Aspect-oriented programming with AspectJ

Daniela Gotseva[1] and Mario Pavlov[2]

**[1] Computer Systems Department, Technical University of Sofia
Sofia, Bulgaria**

**[2] Computer Systems Department, Technical University of Sofia
Sofia, Bulgaria**

Abstract
This article describes the fundamental concepts of a complete AOP system. It discusses the AspectJ language specification and implementation. The article examines the use of the language, as well as its features and tooling support. It lays out a common crosscutting problem to illustrate the general syntax of the traditional AspectJ Language. The development tools of the AspectJ language are described and compared to popular Java development tools.

Keywords: *Aspect-oriented Programming, AspectJ, Aspect-oriented Programming System (AOP), AspectJ Development Tools (ADT), Eclipse IDE.*

1. Introduction

1.1 Fundamental concepts

An aspect-oriented programming system (AOP) is a software system that is a realization of the aspect-oriented programming methodology. In general, a programming methodology realization consists of two parts:

- Language specification: an explicit definition of the syntax and semantics of the language

The AOP language specification defines how programming constructs are expressed. An AOP language must include constructs for describing data and behaviour, along with constructs that describe how to combine data and behavior.

- Language reference implementation: a software application that can translate code written in the language into an executable form.

The AOP language implementation performs the steps necessary to convert the source code into executable code according to the language rules. This process is commonly referred to as compilation.

A complete AOP system supports the following fundamental concepts [1, 4-6]:

- Join points: identifiable points in the execution of a system
- Pointcut: a construct for selecting join points
- Advice: a construct to introduce or alter execution behaviour
- Static crosscutting: constructs for altering the static structure of a system
- Aspect: a module to express all corsscutting constructs

1.2 Historical perspective

AspectJ is the first complete AOP system. At the time of writing this article, AspectJ is also the best and most widely used AOP system. The initial development of AspectJ was the work of a team at Xerox PARC, led by Gregor Kiczales. He also coined the terms "crosscutting" and "aspect-oriented programming". After a few releases, Xerox donated AspectJ to the free software community at http://eclipse.org. A few years later another AOP system (AspectWerkz) merged with AspectJ, adding features, such as annotation based syntax. At the moment, AspectJ has an alternative implementation (AspectBench), used for experimenting with new features and optimizations.

2. AspectJ as an AOP system

AspectJ is an extension to the Java programming language that adds AOP capabilities to Java. The AspectJ implementation consists of the following components [2]:

- A compiler (ajc)
- A debugger (ajdb)
- A documentation generator (ajdoc)
- A program structure browser (ajbrowser)
- Integration with Eclipse, Sun-ONE/Netbeans, GNU Emacs/XEmacs, JBuilder, and Ant.

The AspectJ compiler (ajc) is often called a weaver. The name originated because of its primary role to weave the AspectJ extensions into the Java code, that is to weave aspects into classes and produce the final executable code. AspectJ provides three weaving mechanisms:

- Source weaving: The source code for aspects and classes is fed to the AspectJ compiler/weaver which produces the final executable code. This mechanism is useful when the entire source code (aspects and classes) is available. It enables both the compilation and weaving to be done in one go and highly simplifies the building process.
- Binary weaving: The input to the weaver is in byte-code form, that is classes and aspects are compiled beforehand. The idea is to compile the classes and aspects without weaving and then weave the resulting binary file. In other words, this is a three-step process (see Fig.h 1):
  1. Compile classes
  2. Compile aspects
  3. Weave aspects into classes to produce the final binary files

The binary weaving mechanism is very flexible and enables weaving even when some or all of the Java source code is not available, for example the source code is part of a third party library or for some reason cannot be compiled alongside the AspectJ code.

2.1 Load-time weaving

A load-time weaver takes compiled classes and aspects, as well as an XML configuration. The load-time weaver can take various forms, such as JVMTI agent or a classloader. The load-time weaver weaves aspects into classes as they are loaded into the system prior to first use. The XML file (aop.xml) specifies the weaving configuration. The aop.xml can be placed under the META-INF directory and defined in the MANIFEST.MF as a classpath component. Load-time weaving greatly reduces the risk of adopting and experimenting with AOP because it makes switching AspectJ on and off just a matter of configuration.

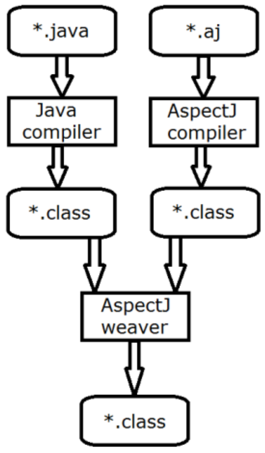


Fig. 1 Three-step process.

The AspectJ documentation tool (ajdoc) is very similar to the Java documentation tool (javadoc). It produces similar output, but also provides additional information, such as advice affecting a particular method or all code affected by a given aspect, allowing developers to examine easily the crosscutting structure of a system.

3. AOP with AspectJ

3.1 Implementing a cache

To describe and illustrate an AOP with AspectJ, we are going to discuss a rather common and wide-spread problem: implementing a cache. A cache is a classical crosscutting concern. With traditional object-oriented programming, the caching concern cannot be completely isolated or separated from the main concern. You can work around this problem to a certain extent leveraging design patterns such as proxy and decorator, but it cannot be completely eliminated.

AOP provides a complete and elegant solution to such a problem. A very simple illustration is shown in code listing 1 (Note that this code listing shows three files: FileSystem.java, FileSystemCache.java and FileSystemCacheAspect.aj):

Code listing 1: File System Interface.

```
public interface FileSystem {

        Collection<File> list(File entry);

        byte[] getContents(File entry);
}

public interface FileSystemCache {

        byte[] getContents(File entry);

        void putContents(File entry, byte[] contents);
}

public aspect FileSystemCacheAspect {

        private FileSystemCache cache = new
        FileSystemCacheImpl();

        pointcut fileSystemAccess(File entry) :
        execution(public * FileSystem+.get*(File)) &&
        args(entry);

        byte[] around(File entry) : fileSystemAccess(entry) {
                byte[] contents = cache.getContents(entry);
                if(null == contents) {
                        contents = proceed(entry);
11                              cache.putContents(entry, contents);
12                    }
13                    return contents;
14          }
15 }
```

The example shows the traditional AspectJ syntax. The FileSystemCacheAspect contains one pointcut and one advice that are defined on lines 5 and 7 respectively. Each pointcut definition is of the form:

pointcut *pointcutIdentifier : joinPointSelectionCriteria;*

Each advice definition is of the form

[strictfp] AdviceSpec [throws typeList] : pointcutIdentifier { body }

where AdviceSpec is one of

before*(formals)*
after*(formals)* ***returning*** *[(formal)]*
after*(formals)* ***throwing*** *[(formal)]*
after*(formals)*
Type ***around****(formals)*

and where formal refers to a variable binding like those used for method parameters, and formals refers to a comma-delimited list of formal.[2]

The following line:
pointcut *fileSystemAccess(File entry) :* ***execution****(public * FileSystem+.get*(File)) && args(entry);*
creates a pointcut with the name fileSystemAccess. The part after the colon is the join point selection criteria. In this case, it will match on execution of public methods that fulfil the following criteria:

- their name starts with “get”
- the return type is any (including void)
- takes one parameter of type File
- resides in classes that are direct or indirect descendants of FileSystem or FileSystem itself

This pointcut also collects the actual method parameter that is passed to each method execution matched by the pointcut. AspectJ provides a wide variety of powerful pointcut constructs used to select different join points. Most of the pointcut constructs are shown in Table 1. [2]

AspectJ offers a wide variety of pointcut constructs that cover virtually every possible join point in a system. The pattern syntax is as follows: [2]

```
MethodPattern =
  [ModifiersPattern] TypePattern
        [TypePattern . ] IdPattern (TypePattern | ".." , ... )
        [ throws ThrowsPattern ]
```

Table 1: Pointcut Construct Structures.

Pointcut construct	***Used to select...***
call*(MethodPattern or ConstructorPattern)*	calls to the methods or constructors
execution*(MethodPattern or ConstructorPattern)*	executions of methods or constructors
get*(FieldPattern)*	field read access
set*(FieldPattern)*	field write access
handler*(TypePattern)*	catch blocks
within*(TypePattern)*	executions of code defined in classes
withincode*(MethodPattern or ConstructorPattern)*	executions of code defined in methods or constructors
initialization*(ConstructorPattern)*	object initializations
preinitialization*(ConstructorPattern)*	object pre-initializations
staticinitialization*(TypePattern)*	static initializer executions
adviceexecution*()*	all advice executions
cflow*(Pointcut)*	join points in the control flow of any join point P picked out by Pointcut, including P itself
cflowbelow*(Pointcut)*	join points in the control flow of any join point P picked out by Pointcut, but not P itself
this*(Type)*	join points where the currently executing object is an instance of Type
target*(Type)*	join points where the target object (the object on which a call or field operation is applied to) is an instance of Type
args*(Type)*	join points where the arguments are instances of Type

```
ConstructorPattern =
  [ModifiersPattern ]
        [TypePattern . ] new (TypePattern | ".." , ...)
        [ throws ThrowsPattern ]

FieldPattern =
  [ModifiersPattern]   TypePattern   [TypePattern   .   ]
IdPattern

ThrowsPattern =
  [ ! ] TypePattern , ...

TypePattern =
    IdPattern [ + ] [ [] ... ]
    | ! TypePattern
    | TypePattern && TypePattern
    | TypePattern || TypePattern
    | ( TypePattern )
```

IdPattern =
Sequence of characters, possibly with special * and .. wildcards

ModifiersPattern =
[!] JavaModifier ...

Pointcuts can also be combined logically, for example:
***pointcut** staticExecution : execution(static * *(..)) && !within(com.example..*);*

The following line:
***byte**[] **around**(File entry) : fileSystemAccess(entry)*

is the advice that executes somewhat "around" the advised method or in other words behavior can be added before and/or after the method execution, and the behavior can be based on the result of the method execution. The code in the body of the advice first checks if the requested file contents is already cached: if yes, the cached value is returned; if no, the actual method from the file system implementation is invoked using the special construct "proceed()", the return value is cached and then returned. This advice augments the join points selected by the fileSystemAccess pointcut, that is the getContents() method from any FileSystem implementation.

AspectJ has two more types of advice: before and after that execute before or after the matched join point respectively. An advised method is each method that satisfies the join point criteria defined in the pointcut. Actually, every join point that is supported by the AOP system, such as field access, constructors and exceptions, can be advised.

The code in code listing 1 illustrates a very simple file system that supports caching. If traditional object-oriented programming were applied, the caching logic would have been scattered across all methods of the file system implementation. And if at some point the implementation had to be changed, the caching code would have to be presented in the new implementation as well. Even if the decorator design pattern were used, some changes still would be necessary to make the decorator decorate the correct file system implementation. With the aspect-oriented approach, the caching logic resides only in the aspect, which essentially means that modifications to the file system implementation or even replacing the entire implementation will have no effect on the caching aspect (literally speaking) of the file system. In other words, the file system cache is a completely separate module and modifications to the file system implementation and/or the cache can be done independently. So the code for the file system and the cache becomes simpler and therefore easier to maintain.

3.2 Program tracing

Another very good example of AOP with AspectJ is the problem of program tracing. Tracing is a special case of logging with the purpose of tracing the execution of a program. In other words, tracing is a technique to obtain records of everything executed in a program. Traditionally tracing is done by adding logging to the beginning and end of methods similar to the code in code listing 2.

Code listing 2: File System Implementation.

```
public class FileSystemImpl implements FileSystem {

        public Collection<File> list(File entry) {
                logger.info("Entered:  FileSystemImpl.list()");

                // code omitted for brevity

                logger.info("Left: FileSystemImpl.list()");
        }

        public byte[] getContents(File entry) {
                logger.info("Entered:
        FileSystemImpl.getContents()");

                // code omitted for brevity

                logger.info("Left:
        FileSystemImpl.getContents()");
        }

        public static void main(String[] args) {
                logger.info("Entered:
        FileSystemImpl.main()");

                FileSystem fs = new FileSystemImpl();
                Collection<File> files = fs.list(new File("/"));
                fs.getContents(files.iterator().next());

                logger.info("Left: FileSystemImpl.main()");
        }
}
```

Obviously, the calls to the logger should be added to each method in the system, which increases code scattering immensely, not to mention that the code becomes a maintenance nightmare. <<tsvety: had to re-wire the previous sentence a bit; nothing wrong with it in a conversation, but it was too informal for writing.>> Fortunately AspectJ solves this problem with literally several lines of code. A simple solution to the tracing problem using AspectJ is shown in code listing 3.

Code listing 3: Tracing Aspect.

```
public aspect TracingAspect {
        private Logger logger =
        Logger.getLogger(TracingAspect.class.getName());
        pointcut tracedExecution() : execution(public * *.*(..))
        && !within(TracingAspect);

        before() : tracedExecution() {
                Signature methodSignature =
        thisJoinPointStaticPart.getSignature();
                logger.info("Entered: " +
        methodSignature.toShortString());
        }

        after() : tracedExecution() {
                Signature methodSignature =
        thisJoinPointStaticPart.getSignature();
                logger.info("Left: " +
        methodSignature.toShortString());
        }
}
```

The pointcut in the TracingAspect obtains the execution of every public method in the system, except anything in the aspect itself, and weaves the calls to the logger before and after each execution. The thisJoinPointStaticPart reference is accessible within every advice. It refers to the static part of the current join point and it can be used to access useful information like the join point signature, as shown in code listing 3. The AspectJ runtime API documentation is located at http://www.eclipse.org/aspectj/doc/next/runtime-api/index.html. When the code in code listing 2 is executed with the TracingAspect, the logger output should look similar to the following:

INFO: Entered: FileSystemImpl.main(..)
INFO: Entered: FileSystemImpl.list(..)
INFO: Left: FileSystemImpl.list(..)
INFO: Entered: FileSystemImpl.getContents(..)
INFO: Left: FileSystemImpl.getContents(..)
INFO: Left: FileSystemImpl.main(..)

The TracingAspect, as is, will work for any system. When implemented as shown in code listing 3, tracing is completely separated from the system and the aspect itself is fully reusable.

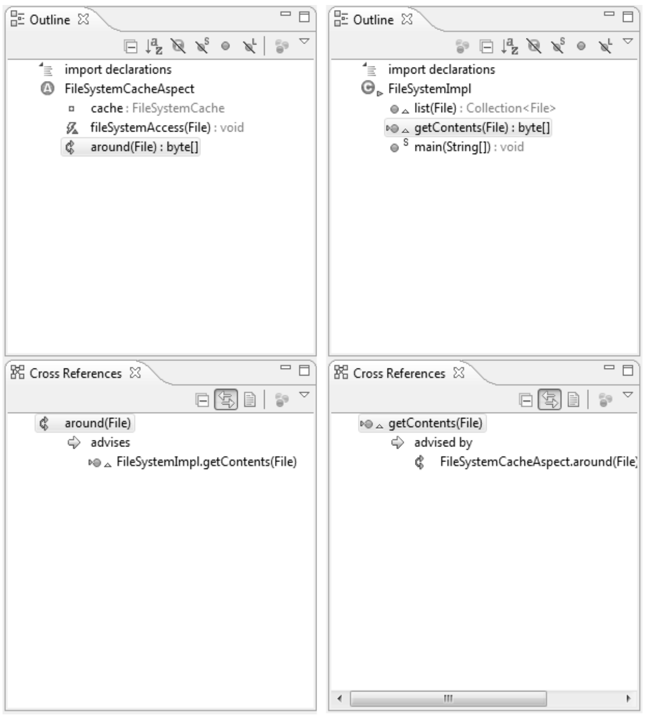
4. AspectJ tooling support

4.1 Eclipse integration

AspectJ is well-integrated with the Eclipse IDE. The AspectJ Development Tools (AJDT) project provides Eclipse platform based tool support for aspect-oriented software development with AspectJ. AJDT's home page is at http://eclipse.org/ajdt/. AJDT is very similar and consistent with the Java Development Tools (JDT). It is integrated with the "New Project" wizard and it has its own "New Aspect" wizard. AJDT has an editor for aspects that is also similar and consistent with the Java editor. The aspect editor shares many of the features of the Java editor including [3, 5]:

- The outline view populates as you type
- Syntax errors are underlined in red
- Organize imports works the same across aspects and classes
- The editor supports folding (for example, collapsing comment blocks)
- Reformat file function
- Helpful content-assist via control-space function

Another very useful part of AJDT is the Cross References view. It shows the crosscutting relationships of the selected element. This view plays very well with the standard Outline view that normally shows the structure of a class/aspect. For example, clicking within a method in the editor or the Outline view causes the Cross References view to show any crosscutting information for that method, for example which advice are applied to this method. Likewise clicking within an advice in the editor or the Outline view causes the Cross References view to show which methods are advised by the selected advice as shown in Fig. 2.



```
import declarations
FileSystemCacheAspect
    cache : FileSystemCache
    fileSystemAccess(File) : void
    around(File) : byte[]
```

```
import declarations
FileSystemImpl
    list(File) : Collection<File>
    getContents(File) : byte[]
    main(String[]) : void
```

```
around(File)
    advises
        FileSystemImpl.getContents(File)
```

```
getContents(File)
    advised by
        FileSystemCacheAspect.around(File
```

Fig. 2 Outline and Cross References

Debugging AspectJ code is done virtually in the same way that Java code is debugged.

The AspectJ compiler used in AJDT is based on the Eclipse Java compiler, and aims to be as efficient. The compilation process is made more complex by the crosscutting nature of AspectJ, but even the simplest changes to both classes and aspects will still result in a fast incremental compile. Some changes, such as changes to pointcuts or advice that has been inlined, will require a full build.

4.2 Ant integration

The AspectJ compiler is integrated with Ant and can be run from any Ant build project. The aspectjtools.jar includes three Ant tasks: [2, 6]

- AjcTask (iajc), a task to run the AspectJ post-1.1 compiler, which supports all the Eclipse and ajc options, including incremental mode.
- Ajc11CompilerAdapter (javac), an adapter class to run the new compiler using Javac tasks by setting the build.compiler property
- Ajc10 (ajc), a task to run build scripts compatible with the AspectJ 1.0 tasks

Adding the code in code listing 4 to any Ant build file makes the AspectJ Ant tasks accessible:

Code listing 4: Ant Integration.

```
<taskdef
        resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdef
        s.properties">
        <classpath>
                <pathelement
        path="${aspectj.lib.dir}/aspectjtools.jar" />
        </classpath>
</taskdef>
```

Where ${aspectj.lib.dir} points to the lib directory in the AspectJ distribution. The <iajc> task should be used in most cases. Some of the important attributes are shown in Table 2. [2]

A complete list of all supported attributes is located at http://eclipse.org/aspectj/doc/released/devguide/antTasks-iajc.html. A simple example of a functional Ant build file is shown in code listing 5.

Code listing 5

```
<project name="simple-example" default="compile" >
        <taskdef
        resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdef
        s.properties">
                <classpath>
                        <pathelement
        path="${aspectj.lib.dir}/aspectjtools.jar" />
                </classpath>
        </taskdef>

        <target name="compile" >
                <iajc
        outJar="${project.output.dir}/application.jar"
        sourceroots="${project.src.dir}"
        classpath="${aspectj.lib.dir}/aspectjrt.jar" />
        </target>
</project>
```

Where *${project.src.dir}* points to a source directory that contains Java classes and aspects and *${project.output.dir}* points to a directory into which the final binary file (application.jar) goes. In this case the AspectJ compiler performs source weaving.

The AspectJ Ant integration is very convenient and simplifies to a great extent adding AspectJ to existing projects whose builds are based on Ant.

Table 2: Important Attributes

Attribute	***Description***
sourceRoots	Directories containing source files (ending with .java or .aj) to compile. (can be used like a path-like structure)
inPath	Read .class files for bytecode weaving from directories or zip files. (can be used like a path-like structure)
classpath	The classpath used by the sources being compiled. When compiling aspects the same version of aspectjrt.jar has to be included. (can be used like a path-like structure)
destDir	The directory in which to place the generated class files. Only one of destDir and outJar may be set.
outJar	The zip/jar file in which to place the generated output class files. Only one of destDir and outJar may be set.
target	Target class file format.
source	Source compliance level.

5. Conclusion

AspectJ slowly but surely gains popularity and the number of big projects that are using it has increased. A good example is the Spring framework that allows AOP and AspectJ code in particular to be seamlessly integrated and used with the framework. AspectJ is also very popular in the academic setting. Researchers often use it for their research in the area of AOP, for example in software design optimizations.

Although AOP with AspectJ is not mainstream at the moment, it has potential to become very popular in the near future because it is easy to use and very powerful.

References

[1] R. Laddad, AspectJ in Action, Enterprise AOP with Spring, Manning Publications, 2010.
[2] http://www.eclipse.org/aspectj
[3] http://www.eclipse.org/articles
[4] R. Miles, AspectJ Cookbook, O'Reilly Media, 2004.
[5] A. Colyer, A. Clement, G. Harley and M. Webster, Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and Eclipse AspectJ Development Tools, Addison-Wesley Professional, 2004.
[6] J. D. Gradecki and N. Lesiecki, Mastering AspectJ: Aspect-Oriented Programming in Java, Wiley, 2003.

Daniela Gotseva is associate professor, PhD and Vice Dean of Faculty of Computer Systems and Control, Technical University of Sofia, from 2008 with primary research interest of programming languages and fuzzy logics. She is a member of the IEEE and the IEEE Computer Society.

Mario Pavlov is PhD student at Faculty of Computer Systems and Control, Technical University of Sofia, from 2010, with primary research interest of aspect oriented programming and programming languages.