Aspect-Oriented Programming (AOP) in Java

Mark Volkmann

Partner

Object Computing, Inc. (OCI)

August 14, 2003

AOP Overview

- Provides "separation of concerns"
  - separating **common needs** of possibly **unrelated classes** from those classes
  - can **share** a single implementation **across many classes**
    - much better than modifying many existing classes to address a concern
  - changes can be made in **one place** instead of in multiple classes
- Provides a way to describe concerns
  - concerns are encapsulated into "aspects" (more on this later)
- Removes "code tangling"
  - implementing more than one concern in one class
- Removes "code scattering"
  - implementing the same concern in multiple classes

both of these reduce potential for reuse

- Not a replacement for object-oriented programming (OOP)
  - used in conjunction with it

Common Uses For AOP

(called “concerns” in AOP lingo)

- Authentication
- Caching
- Context passing
- Error handling
- Lazy loading
- Debugging
  - logging, tracing, profiling and monitoring
- Performance optimization
- Persistence
- Resource pooling
- Synchronization
- Transactions

AOP Terminology

- concern - functionality to be consolidated (see common uses on previous page)
- advice - code that implements a concern
- join point - a location in code where advice can be executed
- pointcut - identifies sets of join points
- introduction
  - modify a class to add fields, methods or constructors
  - modify a class to extend another class or implement a new interface
- aspect - associates join points/pointcuts/advice and applies introductions
- crosscutting - what aspects do to application classes (see next page)
- weaving - the process of inserting aspect code into other code
- instrumentor - tool that performs weaving

pointcuts can also identify context information to be made available to advice

can be done at build-time, load-time and run-time

Concerns: Crosscutting or Integral?

- Before AOP
  - implementations of common concerns were typically shared between multiple classes by inheriting from a common base class
- All want same?
  - when all potential users of the classes would want the same implementation, the concern is "integral"
  - in this situation, inheriting from a common base class is fine
- Some want different?
  - when some potential users of the classes may want a different implementation, the concern is "crosscutting"
    - all the typical uses of AOP listed on page are potentially crosscutting
  - it's best to separate these from the classes in order to **maximize their reusability**
  - **AOP gives us this capability!**

Join Points

This is a list of join points supported by AspectJ. Other implementations tend to support a subset of these. For example, Nanning only supports “method call”.

- Support for specific kinds of join points varies
- Some to look for include
  - method call - in calling code where call is made
  - method execution - in called method before code is executed
  - constructor call - in calling code where call is made
  - constructor execution -

    in called constructor after `super` or `this` calls, but before other code is executed
  - field get - when the value of a field is accessed
  - field set - when the value of a field is modified
  - exception handler execution - before a `catch` block for an exception executes
  - class initialization - before execution of “`static { code }`” blocks
  - object initialization - before execution of “`{ code }`” blocks

Development vs. Production Aspects

- Development aspects
  - may want to insert them after code is placed in production and remove them when finished using
  - used for debugging concerns
- Production aspects
  - intended to be used in production code
  - used for all other concerns listed on page 3
- Some AOP frameworks don't support insertion of aspects into production code at run-time and later removal

Java Weaving Approaches

- Source Generation
  - parse Java source and generate new Java source
- Bytecode Modification
  - three varieties
    - modify .class files at build-time
    - modify bytecode at run-time as it is loaded into the JVM
    - modify bytecode at run-time after it has been loaded into the JVM
      - great for debugging concerns
- Dynamic Proxies
  - create proxy objects at run-time that can delegate to the target object
  - can only be used with classes that implement some interface
  - code must explicitly create proxy objects
    - typically done in a factory method
    - if target objects are created using their constructors then aspects won't be utilized

Any form of source generation is an alternative to build-time AOP. For example, **XSLT** can be used to generate source code from an XML document that describes a database schema.

Java-based AOP Frameworks

- The following AOP frameworks are discussed later
  - AspectJ
  - AspectWerkz
  - Nanning
  - Prose (PROgrammable Service Extensions)

There is debate over whether frameworks that only provide method interception such as Nanning represent real AOP. Some refer to them as **Aspect-like** rather than **Aspect-Oriented**.

Dynamic Proxies

- Overview
  - dynamically generates classes at run-time that implement given interfaces
  - instances of those classes are called “dynamic proxies”
  - used as the basis of some AOP frameworks such as Nanning
- Limitations
  - can only act as a proxy for classes that implement some interface
  - when overriding methods of existing classes, callers must typically obtain an object from a factory method instead of using a constructor
    - existing code that uses constructors must be modified
- Simple to use!
  - see example on next page

Dynamic Proxy Example

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class DynamicProxyDemo implements InvocationHandler {

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

  private DynamicProxyDemo() {
    Adder proxy = getAdder();
    System.out.println("sum = " + proxy.add(19, 3));
  }
```

```
public interface Adder {
  int add(int n1, int n2);
}
```

Dynamic Proxy Example (Cont'd)

```
public Adder getAdder() {
  // What interfaces should the proxy implement?
  Class[] interfaces = new Class[] {Adder.class};

  // What class will handle invocations on the proxy?
  InvocationHandler ih = this;

  // Create the proxy.
  ClassLoader cl = getClass().getClassLoader();
  return (Adder) Proxy.newProxyInstance(cl, interfaces, ih);
}
```

clients of the Adder interface
would call this method
to get an instance

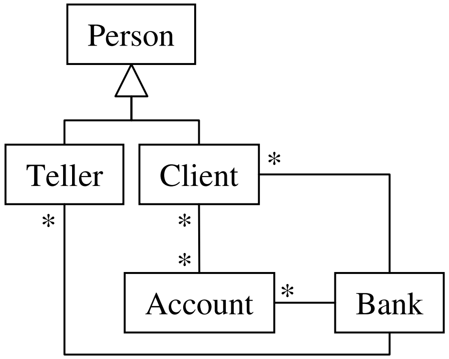
Dynamic Proxy Example (Cont’d)

```
  public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    if (!(proxy instanceof Adder)) {
      throw new IllegalArgumentException("bad proxy");
    }

    if (!method.getName().equals("add")) {
      throw new IllegalArgumentException("bad method");
    }

    // Can also test parameter types of the Method.

    // Typically delegate to methods of other classes.

    int n1 = ((Integer) args[0]).intValue();
    int n2 = ((Integer) args[1]).intValue();
    return new Integer(n1 + n2);
  }
}
```

only method in InvocationHandler interface

AOP Examples

- Upcoming examples address the following concerns
  - access
    - log access (or calls) to specific methods
  - context
    - pass “context” data to specific methods
      so they can include it in their log messages
      - examples could include the name of the application making the call
        and the name of the user running the application
  - exceptions
    - log the occurrences of specific exceptions
  - performance
    - log the time it takes to complete specific method calls
- Domain classes used
  - see diagram to the right



AspectJ

for **historical perspective**,
see Gregor's 1997 paper at
http://www.parc.com/research/csl/projects/aspectj/downloads/ECOOP1997-AOP.pdf

- Open-source AOP framework started by Gregor Kiczales
  - based on research at Xerox Palo Alto Research Center (PARC)
    - over 10 years so very mature
    - funded by Xerox, a U.S. grant and a DARPA contract
  - available at http://eclipse.org/aspectj

Defense Advanced Research Projects Agency

- AspectJ Compiler (ajc)
  - based on IBM's Eclipse Java compiler
    - this isn't based on Jikes, but some of the Jikes developers work on it
  - compiles aspect code and Java classes
  - doesn't require a special JVM to execute

can also operate on .class files produced by another compiler when source is not available using the `-injars` option

- How are aspects specified?
  - using proprietary Java extensions that are compiled with ajc
  - just have to compile aspects (typically in .aj files) along with Java classes
  - no other configuration files are needed

AspectJ (Cont'd)

- Weaving
  - version 1.0 and earlier used source generation weaving
  - version 1.1 (current version)
    uses bytecode weaving into .class files before run-time
  - will supply a custom classloader soon that provides bytecode weaving as it is loaded into the JVM
- Features
  - supports more AOP features than others
    - has a corresponding learning curve
  - aspect browser (ajbrowser) - more on this later
- Run-time library size - 29K
  - aspectjrt
  - small because all weaving is done at build-time

AspectJ Support in IDEs

- Two features are typically supported
  - compiling with the AspectJ compiler
  - browsing relationships between classes and aspects
- Currently available for these IDEs/tools
  - Eclipse, NetBeans, Emacs, JBuilder, Ant
- Currently Eclipse is the only IDE with good support for AspectJ debugging

IntelliJ is working on adding support for IDEA

AspectJ AccessAspect.aj

```
package com.agedwards.aspects;

import com.agedwards.bank.Account;

aspect AccessAspect {

  pointcut accountMethod(): execution(* Account.*(..));

  before(): accountMethod() {
    String className = thisJoinPoint.getTarget().getClass().getName();
    String methodName = thisJoinPoint.getSignature().getName();
    System.out.println
      ("Access: " + className + " method " + methodName + " was called");
  }
}
```

Logs calls to all methods in the Account class

AspectJ ExceptionAspect.aj

```
package com.agedwards.aspects;

import com.agedwards.bank.Demo;

aspect ExceptionAspect {

  pointcut demoRun(): execution(void Demo.run());

  after() throwing(Exception e): demoRun() {
    System.out.println("EXCEPTION: " + e.getMessage());
  }
}
```

Logs all exceptions thrown
out of the run method
of the Demo class

AspectJ PerformanceAspect.aj

```
package com.agedwards.aspects;

import com.agedwards.bank.Account;

aspect PerformanceAspect {

  pointcut accountDeposit(): execution(void Account.deposit(..));

  void around(): accountDeposit() {
    long startTime = System.currentTimeMillis();
    proceed();
    long stopTime = System.currentTimeMillis();
    long elapsedTime = stopTime - startTime;
    System.out.println("Perf: time to deposit = " + elapsedTime + " ms");
  }
}
```

Logs the elapsed time for all calls to the deposit methods in the Account class

"around" advice is run instead of the method it wraps. `proceed()` invokes the wrapped method.

AspectJ ContextAspect.aj

Logs all calls to the deposit method in the Account class including data in the current Context object

```
package com.agedwards.aspects;

import com.agedwards.bank.Account;
import com.agedwards.bank.Context;
import com.agedwards.bank.Demo;
import java.lang.reflect.*;
import org.aspectj.lang.reflect.MethodSignature;
```

includes a reference to the Bank and Teller associated with a transaction

```
aspect ContextAspect {

  public interface ContextPasser {}
  private Context ContextPasser.context;
  declare parents: Demo implements ContextPasser;
```

adds a “context” field to the Demo class

```
  public interface ContextReceiver {}
  declare parents: Account implements ContextReceiver;
```

adds an “invoke” method to the Account class (see next page)

AspectJ ContextAspect.aj (Cont'd)

```
private Object ContextReceiver.invoke(Context context,
  String methodName, Class[] types, Object[] args) {
  Class clazz = getClass();
  System.out.println("Context: " + clazz.getName() +
    " method " + methodName + " called, context = " + context);

  Object result = null;
  try {
    Method method = clazz.getMethod(methodName, types);
    result = method.invoke(this, args);
  } catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
  }
  return result;
}
```

invokes the specified method using reflection

AspectJ ContextAspect.aj (Cont’d)

```
pointcut demoSetup(Demo demo):
  execution(void Demo.setup()) && this(demo);

after(Demo demo): demoSetup(demo) {
  demo.context = new Context(demo.getBank(), demo.getTeller());
}
```

sets the “`context`” field in the Demo object when the data it needs is available

AspectJ ContextAspect.aj (Cont'd)

intercepts all deposits and passes the data needed to invoke the real method, along with associated Context, to the real target (see invoke method on page 22)

```
  pointcut accountDeposit(ContextPasser passer):
    call(* Account.deposit(..)) && this(passer);

  void around(ContextPasser passer): accountDeposit(passer) {
    ContextReceiver receiver =
      (ContextReceiver) thisJoinPoint.getTarget();
    MethodSignature signature =
      (MethodSignature) thisJoinPoint.getSignature();
    String methodName = signature.getName();
    Class[] types = signature.getParameterTypes();
    Object[] args = thisJoinPoint.getArgs();
    receiver.invoke(passer.context, methodName, types, args);
  }
}
```

AspectJ Ant build.xml

```
<project name="AspectJDemo" default="run">
  <property name="aspectj.home" value="C:\Java\AOP\AspectJ\aspectj1.1"/>
  <property name="build.dir" value="classes"/>
  <property name="src.dir" value="src"/>

  <path id="classpath">
    <pathelement location="${build.dir}"/>
    <fileset dir="${aspectj.home}/lib" includes="*.jar"/>
  </path>

  <taskdef name="ajc" classname="org.aspectj.tools.ant.taskdefs.AjcTask"
    classpath="${aspectj.home}/lib/aspectjtools.jar"/>

  <target name="clean">
    <delete dir="${build.dir}"/>
  </target>
```

AspectJ Ant build.xml (Cont’d)

```
  <target name="compile" depends="prepare">
    <ajc srcdir="${src.dir}" destdir="${build.dir}”
      classpath="${aspectj.home}/lib/aspectjrt.jar"/>
  </target>

  <target name="prepare">
    <mkdir dir="${build.dir}"/>
  </target>

  <target name="run" depends="clean,compile">
    <java classname="com.agedwards.bank.Demo"
      classpathref="classpath" fork="yes"/>
  </target>

</project>
```

AspectJ Aspect Browser - ajbrowser

- Simple IDE that shows where aspects are used
- Requires a “build file”
  - just a text file with the path to each aspect and Java source file on separate lines
  - typically has “.lst” extension
- To launch the browser
  - ajbrowser {*build-file*}

build file example

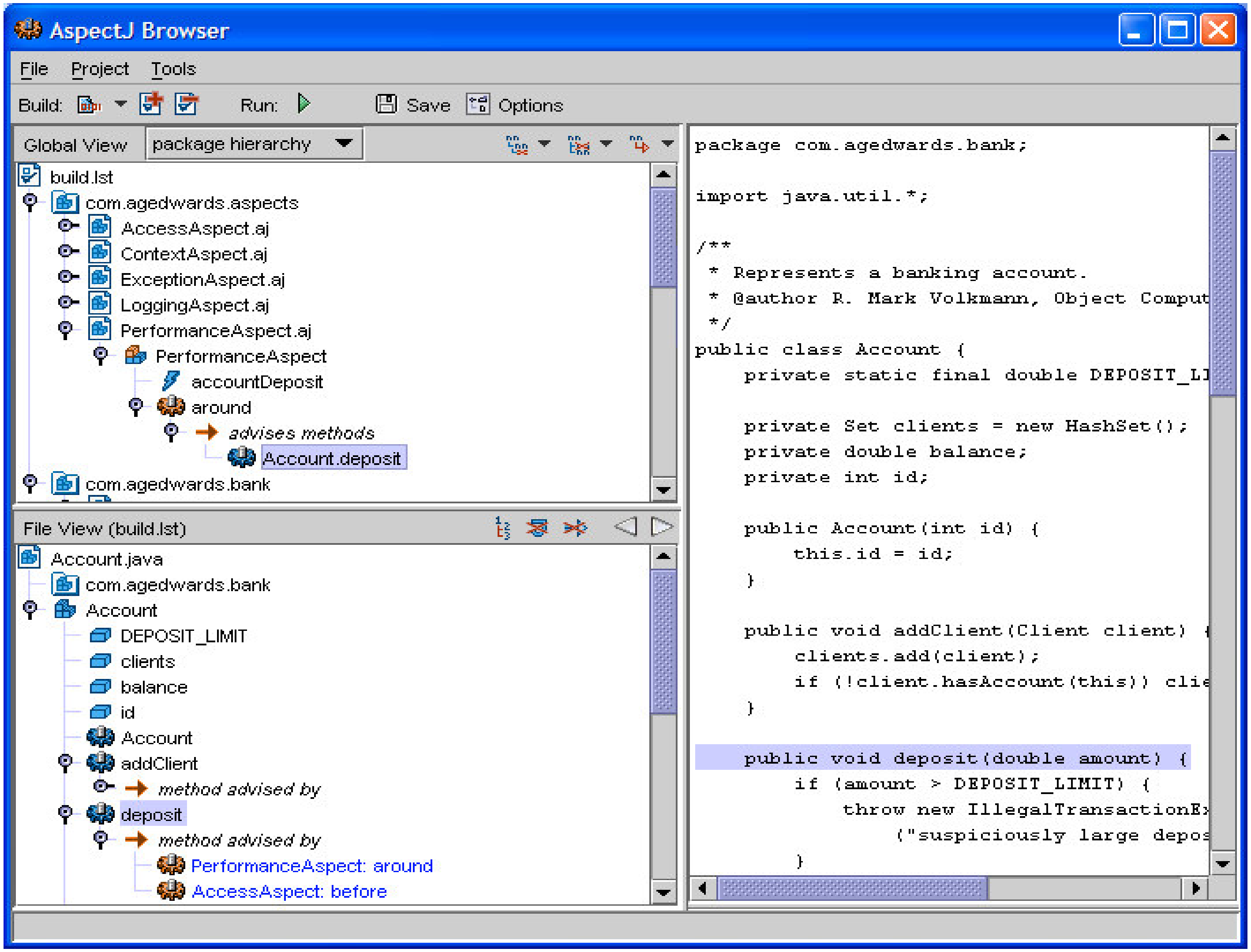
```
src/com/agedwards/aspects/PerformanceAspect.aj
src/com/agedwards/bank/Account.java
```

AspectJ Aspect Browser - ajbrowser (Cont’d)

In the **upper-left pane**, PerformanceAspect.aj is expanded to show that it affects the deposit method in the Account class.

Clicking on the “Account.deposit” causes the source code to be displayed in the **right pane**.

The **lower-left pane** shows that the deposit method is advised by both PerformanceAspect and AccessAspect.



```
package com.agedwards.bank;

import java.util.*;

/**
 * Represents a banking account.
 * @author R. Mark Volkmann, Object Comput
 */
public class Account {
    private static final double DEPOSIT_LI

    private Set clients = new HashSet();
    private double balance;
    private int id;

    public Account(int id) {
        this.id = id;
    }

    public void addClient(Client client) {
        clients.add(client);
        if (!client.hasAccount(this)) clie
    }

    public void deposit(double amount) {
        if (amount > DEPOSIT_LIMIT) {
            throw new IllegalTransactionE
                ("suspiciously large depos
        }
```

AspectWerkz

seems to be the most popular alternative to AspectJ

- Open-source AOP framework started by Jonas Bonér
  - available at http://aspectwerkz.codehaus.org
- Uses run-time bytecode weaving
  - unlike AspectJ, doesn't require a special compiler
- How are aspects specified?
  - aspect are specified using an XML configuration file ← typically named `aspectwerkz.xml`
  - advice is specified with normal Java interfaces and classes
  - when using introductions, a "weave model" must be produced
    - a tool to create these is provided (along with a custom Ant task to invoke it)
    - more on next page
  - the application must be executed using a supplied script ← `aspectwerkz.bat`
    - uses `org.cs3.jmangler.offline.starter.Main` to weave bytecode as it is loaded into the JVM

AspectWerkz (Cont'd)

- Meta-data
  - allows arbitrary objects to be attached to others using Map-like syntax
  - alternative to adding a field using introduction

```
((MetaDataEnhanceable) target).___AW_addMetaData(key, value);
Object value = ((MetaDataEnhanceable) target).___AW_getMetaData(key);
```

- Weave models
  - serialized objects that contain data needed by the bytecode weaver at application startup
  - required when introductions or meta-data is used
  - created by a separate step in the build process using SourceFileMetaDataCompiler or ClassFileMetaDataCompiler
    - see example build.xml later
- Run-time library size - 2082K
  - aspectwerkz, bcel, commons-jexl, concurrent, dom4j, jisp, jmangler, qdox, trove

AspectWerkz aspectwerkz.xml

```
<aspectwerkz>
  <advice-def name="accessAdvice"
    class="com.agedwards.advice.AccessAdvice"/>
  <advice-def name="contextAdvice"
    class="com.agedwards.advice.ContextAdvice"/>
  <advice-def name="exceptionAdvice"
    class="com.agedwards.advice.ExceptionAdvice"/>
  <advice-def name="performanceAdvice"
    class="com.agedwards.advice.PerformanceAdvice"/>

  <introduction-def name="contextPasser"
    interface="com.agedwards.bank.ContextPasser"
    implementation="com.agedwards.bank.ContextPasserImpl"
    deployment-model="perInstance"/>

  <introduction-def name="contextReceiver"
    interface="com.agedwards.bank.ContextReceiver"
    implementation="com.agedwards.bank.ContextReceiverImpl"
    deployment-model="perInstance"/>
```

associate advice names with advice classes

associate introduction names with introduction interfaces and implementation classes

AspectWerkz aspectwerkz.xml (Cont’d)

Logs calls to all methods in the Account class

```
<aspect name="accessAspect">
  <pointcut-def name="methods" type="callerSide"
    pattern="*->* com.agedwards.bank.Account.*(..)"/>
  <advice pointcut="methods">
    <advice-ref name="accessAdvice"/>
  </advice>
</aspect>
```

the first * in this pattern represents the caller type

In the AspectJ example, these calls were intercepted inside the called method. Here they are intercepted in the caller just to demonstrate another alternative.

AspectWerkz aspectwerkz.xml (Cont'd)

```
<aspect name="exceptionAspect">
  <pointcut-def name="methods" type="throws"
    pattern="void com.agedwards.bank.Demo.run()#*"/>
  <advice pointcut="methods">
    <advice-ref name="exceptionAdvice"/>
  </advice>
</aspect>
```

Logs all exceptions thrown out of the run method of the Demo class

represents any kind of exception

```
<aspect name="performanceAspect">
  <pointcut-def name="methods" type="method"
    pattern="* com.agedwards.bank.Account.deposit(..)"/>
  <advice pointcut="methods">
    <advice-ref name="performanceAdvice"/>
  </advice>
</aspect>
```

Logs the elapsed time for all calls to the deposit method in the Account class

AspectWerkz aspectwerkz.xml (Cont’d)

Logs all calls to the deposit method in the Account class including data in the current Context object

```
  <aspect name="contextAspect">
    <introduction class="com.agedwards.bank.Demo">
      <introduction-ref name="contextPasser"/>
    </introduction>
    <introduction class="com.agedwards.bank.Account">
      <introduction-ref name="contextReceiver"/>
    </introduction>

    <pointcut-def name="methods" type="method"
      pattern="* com.agedwards.bank.Account.deposit(..)"/>
    <advice pointcut="methods">
      <advice-ref name="contextAdvice"/>
    </advice>
  </aspect>

</aspectwerkz>
```

adds a “`context`” field to the Demo class

adds an “`invoke`” method to the Account class (see page 43)

AspectWerkz AccessAdvice.java

Logs call to the method associated with the given JoinPoint

```
package com.agedwards.advice;

import org.codehaus.aspectwerkz.advice.PreAdvice;
import org.codehaus.aspectwerkz.joinpoint.CallerSideJoinPoint;
import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class AccessAdvice extends PreAdvice {

  public void execute(JoinPoint joinPoint) throws Throwable {
    CallerSideJoinPoint cjp = (CallerSideJoinPoint) joinPoint;
    System.out.println("Access: " + cjp.getCalleeClassName() +
      " method " + cjp.getCalleeMethodName() + " was called");
  }
}
```

AspectWerkz ExceptionAdvice.java

Logs exeception thrown from the given JoinPoint

```
package com.agedwards.advice;

import org.codehaus.aspectwerkz.advice.ThrowsAdvice;
import org.codehaus.aspectwerkz.joinpoint.JoinPoint;
import org.codehaus.aspectwerkz.joinpoint.ThrowsJoinPoint;

public class ExceptionAdvice extends ThrowsAdvice {

  public void execute(JoinPoint joinPoint) throws Throwable {
    ThrowsJoinPoint tjp = (ThrowsJoinPoint) joinPoint;
    System.out.println
      ("EXCEPTION: " + tjp.getException().getMessage());
  }
}
```

AspectWerkz PerformanceAdvice.java

Logs elapsed time to execute the method associated with the given JoinPoint

```
package com.agedwards.advice;

import org.codehaus.aspectwerkz.advice.AroundAdvice;
import org.codehaus.aspectwerkz.joinpoint.JoinPoint;
import org.codehaus.aspectwerkz.joinpoint.MethodJoinPoint;

public class PerformanceAdvice extends AroundAdvice {
```

AspectWerkz PerformanceAdvice.java (Cont’d)

```
  public Object execute(JoinPoint joinPoint) throws Throwable {
    long startTime = System.currentTimeMillis();
    Object result = joinPoint.proceed();
    long stopTime = System.currentTimeMillis();
    long elapsedTime = stopTime - startTime;

    MethodJoinPoint mjp = (MethodJoinPoint) joinPoint;
    String targetMethod =
      mjp.getTargetClass().getName() + "." + mjp.getMethodName();
    System.out.println
      ("Perf: " + targetMethod + ' ' + elapsedTime + "ms");

    return result;
  }
}
```

AspectWerkz ContextAdvice

Logs call to the method associated with the given JoinPoint

```
package com.agedwards.advice;

import org.codehaus.aspectwerkz.advice.AroundAdvice;
import org.codehaus.aspectwerkz.joinpoint.JoinPoint;
import org.codehaus.aspectwerkz.joinpoint.MethodJoinPoint;
import com.agedwards.bank.*;

public class ContextAdvice extends AroundAdvice {
```

AspectWerkz ContextAdvice (Cont'd)

intercepts all deposits and passes the data needed to invoke the real method, along with associated Context, to the real target (see invoke method on page 43)

```
public Object execute(JoinPoint joinPoint) throws Throwable {
  ContextReceiver receiver =
     (ContextReceiver) joinPoint.getTargetObject();

   MethodJoinPoint mjp = (MethodJoinPoint) joinPoint;
   String methodName = mjp.getMethodName();
   Class[] types = mjp.getParameterTypes();
   Object[] args = mjp.getParameters();

   ContextPasser passer = null;
   return receiver.invoke
     (passer.getContext(), methodName, types, args);
  }
}
```

In the current version of AspectWerkz (0.7) there is no way to determine the calling object in an AroundAdvice, so **this code doesn't work!** The author is working on adding this capability.

AspectWerkz ContextPasser Introduction

- ContextPasser.java

```
package com.agedwards.bank;

public interface ContextPasser {
  Context getContext();
}
```

- ContextPasserImpl.java

```
package com.agedwards.bank;

public class ContextPasserImpl implements ContextPasser {
  private Context context;

  public ContextPasserImpl(Bank bank, Teller teller) {
    context = new Context(bank, teller);
  }

  public Context getContext() { return context; }
}
```

AspectWerkz ContextReceiver Introduction

- ContextReceiver.java

```
package com.agedwards.bank;

public interface ContextReceiver {
  Object invoke(Context context, String methodName,
                Class[] types, Object[] args);
}
```

- ContextReceiverImpl.java

```
package com.agedwards.bank;

import java.lang.reflect.*;

public class ContextReceiverImpl implements ContextReceiver {
```

Logs calls to the method associated with the given JoinPoint, including data in the given Context object

continued on next page

AspectWerkz ContextReceiver Introduction (Cont'd)

```
  public Object invoke(Context context, String methodName,
                       Class[] types, Object[] args) {
    Class clazz = getClass();
    System.out.println("Context: " + clazz.getName() +
      " method " + methodName + " called, context = " + context);

    Object result = null;

    try {
      Method method = clazz.getMethod(methodName, types);
      result = method.invoke(this, args);
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(1);
    }

    return result;
  }
}
```

invokes the specified method using reflection

AspectWerkz Ant build.xml

```
<project name="AspectWerkzDemo" basedir="." default="run">
  <property environment="env"/>
  <property name="aspectwerkz.script"
    value="${env.ASPECTWERKZ_HOME}/bin/aspectwerkz.bat"/>
  <property name="build.dir" value="classes"/>
  <property name="definition.file" value="aspectwerkz.xml"/>
  <property name="metadata.dir" value="${build.dir}"/>
  <property name="src.dir" value="src"/>

  <path id="classpath">
    <pathelement location="${build.dir}"/>
    <fileset dir="${env.ASPECTWERKZ_HOME}/lib" includes="*.jar"/>
  </path>

  <taskdef name="compileWeaveModelFromSources"
    classname="org.codehaus.aspectwerkz.task.SourceFileMetaDataCompilerTask"
    classpathref="classpath"/>
```

script used to run application

where weave model will be generated

AspectWerkz Ant build.xml (Cont’d)

```
<target name="clean">
  <delete dir="${build.dir}"/>
</target>

<target name="compile" depends="prepare">
  <javac srcdir="${src.dir}" destdir="${build.dir}"
    classpathref="classpath" deprecation="on" debug="on"/>

  <!-- This is required when using introductions or metadata. -->
  <compileWeaveModelFromSources definitionFile="${definition.file}"
    sourceDir="${src.dir}" metaDataDir="${metadata.dir}"
    uuid="${ant.project.name}"/>
</target>

<target name="prepare">
  <mkdir dir="${build.dir}"/>
</target>
```

AspectWerkz Ant build.xml (Cont'd)

```
  <target name="run" depends="clean,compile">
    <property name="cp" refid="classpath"/>
    <exec executable="${aspectwerkz.script}">
      <arg line="-Daspectwerkz.metadata.dir=${metadata.dir}"/>
      <arg line="-cp ${cp}"/>
      <arg line="com.agedwards.bank.Demo"/>
    </exec>
  </target>

</project>
```

Nanning

- Open-source AOP framework started by Jon Tirsen
  - available at http://nanning.codehaus.org
- Uses dynamic proxies
  - clients of instrumented objects must use special code to obtain them
    - use of the factory pattern is suggested
  - can only instrument classes that implement some interface
  - these issues limit the applicability of the framework
- Run-time library size - 1449K
  - commons-beanutils, commons-collections, commons-digester, commons-jelly, commons-logging, concurrent, dom4j, nanning, nanning-contract, nanning-locking, nanning-profiler, prevayler, qdox

Prose

- Open-source AOP framework started by Andrei Popovici
  - available at http://prose.ethz.ch
- Uses run-time bytecode weaving
  - happens while the application is running, not just when classes are loaded
- Aspects are specified with normal Java classes
  - these classes must extend one of the following Prose classes
    - CatchCut, GetCut, MethodCut, SetCut and ThrowCut
      - these all extend from AbstractCrosscut which implements Crosscut
- Steps to build and run
  - aspect classes are compiled with a normal Java compiler (such as `javac`)
  - weaving is performed at run-time by invoking

    ```
    ProseSystem.getAspectManager().insert(aspect-object);
    ```

  - must run application with a **Prose-specific JVM**

    ```
    prose -classpath classpath main-class
    ```

may not trust it
for production use

Recommendation

- The recommended AOP framework is AspectJ
- The reasons for this recommendation include
  - maturity compared to other frameworks
  - number of supported features compared to other frameworks
  - promise of upcoming support for run-time bytecode weaving
    - through a custom class loader
  - availability of books on using it
    - Mastering AspectJ - Wiley
    - Aspect-Oriented Programming with AspectJ - SAMS
    - AspectJ in Action - Manning
- Recommended reading
  - “I want my AOP!”, a three-part article at JavaWorld
    - http://www.javaworld.com/javaworld/jw-0118-aspect_p.html