AOP 101: Intro to Aspect Oriented Programming

Ernest Hill
ernesthill@earthlink.net
AOP 101: Aspect Oriented Programming

- Goal of Software
- History of Programming Methodology
- Remaining Problem
- AOP to the Rescue
- AOP Terminology
- AOP Implementation
- Summary
Goal of Software

- Systems are built to deal with concerns
- A concern is functionality required by the system which can be addressed in code
History of Programming
Methodology

- Structured Programming
  - built functions to deal with these concerns

- Object Oriented Programming
  - decomposed systems into objects which deal with these concerns
Problem Addressing Crosscutting Concerns

• Crosscutting Concerns
  – Concerns that cut across objects or are global
  – Examples: Logging and Authorization

• Implementing code to deal with these concerns
  – Leads to duplicated code
  – Leads to tangled code
  – Leads to scattered code
AOP to the Rescue

- AOP aims to separate out crosscutting concerns
- Modularize code dealing with each concern
- Modularized code is an aspect
- Aspects are weaved in at compile or runtime
AOP Benefits

• Aspects are easier to maintain
  – Code dealing with an aspect in one place

• System is easier to evolve
  – Crosscut module unaware of aspect, so easy to add new aspect

• Some design decisions can be deferred
  – Aspect are implemented separately
AOP Terminology

- Join point
- Pointcuts
- Advice
- Introduction
- Weaving
Join Point

• Fundamental concept of AOP
• Any identifiable execution point in the code
• A location in the code where a concern will crosscut the application
• Not all join points are supported
Typical Join Points

- Constructor call
- Constructor call execution
- Method call
- Method call execution
- Field get
- Field set
- Exception handler execution
- Class Initialization
- Object Initialization
Constructor Call

• Defined when a constructor is called during creation of a new object

• Defined within the context of the calling application

Widget widget = new Widget();
Constructor Call Execution

- Defined when the constructor is called on an object
- Trigger before the constructor code executes
- Occurs after Constructor Call join point

```java
public Widget {
    private Thingee thingee;
    public Widget( Thingee aThingee) {
        this.thingee = aThingee;
    }
}
```
Method Call Join Point

- Defined when any method call is made by an object or static method if no object is defined
- Defined within the calling object or application

```java
widget.fill();
```
Method Call Execution Join Point

- Defined when a method is called on an object and control transfers to the object
- Occurs before method code is executed
- Occurs after method call join point

```java
public class Widget {
    ...
    public void fry() {
        this.status = FRIED;
    }
}
```
Field Get Join Point

• Defined when an object attribute is read

```java
public class Widget() {

    String name;

    ...

    public String toName() {
        return "Widget" + name;
    }
}
```
Field Set Join Point

• Defined when object attribute is written

```java
public class widget {

    int temp;

    ...

    public void heatTo(int temperature) {
        temp = temperature;
    }
}
```
Exception Handler Join Point

- Defined when exception handler is executed

```java
try {
    widget.heat( 212 );
} catch ( WidgetOverheatedException ex ) {
    postMessage(ex);
}
```
Class Initialization Join Point

- Defined when any static initializers are executed
- If no initializers, no join points

```java
public class widget {

    static {
        try {
            System.loadLibrary("widgethandler");
        } catch (UnsatisfiedLinkError ex) {
            ... deal with the exception
        }
    }

    ... 
}
```
Object Initialization Join Point

- Defined when a dynamic initializer is executed for a class
- After call to object's parent constructor
- Just before return of object's constructor

```java
public class Widget extends Thingee {
    ...
    public Widget(boolean isPalpable) {
        super();
        this.isPalpable = isPalpable;
    }
    ...
}
Pointcut

• A set of join points defined to specify when the advice should be executed

• Often described using regular expressions or pattern match syntax

• Some frameworks support composition of pointcuts
Advice

• The actual code to be executed when the poincut is triggered

• Types of Advice
  – Before
  – After
  – Around
Before Advice

- Simplest type of advice
- Invoked before the join point is invoked
After Advise

- Three types of after advise
  - After returning
    • Runs after join point is executed, if no exception was thrown
  - After throwing
    • Run if the joint point threw an exception
  - Unqualified
    • Runs no matter what the outcome of the join point
Around Advice

- Most intrusive
- Given control
- May invoke the joint point if it chooses
Introduction

• Adding methods or fields to an existing class
• Can be used to have a class implement a new interface
Weaving

- Assembling to modules into its final form
- Aspect define the rules for assembly
- Can be performed at compile time or run time
AOP Implementation Strategies

- Dynamic proxies
- Dynamic byte code generation
- Java source code generation
- Custom class loader
- Language extension
Dynamic Proxy

- Allows implementation of one or more interface on the fly
- Around advice
  - Proxy will invoke chain of interceptors
  - Last interceptor will invoke target
- Only uses standard Java
- Only works with interfaces
- Used by Spring
Dynamic Byte Code Generation

- Generate dynamic subclasses
- Methods have hooks to invoke advice
- Subclass cannot proxy final methods
- Used by Spring
Java Source Code Generation

- Generate new source that includes crosscutting code
- Used by EJB
Use a Custom Class Loader

- Advice can be applied when class is loaded
- Problematic to control class-loading hierarchy
- Used by AspectWerkz
Language Extension

- Pointcuts and aspects can be first-class language constructs
- Aspects can participate in inheritance
- Used by AspectJ
AOP Implementations

- AspectJ
- AspectWerkz
- Spring
AspectJ

- Most complete & mature AOP implementation
- Pointcuts and aspects are first-class language constructs
- Aspects can inherit from aspects
- Pointcuts based upon a range of criteria
- Compile-time declaration allows addition of compile-time warnings and errors
- Weaving occurs at compile-time
AspectJ Trace Aspect

public aspect JoinPointTraceAspect {
    private int _callDepth = -1;

    pointcut tracePoints() : !within(JoinPointTraceAspect);

    before() : tracePoints() {
        _callDepth++;
        print("Before", thisJoinPoint);
    }

    after() : tracePoints() {
        print("After", thisJoinPoint);
        _callDepth--;
    }

    private void print(String prefix, Object message) {
        for(int i = 0, spaces = _callDepth * 2; i < spaces; i++) {
            System.out.print(" ");
        }
        System.out.println(prefix + ": " + message);
    }
}
Compile Time Declaration

- Allows addition of compile time warnings and errors
- Can be used to enforce coding standards
- Unique to ApectJ
- No class file modification occurs
Compile Time Declaration

aspect DetectPublicAccessToMembers {
    declare warning :
        get(public !final * *) || set(public * *) :
            "Please consider using non-public access";
}

aspect DetectSystemErrUsage {
    declare error :
        call ( * System.err.print*(..) :
            "Please use Logger.log() instead";
}
Eclipse support for AspectJ

- Outline for aspect lists
- join points advised
Eclipse Support for AspectJ

- Different gutter annotation depending on type of advise
AspectWerkz

- Good documentation
- Support per JVM, per class, per instance and per thread advice
- Ability to add or remove advice at runtime
- Class loader approach to weaving
- Also support code weaving at compile time
package testAOP;

import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class MyAspect {

    public void beforeGreeting(JoinPoint joinPoint) {
        System.out.println("before greeting...");
    }

    public void afterGreeting(JoinPoint joinPoint) {
        System.out.println("after greeting...");
    }
}
AspectWerkz PointCut

<aspectwerkz>
  <system id="AspectWerkzExample">
    <package name="testAOP">
      <aspect class="MyAspect">
        <pointcut name="greetMethod" expression="execution(* testAOP.HelloWorld.greet(..))"/>
        <advice name="beforeGreeting" type="before" bind-to="greetMethod"/>
        <advice name="afterGreeting" type="after" bind-to="greetMethod"/>
      </aspect>
    </package>
  </system>
</aspectwerkz>
Spring

- Expressive and extensive pointcut model
  - Regular expressions supported
  - Composition supported

- Control flow pointcuts supported
  - Such as “all methods invoked from MVC controller”

- Programmatic of configuration driven proxing

- Must get advised objects from Spring IoC container or use AOP framework programmatically
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class TracingBeforeAdvice
    implements MethodBeforeAdvice
{
    public void before(Method m, Object[] args, Object target)
        throws Throwable
    {
        System.out.println("Hello world! (by " +
                           this.getClass().getName() + "")");
    }
}
Spring Configuration

```xml
<bean id="businesslogicbean"
  class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces">
    <value>IBusinessLogic</value>
  </property>
  <property name="target">
    <ref local="beanTarget"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>theTracingBeforeAdvisor</value>
    </list>
  </property>
</bean>

<!-- Bean Classes -->
<bean id="beanTarget" class="BusinessLogic"/>

<!-- Advisor pointcut definition for before advice -->
<bean id="theTracingBeforeAdvisor"
  class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="advice">
    <ref local="theTracingBeforeAdvice"/>
  </property>
  <property name="pattern">
    <value>.*</value>
  </property>
</bean>

<!-- Advice classes -->
<bean id="theTracingBeforeAdvice" class="TracingBeforeAdvice"/>
```
Summary

• AOP modularizes code to deal with crosscutting concerns

• Aspect is made up of
  – Joinpoint – An identifiable execution point in the code
  – Pointcut - A set of join points defined to specify when the advice should be executed
  – Advise - The actual code to be executed when the pointcut is triggered

• A number of implementations available
Resources

• Aspect Oriented Software Development
  – http://www.aosd.org

• AspectJ
  – http://www.aspectj.org

• AspectWerkz
  – http://aspectwerkz.codehaus.org/

• Spring
  – http://springframework.org