AspectWerkz
Dynamic AOP For Java

Amit Shabtay

Aspect Oriented Programming

- Cross-cutting concerns
  - System wide common handling
  - Security, logging, persistence, synchronization
- Advices
  - The additional code applied to the existing model
- Point-cuts
  - Where the advice should be applied
- Aspects
  - Combinations of point-cuts and advices

Current AOP Implementations

- Focusing on Java
  - Other implementations for .NET, C++, Ruby, Smalltalk and more
- AspectJ
  - Extend the Java language itself
  - Introduce the notion of an aspect for both design and coding.

Java AOP Implementations

- Other implementations
  - Work within Java
  - Dynamic proxies, reflection, bytecode manipulation, custom class loading
  - Aspect, advice and point-cut are plain Java classes
- Several implementations:
  - AspectWerkz
  - JBossAOP
  - Spring framework
  - And more

AspectWerkz

- A dynamic, lightweight AOP framework for Java
- Allows adding aspects to new code as well to existing applications
- Advices can be plain (old) Java objects (POJO)
- http://aspectwerkz.codehaus.org/

Example Advice

```
public Object
  myAroundAdvice(JoinPoint
  joinPoint) throws Throwable {
   // do stuff
   Object result =
     joinPoint.proceed();
   // do more stuff
   return result;
}
```

JoinPoint

- A well-defined point in the program flow
- Provides information on:
  - The signature of the code (method/field/…)
  - RTTI information
- `JoinPoint.proceed()` – telling the program to continue

Advice types

- Around advice
  - Invoked "around" the join point
- After advice
  - Invoked after the join point
- Before advice
  - Invoked before the join point
- `JoinPoint.proceed()` can only be called in around advices, otherwise you have infinite loops

Definitions

- There are two aspect definition models
  - External XML file
  - Javadoc-like attributes
- Using XML
  - … is using well known and supported format
  - … is more complex, no refactoring

Point-cut Types (1)

- `execution(<method or constructor pattern>)`
  - picks out join points defining method (static or member) or constructor execution
- `call(<method or constructor pattern>)`
  - picks out join points defining method (static or member) or constructor call
- `get/set(<field pattern>)`
  - picks out join points defining field access/modification.
- Valid advice for these pointcuts are around , before and after

Point-cut Types (II)

- handler(<exception class pattern>)
  - picks out join points defininging a catch clause.
    Valid advice for this pointcut is before

Point-cut selection pattern language

- Defines where To Add the Advice
- * wildcard
  - One package level or one method parameter
  - **At least one** character (package name)
  - **Zero or more** characters (otherwise)
- .. Wildcard
  - Any sequence of characters that start and end with a ".",
  - `org.codehaus..*` will match all classes in all subpackages starting from org.codehaus

How To Choose Classes?

- `foo.bar.*` - will match `foo.bar.FooBar2` as well as `foo.bar.FooBear`
- `foo.*.FooBar` - will match `foo.bar.FooBar` as well as `foo.bear.FooBar` but not `foo.bear.FooBar2`
- `foo.*.FooB*` - will match `foo.bar.FooBar2` as well as `foo.bear.FooBear` as well as `foo.bear.FooB`

How To Choose Fields?

- `int foo.*.Bar.m_foo` - will match `int m_foo` but not `int s_foo` or `long m_foo`
- `* foo.*.Bar.m_foo` - will match `int m_foo` as well as `java.lang.String m_foo`
- `java.lang.* foo.*.Bar.m_foo` - will match `java.lang.String m_foo` as well as `java.lang.StringBuffer m_foo`

How To Choose Methods? (I)

- `int foo.*.Bar.method()` - will match `int method()` but not `int method(int i)`
- `int *.method(*)` - will match `int Foo.method(int i)` but not `int Foo.method()`

How To Choose Methods? (II)

- `int foo.*.*.method(*,int)` - will match `int method(String s, int i)` as well as `int method(int i1, int i2)`
- `int foo.*.Bar.method(..)` - will match `int method()` as well as `int method(String s, int i)` as well as `int method(int i, double d, String s, Object o)`

How To Choose Methods? (III)

- There are many more examples
- Variation of return type, method name and parameters
- Check http://aspectwerkz.codehaus.org/definition_issues.html

What About Constructors?

- Use "new" as method name
- `foo.*.Bar.new()` - will match `new Bar()` but not `new Bar(int i)`
- `*.new(String)` - will match `new Foo(String name)` and `new Bar(String name)` but not `new Foo()`

Subtyping

- As we saw, the pattern `* foo.Bar.*(..)` will match any method declared in foo.Bar
- What about foo.Bar2 who extend foo.Bar?
- The pattern is `* foo.Bar+.*(..)`

Point-cut composition

- Using the logical operators &&, ||, ! and ()
- `execution(* foo.bar.Baz.*(..)) || call(* foo.bar.Baz.*(..))`
- `(set(* foo.bar.*) || get(* foo.bar.*)) && cflow(* foo.bar.Buzz.(..))`
- `handler(java.lang.Exception+) && !cflow(* foo.bar.Buzz.(..))`

Example Code

```
/** @Aspect perInstance */
public class LoggingAspect extends Aspect {
    /** @Call * foo.bar.*.*(..) */
    Pointcut logMethodCall;

    /** @Execution * foo.baz.*.*(..) */
    Pointcut logMethodExecution;

    /** @Before logMethodCall */
    public void logEntry(JoinPoint joinPoint) { ... }

    /** @After logMethodCall */
    public void logExit(JoinPoint joinPoint) { ... }

    /** @Around logMethodExecution */
    public Object logExecution(JoinPoint joinPoint) { ... }
}
```


Equivalent XML (I)

```
<aspectwerkz>
  <system id="sample">
    <aspect class="examples.LoggingAspect">
      <pointcut name="logMethodCall"
                expression="call(*
                          foo.bar.*.*(..))"/>
      <pointcut name="logMethodExecution"
                expression="execution(*
                          foo.baz.*.*(..))"/>
```


Equivalent XML (II)

```
      <advice name="logEntry"
              type="before"
              bind-to="logMethodCall"/>
      <advice name="logExit"
              type="after"
              bind-to="logMethodCall"/>
      <advice name="logExecution"
              type="around"
              bind-to="logMethodExecution"/>
    </aspect>
  </system>
</aspectwerkz>
```


Caching Example

```
/**
 * @Around execution(int
   examples.caching.Pi.getPiDecimal(int))
 */
public Object cache(final JoinPoint joinPoint)
  throws Throwable {
  MethodRtti rtti = (MethodRtti)joinPoint.getRtti();
  final Long hash = new Long(calculateHash(rtti));
  final Object cachedResult = m_cache.get(hash);
  if (cachedResult != null) {
    return cachedResult;
  }
  final Object result = joinPoint.proceed();
  m_cache.put(hash, result);
  return result;
}
```

Exception Handling Example

```
public class ExceptionHandlingAspect {

  /**
   * @Before handler(java.lang.Exception)
   */
  public void logEntry(final JoinPoint joinPoint)
    throws Throwable {
    CatchClauseRtti rtti =
      (CatchClauseRtti)joinPoint.getRtti();
    Exception e = (Exception)rtti.getParameterValue();
    System.out.println(
       "[From advice] exception catched:" +
       e.toString());
  }
}
```


How To Use It?

- How do we add aspect to existing code?
- There are two modes:
  - Online – "using" AspectWerkz as the virtual machine (instead of java)
  - Offline – the class file are post-processed and the aspects are added

Offline mode (I)

- Command line tool:
- `aspectwerkz -offline <definition file> <options> [-cp <classpath>]* <target to transform>+`

Offline mode (II)

- Instead of `aspectwerkz` you can write "`java org.codehaus.aspectwerkz.compiler.AspectWerkzC`"
- Why this is important?
  - Ant task

Where can you find it?

- `~ood/aspectwerkz`
- Set the `ASPECTWERKZ_HOME` environment variable
- Add `$ASPECTWERKZ_HOME/lib` to the classpath
- Add `$ASPECTWERKZ_HOME/bin` to `PATH`

More Information

- There many examples in `~ood/aspectwerkz/src/samples`
  - run them and see that you understand them
- http://aspectwerkz.codehaus.org/
- http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html