Technische Universität Darmstadt

Bachelor Thesis

Language-independent visualization of cross-cutting program structure

Software Technology Group

by
Jannik Jochem

Advisor:
Dr.-Ing. Christoph Bockisch

Supervisor:
Prof. Dr.-Ing. Mira Mezini

September 18, 2008

Erklärung

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

(Ort, Datum)	(Unterschrift)

Contents

1 Introduction

Today, developers expect highly integrated development environments (IDE). Such IDEs support automatic code transformations and help the developer by providing various kinds of visualizations that simplify understanding the code.
Aspect-Oriented Programming (AOP) allows the developer to alter the behavior of one module from another module. In AspectJ, this is realized by having conventional Java classes as well as *aspects* that contain the aspect-oriented parts of a program. An *aspect* contains *pointcuts* with associated *advice*. A *pointcut* is an expression that can select points in the execution of a program. A pointcut has a static part that can be evaluated during weaving. It can also have a *runtime check* that can only be evaluated when one of the points selected by the pointcut is reached. An *advice* is simply a piece of conventional Java code. Those instructions that can potentially be matched by a pointcut are called *join point shadows*. When this happens, the advice that is associated with the pointcut is executed before, after or instead of the code at the join point shadow. In aspect-oriented terminology, one says the join point shadow is *advised by* the advice code. In this way, a bidirectional *advises / advised by relationship* is created between the join point shadow and the advice.
This requires the developer to understand which advice apply at which points in the program flow. The ends of the advises relationship are usually part of different compilation units and thus cannot be easily related to each other by looking at one compilation unit in isolation. To remedy this situation, modern AOP tools visualize the cross-cutting program structure in such a way that the advises relationship is always displayed explicitly. In the AspectJ Development Tools (AJDT), this is realized by markers at advice declarations and at the join point shadows the advice applies to. Using the markers, the developer can navigate from advice to advised code and vice versa. Additionally, a tree view of all cross references of the currently selected program element is provided.
This visualization is based on a code model that is generated during compilation and weaving. However, recent developments in the implementation of aspect-oriented languages include the possibility of weaving advice at load- or runtime instead of at compile time. AspectJ for instance provides a load-time weaving facility. If this is employed, the AspectJ compiler is stopped before the weaving phase and weaving is performed when the classes of the application are loaded. The Framework for Implementing Aspect Languages (FIAL) [Boc08, chapter 3.3] works in a similar manner, with the difference that weaving is performed at runtime. However, when the application is not woven at build time, the models that drive the discussed code visualizations are left incomplete.
The goal of this bachelor thesis is to show that it is possible to employ runtime weaving and still be able to use the advanced code visualization provided by AJDT. In order to reach this goal, an additional build step is performed after the compilation of the code

under development. This build step is similar to weaving in that it analyzes the program structure to obtain advice and join point shadows. This information is then provided to FIAL in order to acquire the advises relationship by simulating a weaving run. However, no code is generated. Instead, the matching information provided by FIAL is used to complete the code models that were built by the compiler in the previous step. As a result, visualizing where advice match is possible without actual weaving being performed. FIAL is designed to be language-agnostic. This will make it possible to create a single aspect-visualization plugin per IDE that provides support for a multitude of aspect-oriented languages.
In the next chapter, a general overview of the relevant software modules will be given. Chapter 3 discusses the approach taken in this thesis in detail as well as the technical and conceptual issues that have arisen during implementation. Chapter 4 discusses related work in the area of aspect-oriented development environments. Finally, chapter 5 sums up the result of this work and in chapter 6 opportunities for further development are discussed.

2 Problem analysis

When weaving is performed at load- or runtime, the visualizations the developer is used to are no longer available because the cross-cutting information is not generated at build time. The solution that is proposed in this thesis is based on AspectJ [KHH+01] as the AOP language with Eclipse as the development environment. The IDE integration for Eclipse is provided by the AspectJ Development Tools (AJDT) [CCK03]. It provides different kinds of visualizations for cross-cutting program structure that are based on the Abstract Structure Model (ASM) [KCCC06, Ker02]. When load- or runtime weaving is used, the ASM is not completed at build time. Therefore, this thesis proposes using the Framework for Implementing Aspect Languages (FIAL) [Boc08, chapter 3.3] to gather cross-cutting information and then transforming the FIAL representation into an ASM representation of the application under development. As a result, the code visualizations are available even when weaving is deferred until runtime.

2.1 AspectJ and AJDT

The AspectJ Development Tools (AJDT) consist of a set of Eclipse plugins that support the developer when writing applications in AspectJ. Because AspectJ uses Java as its base language, the AJDT is largely based on the Java Development Tools (JDT) for Eclipse. Therefore, the developer can use the features provided by the JDT for editing the Java code of the application.
The AJDT offers three kinds of visualizations that are unique to AspectJ projects. Inline *markers* are used to indicate places in the code where advice match. If an advice has a runtime check associated with it, this is reflected by a question mark on the marker icon. By right-clicking on a marker, the developer can navigate from advice to the code places that are advised as well as from advised code places to the corresponding advice. Figure 2.1 shows the markers as well as the context menu for navigating the advises relationship of an aspect.
Analogous to the outline that shows the static structure of Java classes, a so-called *cross-references view* is provided. This is a tree view that shows any cross-cutting relationships in the code structure. Figure 2.2 shows the cross-references view for an example class.
As an additional help, the AJDT contains a data source for the Eclipse visualizer. This visualizer shows advised code using bar diagrams. This is for example useful for estimating the overall effect of aspect-oriented program parts on the project or on a package. Figure 2.3 shows the visualizer for a small AspectJ project.
For building, AJDT contains an AspectJ builder that drives the AspectJ compiler. Just like the Java builder, this is an incremental project builder that re-compiles only those

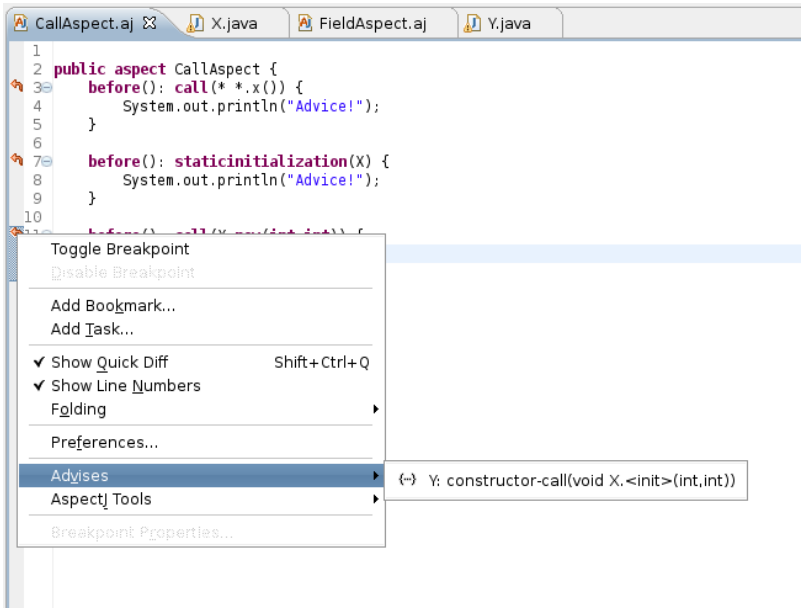

Figure 2.1: AJDT markers with the navigation context menu

source files that were actually changed. The AspectJ compiler compiles the Java source files and aspects in the project and performs weaving.
In addition to the java bytecode files that are created, the AspectJ compiler generates a code model of the cross-cutting structure of the program under compilation. This model is called the Abstract Structure Model (ASM) and is used to realize the visualizations that were previously discussed.
The ASM consists of two parts: a program element hierarchy and a relationship map. The hierarchy is a tree of program elements such as classes, join point shadows, aspects and advice. The relationship map defines certain one-to-many relationships between those program elements. To illustrate this, the following example project is used.

```
1 public class X {
2   public void x() {
3     System.out.println("");
4   }
5
6   public void y() {
7     x();
8   }
9 }
```

Listing 2.1: An example base class in Java

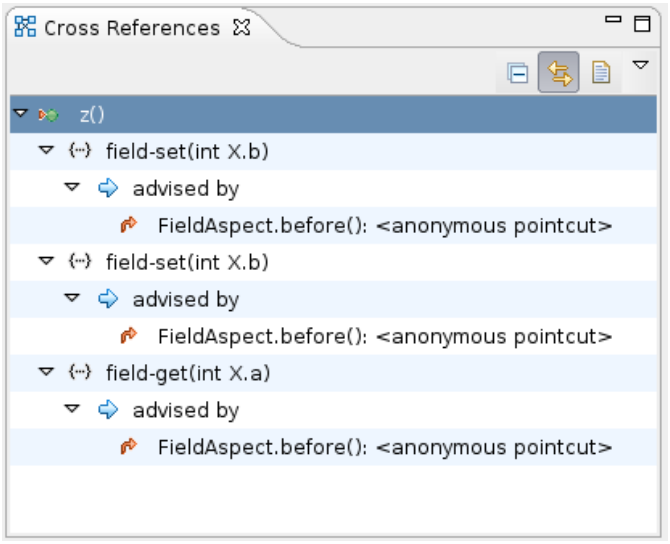

Figure 2.2: Visualizing cross-cutting structure in a tree view

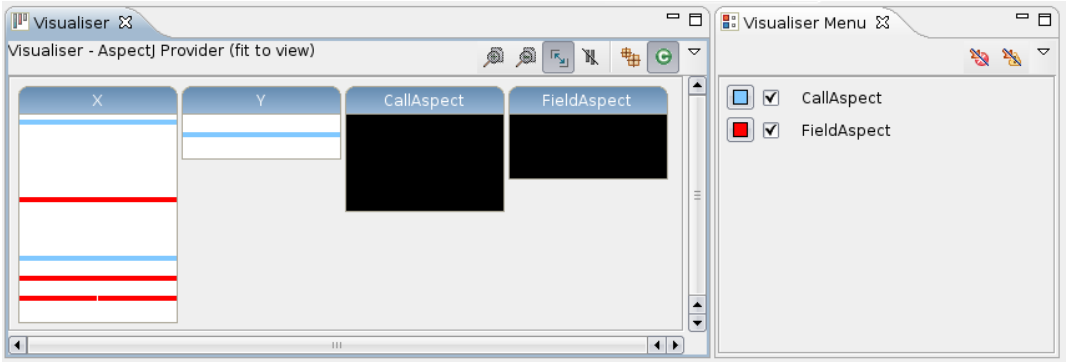

Figure 2.3: A visualization that allows estimating the overall effect of advice on a project

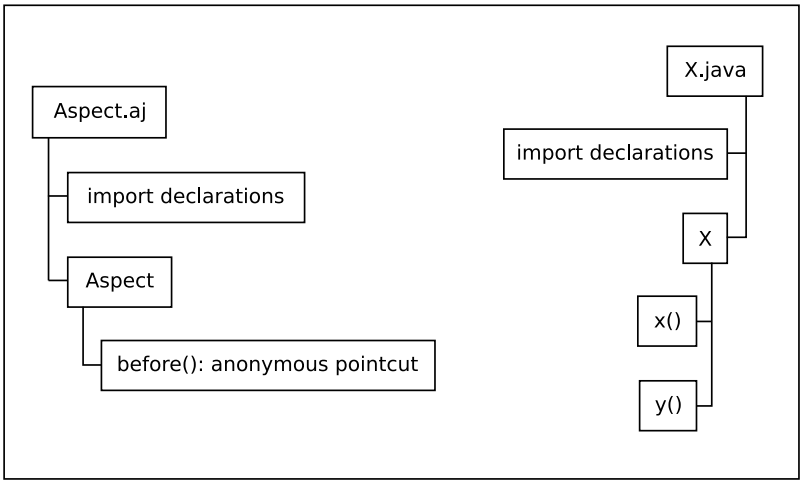

Figure 2.4: After compilation, join point shadows are still missing from the hierarchy

```
1 public aspect Aspect {
2
3   before(): call(void X.x()) {
4     System.out.println("Advice!");
5   }
6 }
```

Listing 2.2: An AspectJ aspect advising a method in class X

After compilation, but before weaving, the hierarchy contains program elements for all aspects and advice, as well as for all classes and their methods. Program elements for join point shadows are not yet generated. Program elements contain line number information that links the program element object to the corresponding element of the source code. Figure 2.4 shows the state of the hierarchy just after compilation, but before weaving is performed.

When the program is woven, any join point shadows that are advised by some piece of code are also added to the hierarchy. Additionally, the relationship map is populated with cross-cutting relationships. Those are one-to-many relationships that have one program element as source and any number of program elements as target. Each relationship has a kind that specifies what type of cross-cutting behavior it describes. ASM offers a number of different cross-cutting relationships. This includes the *advises* relationship as well as relationships for advanced language features such as inter-type declarations and pointcut-generated compiler errors and warnings (*declare error / declare warning*). For this work, only the advice relationships are relevant. A relationship also has a direction that is reflected in the displayed name, such as “advises” as opposed to “advised by”. Additionally, each relationship has a flag that is set to `true` if the cross-cutting behavior depends on a runtime check.

Figure 2.5 shows the hierarchy after weaving, with the added program element for the advised join point shadow and the contents of the relationship map illustrated as arrows.

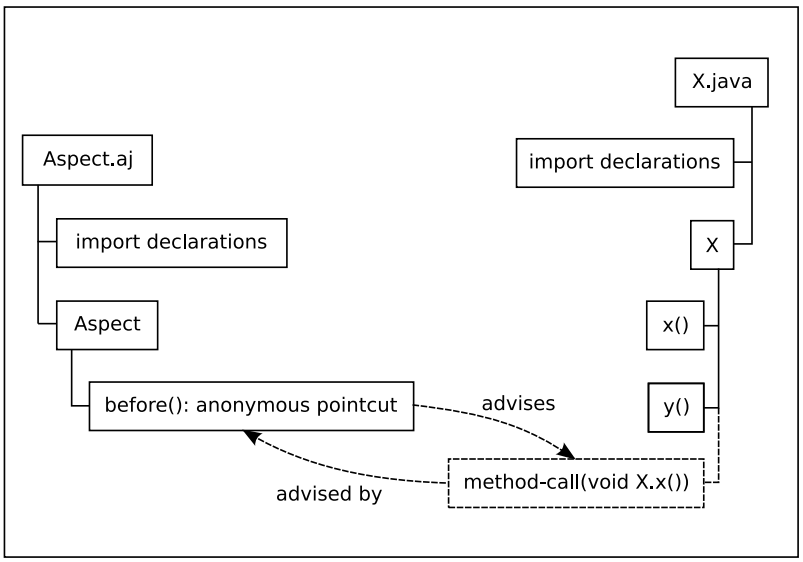

Figure 2.5: After weaving, join point shadows are available in the hierarchy and cross-cutting information is represented in the relationship map

The ASM is the IDE-independent code model for AspectJ and is part of the incremental compiler. The AJDT adds another model layer, called the AJProjectModel, which is Eclipse-specific and provides model persistence to preserve code structure information when the IDE is closed.

2.2 FIAL

The Framework for Implementing Aspect Languages (FIAL) consists of a meta-model for expressing the semantics of pointcut-advice based aspect-oriented languages. FIAL is open towards the language that is implemented as well as the runtime implementation of weaving. Thus, it has a language-oriented *front-end* as well as an execution-oriented *back-end* that implements weaving functionality. Ideally, the front-end and back-end can be implemented independently of each other. The back-end implementation instantiates the framework and can thus supply its own subclasses of the model entities. The front-end uses the generic framework classes to build a model representation of the cross-cutting structure of the application that is being loaded [Boc08, chapters 3.3, 3.4]. Figure 2.6 shows the classes of the meta-model that are relevant to this work in a UML class diagram.

Instructions where advice may apply are modeled by `JointPointShadow` entities. A `JoinPointShadow` has a `Signature`, the type of which describes the kind of join point shadow that is being modeled. The five currently supported signature types are listed in table 2.1. The `Signature` contains information such as the declaring class, as well as access modifiers and field or method names where applicable. This information is used to select join point shadows using a pointcut. When a join point shadow is reached, any code advising that join point shadow may depend on a runtime check. To manage the

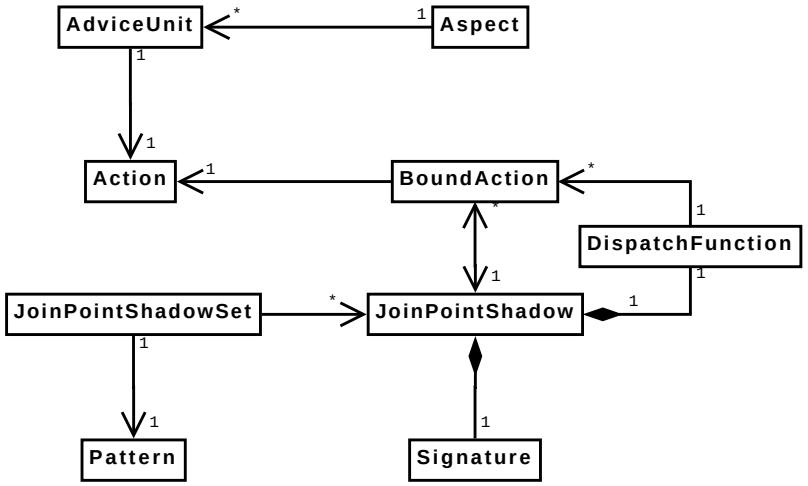

Figure 2.6: The FIAL meta-model simplified to contain only classes that are relevant to this thesis

execution of advice during runtime, each `JoinPointShadow` has a `DispatchFunction` that determines what code is executed when the join point shadow is reached.
Because a pointcut ultimately describes a selection of join point shadows, the corresponding model entity is called a `JoinPointShadowSet`. `JoinPointShadowSet`s always have a single `Signature` type that determines which `JoinPointShadow`s they can refer to. For example, a `JoinPointShadowSet` for `MethodCallSignature`s can only contain `JoinPointShadow`s with `MethodCallSignature`s. Each `JoinPointShadowSet` has a `Pattern` that models part of the pointcut expression. Which `JoinPointShadow`s belong to a `JoinPointShadowSet` is determined by matching the `Signature` of all `JoinPointShadow`s against this `Pattern`. `Pattern`s are usually generated from the corresponding pointcut definition in the aspect-oriented language.
Aspects are modeled by an `Aspect` model entity. An `Aspect` has any number of `AdviceUnit`s that model pointcut-advice pairs. The advice itself is modeled as an `Action`. When an `Aspect` is *deployed*, this has the effect of deploying all its `AdviceUnit`s. After deployment of an `AdviceUnit`, its `Action` is *attached* to any matched `JoinPointShadow`s. This attached `Action` is represented by the `BoundAction` model entity. The relationship between `JoinPointShadow`s and `BoundAction`s is bidirectional, so `BoundAction`s are navigable from their corresponding `JoinPointShadow`.

2.3 Problem statement

AspectJ can be integrated with FIAL in such a way that the standard AspectJ compiler can be used. In this approach, the AspectJ compiler is stopped before the weaving phase. However, an abstract definition of the aspect-oriented parts of the program is generated in the form of an XML file that contains the class names of all aspects. These classes have special Java annotations that describe pointcut expressions and advice types. An

Signature type	Description
MethodCallSignature	when a method is invoked
ConstructorCallSignature	when a constructor is invoked
FieldReadSignature	when a field of a class or object is read
FieldWriteSignature	when a field of a class or object is set
StaticInitializerSignature	when a class is loaded and static initialization is performed

Table 2.1: Signature types in FIAL

importer reads that information and generates meta-model entities from it that represent the aspect-oriented part of the program, thus allowing FIAL to perform the final weaving at runtime [BMH+07].
Using the AspectJ compiler to develop with FIAL enables the developer to use the AJDT for editing and running code. The drawback of this approach is that cross-cutting information is not generated at build time. Therefore, none of the visualizations that are provided by AJDT are available. This is because the ASM is left incomplete in two respects. First, the hierarchy does not contain any program elements for join point shadows. Second, the relationship map is empty, so it does not reflect the cross-cutting structure of the program.
The goal of this thesis is to remedy this situation by instantiating the FIAL framework and using it to generate the missing information at build time. On an abstract level, this requires a number of steps to be performed:

1. Collect join point shadows from the application and provide them to the framework.
2. Import the aspects in the application into the framework.
3. Create program elements for matched join point shadows.
4. Create advises-relationships for join point shadow - advice pairs.

3 Proposed solution

The problem of visualizing cross-cutting program structure in the absence of static weaving that has been identified in the previous chapter is solved by the AspectJ/FIAL Eclipse plugin. This plugin gathers all the information that is needed to build a FIAL model of the application under development and then uses the cross-cutting information supplied by FIAL in order to complete the ASM.

3.1 Approach

The features implemented as part of this thesis are provided as the AspectJ/FIAL Eclipse plugin. The Eclipse IDE provides a number of extension points that allow new features to be added to the IDE or existing ones to be modified. The AspectJ/FIAL plugin contains a *project nature* that can be added to any AspectJ project, thus making the project an *AspectJ/FIAL project*. The AspectJ/FIAL plugin modifies the AspectJ build options so that the AspectJ compiler skips its weaving step but generates all the information necessary for performing load time weaving. This includes the file `META-INF/aop-ajc.xml` that lists all aspect classes in the project. Also, the compiler generates Java annotations that encode pointcut and advice declarations in the aspect class files. This is the required standard configuration for using the importers approach described in section 2.3 with AspectJ.
When an AspectJ/FIAL project is built, the standard AspectJ builder is run first. Because no weaving is done, the cross-cutting information is still missing from the Abstract Structure Model (ASM) after this build step. Thereafter, the AspectJ/FIAL builder is run which contains the logic to complete the ASM. The AspectJ/FIAL builder is discussed in detail in section 3.2.
As mentioned in section 2.3, as a first step to obtaining cross-cutting information for a given project, the join point shadows of all conventional Java classes need to be collected. This is done using *ASM Traversal*, a framework for collecting join point shadows that is based on the ASM Bytecode Toolkit. For clarity, the latter is always referred to as the ASM Bytecode Toolkit, whereas only ASM refers to the Abstract Structure Model of the AspectJ compiler. ASM Traversal takes any number of class names as input and outputs a number of `JoinPointShadow` meta-model objects suitable for passing to FIAL.
To import the aspect-oriented parts of the program, the *AspectJ annotations importer* that was developed for FIAL is used. The importer finds any aspect class files in the current class path using the XML file that is output by the AspectJ compiler. Those class files are then analyzed using the ASM Bytecode Toolkit. From the Java annotations previously generated by the AspectJ compiler, the importer builds `Aspect` meta-model

objects that describe the contents of the analyzed aspect. These are then imported into FIAL.
The two FIAL modules *ReferenceArchitecture* and *Patterns* are used. ReferenceArchitecture provides management of join point shadows and aspect-oriented constructs and also controls the other parts of the framework. Patterns is used to perform matching over the `Signature`s of `JoinPointShadow`s.
When matching has been performed by FIAL, the resulting information can then be transformed into the ASM representation of cross-cutting program structure. This is discussed in section 3.2.2. The ASM for any AspectJ project in the Eclipse workbench is available because the AspectJ compiler is an Eclipse plugin itself. As a last step, the visualizations provided by the AJDT are refreshed to display the cross-cutting information that is now available. Manipulating the ASM directly has the advantage that the model persistence features provided by AJDT are automatically used, even when the information is gathered by other means than the AspectJ compiler.

3.2 The AspectJ/FIAL builder

The AspectJ/FIAL builder is the central part of the AspectJ/FIAL plugin that controls the process of generating cross-cutting information that the AspectJ builder does not provide. Figure 3.1 shows an overview of the build process, including the software modules, output files and models involved and is explained in the following.
After the AspectJ builder was run (1), the AspectJ/FIAL builder is started (2). First, a list of all conventional Java classes that were compiled from the Java source files in the project is built. Then, join point shadows are collected from those classes (2.1.a). The process of collecting join point shadows using ASM Traversal is described in detail in section 3.2.1. After this process, join point shadows in the output classes of the project are represented as `JoinPointShadow` objects and are imported into FIAL.
After the object-oriented parts of the program have been handled by the builder, the AspectJ annotations importer is used to create `Aspect` instances for any aspects on the project's build path (2.1.b). The resulting list of `Aspect`s is then imported into FIAL and deployed. After this process, cross-cutting information is available from FIAL (2.2).
The relationship manager (2.3) iterates the `JoinPointShadowSet`s provided by FIAL, creating the corresponding advises and advised by relationships in the relationship map of the ASM in the process. Exactly how those relationships are generated is discussed in section 3.2.2.
As a final step, the AJDT plugin is notified of the changes to the ASM and is asked to refresh its visualizations. This is necessary so that the cross-references view actually shows the relationships of the program element currently being edited and the markers are re-generated.

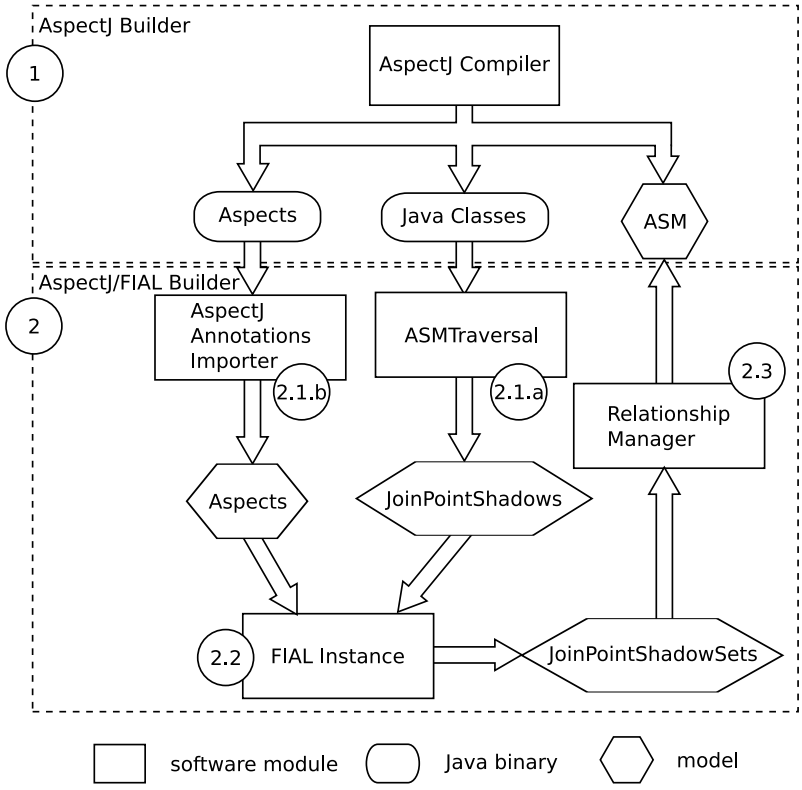

Figure 3.1: Overview of the build process

instruction	possible join point shadow signature type
for every class	StaticInitializerSignature
`GETFIELD` or `GETSTATIC`	FieldReadSignature
`PUTFIELD` or `PUTSTATIC`	FieldWriteSignature
`INVOKESPECIAL` with "<init>" method name	ConstructorCallSignature
`INVOKESPECIAL`, `INVOKEVIRTUAL`, `INVOKESTATIC`, `INVOKEINTERFACE`	MethodCallSignature

Table 3.1: Bytecode instructions and the join point shadows they result in

3.2.1 Collecting join point shadows

Join point shadows are collected using the ASM Traversal framework. It was initially developed by Daniel Müller and Florian Breuning as part of a FIAL based static weaver and has been extended into a more general framework for collecting join point shadows from Java bytecode. As part of this thesis, it was extended to offer the user of the framework more control over various aspects of shadow collection. ASM Traversal uses the ASM Bytecode Toolkit to extract join point shadows from Java class file binaries. ASM Traversal also provides a partial instantiation of the FIAL back-end.
The ASM Bytecode Toolkit offers visitor [GHJV95, 331-349] interfaces that allow Java bytecode files to be analyzed for their structure. The process of analyzing a class for its join point shadows is called *traversing* the class in ASM Traversal. When a class is traversed, a static initializer `JoinPointShadow` for the class is created first. Then, the methods of the class are visited and analyzed for any bytecode instructions that represent join point shadows. `JoinPointShadow`s are created for each of those instructions along with a `Signature` matching the join point type as in table 2.1. Because the access modifiers and `throws` declarations of called methods are also part of the `Signature`, but are not available at the call site, the class that declares the invoked method is also analyzed using a special visitor that retrieves this information.

ASM Traversal also provides notification of dependencies between traversed classes. This means that if a class depends on another class in some particular way, ASM Traversal notifies the client application of this fact. To determine which classes should be traversed, the callback interface `TraversalController` is provided and must be implemented by the client application.
The possible dependency types between classes are listed in table 3.2. This is useful for weaving an application when only the class with the main method is supplied to the ASM Traversal framework. For the AspectJ/FIAL plugin this is not necessary because all classes in the output folders of the project are traversed anyway. This means that `RelationshipType.None` is always used, which is just a shortcut to force traversal of a particular class.

As an additional feature that makes the process of collecting join point shadows more flexible, the `TraversalController` can provide an observer interface that receives any `JoinPointShadow` objects as soon as they are created. Also, the client application using

RelationshipType	**description**
None	force traversal of a particular class
Extends	superclass of traversed class
Implements	interface implemented by traversed class
CallsMethodOn	class that declares a method which was called by traversed class

Table 3.2: Relationship types between traversed classes

ASM Traversal can provide an abstract factory for creating JoinPointShadows. In AspectJ/FIAL, this is used to associate each JoinPointShadow with its program element in the ASM.
The FIAL framework requires instantiations to provide an implementation of the FIAL TypeDescriptor model entity as well as an implementation of the ClassHierarchyProvider abstract factory [GHJV95, 87-95] to create TypeDescriptor instances. The TypeDescriptor is used to express types of the base language in the aspect-oriented meta-model such as classes or argument and return types of methods. This is used to build Signature objects and match against them. ASM Traversal provides a TypeDescriptor implementation that is an adapter [GHJV95, 139-150] for the type descriptor implementation of the ASM Bytecode Toolkit, which in turn represents Java's internal bytecode type descriptor format. As an additional feature, the abstract factory that is used to create TypeDescriptors can also be queried for the type hierarchy of a particular type. This functionality is provided by using the information that was collected when the type hierarchy of the classes under development was traversed. This is required in order to allow subtype matching as with AspectJ's "+" matching operator to take place.
The TraversalController in AspectJ/FIAL is called the ClassDependencyManager. It controls the ShadowManager that manages the join point shadows in a project. The JoinPointShadows are stored per-class. This means that once it has been determined that a class has changed, all its JoinPointShadows are disposed of by the ShadowManager. Then, the class is traversed again in order to acquire a new and correct set of JoinPointShadows for the class.

3.2.2 Building relationships

When join point shadows and aspects have been loaded into the FIAL instance, the cross-cutting information that is necessary to generate the advises relationship is available from FIAL. The part of the AspectJ/FIAL plugin that transforms the matching information from FIAL into the ASM representation is called the RelationshipManager. FIAL provides a set of JoinPointShadowSets for each signature type (see table 2.1). Every JoinPointShadowSet contains a set of JoinPointShadows that were matched by the former. For each matched JoinPointShadow, a program element is created in

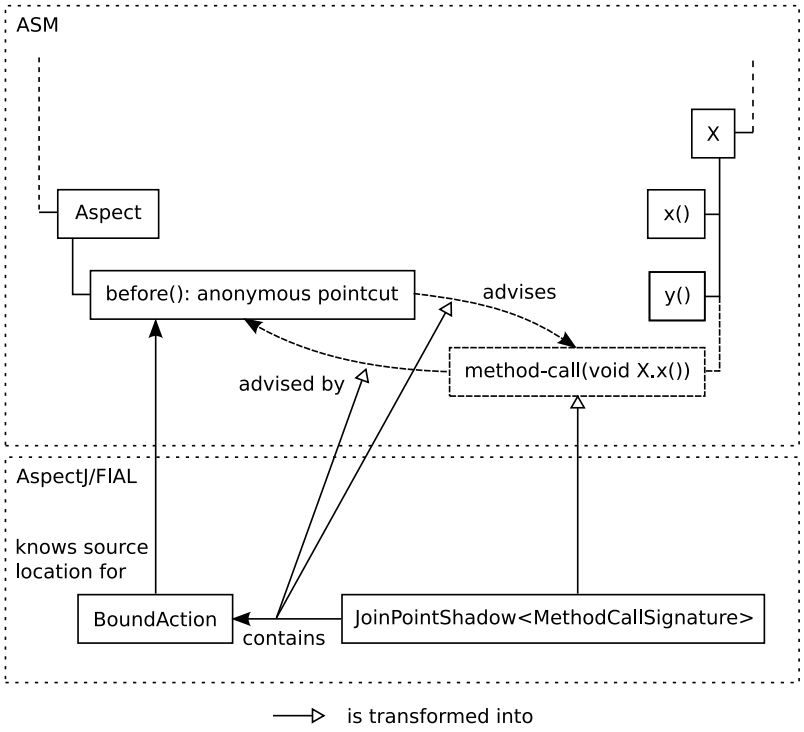

Figure 3.2: Model transformation from FIAL to ASM

the ASM hierarchy. For creating the advises relationship, all `BoundActions` of a `JoinPointShadow` are inspected. For every join point shadow, FIAL creates a `BoundAction` representing the code of the join point shadow itself. This is simply ignored by the `RelationshipManager`. The actual advice that is associated with the join point shadow is contained in the `Action` part of the `BoundAction`. For visualization, only the source code location of the advice declaration is of interest. This information is available directly from the `BoundAction` (see section 3.2.3). This way, a pair of advises / advised by relationships can be created from the `BoundAction`. Figure 3.2 shows how FIAL model entities are transformed into ASM entities.

Any relationship in the relationship map also has a flag that is set when the advice depends on a runtime check. In FIAL, runtime checks are modeled as *dynamic properties*. Every `JoinPointShadow` has a `DispatchFunction` that controls the order of code execution once the join point shadow is reached. In order to determine whether any advice depends on a runtime check, the `RelationshipManager` has to find out whether the dispatch function for a given `JoinPointShadow` and a given `BoundAction` is constant. On an abstract level, this requires the reduction of the general `DispatchFunction` for all `BoundActions` of a `JoinPointShadow` into a `DispatchFunction` that handles only the dispatch of the `BoundAction` that is currently being inspected. This operation is called a *projection* of the `DispatchFunction`. If the structure of the resulting projected `DispatchFunction` is non-trivial, the relationship in question has a runtime check.

3.2.3 Implementation-specific problems

Limitations of FIAL model management

FIAL is a statically-initialized singleton framework. This means that a single instance of the framework has to be shared between all AspectJ/FIAL projects in the Eclipse workbench, which leads to unnecessary matching operations being performed between join point shadows and pointcuts of unrelated projects. This results in a performance loss.
Since FIAL is geared towards runtime weaving of applications, it is not expected that the application being processed changes during the lifetime of the framework instance. Exactly this is the case when developing AspectJ applications in an IDE. When code is edited, join point shadows as well as pointcuts and advice in the application may change. FIAL does not currently provide any means to express such changes in its meta-model. It is not possible to remove single `JoinPointShadow`s or `Aspect`s from the system. Nor can `Aspect`s or pointcut expressions be changed after they have been loaded into the system.
Because of those limitations, the first build step performed by the AspectJ/FIAL builder is to completely remove any existing model information in the framework. This was not previously possible in FIAL. For this reason, two methods have been added to the `AspectModelSystem` class that serves as the controller of FIAL.
The method `clearShadows()` removes all `JoinPointShadow` objects from the system. However, the `JoinPointShadow`s are still stored in the `ShadowManager` by the AspectJ/FIAL plugin for later reuse.
A second method `clearJoinPointShadowSets()` that removes all `JoinPointShadowSet`s stored in the system has also been added.
When an AspectJ/FIAL build is started, the system is reset by first calling `clearShadows()`. Then, all Aspects are undeployed and `clearJoinPointShadowSets()` is called. This results in a FIAL instance that behaves as if it was just initialized.

Incremental building

When a project is built, the most simple approach is to recompile all source files in the project. This is called a *full build*. However, Eclipse provides incremental project builders. When an incremental project builder is invoked on a project that was previously compiled, it can perform an *incremental build* that only rebuilds information or output files that need to be rebuilt because the corresponding input files have changed. The AspectJ/FIAL builder partly supports incremental compilation. Because FIAL does not allow the aspect-oriented model of an application to be changed on the fly, only the collection of join point shadows is performed incrementally.
Figure 3.3 shows how collecting shadows incrementally is realized. Shadows are managed on a per-class basis. The circles represent the information supplied by the build system, which consists of three sets of classes: classes that were deleted, classes that were modified and classes that were added to the project. When an incremental build is performed,

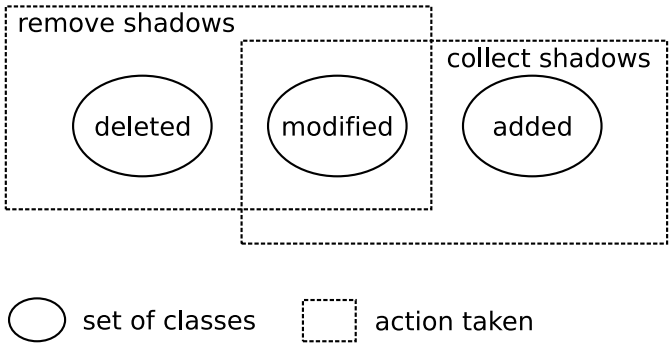

Figure 3.3: Collecting shadows incrementally

the builder first removes all `JoinPointShadow`s that were collected from deleted or modified classes from its internal pool of `JoinPointShadow`s. Then, `JoinPointShadow`s are collected from all classes that were either modified or added to the system. This results in a set of `JoinPointShadow`s that are consistent with the incrementally compiled Java code.

Source code locations

The processing of join point shadows and advice takes place on the bytecode level. However, the desired code visualizations actually work on source code elements. For this reason, the generated `JoinPointShadow` and `AdviceUnit` objects need to be linked to their original statements in the source code. For this, the line number of the statement that the model entity represents is needed.
For join point shadows, this is straight-forward. Because a join point shadow is actually a bytecode instruction, the number of the line from which it was created can be obtained from debugging information in the bytecode. For advice, this is not done as easily.
When the AspectJ compiler compiles an Aspect, a normal Java class file is generated. For each advice, this class file contains a method with the code of the advice body. The debugging information in this method only contains the line numbers of the actual advice body, not the declaration.
In order to associate model elements with the piece of source code they were generated from, a `SourceLocation` class was introduced in FIAL. This includes the file name of the source code file, the name of the java class as well as a start line and an end line for the code in question.
The AspectJ compiler was modified to generate special `@LineNumbers` annotations for each advice method. The `@LineNumbers` annotation contains the start and end lines of the advice, including the advice declaration. This annotation is then processed by the AspectJ annotations importer, which creates a `SourceLocation` object that is attached to every `AdviceUnit` as well as to every `BoundAction` that is created when the former is deployed. In this way, determining the start lines of advice declarations is straightforward when processing FIAL model objects.
The `SourceLocation` associated with `JoinPointShadow`s is currently only required for

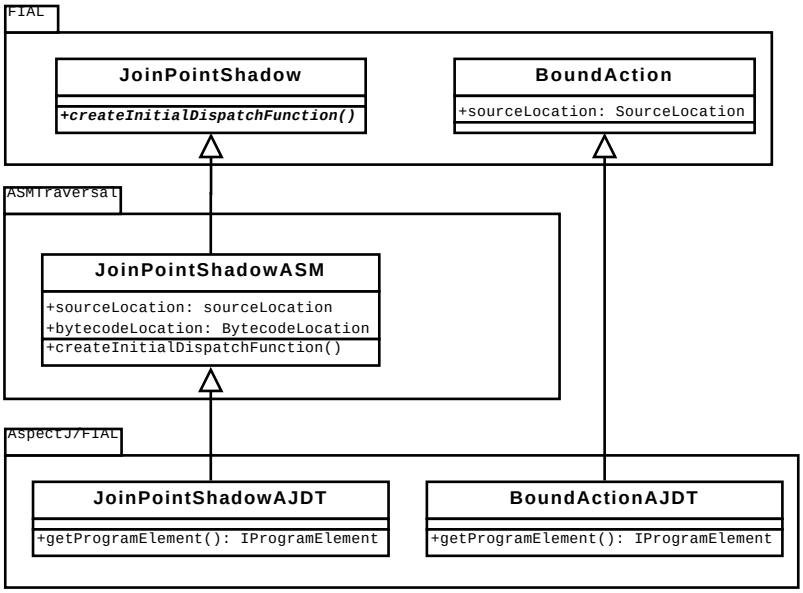

Figure 3.4: Extensions to FIAL make linking model entities with source code lines possible, thus allowing FIAL model elements to be linked with ASM hierarchy elements.

the AspectJ/FIAL-specific subclass of `JoinPointShadow`. This could be made mandatory for all FIAL instances by requiring a `SourceLocation` to be present in the `JoinPointShadow` base class provided by FIAL.
Figure 3.4 shows how FIAL was extended by ASM Traversal and the AspectJ/FIAL plugin. Information about the source code token that is represented by a model object is contained in the `SourceLocation` attributes of `BoundAction` and `JoinPointShadowASM`. The classes `BoundActionAJDT` and `JoinPointShadowAJDT` use that information to link FIAL model objects to `IProgramElements` in the ASM hierarchy.

4 Related work

This chapter gives an overview of some existing development tool support in the domain of aspect-oriented programming languages. CaesarJ and Compose* are both aspect-oriented languages for which cross-cutting visualization support exists. However, the approaches differ with respect to the language itself, the kinds of visualizations that are supported and the code models that are used to enable those visualizations.

4.1 CaesarJ

CaesarJ is an extension to Java that is being developed at TU Darmstadt. It supports collaborations of multiple classes as a formalized language construct called a *cclass*. This is combined with mixins and virtual classes as a form of aspect-oriented programming. Pointcut-advice AOP features as seen in AspectJ are also supported in CaesarJ [AGMO06].
Development tool support is available in the form of the CaesarJ Development Tools (CJDT). The CJDT is a version of AJDT that was modified for the CaesarJ language. It includes the *CaesarJ outline view*. This is similar to the Java outline view in the Java Development Tools (JDT) in that it displays the members of the current class or cclass in a tree view. Additionally, the *advises* relationship is also displayed in the same tree view. This is different from the approach taken in AJDT where cross-cutting structure is displayed in a separate tree view, the cross-references view. Also, a *CaesarJ hierarchy* is available that displays the structure of cclasses along with their mixins [UZ04, chapter 5].
The visualizations are generated from the AST that is built by the CaesarJ compiler. Because the latter does not support incremental compilation, the visualizations are only updated after a full build of the project, not while editing code [UZ04, chapter 6].

4.2 Compose*

Compose* is an aspect-oriented language that is based on the composition filters approach and is being developed at the University of Twente. In this approach, object-oriented classes are composed in *concerns*. A *concern* can intercept messages (method calls) sent to and from the objects it contains using input filters and output filters. These filters select messages by pattern matching, for example by the receiving class or by method name in a similar way as pointcuts are defined in AspectJ. The filters can be used to express cross-cutting by modifying the way the message is dispatched, for example by redirecting the message to a different object or by suppressing the message [BA01].

Compose* allows superimposition of filters on classes written in any language supported on the .NET or Java platforms. For Compose*, a prototypical graph-based visualization is available. It currently supports three views: the *program view*, the *filter view* and the *filter action view*. The program view is based on the UML class diagram and visualizes all concerns of the program with their superimposition as a way to give the developer an overview of where cross-cutting behavior is defined. The filter view visualizes the dispatch of messages by showing all filters that apply to a particular class on top of an UML class diagram of the latter. The filter action view shows the dispatch of a particular message as it is filtered. This is realized by a flowchart that is laid out along a number of lanes. The lanes are used to visualize in which concern the part of the processing takes place [Hen07, chapter 6.3].
The visualizations are created from the Compose* repository, in which the Compose* compiler gathers all information that is needed for weaving. This differs from the ASM, which was designed exclusively for enabling visualizations and code transformations in AspectJ. Incremental compilation is supported in Compose* so up-to-date model information is available after editing code [Spe06].
These visualizations are not part of an IDE but are realized as a stand-alone program [Hen07, chapter 6.4]. This makes the level of integration that is possible with the AJDT or the CJDT difficult to achieve, especially with regard to navigating source code interactively. A solution to this is proposed in [Hen07, chapter 6.4], but it is not yet implemented in the current Eclipse version of the Compose* plugin.

5 Summary

The problem of providing visualization of cross-cutting program structure in the absence of static weaving has been approached in this thesis. It was further concretized towards a specific language and a specific kind of implementation of runtime weaving. It has been shown that one can employ runtime weaving for AspectJ as the aspect-oriented language and still be able to use AJDT's visualization features. This is possible because most of the infrastructure that is needed for identifying cross-cutting relationships is provided by the Framework for Implementing Aspect Languages (FIAL).
In order to reach the goal of this thesis, an Eclipse plugin was developed that transforms a model of cross-cutting program structure from the FIAL representation to the ASM representation used in the AJDT. As part of this integration work, a number of problems have been identified. These mostly stem from the fact that AspectJ and AJDT were designed with development tool support in mind, while FIAL's design focus is execution-oriented.
This for example means that incremental compilation is only partially supported with the approach taken in this thesis. While join point shadow information can be collected incrementally, the cross-cutting relationships need to be re-generated during every build.
The AspectJ/FIAL plugin provides the developer with visualization support for the advises relationship when no static weaving is done. This is not only useful in conjunction with FIAL runtime weaving, but can also be used when AspectJ's load-time weaving facility is employed.

6 Future work

6.1 Building relationships incrementally

Currently, relationships are always fully re-generated on an incremental build. This is because of limitations in FIAL, which has only a singleton instance that manages the application meta-model. Also, removing individual join point shadows or aspects is impossible.
As a precondition to incremental relationship building, it is required that FIAL can be instantiated in a multiton fashion. This allows one FIAL instance per Eclipse project and thus prevents unnecessary matching. This way, meta-model data can be retained in FIAL over incremental builds.
As a second precondition, FIAL has to support some way of removing or altering individual meta-model entities. This is non-trivial because the framework has to perform re-matching on any meta-model entities that have a relationship to the removed or altered meta-model entities. Also, for the new matching data to be useful in conjunction with incremental building, some form of change notification for the meta-model has to be provided. For the current AspectJ/FIAL implementation, this is ideally a list of all `BoundActions` that have been removed or added as a result of changes in the current meta-model.
The `AJProjectModel` that is AJDT's model abstraction layer for the ASM already provides facilities to remove relationships that are no longer valid because the compilation unit that contained one of the ends of the relationship has changed. This can be used for removing invalid relationships once the `BoundAction` for the relationship that has become invalid is known.
Once the aforementioned preconditions are met by FIAL, a relationship can easily be generated using the `RelationshipManager` of AspectJ/FIAL.

6.2 Full AspectJ support

As was mentioned briefly in section 2.1, AspectJ supports cross-cutting that is not expressed by the *advises* relationship. This includes *inter-type declarations* that can add fields or methods to matched classes. Also, the *declare error / declare warning* facility can be used to cause compiler warnings and errors on lines that are matched by a pointcut expression. Those are available as relationships in the ASM. The feature of named pointcuts, which is not cross-cutting, is also expressed as a relationship in the ASM by the *uses pointcut* relationship between advice and named pointcuts. Those relationships are currently not supported by AspectJ/FIAL.

The features mentioned before can be modeled in FIAL. However, the AspectJ annotations importer that generates the aspect-oriented parts of the application meta-model used by AspectJ/FIAL does not yet support this. Once it does, supporting AspectJ fully in the AspectJ/FIAL plugin is straight-forward. The only precondition that needs to be satisfied is that the source code location of both ends of the relationship in question is known. For named pointcuts, this involves making the location of the pointcut declaration available in the corresponding meta-model entity in a similar way as has been done with advice.

6.3 Cross-project references and binary aspects

AspectJ/FIAL only analyzes the Eclipse project's output folders for class files from which to collect join point shadows. This limits the visualization of cross-cutting relationships to the current project. However, this limitation does not hold for AspectJ itself. AspectJ can match join point shadows that are in any project that the current project depends on. Also, join point shadows in JAR files can be matched. This can be supported in AspectJ/FIAL by traversing the entire class path of the project using the class dependency-oriented facilities provided by ASM Traversal.
The AspectJ build options also allow an ASPECTPATH to be specified. This path is searched for any pre-compiled aspect class files that are then woven into the current project. This can be supported by AspectJ/FIAL by either using a dedicated AspectJ annotations importer for each ASPECTPATH entry or by modifying the AspectJ annotations importer to accept an optional ASPECTPATH that is then searched by the importer for binary aspects. Of course, this requires that the aspects in the ASPECTPATH entries have been compiled in such a way that the importer can actually extract aspect-oriented meta information from them.

A Installation

AspectJ/FIAL is only available for Eclipse Europa (3.3). In addition to an Eclipse installation, the AJDT feature is required (tested with `ajdt-1.5.3_200805131655`).
AspectJ/FIAL is experimental software and contains a modified version of the AspectJ compiler binaries that replace the currently installed version of the AspectJ compiler. For this reason, it is recommended to install AspectJ/FIAL in its own Eclipse installation. Installing AspectJ/FIAL in an Eclipse installation that is used productively is not recommended at all.
Currently, the AspectJ/FIAL plugin is only available as an archive of Eclipse plugins. To install AspectJ/FIAL, unpack the archive in the `plugins` folder of your Eclipse installation and restart Eclipse. After Eclipse has been restarted, the AspectJ/FIAL sub-menu is available in the context menus of projects.

Bibliography

[AGMO06] Aracic, Ivica, Vaidas Gasiunas, Mira Mezini and Klaus Ostermann: *An Overview of CaesarJ.* In *Transactions on Aspect-Oriented Software Development I. LNCS*, pages 135–173, February 2006.

[BA01] Bergmans, Lodewijk and Mehmet Aksit: *Composing crosscutting concerns using composition filters.* Commun. ACM, 44(10):51–57, 2001.

[BMH+07] Bockisch, Christoph, Mira Mezini, Ralph Havinga, Lodewijk Bergmans and Kris Gybels: *Reference model implementation.* Technical Report AOSD-Europe Deliverable D96, AOSD-Europe-TUD-6, TU Darmstadt, September 2007.

[Boc08] Bockisch, Christoph: *An Efficient and Flexible Implementation of Aspect-Oriented Languages.* PhD thesis, TU Darmstadt, 2008.

[CCK03] Clement, Andy, Adrian Colyer and Mik Kersten: *Aspect-Oriented Programming with AJDT.* In *AOSD Workshop on Analysis of Aspect-Oriented Software*, July 2003.

[GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns.* Addison-Wesley Professional, January 1995.

[Hen07] Hendriks, Michiel: *Construction and Visualization of Dispatch Graphs for Compose*.* Master's thesis, University of Twente, August 2007.

[KCCC06] Kersten, Mik, Matt Chapman, Andy Clement and Adrian Colyer: *Lessons learned building tool support for AspectJ.* In *AOSD - Industry Track Proceedings*, 2006.

[Ker02] Kersten, Mik: *AO Tools: State of the (AspectJ) Art and open Problems.* In *Workshop on Tools for Aspect-Oriented Software Development at OOPSLA '02*, 2002.

[KHH+01] Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold: *An Overview of AspectJ.* In *Proceedings of ECOOP 2001 - Object-Oriented Programming: 15th European Conference, Budapest, Hungary*, pages 327–354, Heidelberg, June 2001. Springer Verlag.

[Spe06] Spenkelink, Dennis: *Incremental Compilation in Compose**. Master's thesis, University of Twente, October 2006.

[UZ04] Unger, Jochen and Daniel Zwicker: *User's Guide for CaesarJ Development Tool*. `http://caesarj.org/index.php/Caesar/CDTUserGuide`, October 2004.