Aspect Oriented Programming

2013-2014

Course 10

Course 10 Contents

- AspectJ Weaving Models

AspectJ weaving models

- Weaving composes classes and aspects into an executable system.
- AspectJ weaving models can be classified based on when it performs weaving and which kind of input it processes.
- Based on when it performs weaving:
  - Build-time weaving weaves classes and aspects together during the build process before deploying the application.
  - Load-time weaving (LTW) weaves just in time as the classes are loaded by the VM, eliminating any pre-deployment weaving.
- Based on the kind of input a weaver processes:
  - Source code weaving accepts input in the source-code form.
  - Byte-code (binary) weaving that accepts input in byte code form produced by a compiler.

Weaving models supported by AspectJ

Weave time	Weaver input	
	Source code	**Byte code (binary)**
Build-time	Yes	Yes
Load-time	Source using XML syntax only	Yes

Weaving models supported by AspectJ

- AspectJ supports build-time weaving that can take either source code or byte code.
- For LTW, the primary supported input is the byte code. LTW supports source code in a limited manner for aspects expressed in XML syntax.
- AspectJ lets you combine these weaving models.
- You may have a few aspects woven in using source-code build-time weaving and others (eg, third-party aspects) using byte-code build-time weaving. Then, you can use XML-based aspects using source-code load-time weaving.
- You may also weave in aspects (such as third-party aspects) using the byte-code load-time weaver.

Build-time weaving

- The AspectJ compiler, `ajc,` enables build-time weaving.
- It can take input in the form of source files, class files, and jar files, each of which may contain classes and aspects.
- The compiler produces woven byte code (as class files or as a jar depending on the compiler options).
- You can then deploy the resulting byte code in any standard compliant VM.
- You do not need to make any changes in deployment, except for adding aspectjrt.jar to the classpath.
- The `aspectjrt.jar` file contains definitions for the various user accessible AspectJ types such as `JoinPoint` and `Signature`, various @AspectJ annotations, as well as internal AspectJ types used during runtime.

Build-time weaving

- Build-time weaving is most suitable when learning AOP.
- Build-time weaving is the only choice when it comes to certain static crosscutting usages.

Eg. An aspect that introduces a method to a class using an inter-type declaration.

The method must be woven in to successfully compile the classes that use the introduced method.

- Waiting until LTW will lead to compilation errors in the accessing classes aborting any further steps.
- Build-time weaving is also useful for refactoring aspects, where the classes often embed aspects.
- Build-time weaving can work with either source or binary code.

Build-time source code weaving

- In weaving mechanism, the weaver accepts classes and aspects in the source format (written using the traditional syntax or the @AspectJ syntax).
- The compiler produces byte code, which can be executed in any standard VM.
- This option is similar to using javac to compile Java sources.
- Using build-time source-code weaving essentially involves replacing javac with ajc. Most options available to javac are available to ajc.

Eg. to weave the tracing logic in all banking classes, you supply sources as follows:

```
> ajc ajia\banking\*.java ajia\tracing\TraceAspect.aj
```

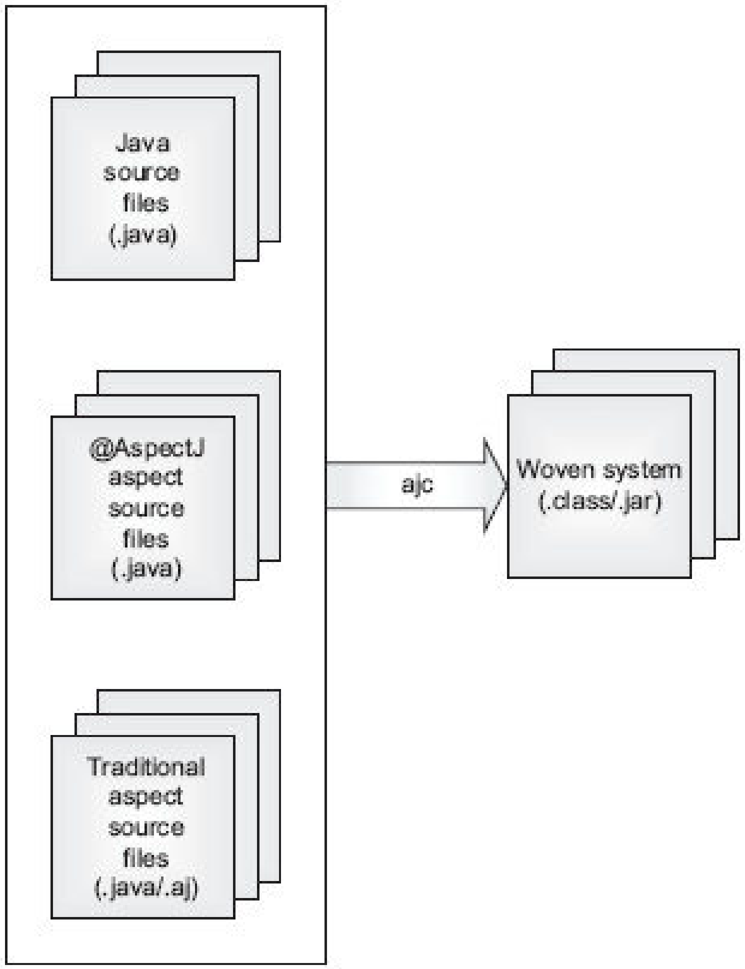
Build-time source code weaving

- javac vs. ajc:
  - With javac, you can compile all source files together or each source file individually without any difference in the output.
  - With ajc, you must pass all the input files together in one invocation.
  - The following two commands will not produce the same result as the command: `> ajc ajia\banking\*.java ajia\tracing\TraceAspect.aj`

```
> ajc ajia\banking\*.java
> ajc ajia\tracing\TraceAspect.aj
```

- Internally, source weaving utilizes binary weaving.
- Even when you present source files as input to the compiler, the AspectJ compiler first compiles the code into bytecode form and then weaves the resulting byte code together.

Build-time source code weaving


Java source files (.java)
@AspectJ aspect source files (.java)
Traditional aspect source files (.java/.aj)
ajc
Woven system (.class/.jar)

Build-time binary weaving

- Often, you do not have access to the source code for the classes or aspects or both (eg. when using a third-party library for classes or aspects).
- AspectJ’s build-time binary weaving offers a solution in such cases. With this option, you perform the following steps:
    - *Use already-compiled classes.* These classes may have been compiled using either a Java compiler or the AspectJ compiler.
    - *Use already-compiled aspects.* Aspects written using the @AspectJ syntax may have been compiled using either the Java compiler or the AspectJ compiler. As for classes, using the Java or AspectJ compiler makes no logical difference.You must use the –g:vars or –g option, when compiling aspects using javac. Aspects written using the traditional syntax must have been compiled using the AspectJ compiler, because javac does not understand the syntax.
    - *Weave classes with aspects by presenting their byte code to another invocation of ajc, which produces a set of classes or a jar file depending on the options provided.* Any standard VM may then execute the resulting byte code.

Build-time binary weaving

- From a usage perspective, the difference between source code weaving and binary weaving is minor.
- With either approach, you still use the ajc compiler. The only difference is the flags that specify sources or byte-code input.
- The following command uses the classes in services.jar as input to be woven in with aspects in monitoring.jar:

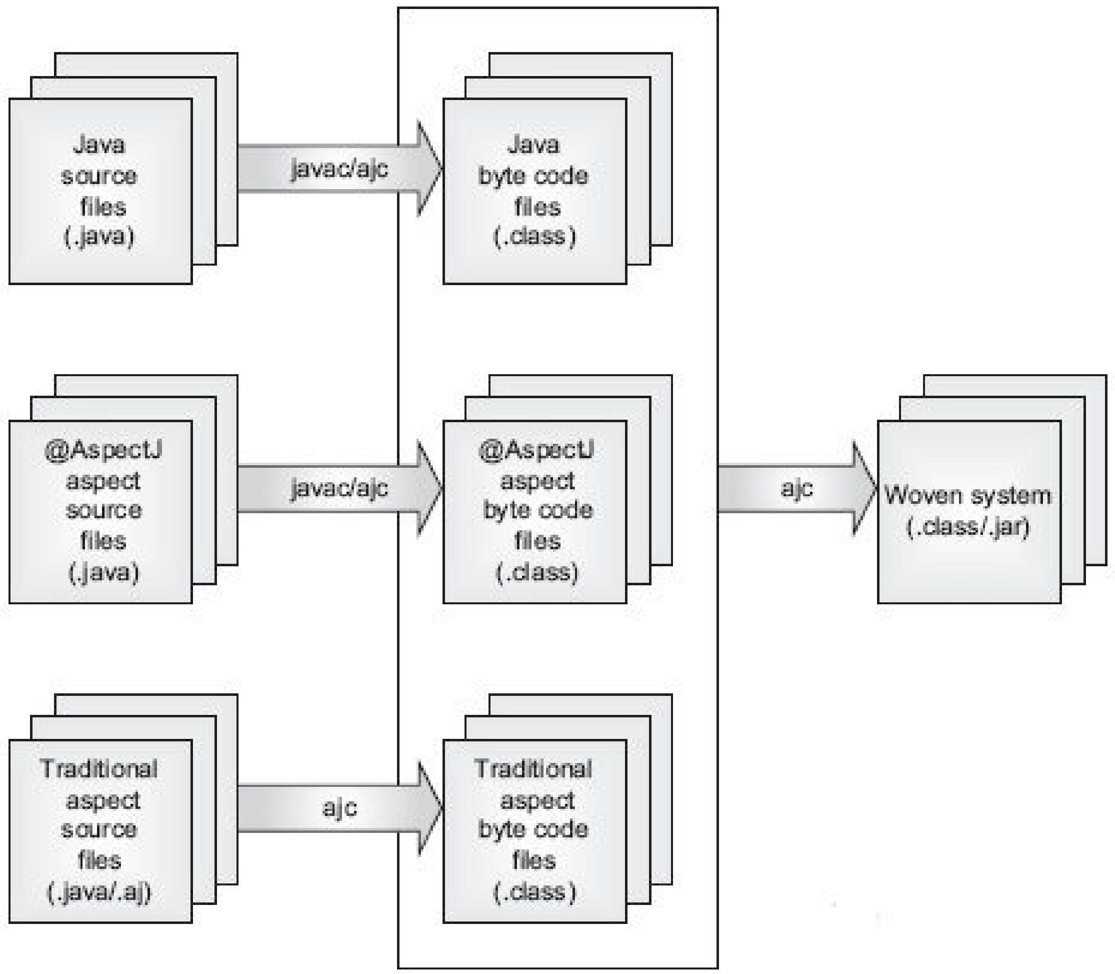
```
> ajc -inpath services.jar -aspectpath monitoring.jar
```

- You can combine source-code and binary weaving in a single command.

Eg. Use source-code weaving for files with a `.java` extension in the banking directory and a `.aj` extension in the tracing directory along with some jar files:

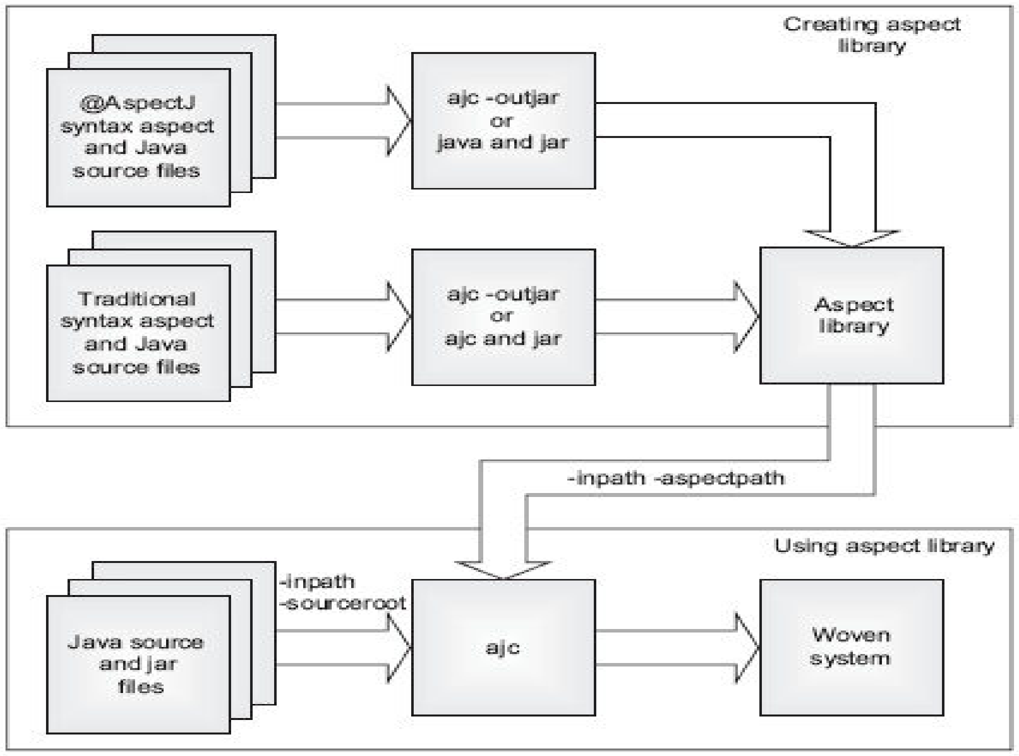
```
>ajc ajia\banking\*.java tracing\*.aj -inpath services.jar
       -aspectpath monitoring.jar
```

Build-time binary weaving


Java source files (.java)
javac/ajc
Java byte code files (.class)
@AspectJ aspect source files (.java)
javac/ajc
@AspectJ aspect byte code files (.class)
Traditional aspect source files (.java/.aj)
ajc
Traditional aspect byte code files (.class)
ajc
Woven system (.class/.jar)

Build-time binary weaving

- Creating an aspect library involves using `-outjar` or explicitly using the jar tool. When you use the library, you need the `-aspectspath` and `-inpath` options. The output of the system can be multiple class files or a jar file.

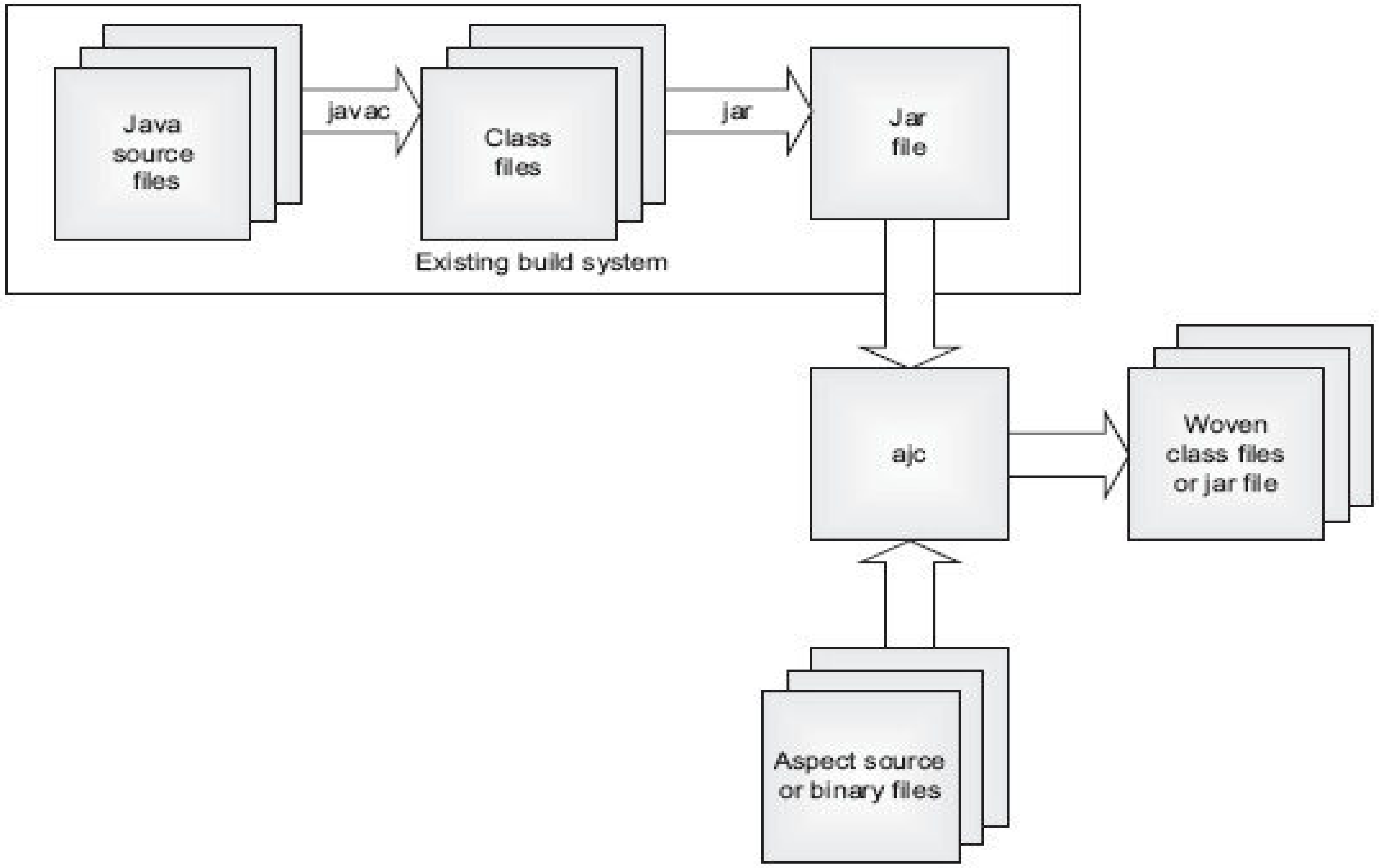

Creating aspect library
@AspectJ syntax aspect and Java source files
ajc -outjar or java and jar
Traditional syntax aspect and Java source files
ajc -outjar or ajc and jar
Aspect library
-inpath -aspectpath
Using aspect library
Java source and jar files
-inpath -sourceroot
ajc
Woven system

Build-time binary weaving

- Build-time weaving can be easy or difficult to start with.
- Easier:
  - You are just replacing `javac` with `ajc`. If you use build-time binary weaving, you can even leave your build system or IDE setup intact except for an additional post-compilation step.
  - When it comes to deployment, except for including `aspectjrt.jar` in the classpath, you can ignore the fact that the system was built using the AspectJ compiler.
- Difficult:
  - It affects how you build your system; and, for a certain class of crosscutting concerns, it may be a bit difficult.

Build-time binary weaving

- Augmenting the existing build system to weave in the aspects



Load time weaving

- Load-time weaving (LTW) provides a way to leave your build system intact and still weave aspects as needed.

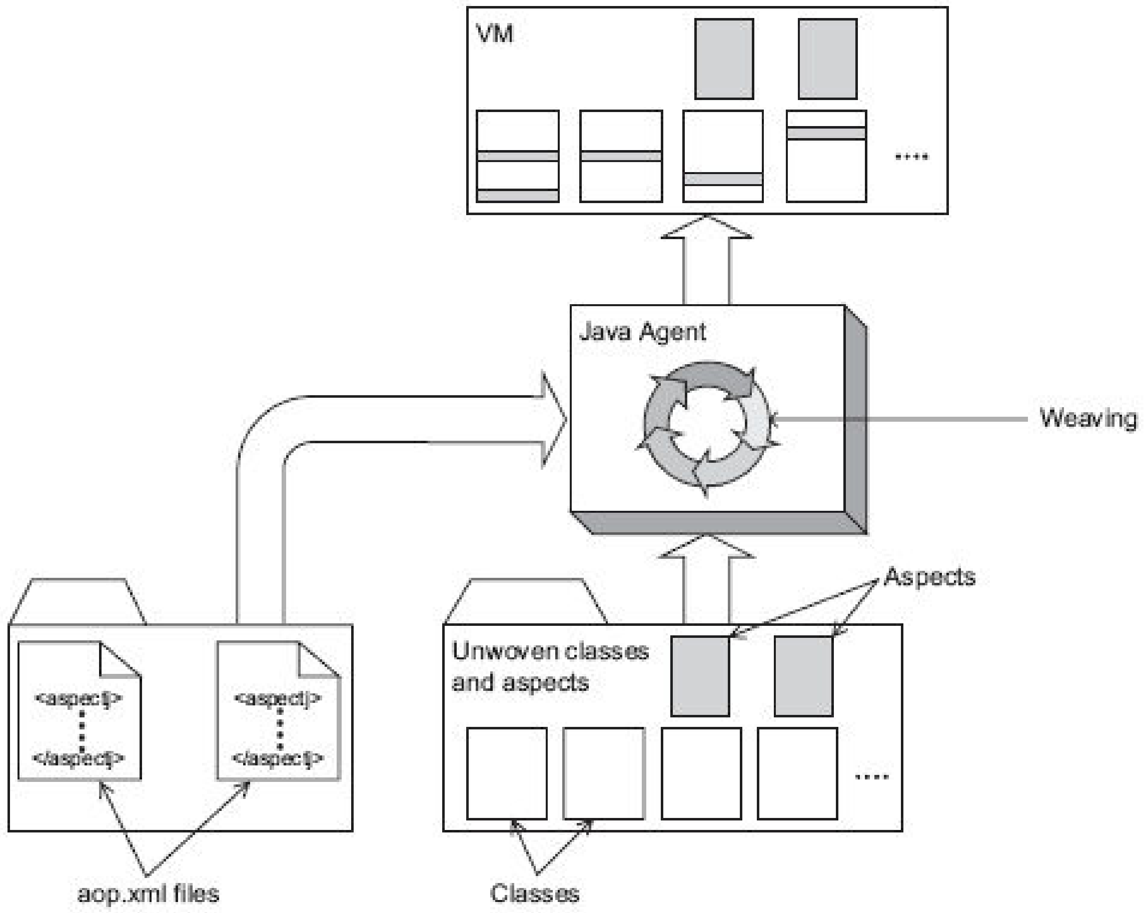
Eg. aspects to monitor the behavior of your application to see if there are any performance bottlenecks, that will not be part of the final system.

- The mechanism used for LTW is an extension of binary weaving.
- LTW takes byte code as input. Unlike binary weaving, there is no explicit weaving step. Instead, the VM is set up to perform weaving as the VM loads the classes.
- The load-time weaver uses the Java Virtual Machine Tools Interface (JVMTI) facility—a new feature introduced in Java 5.
- JVMTI allows a JVMTI agent to, in addition to many other things, intercept the loading of a class. The load-time weaver is a JVMTI agent that uses this functionality to weave in classes as the VM loads them. This agent works only with Java 5 and newer VMs.

Load time weaving

- The load-time weaver needs additional information to decide which aspects to weave.
- The weaver requires that you explicitly specify each aspect in an aop.xml file in the META-INF directory on the classpath.
- Because many classpath entries (jar files and directories) may contain an aop.xml file, the weaver uses them all by combining the information in them.

Load time weaving



Load time weaving

- The steps performed when using LTW:
  1. You deploy an application to use the load-time weaver by using the `-javaagent` option to java to specify `aspectjweaver.jar` as an agent, as follows:

```
> java -javaagent:<path-to>/aspectjweaver.jar [other options]
  <Main-Class>
```

  2. The VM initializes the agent. During initialization, the agent loads all the files matching META-INF/aop.xml in the classpath. It examines each aop.xml file for the list of aspects. It also examines any inclusion or exclusion filter specified. If the agent finds multiple aop.xml files in META-INF directory on the classpath, it logically combines them.
  3. The agent loads the listed aspects that also match the inclusion and exclusion conditions.
  4. The agent registers its interest in the class-loading event so that the VM gives it an opportunity to examine and possibly modify a class being loaded.

Load time weaving

- The steps performed when using LTW:
   5. The system starts its normal execution. The VM then loads classes as needed during execution.
   6. The VM notifies the agent whenever it loads a class. The agent, in turn, inspects the class to determine if any of the loaded aspects need to be woven in. If so, it weaves the class and hands over the modified byte code to the VM.
   7. The VM uses the woven byte code to realize the class.
- LTW has a definite impact on load-time performance.
- The time to load the application and the memory consumed by it are higher.
- After a class is loaded into the VM, there is no additional performance penalty. The woven byte code produced by LTW is identical to the byte code you would obtain using build-time weaving.

Configuring the load-time weaver

- An important component of LTW is the aop.xml files that configure the weaver.
- The LTW configuration specifies the aspects and classes participating in the weaving process, the definitions of pointcuts for abstract aspects, and various debugging options.
- The XML files are a kind of source code useful only for LTW.
- Except for the XML files, all other input must be first compiled using the Java or AspectJ compiler.

Specifying Aspects To Be Woven

- XML syntax offers a simple element for supplying a reusable concrete aspect to the load-time weaver.

```
<aspectj>
  <aspects>
      <aspect name="tracing.Tracing"/>
     <aspect name="transaction.TransactionManagementAspect"/>
     ...
   </aspects>
</aspectj>
```

The `<aspects>` section specifies the aspects to be woven. It is analogous to the `-aspectpath` compiler option.

Each `<aspect>` section specifies an aspect to be woven.

The `name` attribute of the element is a fully qualified type name.

Remark: A type pattern is not allowed as an attribute value.

Defining Concrete Aspects

- Aspect libraries often contain abstract aspects. You must define concrete subaspects for those aspects to be woven in.
- You can use code to define subaspects and use the `<aspect>` elements, but an easier solution is to define concrete aspects using the `<concrete-aspect>` element.

```
<aspectj>
 <aspects>
<!-- Assume monitoring.Monitoring is an abstract aspect with an
  abstract monitored() pointcut -->
   <concrete-aspect name="monitoring.JDBCMonitoring‟
  extends="monitoring.Monitoring">
<pointcut name="monitored‟ expression="call(* java.sql.*.*(..))"/>
</concrete-aspect>
...
</aspects>
...
</aspectj>
```

Defining Concrete Aspects

- The use of XML to define pointcuts offers an easy way to modify the definition and affect system behavior without recompilation.

Eg if you need to monitor the execution of methods in `banking` classes along with JDBC, you change the pointcut to the following definition:

```
<pointcut name="monitored" expression="call(* java.sql.*.*(..))
 || execution(* ajia.banking..*.*(..))"/>
```

- You can replace the binary operators || and && with OR and AND for pointcuts written using XML. This substitution is especially helpful for writing the && operator.
- XML syntax is purposefully limited to define subaspects and pointcuts in those aspects.
- Unlike with traditional or @AspectJ syntax, you cannot define or override methods in the base aspect.

Defining Concrete Aspects

- Pointcuts specified in aop.xml also have a few limitations:
  - *The pointcuts cannot collect a join point's context*. You cannot use an `args()` pointcut to collect arguments to a method. This limitation makes aspects expressed using XML suitable only for scoping purposes.

  Eg., you may have a base class that defines a pointcut to do the primary selection along with an abstract-scope pointcut. You can then provide the definition for the scope pointcut in XML.
  - *The pointcut must use fully qualified type names similar to @AspectJ syntax and for the same reason: XML syntax does not support a feature comparable to Java import statements*. If you want to select only the `Account` class in the `banking.domain` package, you must specify `banking.domain.Account` as the type name. However, you can always use wildcards to select a set of types as usual.

  Eg., if you use `*..Account` as the type pattern, it matches `banking.domain.Account` and `Account` types defined in other packages, if any.

Defining Aspect Precedence

- Without explicit declarations, the order in which advice from multiple aspects apply is arbitrary.
- AspectJ provides a mechanism to control precedence among multiple aspects.
- Aspect precedence is a system-level consideration specified in a precedence-coordination aspect.
- LTW provides a way to specify aspect precedence in the load-time configuration that already acts at the system level.
- To specify aspect precedence, you define a `<concrete-aspect>` element and specify the precedence sequence in its precedence attribute.

```
<concrete-aspect name="SystemLevelPrecedenceCoordinator"
precedence="security.AuthorizationAspect,
     transaction.TransactionManagementAspect"/>
```

- The concrete aspect specified in this manner is just a placeholder that should not extend any abstract aspect or define any pointcuts.

Specifying Weaving Options

- For more advanced use of LTW.
- To specify aspects and classes participating in the weaving process.
- The `<include>` and `<exclude>` elements nested inside `<aspects>` and `<weaver>` elements help with this.

```
<aspectj>
  <aspects>
    <aspect ....
    <concrete-aspect ....
    <include within="ajia..*"/>
    <include within="org.springframework..*"/>
    <exclude within="@ajia.util.Untested *"/>
    <exclude within="ajia.concurrency.DeadLockDetection+"/>
  </aspects>
<weaver options="-verbose -showWeaveInfo">
   <include within="ajia.banking..*"/>
   <exclude within="org.springframework..*"/>
   <dump within="banking..*" beforeandafter="true"/>
</weaver>
</aspectj>
```

Specifying Weaving Options

- The `<aspects>` element specifies the aspects to weave in.
- The `<weaver>` section specifies the classes to weave into, in addition to controlling other weaver parameters.
- The `<aspects>` section plays a role similar to the `-aspectpath` option to the compiler, whereas the `<weaver>` section plays a role similar to the `-inpath`, `-verbose`, and `-showWeaveInfo` options.
- The optional `<include>` elements in the `<aspects>` section specify the aspects to be considered for weaving.
- Each `<include>` element specifies a type pattern for the `within` attribute.
- A common use of the `<include>` element is to weave only a subset of aspects declared in other aop.xml files.

Eg., the aop.xml in an aspect library may list all its aspects. Without an `<include>` (or `<exclude>`) element, this leads to weaving of all those aspects. Another aop.xml may limit weaving to only a subset of those aspects by using an `<include>` element. In the example, all aspects in the `ajia` and `org.springframework` packages are included as well as their direct and indirect subpackages.

- If you do not specify an `<include>` element, all aspects declared using `<aspect>` and `<concrete-aspect>` are candidates to be woven in.

Specifying Weaving Options

- Optional `<exclude>` sections specify type patterns for the aspects to be excluded.
- A common use of `<exclude>` is to exclude aspects defined in other aop.xml files.
- It serves a purpose similar to that of the `<include>` element, except that it allows exclusion instead of inclusion.
- Only aspects that are listed are woven if they match the types specified in the `<include>` but not those in the `<exclude>` elements.

Example:

  - Deadlock-detection aspects are excluded by specifying a type pattern that selects all subaspects of `DeadLockDetection`.
  - All aspects that carry the `@Untested` annotation are also excluded.

- When you do not specify an `<exclude>` element, all aspects declared using `<aspect>` and `<concrete-aspect>` that match `<include>` elements, if any, are woven in.

Specifying Weaving Options

- The `<weaver>` element controls the classes to be woven in along with a few other characteristics of the weaving process.
- The `<weaver>` element can specify the options passed to the weaver using the options attribute.
- Most of these options are the same as those for the compiler, such as `-verbose`, `--showWeaveInfo`, `-nowarn`, `-XmessageHandlerClass`, and `-Xlint`.
- The `<include>` element specifies a type pattern for the types exposed to the weaver, which is thus available to be woven in.
- By default, the weaver weaves in all types exposed to it.
- Example, all banking-related types are exposed to the weaver.

Specifying Weaving Options

- The `<exclude>` element specifies a type pattern for the types that should not be exposed to the weaver.
- The combination of `<include>` and `<exclude>` control the types exposed to the weaver.
- E.g,, any types from the Spring Framework using the `org.springframework..*` type pattern are excluded.
- By default, the weaver excludes all types in direct or indirect subpackages of `java`, `javax`, and `org.aspectj`.
- The `<dump>` section can specify woven types that should be written to the disk for diagnostic purposes.
- The optional `beforeandafter` parameter specifies whether the unwoven classes should be dumped as well. This may be useful to examine generated classes.

Using Multiple aop.xml Files

- When you have multiple aop.xml files in various components in the classpath, AspectJ’s load-time weaver logically combines all those files.
- The effect is the logical combination, so it may have surprising results.

Eg., if an aop.xml file specifies an `<exclude>` element inside an `<aspects>` element, it excludes any aspect matching the pattern (and not just from the classpath component associated with the aop.xml file that specifies it).

- The recommended practice is that the aop.xml file associated with each library should only list aspects using the `<aspect>` elements and not exclude any aspects using wildcards.
- Excluding specific aspects in the library is not a problem, but using a wildcard can end up choosing aspects in other libraries and may lead to surprises.
- You should use an additional aop.xml file to choose aspects to be woven in (similar to an application-level configuration).

Choosing syntax and weaving

- Which weaving model is best suited: build-time or load-time? Which syntax is the best: traditional or @AspectJ?
- Advantages and disadvantages.
  - *The team is using Eclipse, and AspectJ is a core part of the design.* Use the traditional syntax, because it will lead to the most compact code and provide the full power of AspectJ. You can use build-time weaving, because the plug-in makes its usage easy.
  - *The team is not using Eclipse, and AspectJ is a core part of the design.* Use the @AspectJ syntax if you must work in non-Eclipse environments. Let the IDE compile classes and aspects through its normal build process. Use a post-build step to weave classes and aspects together. Even though the IDE is not AspectJ aware, debugging will not be a problem. You can set breakpoints in normal classes as well as those carrying the @Aspect annotation.

Choosing syntax and weaving

- *The team is exploring AspectJ for non-production use.*
  - It is a situation where you have just started with AspectJ. It is very likely that you will be using pre-built third-party (open source or otherwise) aspects instead of developing your own.
  - In this situation it is best to not alter your build environment.
  - If you are using Java 5, due to the simple configuration to set up the weaving agent, LTW should be considered. All you will need to do is make a few modifications to your startup script (add the –javaagent option) and create an aop.xml file (to define aspects to weave in and define concrete subaspects).
  - If you are using pre-Java 5, although you may use a classloader-based LTW, it can be difficult to set up correctly at first. Binary weaving should be considered in this case.