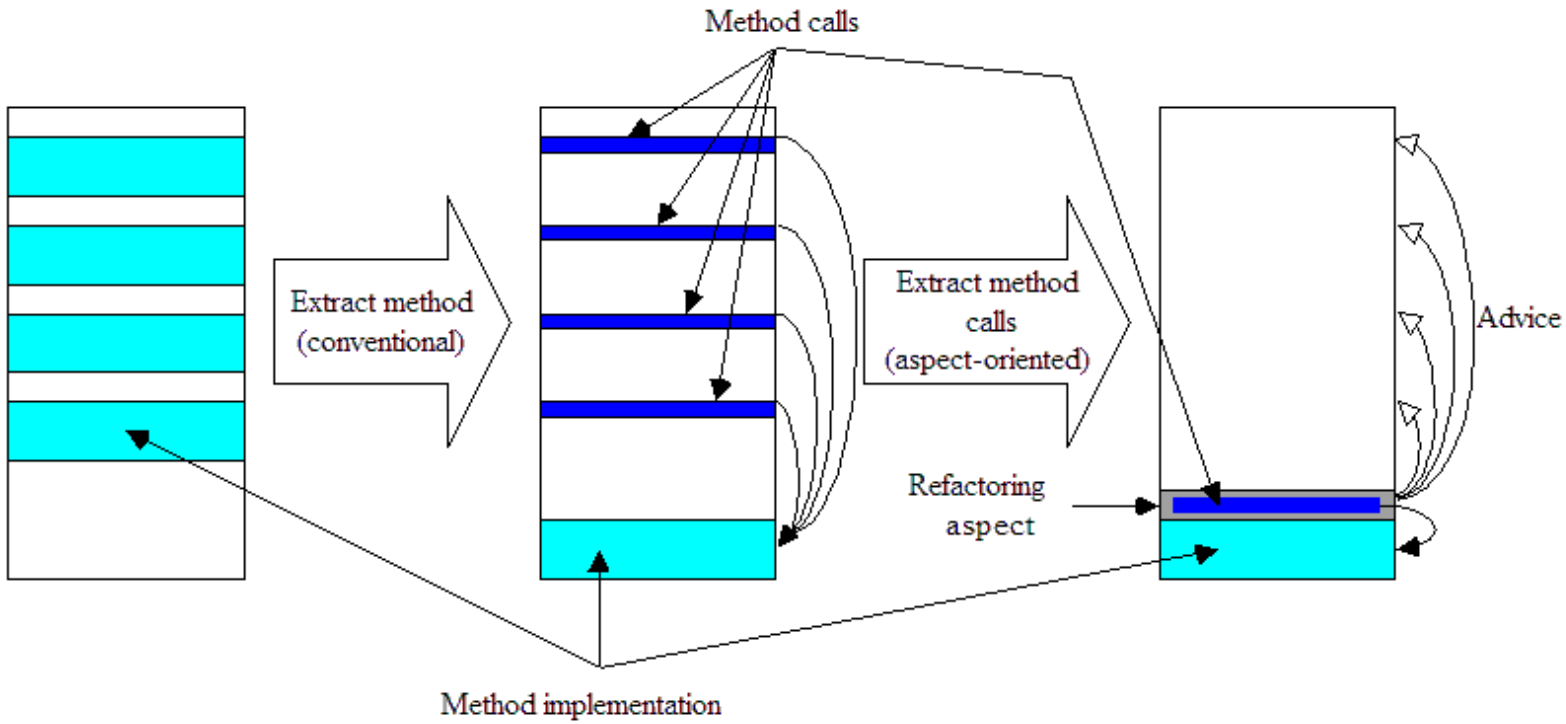
AOP Refactoring

- Design of Aspects – Refactoring Existing Code
  - Extract exception handling
    - wrapping underlying exceptions (in context of layered architecture)
  - Extract concurrency control
    - acquiring read or write locks
  - Extract worker object creation
    - Runnables, PrivilegedActions, SwingUtilities.invoke…
  - Replace argument trickle by wormhole
  - Extract interface implementation
  - Replace override with advice
    - don't bother to subclass
  - Extract lazy initialization
  - Extract contract enforcement

We haven't got time to describe all of these; we just want to give some examples

Implications on Design
AOP Refactoring

- e.g. "Extract Method Calls"



AOP refactoring article on http://www.theserverside.com

Implications on Testing

- Testing OO classes
  - rather than using mock objects, use an aspect with around(...) advice
- Testing of aspects themselves
  - Testing advice is not much harder than testing a method
    - apply aspect to known "base" class
    - use JUnit as per usual
  - How test that pointcuts are correctly defined?
    - naming conventions?
    - marker interfaces?
    - annotations?
    - something else?

Brainstorming Aspects

- Some categories
  - Development Aspects
    - the final software works well without, only be useful during development.
    - Examples of these is logging, tracing and profiling.
  - Product Aspects
    - must be included for the system to work
    - Examples are Authentication and Exception handling.
  - Runtime Aspects
    - make the program work better but they are not required for the program to function.
    - Examples are aspects that raise performance like pooling, caching and buffering.

- Brainstorm aspects and categorise
  - system aspects
    - eg persistence, security etc.
    - the architecture slides will likely give some ideas
  - business aspects
    - e.g. implementing bidirectional referential integrity?
    - is this an appropriate use of AOP?

- For each, ask
  - Could it be made generic and placed into a library?
  - would it be worth doing so?

Brainstorming
Here's a starter (in no particular order)

- logging
- tracing
- security
- transactions
- persistence
- visualization a la Naked Objects
- internationalization
- backgrounding
- pooling
- hot swap
- caching
- immutability (decorator)
- threadsafety (decorator)
- remoting
  - different sets of aspects to create client/server interactions
- policy enforcement standards ("semantic compilation")

Discussion / Group Exercise

- Working in groups
  - Take one or two of the questions posed on the previous slides
  - See if you can come up with some coherent answers :-)
- And for every group, also consider
  - We've assumed that AOP is a "good thing" … is AOP a viable technology?
  - Is it clear how to apply it?
    - where is AOP's sweet spot?
  - Are there dangers in applying AOP?
    - Where should AOP not be used
    - what might make a project using AO fail?
  - Is the technology ready for adoption?

30 minutes

Wrapup

- We covered
  - AOP concepts
  - AspectJ syntax
    - dynamic & context collection pointcuts
    - introductions
    - enforcing constraints
  - Advanced AspectJ topics
    - bindings
    - annotations
  - Eclipse AJDT
- We discussed implications of adopting AO to:
  - development process
  - architecture
  - design
  - testing
- Still to come (if you are interested)
  - A tiny Spring overview
  - Configuring Aspects using Spring
  - Real-world AspectJ
  - Resources & Tools
- But otherwise …
  - that's a wrap!

SPA 2005
AspectJ Tutorial

Dan Haywood
dan@haywood-associates.co.uk
Eóin Woods
ewo@zuhlke.com

Appendices

- Spring and AspectJ
- Real world AspectJ
- Resources
- Other Tools

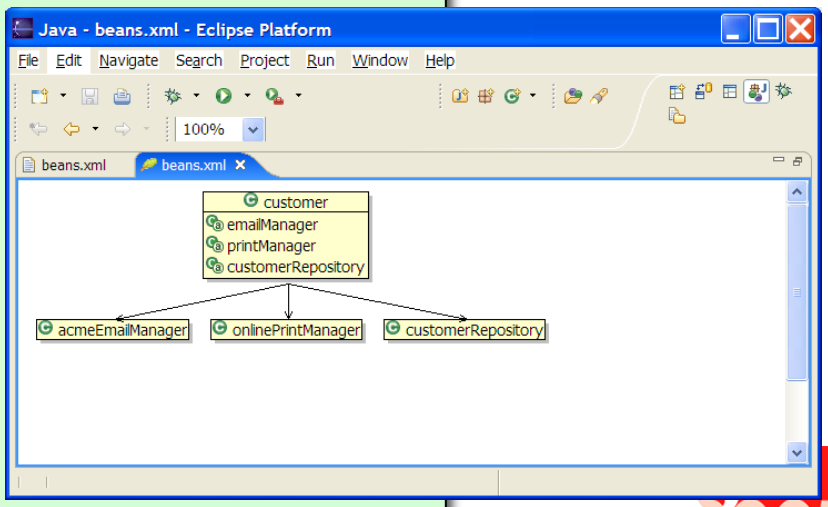
Dependency Injection

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
          "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="acmeEmailManager"
          class="com.mycompany.services.email.acme.AcmeEmailManager"
          singleton="true"/>

    <bean id="onlinePrintManager"
          class="com.mycompany.services.print.online.OnlinePrintManager"
          singleton="true"/>

    <bean id="customerRepository"
          class="com.mycompany.domain.CustomerRepository"
          singleton="true"/>

    <bean id="customer"
          class="com.mycompany.domain.Customer"
           singleton="false">
        <property name="emailManager">
            <ref local="acmeEmailManager"/>
        </property>
        <property name="printManager">
            <ref local="onlinePrintManager"/>
        </property>
        <property name="customerRepository">
            <ref local="customerRepository"/>
        </property>
    </bean>
</beans>
```



BeanFactory

```
public class Main {
  public static void main(String[] args) throws Exception {

    XmlBeanFactory beanFactory =
      new XmlBeanFactory(new FileInputStream("beans.xml"));

    CustomerRepository customerRepository =
      (CustomerRepository)beanFactory.getBean("customerRepository");

    Customer eoin = customerRepository.lookup(123);
    Customer dan = customerRepository.lookup(456);

    System.out.println(eoin);
    System.out.println(dan);
  }
}
```

CustomerRepository is a configured as singleton

```
public class CustomerRepository implements BeanFactoryAware {

  public Customer lookup(final int number) {
    Customer customer =
     (Customer)beanFactory.getBean("customer");

    // populate with serialized data

    return customer;
  }

  private BeanFactory beanFactory;
  public BeanFactory getBeanFactory() { … }
  public void setBeanFactory(BeanFactory beanFactory) { … }
}
```

Customer is configured as a prototype (non-singleton)

Spring / AspectJ integration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
          "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="girl" class="com.mycompany.Girl">
    <property name="kissable"><ref bean="boy"/></property>
  </bean>

  <bean id="boy" class="com.mycompany.Boy"/>

  <bean id="teacher"
        class="com.mycompany.TeacherAspect"
        factory-method="aspectOf">
    <property name="response">
      <value>We'll have none of that please...</value>
    </property>
  </bean>
</beans>
```

```
public aspect TeacherAspect {
  private String response;
  public void setResponse(String response) { … }

  pointcut aGirlKissingABoy():
    call(* Kissable.kiss()) && this(Girl) && target(Boy);

  after() returning : aGirlKissingABoy() {
    System.out.println(response);
  }
}
```

- percthis(), percflow() not yet supported

Exercise

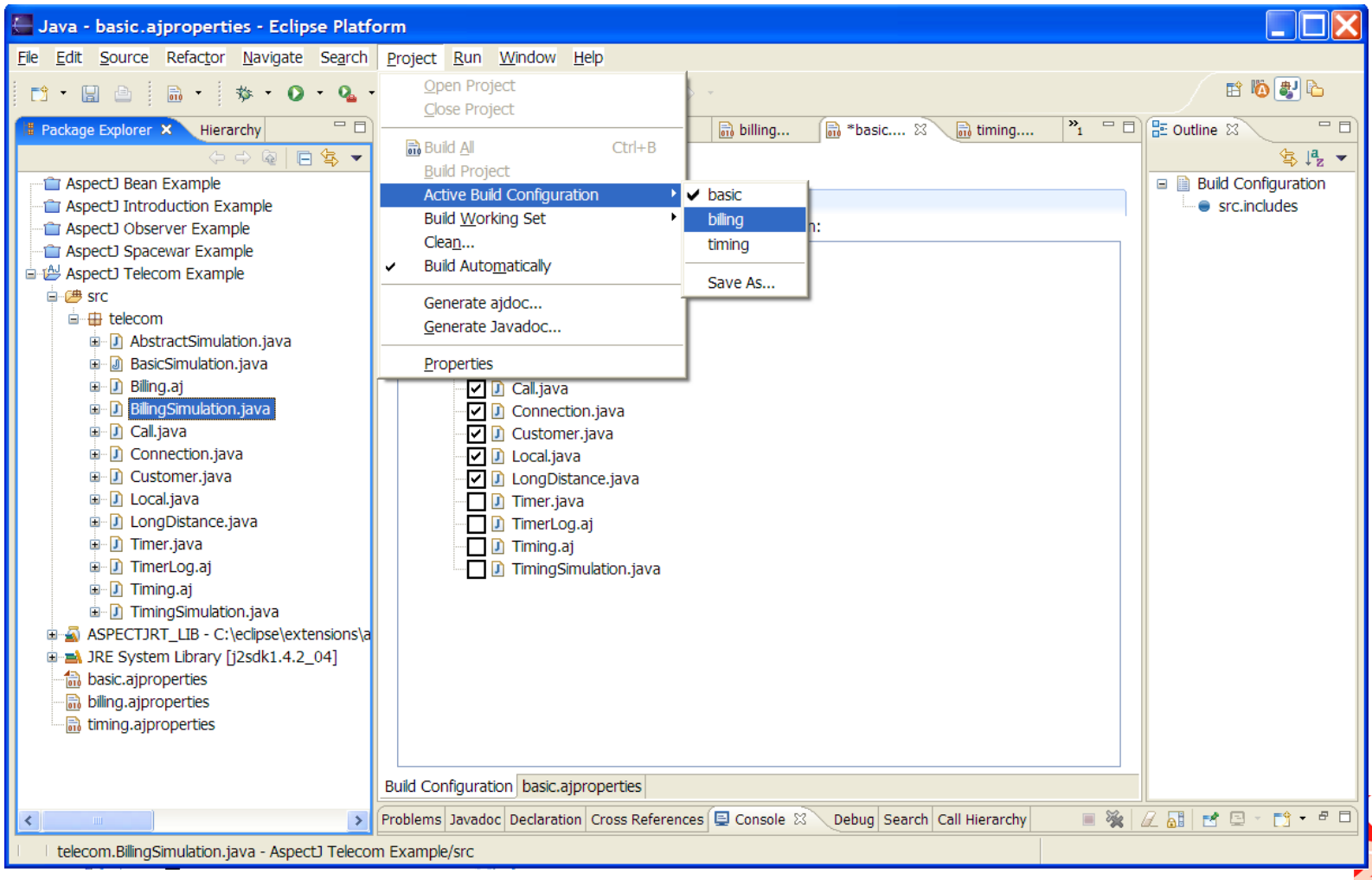
- Use Spring to turn tracing on or off
- Use Spring to make a Security authenticator pluggable into a SecurityEnforcingAspect.

Real-world AspectJ

- Build Configurations
- Using Ant
- CLASSPATH vs INPATH vs ASPECTPATH
- AspectJ Libraries
- Testing Aspects
- Replacing Mock Objects

Build configurations

- Indicate aspects to include in a compile



Using Ant

- Setup to use AJDT build configurations
  - myapp.ajproperties:

```
src.includes = src/
src.excludes = src/tracing/version1/,\
               src/tracing/version2/,\
               src/tracing/version3/
```

  - build.xml:

```
<property file="myapp.ajproperties" />

<target name="compile">
  <iajc classpath="${org.aspectj.runtime.home}/aspectjrt.jar"
        srcdir="."
        includes="${src.includes}"
        excludes="${src.excludes}"/>
</target>
```

- Full instructions in Installation.pdf

CLASSPATH

- Types on the classpath will be found (resolved) by AspectJ during compilation and weaving, exactly as a classpath would be used when compiling with javac.
- Types are unchanged by the compilation and weaving process - they are for lookup only.
  - So if you just put a type on the classpath, it can be found by the compiler/weaver for resolution purposes, but it is not exposed to the weaver for linking with aspects.
- Under AJDT, if a project dependency exists between two projects then the output of the depended project is placed on the CLASSPATH of the dependent project.
  - So the types in the depended project are visible to the compiler, but will not be affected by the weaving process.
  - If you want types in the referenced project to be linked with aspects in the dependent project, you need to use the INPATH

INPATH

- Serves a dual purpose.
  - They are both available for type resolution (as for types on the CLASSPATH)
  - Are also exposed to the weaver for linking with aspects
    - types could themselves be aspects
- Thus the output of a compilation/weave with types specified on the INPATH will result in new (possibly modified) versions of those types being written to the output destination of the compilation.
- Under AJDT
  - specify inpath entries using the AspectJ Inpath page of the project properties.
  - in older versions of ajc, the –injars flag was used

ASPECTPATH & Aspect Libraries

- If you want an aspect (in class file form) to be woven with types exposed to the weaver, you need to place it on the ASPECTPATH
- We often refer to jar files containing aspects that are placed on this path as "aspect libraries".
- In AJDT, if you had one project defining a collection of aspects, and another project that wanted to use those aspects, you would proceed as follows:
  - In the aspect library project, use the "outjar" option on the AspectJ page of the project properties to have AJDT place the output of the compilation into a jar file.
  - In the project using the library, put the outjar created by the library project onto the aspectpath using the AspectJ Aspectpath page of the project properties.

Exercises

- AspectJ Telecom Example
  - 3 different build configurations
- Build using Ant
- Refactor aspects into reusable library
  - e.g. one (or a couple) of the constraint aspects
  - AspectJ Cookbook has details
- Profile for an existing app.
  - eg Ant itself
- Unit test using mocks
  - refactor to use Aspects

Resources
GOF patterns

- A perennial favorite for AO courses, it seems :-)
- A recent reimplementation using AspectJ at
  - http://www.cs.ubc.ca/~jan/AODPs/
  - (downloadable)
- Also, AspectJ Cookbook includes implementations
  - heavy use of the Director pattern

Resources

Other Tools

- Aspect Browser
  - Find and manage crosscutting concerns
  - Eclipse plugin
  - http://www.cs.ucsd.edu/users/wgg/Software/AB
- Feature Exploration and Analysis Tool (FEAT):
  - Explore crosscutting concerns in an existing system
  - Eclipse plugin
  - http://www.cs.ubc.ca/labs/spl/projects/feat
- Aspect Mining Tool (AMT)
  - Mine aspects in an existing system
  - http://www.cs.ubc.ca/~jan/amt
- Eclipse CME
  - Concern Manipulation Environment
  - (not specific to AspectJ)
  - http://www.eclipse.org/cme