Interoperable Trust Assurance Infrastructure

Title of deliverable

Type (distribution level)	Public
Contractual date of Delivery	31-03-2013
Actual date of delivery	27-03-2013
Deliverable number	**D4.1**
Deliverable name	**Analysis and Selection of the AOP framework**
Version	V1.0
Number of pages	69
WP/Task related to the deliverable	WP4/T4.1
WP/Task responsible	WP4/UMA
Author(s)	Lidia Fuentes, Nadia Gámez, José Miguel Horcas, Mónica Pinto (UMA), Wissam Mallouli (MI)
Partner(s) Contributing	*UMA, MI, IT, INDRA*
Document ID	INTER-TRUST-T4.1-UMA-DELV-D4.1-AnaSelecAOPFramework-V1.00

Abstract *This report provides a detailed analysis of the existing AOP (Aspect-Oriented Programming) frameworks and proposes the ones that are best suited to fulfil adequately the INTER-TRUST requirements, regarding the dynamic negotiation of security policies. The document begins with an introduction and a background section presenting the main concepts of AOP. Then, a taxonomy of the variability dimensions used to classify the existing aspect weavers is presented, as well as a list of the main existing proposals. Among existing proposals, only a few of them fit the concrete necessities of the INTER-TRUST project. This subset of AOP frameworks is evaluated to select the ones that best fit the requirements of the project.*

Executive summary

This deliverable is the main output of Task 4.1 (Selection of an existing dynamic AOP framework) in WP4 (Techniques & tools for secure interoperation enforcement and supervision). As part of task 4.1 we have investigated the existing AOP frameworks to identify which ones are the most suitable to support the dynamic enforcement of security requirements demanded by INTER-TRUST. Our investigation was focused on how the different AOP frameworks perform the weaving of crosscutting concerns (modelled as "aspects"). The results of the analysis we have performed is reported in the present deliverable. In order to propose the best AOP solution, we considered in this study mainly the requirements defined in WP1 (D2.1.1) and software architecture defined in WP4 (D4.2.1).

This document is mainly organized in three sections that are **background**, **classification of aspect weavers** and **evaluation and selection of aspect weavers**. AOP introduces a new programming paradigm that needs to be known and understood by all the project stakeholders in order to make a correct, and specially an effective, use of AOP in both the separation of the security concerns modelled by the INTER-TRUST framework, and in the later composition of those concerns with the core functionality of the project case studies. Thus, the purpose of the **background** section is to provide a general introduction to AOP, describing the new terminology used in AOP (e.g. *aspect*, *join point*, *pointcut*, *advice*, *weaver*…). For illustration purposes, a small example of an aspect-oriented application with security is presented.

Regarding the **classification of the aspect weavers** section, it includes the criteria used to perform the study of AOP frameworks that better fit the INTER-TRUST requirements. Study AOP frameworks nowadays has become a challenging task, since the number of AOP approaches has considerably increased during the last years. Also, there is a high variability in the features each AOP framework supports (e.g. programming language, aspect weaving time, etc.). So, in this section we present a taxonomy of features relevant to aspect-orientation that we have defined as part of task 4.1. The goal of this taxonomy is to define the classification criteria used to study the existing AOP approaches both research and industrial ones. Then, a list of existing AOP approaches, classified conformed to these criteria is briefly presented.

The goal of this document is not to perform an exhaustive study of all existing AOP approaches, but only of those that satisfy the INTER-TRUST framework requirements. So, in the **evaluation and selection of aspect weavers** section we establish which aspect-oriented features are relevant for the INTER-TRUST needs, and the list of AOP frameworks that meet these features. In order to select the aspect-oriented relevant features, we consider as inputs the first versions of the INTER-TRUST requirements (D2.1.1) and software architecture (D4.2.1) specifications. We also have taken into account the answers to a questionnaire made to the industrial partners, responsible of the case studies that will be used to test the INTER-TRUST framework.

From the analysis of this evaluation we have to choose the AOP approach(es) that will be used along the project. The main result of the analysis is that there is not a "one-for-all" AOP framework that can be used in the project, basically because of three reasons: (1) the use of different programming languages in different parts of the use cases that are going to be used to evaluate the INTER-TRUST framework; (2) the different degrees of dynamicity that are required by the AOP framework, and (3) the environment constraints that the different AOP frameworks impose make them not suitable to

be used in all the possible execution environments. This does not have to be viewed as a shortcoming of using AOP in the project, basically because the use of a particular AOP framework is only an implementation detail and more than one AOP weaver can be used without a problem in different instantiations of the INTER-TRUST framework. This will be possible because the software architecture of the INTER-TRUST framework will be independent of the use of a particular AOP framework.

Table of Contents

1 Introduction

1.1 Scope of the document

The main objectives of the INTER-TRUST project are to design security mechanisms that allow a secure interoperation between different parties, and to be able to dynamically adapt the security rules that drive that interaction due to changes on the running environment. AOP (*Aspect-Oriented Programming*) can contribute to achieve both objectives by firstly separating the security concerns in aspects and composing them later when needed. On the one hand, security is a typical crosscutting concern [1] and AOP has already been proven as useful to separate the security concern from the core functionality of the applications that require to be secured [2, 3]. On the other hand, the incorporation/weaving of aspects into an application can be performed at different stages of the development, including at runtime.

Due to the large number of AOP approaches existing nowadays, the first step toward the use of AOP in this project is necessarily the selection of the AOP approach that is best suited for satisfying the INTER-TRUST framework requirements. In order to do that, this deliverable has two main objectives. The first one is the **analysis of existing AOP approaches**. The second one is the **selection of an AOP weaver** that satisfies the requirements of the INTER-TRUST framework.

In order to select the most appropriate AOP approach this deliverable has dependencies with these other deliverables:

- D2.1.1 Requirements Specification first version (delivered on month 4 in WP2). The INTER-TRUST framework requirements are taken into account in order to decide the best AOP proposal for the project.

- D2.2.1 Gap and Standards Analysis first version (delivered on month 4 in WP2). This document contains an exhaustive description of the V2x and e-voting use-cases, including their reference architectures. This information is needed in order to better understand the weaving requirements of the use-cases in which the INTER-TRUST framework will be tested.

- D4.2.1 Specification and Design of the INTER-TRUST framework and tools first version (to be delivered on month 6 in WP4). The software architecture of the INTER-TRUST framework will provide relevant information that should be known in order to choose the best AOP weaver for the project.

1.2 Applicable and reference documents

This document refers to the following documents:

- D2.1.1 Requirements Specification first version (delivered on month 4 in WP2).

- D2.2.1 Gap and Standards Analysis first version (delivered on month 4 in WP2).

- D4.2.1 Specification and Design of the INTER-TRUST framework and tools first version (to be delivered on month 6 in WP4).

1.3 Revision History

Version	Date	Author	Description
0.1	08/02/2013	Lidia Fuentes, Nadia Gámez, José Miguel Horcas, Mónica Pinto (UMA)	Initial version and outline
0.2	27/02/2013	Wissam Mallouli (MI)	Review of sections 1, 2 and 3. Initial version of section 4.1 is provided.
0.3	08/03/2013	Lidia Fuentes, Nadia Gámez, José Miguel Horcas, Mónica Pinto (UMA)	Dependencies with other deliverables have been reviewed. Section 3 has been updated. Initial versions of sections 4.1 and 4.2 are provided, integrating MI contribution. Appendix A has been included.
0.4	15/03/2013	Lidia Fuentes, Nadia Gámez, José Miguel Horcas, Mónica Pinto (UMA)	Review and complete the whole document. Sections 4.2, 4.3 and 4.4 have been included. Conclusions and Appendixes have also been included. References have been completed.
1.0	27/03/2013	Lidia Fuentes, Nadia Gámez, José Miguel Horcas, Mónica Pinto (UMA), Samiha Ayed (IT), Leyre Merle (INDRA)	Comments performed by the internal review process (by IT and INDRA) have been taken into account and incorporated into the document (by UMA).

1.4 Notations, Abbreviations and Acronyms

AJDT: AspectJ Development Tool

AOSD: Aspect-Oriented Software Development

AOP: Aspect-Oriented Programming

AJC: AspectJ Compiler

AS: Application Server

CBSE: Component-Based Software Engineering

CLI: Common Language Infrastructure

IDE: Integrated Development Environment

IoC: Inversion Of Control

JAAS: Java Authentication and Authorization Service

JVM: Java Virtual Machine

LTW: Load-Time Weaving

OOP: Object-Oriented Programming

RE: Regular Expression

UML: Unified Modelling Language

2 Background

The purpose of this section is to introduce the Aspect-Oriented Programming (AOP) technology to non-expert readers. We will explain the motivation and main benefits of AOP to the software development and also define the base terminology related with AOP. Finally, we will illustrate AOP with an example in AspectJ, the most well-known aspect-oriented language.

2.1 Separation of concerns

In Object-Oriented Programming (OOP), there are properties of an application that can be *dispersed* in multiple classes. Typical examples are properties such as logging or authentication, whose code is normally scattered in several classes. Even if they are well-encapsulated in a *logging* or an *authentication* class, the rest of classes requiring these properties need to include implicit calls to them. Also in Component-Based Software Engineering (CBSE) there are concepts that are replicated in several components; for instance, extra-functional properties such as security or persistence are defined in multiple different components. A graphical example is shown in Figure 2.1, in which three different services (`CourseService`, `StudentService`, and `MiscService`) encapsulating the main functionality of an application are *crossed* by extra-functional properties such as the `Security` module and the `Transactions` module, among others.

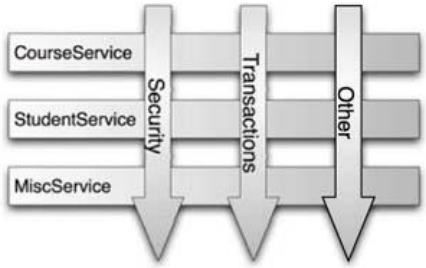

Figure 2.1. Properties that cut across several modules ([4]).

Incorporating these kinds of properties directly into the base code within the main functionality of an application causes **scattered code** – i.e., the specification of a property or functionality is dispersed in more than one module – and **tangled code** – i.e., a module contains descriptions related with several functional and non-functional properties, which are *intermingled* with the base functionality of the module. For instance, left side of Figure 2.2 shows an example of both scattered and tangled code in OOP. `Object1` and `Object2` represent the main functionality of an application while `Object3` and `Object4` represent extra-functional properties (e.g., authentication and trace). In order to incorporate these properties to the base code, `Object1` and `Object2` need to include implicit calls to them. On the one hand, code of `Object1` is tangled with the code of `Object3` and with the code of `Object4`, increasing the size of the code and the cost of maintenance. On the other hand, code of the property represented by `Object4` is scattered in `Object1` and `Object2`, which complicates finding redundant code and performing changes consistently.

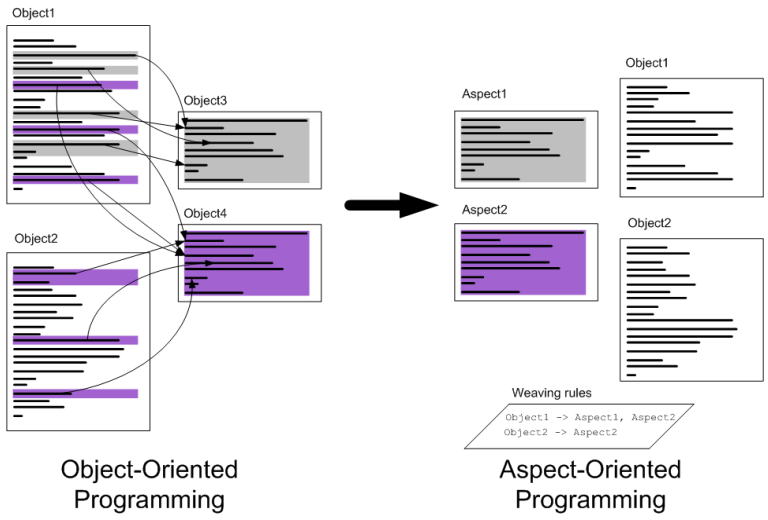

Figure 2.2. Separation of crosscutting concerns.

A property that causes both scattered and tangled code is known as a **crosscutting concern.** Crosscutting concerns appear due to the fact that existing modelling techniques for composition and decomposition of a system support a "dominant dimension" that guides the modelling process, implicitly or explicitly, through a certain hierarchical view of the organization of the system. For instance, OOP supports decomposition based on the dimension of the object: the class models a type of object and all the functionality associated with an object class (e.g., logging, trace) is described within the class. The disadvantage of these partitions is that many of the required functionalities of a system (in particular, the non-functional properties) are not well adapted to this decomposition criteria and have to be mixed with the concerns of the dominant dimension, affecting the separation of concerns. This problem is known as the **Tyranny of the Dominant Decomposition**.

Several techniques (e.g., design patterns, mixing classes) have been developed for dealing with the problem of modularization of crosscutting concerns. One of the most advanced and sophisticated technique is AOP. AOP is a programming paradigm that improves modularization by allowing the separation of crosscutting concerns out of the dominant dimension of the programming language (e.g., out of classes in OOP). Back to Figure 2.2 (right side), in AOP, implementation of crosscutting concerns represented by `Object3` and `Object4` are encapsulated in a new entity named **aspect** (`Aspect1` and `Aspect2` respectively), and code of `Object1` and `Object2` contains only the main functionality of the system excluding any reference to extra-functional properties. Interactions between base code and aspects are expressed through the **weaving** rules that will be described later.

Modelling of crosscutting concerns as a separate entity, such as an aspect, in which its implementation appears encapsulated only in a part of the program, smoothes coupling between modules and increases cohesion of each of them. Moreover, as consequence of **low coupling** and a **high cohesion**, the **maintainability** of the global system improves due to the fact that changes in a module affect only that module; and also the **reusability** improves due to both base code and aspects can be reused easier in different systems. There are a lot of crosscutting concerns that are usually useful to treat separately and so can be modelled as examples of aspects: logging, authentication, trace, coordination, synchronization, security, persistence, failover, error detection and correction, memory management, internationalization, localization, monitoring, product features, data validation, transaction processing, caching…

2.2 Composing aspects

AOP can be considered a meta-programming notation or a language extension pluggable on top of every existing host languages that allow the separation of crosscutting concerns through aspect-oriented abstractions (e.g., aspects). Once crosscutting concerns are separated from the main functionality of the system and encapsulated in aspects, the incorporation of the aspects into the system must be performed in a non-intrusive way – i.e., without modifying the existing code. The mechanism in charge of composing the main functionality of the system and the aspects is the **weaver**.

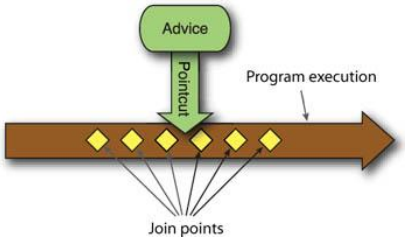

Figure 2.3. An aspect's functionality (advice) is woven into a program's execution at one or more join points ([4]).

To combine or weave the base code and aspects, the base language and the aspects language must interact together in order to add the additional behaviour contained in the aspects. Aspects are often described in terms of **join points**, **pointcuts** and **advices**[1]. Figure 2.3 illustrates how these concepts are tied together: AOP languages must provide a **joint point model** that defines the points of the program execution which can be affected by the aspect (e.g., method call, function execution, field access…). These points are called join points and a mean to refer to these join points is through the pointcuts. A pointcut is an expression that describes a set of join points where the code of the aspect should be executed in addition to the main behaviour of the program – i.e., the weaving rules. The additional code that affects the base program at the selected join points is the advice. Code that that is added to the base program can be within aspect implementation or outside, depending on the language. In asymmetric approaches this code (i.e. the advice) is inside the aspect, but in symmetric ones, the advice is outside, being implemented as a normal class or module of the base programming language. For instance, Figure 2.4 shows a complete example of an aspect, using the most well-known aspect-oriented language which is AspectJ. The aspect `TraceAge` is a class-like unit that encapsulates the implementation of the aspect (the concern *trace*). The additional behaviour and the weaving rules are defined together. The pointcut expression picks out all calls to the method `setAge` with any parameter and any returned value in the context of the variable `i`. The aspect code that will be incorporated is part of the advice and indicates when this code runs, which in this case will be **before** the code of the method `setAge`.

[1] This AOP terminology is strongly influenced by the AspectJ language, but is the most used by the AOP community.

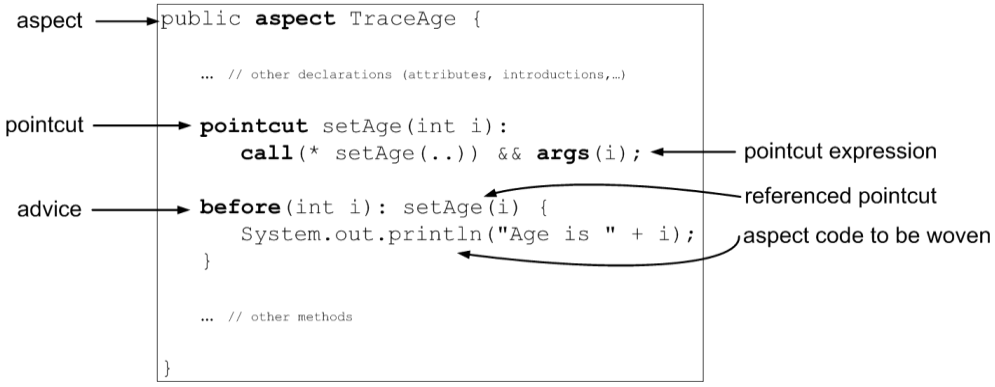

Figure 2.4. Example of an aspect.

The process of weaving that combines the base code with the aspects code by following the weaving rules can take place in different times in the lifetime of the program. Basically, the weaving can be performed at compile-time, at load-time, or at runtime. **Compile-time weaving** consists of adding the code of the aspect into the base code before the compiler produces the executable application, as Figure 2.5 shows. The aspects are strongly related to the classes and the resulting code is optimized. But in this case, aspects cannot be composed dynamically. When the base code is compiled or is in binary form, the process is known as **post-compile time weaving**, **link-time weaving** or **binary weaving**. **Load-time weaving** can be used in interpreted languages (e.g., Java) where aspects are woven when the target class is loaded by the *classloader* of the execution environment (e.g., JVM). Finally, in **runtime weaving**, aspects are woven in sometime during the execution of the application (Figure 2.6). Aspects can be added, adapted, replaced or removed dynamically at runtime as needed. Even the weaving rules (i.e. the pointcut) could be changed, at runtime, being possible to apply an aspect at different points of the running base code. This makes the system more customizable and extensible, but with a penalty in efficiency because it requires extra code to connect and disconnect the aspects and classes at runtime and to verify that these changes are performed safely.

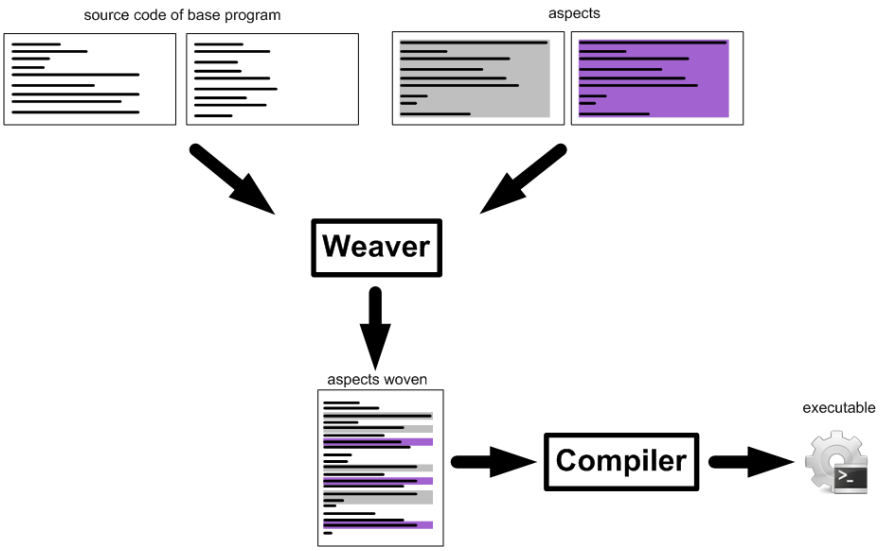

Figure 2.5. Compile-time weaving.

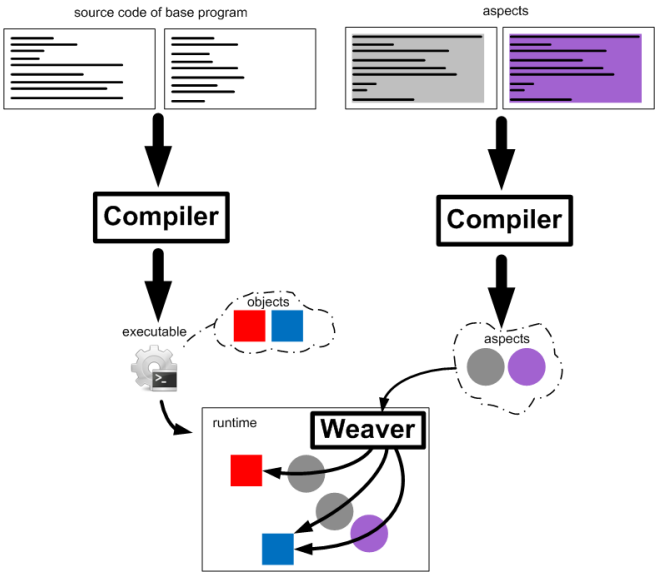

Figure 2.6. Runtime weaving.

Next section presents the basic and general terminology used in AOP.

2.3 Glossary

Like most technologies, AOP has formed its own jargon. Unfortunately, many of the terms used to describe AOP features are not intuitive. Nevertheless, they are now part of the AOP idiom, and in order to understand AOP, you must know these terms. There are over one hundred terms related to AOP. In this section, to simplify, we have chosen a little subset that contains the basic and most important concepts of AOP. An overview of these terms is given in the diagram of the Figure 2.7. The definitions presented in Table 2.1 are based on an AOSD ontology defined in the context of the AOSD-Europe project [5]. We encourage reading that reference work for a deeper and more complete knowledge of AOP terminology.

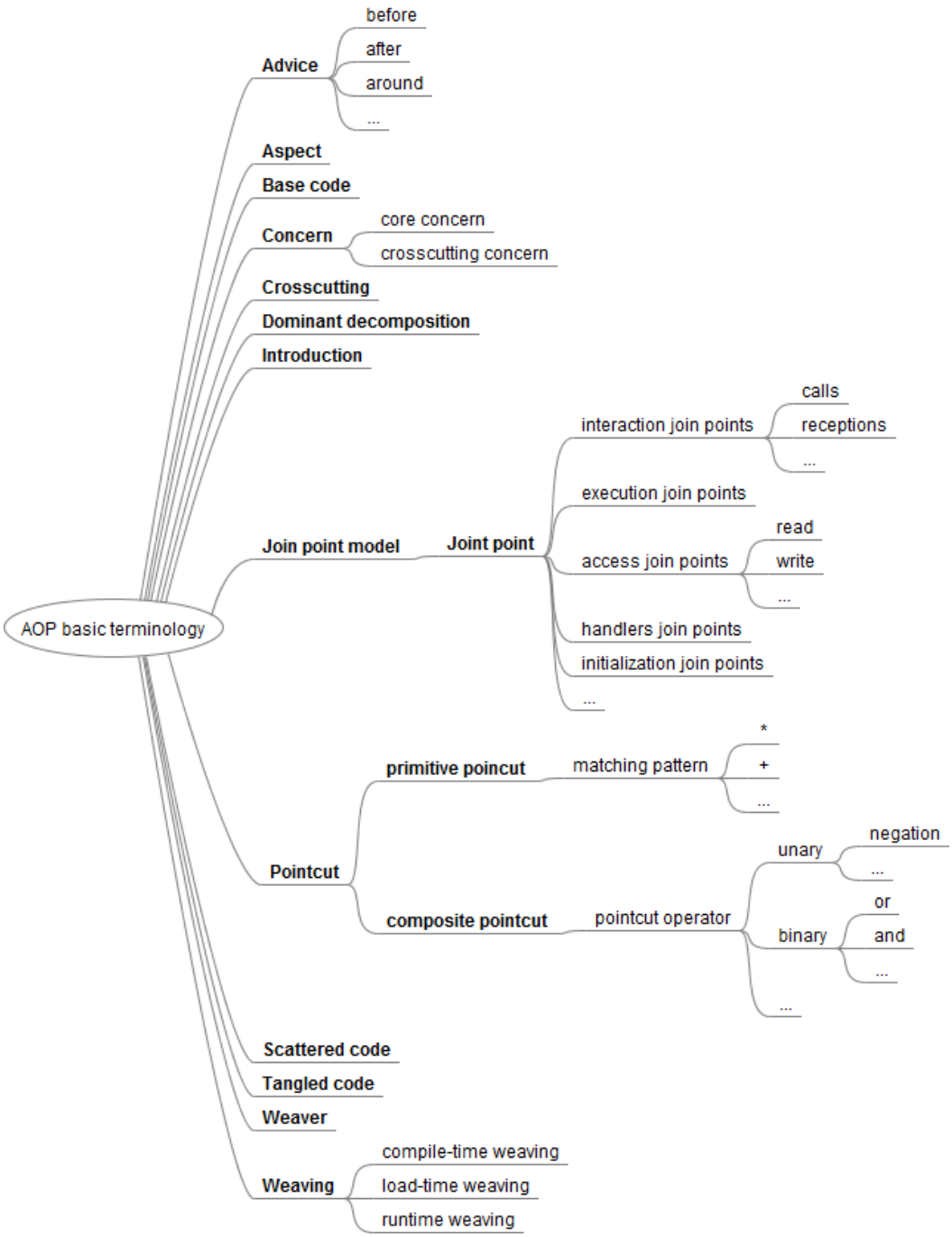

Figure 2.7. AOP basic terminology.

Table 2.1. AOP basic terminology.

Term	Definition
Advice	An advice is a way to express *crosscutting* action at the *join points* that are captured by a *pointcut*. There are usually three kinds of advices: *before*, *after*, and *around*. - **Before**: a before advice executes before the execution of the captured join point. - **After**: an after advice executes after the execution of the captured join point.

	- **Around**: an around advice can bypass the execution of the captured join point, execute it with different argument, execute it multiple times, and/or perform additional execution before and after the join point.
Aspect	An aspect is an entity that modularizes *crosscutting* concerns. Code that expresses the weaving rules can be within the aspect or outside, depending on the language.
Base code	Code of the original program in which *aspects* will be applied. For example, the Java classes and interfaces comprising the implementation of the original object-oriented decomposition.
Concern	A concern is a specific need that must be addressed in order to satisfy the overall system goal. After software modularization approach has been selected, there could be two types of concerns: *core concern* and *crosscutting concern*. - **Core concern**: a core concern captures the central functionality of a module in a system. - **Crosscutting concern**: a crosscutting concern captures requirements that cross multiple modules in a system. It should be noted that a concern can be crosscutting in respect with a certain set of modules (e.g., classes) but it can become non-crosscutting if an alternative set of modules is selected (e.g., features). Thus crosscutting exists only in relation to set modularization constructs.
Crosscutting	Crosscutting is the *scattering* and *tangling* of *concerns* arising due to poor support for their modularization. Crosscutting concerns can be identified at different stages of the software development process such as at specification level, at architectural level, at design level, at implementation level…
Dominant decomposition	There is a “dominant decomposition” when a programming language or software development approach allow only one possible decomposition of a problem domain (e.g. objects in POO), even if some concerns appear scattered/tangled. In this sense, one of the contributions of AOP is that solves the problem of the “tyranny of the dominant decomposition” in POO.
Introduction	An introduction is a static *crosscutting* instruction that introduces changes to the static structure of the system (e.g., classes, components, interfaces, aspects). An example of an introduction is the incorporation of new state attributes or methods to a class.
Join point	A join point is an identifiable point in the execution of a program in a certain context. The context of a join point contains the information about the current execution of the program (e.g., caller object, arguments of functions…). The following join point types are normally used in most AOP approaches: interactions join points (e.g., call method, reception method), execution join points (e.g., procedure execution, function execution), access join points (e.g., variable read/write access), handlers join points (e.g., exception handler, event handler), initialization join points, etc.

Join point model	A join point model is a model of a system in which *join points* are defined. The join point model defines the way to refer to these join points (e.g., through *pointcut)*, and a mechanism to affect the program at the selected join points (e.g., through *advice*).
Pointcut	A pointcut is an expression that selects *join points* by matching certain characteristics and collects context at those points. The matching pattern allows to group together specified join points by multiple signatures using artefacts (e.g. *, .., +). AOP languages usually provide some predefined pointcuts called primitive pointcuts (e.g., calls, receptions, executions…) and operators (e.g., negation, or, and) to compose more complex pointcuts combining primitive pointcuts.
Scattered code	Scattered code is code in which the implementation of a single *concern* is spread over multiple modules.
Tangled code	Tangled code is code in which the implementation of multiple concerns is handled simultaneously in a single module.
Weaver	A weaver is a processor that performs the *weaving* according to weaving rules.
Weaving	Weaving is the process of composing the *base code* with the *aspect* code by following the weaving rules. There are different kinds of weaving depending the moment in the lifetime of the program where the process takes place: *compile-time weaving*, *load-time weaving*, and *runtime weaving*. - **Compile-time weaving:** aspects are incorporated into the base code before the compiler produces the executable application. If base code is compiled or it is in binary form, the process is known as **post-compile time weaving**, **link-time weaving** or **binary weaving**. - **Load-time weaving:** aspects are woven in when the target class is loaded by the classloader of the execution environment (e.g., JVM). - **Runtime weaving:** aspects are woven in sometime during the execution of the program. Aspects can be added, adapted, or removed dynamically at runtime as needed.

2.4 An AOP example

This section describes an example of an aspect in order that the reader reaches a better understanding of all the terms and descriptions presented above. Among all available AOP languages, we have chosen AspectJ language since, as we already mentioned, it is the most commonly used language in AOP and was the first to coin the terms related to AOP.

We will use a **banking system** as example (taken from [6]) and we will integrate **authentication** as an aspect. Figure 2.8 shows the UML class diagram of the system. The `Account` interface defines the

basic public method on an account; the `AccountImpl` class implements the account; the `InsufficientBalanceException` class identifies insufficient balance; and the `InterAccountTransferSystem` class contains a method for transferring funds from an account to another. We show how to implement an authentication aspect for the system as well as we explain the main parts of AspectJ at the same time. We use the *Java Authentication and Authorization Service* (JAAS) that is one of the simplest ways to implement authentication and authorization in Java applications; but we separate it from the main functionality of the banking system.

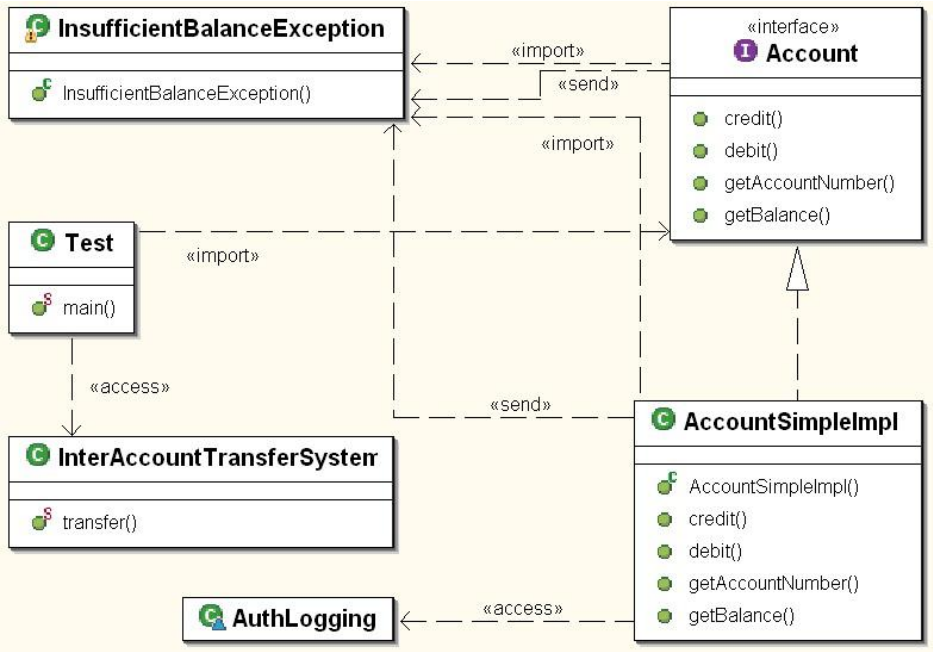

Figure 2.8. Banking system class diagram ([7]).

AspectJ allows intercepting method calls from the source object; method call receptions and executions on the target object. It can also intercept reading/writing of attributes and exceptions handling. Finally, it is possible to intercept the throwing of exceptions and the initialization of classes and objects. Pointcuts of an aspect in AspectJ determine what points of the execution of a program can be intercepted by that aspect. We consider the following pointcut expression (Figure 2.9):

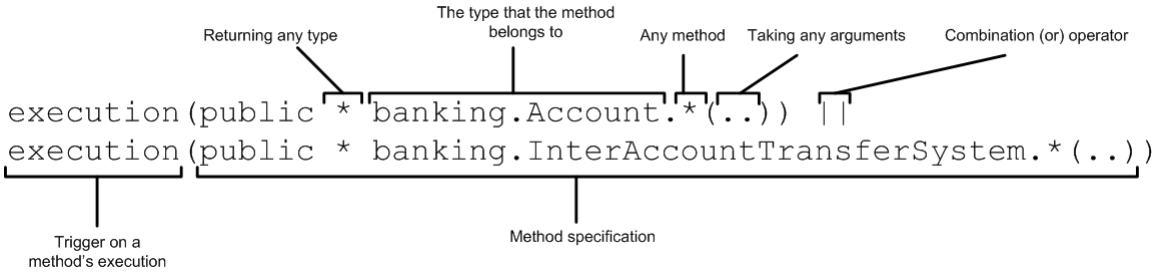

Figure 2.9. Pointcut expression.

This pointcut expression picks out all executions of any public method of the classes `Account` or `InterAccountTransferSystem`, in the package `banking`. The pointcut is completely defined as follows:

```
pointcut authOperations():

     execution(public * banking.Account.*(..)) ||

     execution(public * banking.InterAccountTransferSystem.*(..));
```

Pointcuts capture a set of join points and permit identifying them with a name (`authOperations` in the example). Pointcuts (e.g. `authOperations()`) are syntactic patterns expressed as a predicate composed by AspectJ primitive pointcuts. The following advice implements the crosscutting behaviour of the authentication concern:

```
before() : authOperations() {

     if(_authenticatedSubject != null) {

          return;

     }

     try {

          authenticate();

     } catch (LoginException ex) {

          throw new AuthenticationException(ex);

     }

}
```

The advice body runs before each join point captured by the pointcut `authOperations()`. It ensures that our code performs authentication logic only if this is the first time during the lifetime of the program that a method that needs authentication is being executed. To do this, the aspect stores the authenticated subject in an instance variable:

```
private Subject _authenticatedSubject;
```

and defines a method to perform the authentication with JAAS:

```
private void authenticate() throws LoginException {

     LoginContext lc= new LoginContext("Sample",new TextCallbackHandler());

     lc.login();

     _authenticatedSubject = lc.getSubject();

}
```

By storing the authenticated subject and checking for it prior to invoking the login logic, we avoid asking for a login every time a method that needs authentication is called. After a successful login operation, we can obtain this member from the `LoginContext` object. In our implementation, we will use the whole process as the login scope. Once a user is logged in, he will never have to log in again during the lifetime of the program. Depending on the specific requirements of the use cases,

we may want to move this member to an appropriate place. For example, if you are writing a servlet, you may want to keep this member in the session object. We also assume that a user, once logged in, never logs out. If this is not true according to the requirements of the use cases, we need to set this member to null when the current user logs out.

The first time the advice runs, `_authenticatedSubject` will be null, and the `authenticate` method will be invoked to perform the core authentication logic. When subsequent join points that need authentication are executed, because the `_authenticatedSubject` is already not null the login process will not be carried out. The core authentication operation is performed in the `authenticate` method. If the login fails, it throws a `LoginException` that aborts the program. If the login succeeds, it obtains the subject from the login context and sets it to the instance variable `_authenticatedSubject`. The `LoginException` is defined inside the aspect as:

```
public static class AuthenticationException extends RuntimeException {

     public AuthenticationException(Exception cause) {

          super(cause);

     }

}
```

Since the `LoginException` is a checked exception, the `before` advice cannot throw it. Throwing such exceptions would result in compiler errors. We could have simply softened this exception using the 'declare' soft construct. However, we instead define a concern-specific runtime exception that identifies the cause of the exception, should a caller wish to handle the exception. `AuthenticationException` is simply a `RuntimeException` that wraps the original exception. The completed code of the aspect is:

```
public aspect Authentication {

     private Subject _authenticatedSubject;


     pointcut authOperations():

     execution(public * banking.Account.*(..)) ||

     execution(public * banking.InterAccountTransferSystem.*(..));


     before() : authOperations() {

          if(_authenticatedSubject != null) {

               return;

          }

          try {

               authenticate();
```

```
          } catch (LoginException ex) {
               throw new AuthenticationException(ex);
          }
     }

     private void authenticate() throws LoginException {
          LoginContext lc = new LoginContext("Sample",
                                             new TextCallbackHandler());
          lc.login();
          _authenticatedSubject = lc.getSubject();
     }

     public static class AuthenticationException extends RuntimeException {
          public AuthenticationException(Exception cause) {
               super(cause);
          }
     }
}
```

In this way, we now have a system with authentication modularized in one reusable aspect. Once classes and aspects that form part of the application are developed, both codes should be mixed to build the final application. The *ajc* compiler provided by AspectJ transforms the original code of the Java files to insert aspects code in the suitable join points. Finally, the resulting code is compiled with the traditional Java compiler. If instead of providing the source files of the application, we have provided the compiled classes, the *ajc* compiler mixes the aspects code with the bytecode. AspectJ also allows load-time weaving, deferring the weaving until the point that the class files are loaded into the JVM. In all cases, the separation between the objects and aspects is lost at runtime.

3 Classification of aspect weavers

3.1 Taxonomy/Variability dimensions

There exists a wide variability of AOP frameworks (with their respective aspect weavers) with specific characteristics that make them more suitable than others regarding certain requirements of the target systems. In this section we will describe these characteristics to classify the aspect-oriented frameworks present in the market. The main features of AOP frameworks that must be taken into account in the INTER-TRUST project are mainly related with how each framework performs the weaving.

In order to specify the set of characteristics relevant to INTER-TRUST, we have defined a taxonomy of aspect-weavers. In order to show graphically this taxonomy we use a *Feature Model*, a widely used model in Software Engineering to formally express the variability of features present in a given domain. In this case, the domain is the aspect weaver and the variability is the set of aspect weaver properties that can vary in each AOP framework. Formally, a Feature Model [8, 9] is a hierarchical decomposition of features to specify which elements, or features, of a family of products (aspect weavers in our case) are common, which are variable and the reasons why they are variable, i.e. if they are alternative elements or optional elements. Then, a feature model specifies where the variability is and enables reasoning about all the different possible configurations of a family. A configuration of a feature model is the selection of a set of features belonging to the feature model. A configuration is valid if all features contained in the configuration and the deselection of all other specific features is allowed by the feature model [10, 11]. Every possible valid configuration could represent a potential custom-made aspect weaver, but only a subset of all these possibilities are implemented and available as AOP frameworks. In Section 3.2, we will analyse the main proposals in terms of the features that we present in this section.

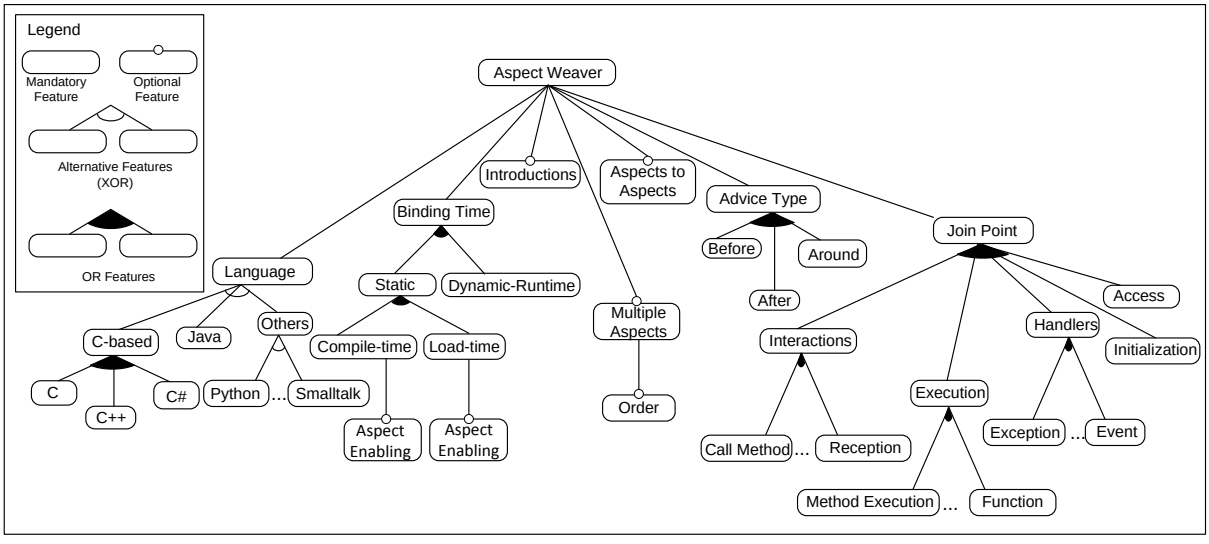

Figure 3.1. Feature Model that represents the aspect weaver characteristics.

Figure 3.1 shows the feature model with the aspect weaver characteristics. It can be observed that the **Language** feature is made up of an exclusive-or (XOR) group of features. This means that only one feature of that group can be selected in a valid configuration. Then, every aspect weaver is

useful for just one language but not for some languages at the same time. There are many **Java** and **C-based** (C, C++ or C#/.NET) aspect weavers. In addition, there are weavers for other languages as **Python** (e.g., Aspyct [12]) or **Smalltalk** (e.g., AspectS [13]), but in this document we will not analyse them since they are not interesting for the target applications and for the end users.

As it is described in the previous section, another important characteristic is the moment when the weaving is done. This is represented with the **Binding Time** feature that is composed by an OR group, since the weavers can be **static** or **dynamic**. The static weaving can be done at **Compile-time** or ad **Load-time** and the dynamic weaving is done at runtime. The **Binding Time** feature is very relevant for the selection of the AOP framework that will be used in INTER-TRUST, so we need to understand deeply what static and dynamic weaving means. In the static weaving, aspects are superimposed onto the primary functionality in an additive manner without altering the existing architecture. These aspects, which are woven, cannot be removed or reconfigured later during the runtime [14]. In next section we will explore the possibility of applying some design patterns, in order to achieve that the aspect could be added and removed at runtime using a static weaver.

On the other hand, once the system starts running, it may be adapted to the changing requirements during runtime. This is especially true in complex distributed systems, which exhibit strong dynamics. Dynamic weavers enable the aspects to be woven and unwoven from the system on the fly, which makes it useful for rapid prototyping, and enables the systems to adapt their services in response to changes in the requirements or in the environmental context. Dynamic weaving is a natural choice for implementing an adaptable system due to the reason that it can apply code retrospectively to a running application [15]. This helps avoiding recompilation, redeployment and restart of the application, and also allows an efficient use of resources since aspects do not have to be pre-loaded in the system. But, some dynamic weavers make excessive use of resources, so the latter advantage could be so beneficial. It is well known that the dynamic adaptation of complex software systems is generally dependent on policies, which all tend to be crosscutting concerns, and, hence are realized as dynamic aspects. This is basically an autonomous policy coordination facility that allows system to continuously adapt itself to a changing environment by determining which policy needs to be changed and how policies are recombined so that the system can keep performing well. Thus, for performing the job of weaving and un-weaving of features, realized as dynamic aspects, the dynamic weaver is an integral feature in the feature model of our family-based adaptable software systems.

Another way to achieve certain degree of dynamicity is exploiting the characteristic allowed by some aspects weavers, where aspects weaved at compile-time or load-time can be enabled or disabled at runtime. This is represented as optional features (**Aspect Enabling**) in the feature model.

Another relevant issue in the classification is if the weaver supports certain programming related extension mechanisms or not. They are represented in the feature model as optional features meaning that they are not so relevant in the selection of the AOP framework for INTER-TRUST, but are important to know them in order to evaluate the future difficulties of programming the INTER-TRUST framework. There are weavers that allow **Introductions** or static crosscutting, to be able to introduce structural changes to classes, interfaces and so on. Furthermore, in some cases, there is a need to define the **order** of execution of the advices, to resolve the conflicts between different advices, affecting the same join points (when **multiple aspects** per joint point are allowed). Finally, several weavers allow intercepting an aspect to apply other aspects (**Aspects to Aspects** optional feature). This helps to define recursive definition of aspects.

The **Advice Type** allowed by the weaver is represented by a feature made up of an OR group with the three main kinds of advices described in Section 2: **Before**, **After** and **Around**. Many weavers support the three ones but others only one of them as the **Around** advice, being the addition of aspects more intrusive than using the before and around advice types.

Finally, the **Join Points** supported indicates which points in the execution of a program could be intercepted. As it was previously detailed there are several types of join points as **Interactions**, **Execution**, **Access**, **Handlers** or **Initializations**. They are represented in an OR group in the feature model and everyone is made up of another OR group with the specific points to be intercepted. As the number of these points is very high, we have only represented few of them (**call method**, **function**, **events**, and so on).

3.2 Main proposals characteristics

The purpose of this section is not detailing every existing AOP framework, but only analysing the most important ones that could suit better with our project, in terms of the features described in the previous section. In the list provided by aosd.net [16] and updated in September 2011, the tools with a significant user base are AspectJ [17], JBoss AOP [18], PostSharp [19] and Spring [20]. The first of two are java-based and the last two are for .NET. Apart from these four proposals we have analysed several others tools proposed by aosd.net and we have included the ones more related with our project (as Spring.NET [21], JAC [22], AspectC++ [23], Aspect# [24] and AspectJS [25]), in Table 3.1.

Now, we will present a short description of the weavers of every tool and after we will summarize their characteristics.

- **AspectJ** [17] is a seamless aspect-oriented extension to Java that enables the modular implementation of a wide range of crosscutting concerns. **AspectWerkz** [26] is a dynamic, lightweight and with high-performance AOP/AOSD framework for Java (from March 2005 AspectWerkz has been merged with AspectJ). AspectJ consists primarily of a compiler and a weaver. The compiler is an extended version of the JDT compiler that understands the new language extensions that AspectJ supports beyond pure Java. The weaver supports weaving at compile time or at class load time, the latter being done via an adapter that plugs into an existing classloader. The bytecode produced by the weaving process is pure java and will run on any JVM. The advantage of compile-time weaving is that AspectJ provides excellent UI feedback on how the aspects are applying to your code, the advantage of load-time weaving is increased flexibility. Load time weaving is done by modifying some existing class loader to delegate to the AspectJ weaving adapter when it loads any bytes.

- **PostSharp** [19] is a build-time weaver for Microsoft .NET. It has a pragmatic design and offers original features, like instantiation, initialization and binary serialization of aspects at build-time. The objective of PostSharp is to bring aspects to people who do not even know what an aspect is. Everyone in .NET knows that a custom attribute is "something that modifies a behaviour" and PostSharp allows to modify add behaviours to code in a natural way.

- **JBoss-AOP** [18] is the Java AOP architecture used for the JBOSS application server. With JBoss AOP advice and interceptor bindings can be changed at runtime. Existing bindings can be unregistered and new bindings can be deployed if the given join points have been instrumented.

Moreover, it is as well possible to weave aspects into application code using a pre-compiler as weaving at load-time.

- **Spring** [20] approach regarding AOP capabilities differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications. Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a Servlet container or application server. Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. Therefore, advice very fine-grained objects cannot do easily or efficiently with Spring AOP. In such cases AspectJ would be the best choice. Spring AOP will never strive to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring AOP and full-blown frameworks such as AspectJ are valuable, and that they are complementary, rather than in competition. In fact, Spring 2.0 seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP to be catered for within a consistent Spring-based application architecture.

- **Spring.NET** [21] AOP support goal, similar as happen with Spring, is not to provide a comprehensive AOP implementation on par with the functionality available in AspectJ. However, Spring.NET AOP provides an excellent solution to most problems in .NET applications that are amenable to AOP. Thus, it is common to see Spring.NET's AOP functionality used in conjunction with a Spring.NET IoC container. Spring.NET AOP is implemented in pure C# and there is no need for a special compilation process, since all weaving is done at runtime. Likewise Spring, it currently supports just the interception of method invocations and provides classes to represent pointcuts and different advice types.

- **AspectC++** [23] is an aspect-oriented extension to the C/C++ programming languages. It is based on source-to-source translation, translating AspectC++ source code to C++. It is available under the GNU GPL, though some extensions specific to Microsoft Windows are only available through pure-systems GmbH.

- **JAC** [22] is a Java framework for aspect-oriented distributed programming. Weaving occurs at runtime whenever a new pointcut is defined. Prior to that, the application-level classes are loaded and transformed by Byte Code Engineering Library. The JAC run-time weaver is implemented as a Java class library.

- **Aspect#** [24] is an AOP framework for the CLI (.Net and Mono). It relies on **Castle's DynamicProxy** [27] and offers a built-in language to declare and configure aspects, and is compliant with AopAlliance. The weaving is done with Castle DynamicProxy, which is used by Castle Windsor to manage crosscutting concerns with the concept of *Interceptors* and the interface *IInterceptor*. The basic is the ability to create a component that is able to intercept every call to methods and or properties of an interface, and deciding with Windsor configuration where to apply this interceptor.

- **AspectJS** [25] is a fully-featured library that implements aspect-oriented programming in JavaScript. AspectJS is an industrial-strength JavaScript component that gives full control over method-call interception. JavaScript's dynamic nature supports a simple principle, such that a call to an object-method can cause other functions to execute as 'prefixes' and/or 'suffixes' to that method. From the point of view of callers and callees, such redirection of execution occurs transparently, and this technique is called Method-Call Interception or 'MCI'.

Table 3.1. Main proposals characteristics.

Proposal	Language	Binding Time	Introductions	Multiple Aspects	Aspects to Aspects	Advice Types	Join Points	Tool Support
AspectJ	Java	Compile-time Load-time (Enabling/ Disabling Aspects)	Yes	Yes (Order)	Yes	Before After Around	Method Constructor Initialization Access Exception	Yes, maintained
PostSharp	C# (.NET)	Load-time	Yes	Yes (Order)	Yes	Before After	Method Access Property Event	Yes, maintained
JBoss-AOP	Java	Load-time Run-time	Yes	Yes (Order)	No	Around	Method Access Within Withincode Has Hasfield	Yes, but no longer maintained
Spring	Java	Run-time	Yes	Yes (Order)	No	Before After Around Throws	Method	Yes, maintained
Spring.NET	C# (.NET)	Run-time	Yes	Yes (Order)	No	Before After Around Throws	Method	Yes, maintained
AspectC++	C++	Compile-time	Yes	Yes (Order)	No	Before After Around	Function Object construct.	Yes, maintained

							Object destruction	
JAC	Java	Load-time Run-time	No	Yes (Order)	No	Around	Method Constructor	Yes, but no longer maintained
Aspect#	C# (.NET)	Run-time	Yes	Yes (Order)	Yes	Around	Method/Interface Properties	Yes, maintained
AspectJS	Java	Load-time	No	Yes (Order)	No	Before After Around	Function, Method , Object, Exception	Yes, maintained

In Table 3.1, we can observe that there are some proposals that only perform static-weaving (compile or load time) as AspectJ, PostSharp, AspectC++ and AspectJS. Instead, in others the weaving can be dynamic as JBoss-AOP, Spring , Spring.NET, JAC and Aspect#. Furthermore, some weavers, as the AspectJ one, support the enabling/disabling of aspects at runtime.

All proposed tools accept multiple aspects per joint point and they allow the indication of the aspect execution order. Most of them (all less JAC and AspectJS) support the introductions or static crosscutting and most of them (but JBoss AOP and AspectJS) support aspects to aspects feature.

There are also differences in the advice types supported. Many of them use the three typical kinds of advices (*Before*, *After* and *Around*) as AspectJ, AspectC++, AspectJS, Spring and Spring.NET (the latter two also define *Throws* advice to be executed if a method throws an exception). However, others only support the *Around* advice as JBoss-AOP, JAC and Aspect#, and others only support the *Before* and *After* but not the *Around*, as PostSharp.

In the join points supported there are also a great difference. In one hand, we can find Spring and Spring.NET approaches, that are only able to intercept the execution of methods. On the other hand, AspectJ or JBoss-AOP are able to intercept many different points, for example, method execution and call, constructor execution and call, initialization, access (get/set), exception handlers, within/withincode or has/hasfield.

All of the proposals included in Table 3.1 have tool support but two of them are no longer maintained (JBoss-AOP and JAC). Finally, Table 3.2 shows a list of AOP frameworks that they are not considered in detail here since they are academic products not reliable to be used in this project.

Table 3.2. Other proposals characteristics.

LOOM.NET is a static aspect weaver that operates on binary .NET assemblies. The RAPIER-LOOM.NET library is a dynamic aspect weaver.
FeatureC++ supports feature-oriented and aspect-oriented programming in C++. The highlight of FeatureC++ is the combination of FOP and AOP concepts.
AspectDNG is a .NET multi-language aspect weaver (Aspects and base code may be written in any programming language (C#, VB.NET, J#...).

Aspyct is an AOP module for Python. It is intended to be powerful while being easy to use. Aspyct uses Python syntax, functions, classes so that no new knowledge is needed. Creating aspect classes is possible, as well as simple function advices. Works from Python 2.4 up to (and including) 3.0
phpaspect is a PHP language extension which implements aspect-oriented programming for PHP 5.
AspectS is an AOP framework for Smalltalk. There are two versions: one for Visual Works, and another for Squeak.
AspectXML is an attempt to sound out, brainstorm, and generally try out the aspect oriented approach in relation to XML.
Aquarium is a full-featured AOP toolkit for Ruby which seeks to provide "enterprise-class" capabilities like AspectJ. It seeks to create a rich, user-accessible pointcut language and to promote aspects as a tool for implementing Domain-Specific Languages (DSL's). Recent enhancements include preliminary support for advising Java classes when using JRuby!
C-mol is a method oriented programming language that is compatible to and can be used together with any C++ code. Using C-mol, programs can be developed and implemented the method oriented way. It supports concepts similar to the aspect oriented concepts, resulting in smaller, non-redundant and well-separated, comprehensive program code—but without breaking the encapsulation that is provided by object oriented programming. C-mol and the method oriented development concepts are very easy to learn since they are along the lines of object oriented concepts and do not introduce tons of new terminology and syntax.
Contract4J supports Design by Contract © for Java 5. Contract tests are embedded in Java 5 annotations. At runtime, AspectJ aspects evaluate the expressions to enforce the contract.
Eos is an aspect-oriented extension to C# for .NET Framework. Eos supports a very simple unified model of AOP.
Motorola WEAVER is an Aspect-Oriented Modeling add-in to Telelogic TAU. It performs weaving of UML 2.0 state machines before code generation.
Nanning is an Aspect Oriented Framework for Java based on dynamic proxies and aspects implemented as ordinary Java-classes.
Object Teams is a programming model for modular aspect-oriented programming with roles and collaborations (=teams). The Eclipse Object Teams Project hosts the development of the language OT/J and the Object Teams Development Tooling which is a feature rich and thoroughly tested IDE for OT/J based on Eclipse. The OTDT itself is built using OT/J and the OT/Equinox technology.
PAT (Persistent Applications Toolkit) is a persistence aspect for plain, Java objects.
SetPoint is an aspect-oriented tool based on semantic pointcuts, developed using Microsoft's .NET framework.
Sophus is a .NET platform high performance and easy to use AOP framework. It is simple and fast.
XWeaver is an extensible, customizable and minimally intrusive aspect weaver for C/C++ — it generates source code which differs as little as possible from the base code (preserves layout and comments).
Teddy's Aspect Weaver is an AOP framework resulting a static aspect weaver based on AspectDNG's ILML library and provides easier configuration format (both meta xml way and custom attribute way) and more advice types support.
MFAOPHP is a simple implementation of AOP in PHP, it supports JoinPoints, PointCuts and the before, after and around Aspects. No pre-PHP processor, but real-time Aspect Weaving. Download your own free version under GPL licence.
AOP Library for PHP This download-free (GPL licence) package can be used to implement Aspect Oriented Programming (AOP) by executing the code of classes that enable orthogonal aspects at run-time.
DotSpect (.SPECT) is a .NET, Compile-time (Static-injection), Language-Independent Aspect Weaver. It provides a language similar to AspectJ with some additional syntactic elements. Advice code can be written in any language. By default, there is support for C# and VB.NET. It also provides an IDE for writing and testing aspects.

4 Evaluation and selection of aspect weavers

4.1 Selection criteria

To be able to evaluate different AOP frameworks, many dimensions are to be considered. In this section we discuss the selection criteria that have been taken into account to choose the aspect weavers that were analysed in section 4.2. This means that those approaches listed in section 3 that do not satisfy these criteria have been directly discarded because they do not satisfy the requirements of the INTER-TRUST project regarding the dynamic weaving of aspects.

In order to choose the selection criteria we have used the information provided by deliverables D2.1.1 and D2.1.2 of WP2, and deliverable D4.2.1 of WP4. The main source to collect the Inter-trust case studies requirements is the deliverable D2.1.1 that provides an initial set of 32 functional and 20 non-functional requirements for the Inter-Trust framework. The requirements have been elicited following the UI-REF methodology [28] taking into account target domains as well as standards and gap analysis, market watch, etc. We have also taken into account the results of a questionnaire that was created by UMA in order to collect more specific requirements regarding the implantation constraints of different case studies, and that INDRA, SCYTL and UMU filled regarding the use-cases they were responsible for[2].

Section 4.1.1 summarizes the main inputs regarding Inter-trust case studies that are relevant for the AOP selection. Then section 4.1.2 specifies the selection criteria to be used to select the AOP framework and a list of requirements to be taken into account.

4.1.1 Inter-trust case studies requirement gathering

The following table summarizes the main inputs regarding Inter-trust case studies that are relevant for the AOP selection.

Table 4.1. Inter-trust case studies requirements

Case Study	Provider	Main Languages	OS	Binding Time	Specific details
V2X	INDRA/UMU	Java & C++ JavaScript & Lua	Linux and Android	Run-time is recommended Interoperability policy should be dynamically deployed within the system	The client side uses a web-app model relying heavily on JSP and HTML. On the server side a range of the most common models based on Java and Javascript are used such as POJO, BEANS, AJAX, JSON, etc.
E-voting	Scytl	Java	Windows	Run-time is recommended	Applets are to be weaved on the client side to add new signature to votes from specific terminals.

[2] This is a summary of the answers to the questionnaire that it is available in Appendix A of this document.

				Interoperability policy should be dynamically deployed within the system	Beans are to be weaved on the server-side to add validation of this extra signature.

Based on requirements specified in D2.1.1, the output of the policy negotiation process between two devices is an agreement about interoperability policies, which must be **dynamically** enforced on the two devices. This implies configuring applications and generating the security aspects that must be integrated into applications. For the two case studies, mechanisms will be defined to generate code or configuration files used by the APIs weaved into the applications to make them secure. The deployment of policies is not limited to the interoperability policies and also encompasses the deployment of reaction policies in response to malicious behaviour and attack detection. Besides, in the case of a changing environment, a security policy can be adapted and a re-negotiation phase can be initiated to agree about different SLA (Security-Level Agreements) to be used to define interoperability policies. These security policies need to be dynamically deployed (based on the AOP framework) during runtime.

4.1.2 Aspect weaver requirements

Within the INTER-TRUST project, we choose to rely on five criteria usually applied in this kind of experiments: the expressiveness, the degree of dynamicity, the versatility, the constraints related to the execution environment (case studies), and finally the strategy. For our comparison, we will focus on those five topics, and a list of more specific aspect weaver requirements related to these topics, to provide a solid base for the decision over the framework selection.

1. The **expressiveness** is mainly considered for the pointcut definition language and the advice predicates. To consider here are, e.g., the ability to supply regular expressions (RE) for matching code entities (method names, signatures, classes or variables accessed), as well as the choice of advices supported (before/after, around/proceed, call hierarchy and exception handling in advices).

2. The **degree of dynamicity** comprises several concepts, too: Being able to enable/disable advices at runtime, to create and modify pointcut definitions at runtime, but also to access and modify the runtime context of a join point during execution.

3. The **versatility** contains aspects of the aforementioned two: The range of programmatical situations in which AOP is supported by the framework. This includes the number of entities that can be detected by a join point, e.g., interception of object creation and destruction, distinction between a method call and a method execution, or the ability to apply aspects on classes or objects and similar issues.

4. The **execution environment** constraints are hard requirements to consider, e.g., the allowed operating systems, platforms, virtual machines to run code in and also how aspects are applied to code in a build process. This information is useful when making a decision upon instating a certain framework for aspect-oriented programming.

5. The **strategy** determines the technical realization of implementing AOP: (i) the partners' skill and expertise regarding a specific framework, (ii) the case study providers' choices to integrate the technology in their solution, and (iii) other similar issues.

Below, we describe 13 requirements that we have identified as important for selecting the AOP framework and aspect weaver. For each requirement, we indicate the selection criteria to which they are related.

AW-Req1 (criteria 1). The pointcut language must be as expressive as possible. The possibility of using regular expressions to identify the set of join points that can be intercepted by an aspect facilitates the use of an aspect-oriented approach. Thus, this is an important requirement that must be considered on selecting our AOP framework and aspect weaver.

AW-Req2 (criteria 2). Different degrees of dynamicity are required. Table 4.2 summarizes the different degrees of dynamicity that can be suitable for the INTER-TRUST framework. We analyse those aspect weavers that provide one or more of them. On the one hand, approaches with the lower degree of dynamicity offer a compile-time that includes a mechanism to enable/disable *at runtime* which aspects are injected into the base code. On the other hand, the higher degree of dynamicity is offered by approaches that provide a run-time weaver, where not only the weaving itself but the decision of which aspects (and which implementation of these aspects) need to be weaved with the base code.

Table 4.2. Degrees of dynamicity

Type of weaver	**Evaluation**	**Discussion – Advantages/Shortcomings**
Compile-time + enabling/disabling of aspects	**Degree of dynamicity.** Low/Medium **Robustness.** High **Execution Performance.** High	Aspects weaved at compile-time can be enabled/disabled at runtime.
		Advantages. This is the most secure option since the robustness and suitability of the aspects to be weaved can be checked before deploying the application.
		Shortcomings. The lists of aspects that can be weaved with the base code, and their implementations, need to be defined at compile time, and cannot be modified at runtime. For instance, it would be possible to negotiate at runtime if a security aspect (e.g. access control) is enabled or not and which specific implementation of an aspect need to be enabled, but no new security aspects or implementations can be weaved with an application if they were not contemplated before compiling the application.
Load-time + enabling/disabling of aspects	**Degree of dynamicity.** Medium	Aspects weaved at load-time can be enabled/disabled at runtime.
		Advantages. This is also a secure option because the list of aspects to be weaved need to be known a-priori. However, it is less robust regarding security because the

	Robustness. Medium **Execution Performance.** Medium	implementations of the aspects can be updated during the execution of the application and before the aspect is first used.
		Shortcomings. The same shortcoming than for compile-time weavers with enabling/disabling of aspects at runtime. All the aspects that can be weaved at load-time need to be defined to the load-time weaver before the application code is loaded.
Run-time	**Degree of dynamicity.** High **Robustness.** Low **Execution Performance.** Low	The weaving of aspects is postponed until the runtime.
		Advantages. This is the only kind of weaver that can be considered completely dynamic, in the sense that security aspects can be incorporated to an application at any moment of the execution, including completely new aspects that were not considered during the design of the application, or even during the initial design of the INTER-TRUST framework.
		Shortcomings. A security aspect that was not even considered during the development of an application can be incorporated at runtime, and this can suppose a security risk that the INTER-TRUST framework should consider. Moreover, the execution performance in approaches providing "real" run-time weaving is usually less efficient than in the other kinds of weavers.

AW-Req3 (criteria 3). Different kinds of join points may need to be intercepted (method constructors, initialization, call/reception of messages, read/write operations on attributes, etc…). As occur with AW-Req5, the information that is currently available in the initial version of the requirements of the INTER-TRUST framework and use-cases is not conclusive regarding the kinds of join points that need to be intercepted, and this prevents us from selecting an aspect weaver that would not be able to intercept the most important join points usually considered in AOP. We have then analysed only the aspect weavers that allow the interception of an important number of join points. Exceptions to this requirement have been made for those aspect weavers that, in spite of providing a limited number of join points, can be easily used in conjunction with other aspect weavers that complement them (for instance, offering the rest of join points but with different degree of dynamicity).

AW-Req4 (criteria 3). Before, after and around advice types may be needed. The information available in the initial version of the requirements for both the INTER-TRUST framework and the use-cases is not conclusive regarding the type of advice that are going to be needed in the project. This prevents us from selecting an aspect weaver that does not include all the advice types that are usually considered in AOP. For this reason, we have been conservative and have only analysed the aspect weavers that allow the injection of aspects before, after and around a join point.

AW-Req5 (criteria 3). Multiple aspects would eventually need to be weaved in the same join point. Nothing in the INTER-TRUST framework or use-cases requirements prevent us for having to weave more than one aspect sequentially in the same join point. Thus, we need to consider that this possibility, which is the most usual one in AOP, will exist and thus we will select an aspect weaver that has this characteristic.

AW-Req6 (criteria 4 + criteria 5). The aspect weaver and its tool support must be functional, up to date and maintained. In the last years, a high number of aspect weavers have been developed, mostly academic ones. Unfortunately, many of them are: (1) prototypes experimenting with some specific aspect-oriented characteristics, (2) without an appropriate tool support, and/or (3) not maintained anymore. Thus, they cannot be used with guarantees in an industrial setting, and this is the reason we have discarded them.

AW-Req7 (criteria 4 + criteria 5). Aspects need to be weaved both at the client and at the server side. After analysing the INTER-TRUST framework requirements in D2.1.1 and the description of the use-cases in D2.2.1, we can conclude that the defined security policies, and therefore the aspects, have to be weaved both at the client and at the server side. These security policies will include different aspects defined by requirements in D2.1.1. Some examples of these requirements, which support this decision, can be seen in Table 4.3[3]. We have used the Requirement No. provided to the requirements in D2.1.1 in order to reference them.

Table 4.3. Justification of AW-Req7 based on the INTER-TRUST framework and the use-cases requirements

Use-Case	**Aspect/Requirement**	
V2x	**Client Side**	- Access Control (V2V001) - Encryption (V2V006, V2I012) - Privacy (V2V002) - Client-side logging (V2I001) - Input validation (V2I009)
	Server Side	- Access Control (V2V001) - Privacy (V2V002) - Server-side logging (V2I002) - Concurrency checking (V2I010)
E-voting	**Client Side**	- Delegation (EV003) - Encryption (EV007)

[3] This table does not include all the use cases aspects, only a few of them just to justify that they have to be weaved both at the side of the client and at the side of the server.

	Server Side	- Encryption in the server side by delegation of the client (EV007)

AW-Req8 (criteria 4 + criteria 5). Applications on the client side must run on different operating systems. According to the use case requirements (e.g. requirement V2I005), the applications using the INTER-TRUST framework need to run on different operating system, including mobile operating systems. There are two possibilities that need to be analysed. One of them is to consider that these applications will be client/server applications where the client application will be accessible using a web browser of the client device. Another one considers applications that are implemented as native applications (e.g. as an Android application). Our analysis considers both possibilities.

AW-Req9 (criteria 4 + criteria 5). Aspects need to be weaved mainly with Java code. The e-voting use-case is implemented in Java and V2x combines both Java and C/C++ code. We have analysed all existing java-based aspect weavers.

AW-Req10 (criteria 4 + criteria 5). Aspects may need to be weaved with C/C++ code. In the V2x use-case, the OBU (vehicle On-board Unit or mobile router) and the RSU (Road-Side Unit or fixed router) are implemented in C. We have then analysed existing aspect weavers for C/C++ to take into account the case in which aspects need to be weaved in these devices.

AW-Req11 (criteria 4 + criteria 5). Aspects need to be woven with Applets. Both use-cases use applets at the client side. Since according to AW-Req2 aspects need to be weaved at both sides, we have analysed whether the aspect weaver works well or not with applets.

AW-Req12 (criteria 4 + criteria 5). Aspects may need to be woven with Android native code. According to AW-Req4, the client-side of the use-cases may run on devices with different operating systems, including Android. Thus, we need to analyse whether the aspect weavers work for that operating systems.

AW-Req13 (criteria 4 + criteria 5). The possibility of integrating Java and .NET worlds need to be explored. In order to be able to use the INTER-TRUST framework not only in the Java world, but also in the .NET world, we will analyse the possibility of using an aspect weaver with both java and .NET applications.

4.2 Existing proposals that meet the selection criteria

In this section we analyse with more details those proposals of section 3 that meet the selection criteria. In sections 4.2.1 to 4.2.7, each proposal has been analysed according to the general criteria selection and to the list of aspect weaver requirements that were described in previous section[4].

[4] There is not any AOP framework that satisfies AW-Req13 because each framework allows the use of a single programming language. There is not an AOP framework able to inject the same aspects into applications implemented in different programming languages. A transformation process from Java to J# or C# would be needed as discussed in Appendix C.

Then, in section 4.2.8 the different tests that we have performed to check that these proposals can be used in the INTER-TRUST project are briefly discussed[5].

4.2.1 AspectJ

Strategy. AspectJ is an aspect oriented extension for the Java programming language. His major versions match up with Java versions (AW-Req9). For instance, currently[6] Java is at 1.7 and AspectJ is about to release the 1.7 version after 13 releases at 1.6 (1.6.0 -> 1.6.12). The AspectJ project then releases service refreshes to that major version every few months. The current release model is at least something every three months - sometimes a milestone, sometimes a full release (AW-Req6).

AspectJ has become the widely used de-facto standard for AOP by emphasizing simplicity and usability for end users. AspectJ is available in Eclipse Foundation open-source projects, both stand-alone and integrated into Eclipse through the AspectJ Development Tools (AJDT). IDE support for emacs, NetBeans, JBuilder, and Oracle's JDeveloper is also supported.

Environment constraints. AspectJ is suitable for using AOP in all systems and platforms that have Java support, because AspectJ works with any Java application. AspectJ provides a compiler and bytecode weaver (ajc) for the AspectJ and Java languages.

Degree of dynamicity. The degree of dynamicity of AspectJ is medium basically because AspectJ does not support weaving at runtime. Based on when it performs weaving, AspectJ offers two weaving models (AW-Req2):

- Build-time or compile-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, obviating any pre-deployment weaving.

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.

When using AspectJ with other platforms or execution environments, the weaving process is usually limited to compile-time weaving. For example, when using AspectJ with Android (AW-Req12). However, if the execution environment allows using a weaving agent or changing the classloader, you still may perform LTW. In the first case, you can specify the following option to the JVM to enable LTW:

```
-javaagent:pathto/aspectjweaver.jar
```

To enable LTW in environments where no weaving agent is available, it is necessary to change the default classloader by instantiating a weaver and weaving classes after loading and before defining them in the JVM. For instance, to perform LTW in Java Applets (AW-Req11) you need to change the

[5] The results of each of the performed tests are details in Appendix C.

[6] March 2013.

classloader used by the JVM in the Java Web Start with the AspectJ weaver using the `WeavingURLClassLoader` and the `WeavingAdapter` (see Appendix C for more details).

However, in all cases, AspectJ allows increasing the degree of dynamicity by coding patterns that can support dynamically enabling and disabling advice in aspects. This is done through the `if(...)` pointcut that AspectJ provides to define a conditional pointcut expression which will be evaluated at runtime for each candidate join point. A simple way to control an aspect is by using an `if()` check evaluating a Boolean field and exposing that field through JMX (Java Management Extensions) technology. If you need to control the applicability of the aspect at application startup, you can assign the boolean field a system property (or a property read from a property file). You can also combine these two techniques to control the default applicability of the aspect as well as allow runtime control.

Expressiveness and versatility. AspectJ allows intercepting many kinds of join points, being the proposal with the most complete join point model. In Appendix B.1, Table B.1.1 takes a quick look at the categories of the join points exposed by AspectJ and their semantics (AW-Req 3).

The AspectJ's pointcut language includes:

- **Named pointcuts** that are elements that can be referenced from multiple places, making them reusable.
- **Anonymous pointcuts** that are defined at the place of their usage, such as a part of advice, a part of static crosscutting constructs, or when another pointcut is defined.
- **Wildcards** in the signature syntax to select program elements that share common characteristics.
- A **unary negation operator** (`!`) and two **binary operators** (`||` and `&&`) to form complex matching rules by combining simple pointcuts.
- Several **pointcut designators**, which when combined with the signatures, form pointcuts.

Pointcuts match join points in two ways: (1) **kinded pointcuts** — i.e., pointcuts that directly map to join point categories or kinds, to which they belong (Table B.1.2 in Appendix B.1 shows the syntax for each of the kinded pointcuts); and (2) **non-kinded pointcuts** — i.e., pointcuts that select join points based on information at the join point, such as runtime types of the join point context, control flow, and lexical scope. These pointcuts select join points of any kind as long as they match the prescribed condition. Some of the pointcuts of this type also allow the collection of context at the selected join points. All these characteristics of the AspectJ's pointcut language make AspectJ the most expressively AOP language (AW-Req1).

AspectJ supports dynamic crosscutting through advice offering three kinds of advice (AW-Req4):

- **Before advice** executes prior to the join point's execution.
- **After advice** executes following the join point's execution. After advice has three variations based on the outcome of the join point execution:
  - **After (finally)** executes after execution of the join point, regardless of the outcome.

  - **After returning** executes after successful execution of the join point—that is, without throwing an exception.
  - **After throwing** executes after failed execution of the join point—that is, throwing an exception.

- **Around advice** surrounds the join point's execution. This advice is special in that it has the ability to continue the original execution with the same or altered context, zero or more times.

Join points exposed by AspectJ are the only points where you can apply advice. AspectJ allows the definition of multiple aspects advising the same join point, as well as ways to control the ordering of application of several aspects in the same join point (AW-Req5).

Another interesting characteristic of AspectJ is the possibility of converting a Java class into an aspect using annotations. AspectJ 5, in addition to the familiar AspectJ code-based style of aspect declaration, also supports an annotation-based style of aspect declaration (called @AspectJ style).

4.2.2 Spring AOP

Strategy. Spring is the most popular lightweight framework for enterprise applications. To satisfy the needs of enterprise applications, it includes an AOP system based on interceptors and the proxy design pattern. Spring is up to date, maintained and well documented. Spring Framework 3.2.1.RELEASE is the current production release and requires Java 1.5+ (AW-Req6).

All of the advice you create within Spring is written in a standard Java class. That way, you get the benefit of developing your aspects in the same integrated development environment (IDE) you'd use for your normal Java development (AW-Req9). What's more, the pointcuts that define where advice should be applied are typically written in XML in your Spring configuration file. This means both the aspect's code and configuration syntax will be familiar to Java developers.

Environment constraints. Spring is a logical choice in many scenarios, from applets to full-fledged enterprise applications that use Spring's transaction management functionality and web framework integration (AW-Req7). A typical full-fledged Spring web application is shown in Figure B.2.1 in Appendix B. The Spring Framework serves as the foundation for the wider family of Spring open source projects, including: Spring Security, Spring Integration, Spring Web Services, Spring Mobile, Spring Android, among others. In particular, Spring Mobile is an extension to Spring MVC that aims to simplify the development of mobile web applications including intelligent device detection and progressive rendering options (AW-Req8). Spring for Android is also an extension of the Spring Framework that aims to simplify the development of native Android applications (AW-Req13).

Degree of dynamicity. In Spring AOP, aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class (AW-Req2). As illustrated in Figure 4.1, the proxy class poses as the target bean, intercepting advised method calls and forwarding those calls to the target bean. Between the time when the proxy intercepts the method call and the time when it invokes the target bean's method, the proxy performs the aspect logic. Spring doesn't create a proxied object until that proxied bean is needed by the application. If you're using an ApplicationContext, the proxied objects will be created when it loads all of the beans from the BeanFactory. Because Spring creates proxies at runtime, you don't need a special compiler to weave aspects in Spring's AOP.

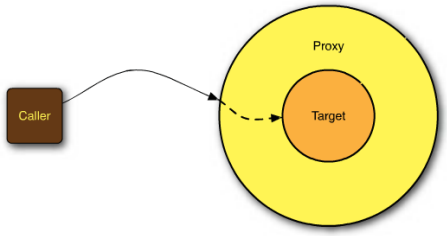

Figure 4.1. Spring aspects are implemented as proxies that wrap the target object. The proxy handles method calls, performs additional aspect logic, and then invokes the target method ([4]).

Expressiveness and versatility. Spring's support for AOP comes in four flavors: (1) Classic Spring proxy-based AOP, (2) @AspectJ annotation-driven aspects, (3) pure-POJO aspects, and (4) injected AspectJ aspects. The first three items are all variations on Spring's proxy-based AOP. Consequently, because it's based on dynamic proxies, Spring's AOP support is limited to method interception (AW-Req3). If the requirements need exceed simple method interception (e.g., constructor or property interception), there must consider implementing aspects in AspectJ, perhaps taking advantage of Spring DI to inject Spring beans into AspectJ aspects. Spring's lack of field pointcuts prevents you from creating very fine-grained advice, such as intercepting updates to an object's field. And without constructor pointcuts, there's no way to apply advice when a bean is instantiated.

In Spring AOP, pointcuts are defined using AspectJ's pointcut expression language. Table B.2.1 in Appendix B.2 lists the AspectJ pointcut designators that are supported in Spring AOP (AW-Req1 and AW-Req5). Attempting to use any of AspectJ's other designators will result in an IllegalArgument-Exception being thrown. In addition to the designators listed in Table B.2.1, Spring 2.5 introduced a new `bean()` designator that lets you identify beans by their ID within a pointcut expression. `bean()` takes a bean ID or name as an argument and limits the pointcut's effect to that specific bean.

Spring aspects can work with five kinds of advice (AW-Req4):

- **Before**: The advice functionality takes place before the advised method is invoked.
- **After**: The advice functionality takes place after the advised method completes, regardless of the outcome.
- **After-returning:** The advice functionality takes place after the advised method successfully completes.
- **After-throwing**: The advice functionality takes place after the advised method throws an exception.
- **Around**: The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

Additionally, Spring AOP can increase his expressiveness by combining with AspectJ. Spring AOP, although not as powerful as AspectJ, offers a pragmatic solution to modularize commonly encountered crosscutting concerns in a typical enterprise application. If you need more power, you

can always use AspectJ byte-code weaving, because AspectJ works with any Java application. Spring provides a few options to simplify AspectJ usage, as well. Spring simplifies the programming model through AspectJ integration that still works within the proxy-based framework (and hence doesn't require explicit byte-code weaving). Specifically, under this model, you can write aspects using:

- A subset of the @AspectJ syntax.
- Plain Java (without annotations), with AOP constructs expressed in XML (the schema-style AOP). This is an especially useful alternative if you can't use Java 5 or above and, hence, annotations.

Spring also simplifies the use of full-power AspectJ that relies on byte-code modification techniques. Specifically, it supports:

- Configuring aspect instances as Spring beans.
- Simplifying load-time weaving (LTW) without requiring modifications to the launch script (to add the weaving agent) for a few web and application servers.

4.2.3 JBossAOP

Strategy. JBoss is an open source application server that offers an AOP solution. JBoss AOP is a 100% Pure Java 'aspected' oriented framework (AW-Req8). Actually, the project is no longer under active development, being JBoss AOP 2.1.8 his latest release at 2010 (AW-Req6). However, the existing version is stable and fully functional (AW-Req6).

Environment constraints. JBoss AOP is usable in any programming environment or tightly integrated with the JBoss Application Server (AS) (AW-Req8).

Degree of dynamicity. JBoss AOP supports dynamic AOP by rebuilding interceptor chains associated with a joinpoint when aspects are added and/or removed (AW-Req2). There are two ways of using dynamic AOP with JBoss AOP: (1) hot deployment (i.e., the possibility of deploying aspects at runtime when running the application in the AS), and (2) through the API. If hot deployment is enabled, JBoss AOP also instruments the code at runtime to insert/remove invocations to these chains whenever appropriate. Otherwise, the calls to chains are inserted during compilation or class loading (depending on whether you are using compile time or load time weaving).

In order to perform dynamic AOP operations, JBoss AOP must know beforehand which joinpoints need to be instrumented, so that the invocations to interceptor chains are added to the code before it gets executed. For weaving, JBoss AOP inserts auxiliary fields and methods into your class bytecodes. This step must be performed before your code gets executed and, hence, the joinpoints to be instrumented must be known at the first weaving stage (compile time or loadtime weaving).

Expressiveness and versatility. JBoss AOP includes a pointcut language similar to that of AspectJ (AW-Req1). The pointcuts expressions include two types of wildcards (* and ..) and logical operators for pointcut composition (!, AND; OR). The joinpoint model of JBoss AOP is shown in Figure B.3.1 in Appendix B.3 (AW-Req3).

JBoss AOP only supports *around* advices. However, using the invocation `invocation.invokeNext()` statement (identical to the `proceed`() statement of AspectJ) is

possible to implement *before* and *after* advices (AW-Req4). All aspects, advices and interceptors are represented by factories. Internally, JBoss AOP does not differentiate advices from interceptors. They are both represented as interceptor instances, which results in uniform, transparent handling of interceptors and advices. This way, an interceptor chain is used for intercepting a joinpoint every time it is executed, and may contain even typed advices.

4.2.4 AspectC

Strategy. AspeCt-oriented C (AspectC for short) consists of a compiler that translates code written in AspeCt-oriented C into ANSI-C code. This code can be compiled by any ANSI-C compliant compiler, like for example `gcc`. The latest version of AspectC is V0.9 RC, released on 2010 (AW-Req6).

Environment constraints. AspectC supports Linux, MacOS X, Solaris and Windows (under Cygwin) platforms (AW-Req8).

Degree of dynamicity. The `acc` compiler of AspectC only supports compile-time weaving (AW-Req2).

Expressiveness and versatility. The AspectC language design follows the ideas of AspectJ programming language adapting it to the C language, resulting in an expressive language with a rich set of keywords to specify and defines pointcuts and advices (AW-Req1 and AW-Req4). Table B.4.1 in Appendix B.4 resumes some of these keywords. AspectC also provides the wildcard character matching and operators for pointcut composition (`&&`, `||`, `!`). An illustration of the joint point model of AspectC is presented in Figure B.4.1 in Appendix B.4 (AW-Req3).

4.2.5 AspectC++

Strategy. AspectC++ is a project that extends the AspectJ approach to C/C++ with a set of C++ language extensions to facilitate AOP with C/C++ and including an AspectC++ compiler (`ac++`). AspectC++ 1.1 is the latest version that was released on 2012. The project is up to dated, maintained and well documented in his project site (AW-Req6).

Environment constraints. AspectC++ binary version is available for Linux and Windows (AW-Req8).

Degree of dynamicity. The degree of dynamicity in AspectC++ is low due to actually only support compile-time weaving (AW-Req2).

Expressiveness and versatility. AspectC++ project extends the AspectJ approach to C/C++. So, AspectC++ is as expressive as AspectJ (AW-Req1). It includes many operators and wildcard characters for matching patterns (%, %%, +=, %*…), and also provides the before, after and around advices (AW-Req4). The API of the join point model of AspectC++ is shown in Figure B.5.1 in Appendix B.5 (AW-Req3).

4.2.6 FeatureC++

Strategy. FeatureC++ is a C++ language extension to support FOP and AOP. FeatureC++ comes in form of a C++ preprocesser (`fc++`) that transforms FeatureC++ code into native C++ code. The latest version of FeatureC++ is v0.7 and released at 2010.

Environment constraints. FeatureC++ is available for Linux, Mac OS X, and Windows (AW-Req8).

Degree of dynamicity. FeatureC++ compiler only supports compile-time weaving (AW-Req2).

Expressiveness and versatility. FeatureC++ is a bit different from the previous (AspectC++). Programming in FeatureC++ is not much different than programming in C++. Actually, programming in FeatureC++ is even simpler due to modifications of the C++ language. FeatureC++ provides only two new keywords: (1) `refines`, that is used to specify extension of an existing class, and (2) `super`, that is used to call an overridden method. These keywords and the classes' heritance of OOP are sufficient to provide a minimum implementation of AOP. Due to the fact that FeatureC++ is a language proposal for FOP in C++ instead of a pure AOP language, it does not define a join point model or a pointcut specification as other AO proposals (AW-Req1, AW-Req3, and AW-Req5). FeatureC++ uses folders to represent features and a folder hierarchy can be used to represent a hierarchy of features. Classes are implemented as in plain C++ but are naturally distributed over multiple features. In this way, aspects are classes that "refine" a super class to override the behavior of the methods; through the `super` keyword it is possible to invoke the original code in any moment. This provides a way to implement before, after and around advices (AW-Req4).

4.2.7 Spring.NET AOP

Spring.NET is the .NET counterpart of the Spring Framework. It includes AOP support that is similar to Spring AOP. So, the characteristics of both Spring AOP (for Java) and Spring.NET AOP are identical.

Integrating Java and .NET worlds. AOP has generated quite a bit of interest in the .NET world. Due to the use of byte code representations and the possibility of using proxies, .NET offers choices similar to those available in the Java world. In addition to Spring.NET, prominent AOP solutions in .NET include PostSharp, and Aspect#. LOOM.NET is a research project that's exploring static and dynamic weaving in .NET.

IKVM.NET is an implementation of Java for Mono and the Microsoft .NET Framework. It includes the following components: (1) a Java Virtual Machine implemented in .NET, (2) a .NET implementation of the Java class libraries, and (3) tools that enable Java and .NET interoperability. IKVM.NET includes ikvmc, a Java bytecode to .NET IL translator. This ikvmc tool allows using a Java library in a .NET application by converting Java bytecode to .NET dll's and exe's. IKVM provides a way for you to develop .NET applications in Java. Although IKVM.NET does not include a Java compiler for .NET, you can use any Java compiler to compile Java source code to JVM bytecode, and then use ikvmc to produce a .NET executable. You can even use .NET API's in your Java code using the included ikvmstub application. The ikvmstub tool generates Java stubs from .NET assemblies.

4.2.8 Tests

AspectJ

- AspectJ in a standalone application

  AspectJ is usable in any environment that supports Java language. Compile-time weaving works directly due to the fact that compiled programs with the AspectJ compiler (ajc) are simple Java programs. However, LTW depends on the classloader used by the JVM of the execution environments. LTW always requires specifying the AspectJ weaver as a java agent to the JVM.

- AspectJ + Servlets/Applets/JSPs

AspectJ works with Servlets, Applets and JSPs applications. On the one hand, in Servlets and Applets, both compile-time and load-time weaving can be performed. Applets must be adapted in order to use LTW by defining another Applet that uses the AspectJ weaving classloader and by using the Java Web Start with a `.jnlp` configuration file. See Appendix C to know how to proceed in that case. On the other hand, weaving aspects in JSP pages is limited to load-time weaving because of JSP pages are compiled at runtime in the web server. In all cases, load-time weaving requires using the java agent `aspectjweaver.jar`

- AspectJ Enable/Disable aspects capability at runtime

  The enable/disable aspects capability at runtime works thanks to the `if()` pointcut of AspectJ. Its usage is independent of the weaving time, so it can be used in compile-time and load-time weaving.

- AspectJ + Android

  When AspectJ is used in Android, we are limited to compile-time weaving, which basically means that you can only intercept code you own. For instance, you can write a pointcut that picks up the executions of the method `onCreate()` or `onDestroy()` of an activity because when using the `execution()` pointcut the code gets weaved "inside" your Activity method `onCreate()` or `onDestroy()`. But, you cannot pick up the calls to these methods because the advice would have to get weaved into the methods that call your activity method `onCreate()` or `onDestroy()`.

Spring AOP

- Spring AOP in a standalone application

  Spring AOP is usable in any Java application: standalone as well as application server based. However, Spring AOP implies the usage of the Spring Framework technology (dependency injection, IoC) and the definition of the classes and the aspects as beans in the Spring beans container.

- Spring AOP + Servlets/Applets/JSPs

  It is also possible using Spring AOP with Servlet, Applets and JSPs. However, dynamic weaving requires the usage of the beans container of Spring. That is, classes that will be woven must be taken from the application context, which is provided by the Spring container. The application context provides instances of the declared beans in the bean configuration file of Spring. See Appendix C.2.1 to know in detail how to use dynamic weaving with Spring AOP. Moreover, Spring framework provides his own way to implement Servlets through the dependency injection and inversion of control technologies, and it is high recommended to use these features if Spring is used.

- Dynamic changing of aspects in Spring AOP

  Dynamic changing of aspect in Spring AOP is supported within the Spring framework due to that the proxy-based dynamic weaving is supported only when using the beans container of Spring. The proxy-based dynamic weaving allows defining more than one application context in which the beans configuration files are specified. These files can be added and removed at

runtime in a programmatic way, so aspects and pointcuts can also be added and removed at runtime. However, these changes cannot be performed over the same instance of the bean. Instead, a new instance of the bean must be taken from the application context after the changes have been done.

- Spring AOP + AspectJ

  Spring simplifies the programming model through AspectJ integration. On the one hand, it is possible using the AspectJ language to define aspects, but dynamic weaving still works within the proxy-based framework. Moreover, some limitations exist, for example it is not possible to have aspects themselves be the target of advice from other aspects. The @Aspect annotation on a class marks it as an aspect, and hence excludes it from auto-proxying. On the other hand, LTW can also be performed. In Appendix C.2.2 and C.2.3 is explained how to use Spring AOP and AspectJ in these two ways.

JBOSS AOP

- JBoss AOP in a standalone application

  JBoss AOP works with any Java program by using the API of JBoss AOP. The library that includes the API is `jboss-aop-jdk50.jar`. Apart from the API, JBoss AOP works with the JBoss Application Server in a naturally way. A requirement to perform dynamic weaving in both cases is that classes that will be woven must be prepared before running the application. Appendix C.3 shows how to prepare the classes for dynamic weaving.

- JBoss AOP in Serlvets/Applets/JSP

  JBoss AOP works with Servlets, Applets and JSP in JBoss AS and using the API. However, the limitation of preparing the classes exists in all cases. Compiling is not a trivial task when the JBoss AOP API is used and can be necessary the use of a build file such as an ant file. Appendix C.3 shows information about how to perform the entire process for using the API.

- Dynamic changing of aspects in JBossAOP

  On the one hand, hot deployment is available for JBoss AS. In this case, all deployment units, such as ears, wars, jars, and JBoss AOP deployments unit (that have the .aop suffix) can be hot deployed. JBoss AS will automatically load the applications added to the deploy folder, and unload them when their units are removed from the deploy folder. In addition, if you are running the server with load time weaving, all the applications deployed after this moment will also be instrumented by JBoss AOP to insert the aspects contained in the deployment unit, according to the bindings and pointcuts it contains. Similarly, when a `.aop` file is removed from the `deploy` dir, the bindings it contains will be automatically removed from the running applications. As an alternative to using `.aop` units, it is possible using other deployment units and xml files to hot deploy aspects to the server.

  On the other hand, when using the API of JBoss AOP is also possible the addition and removal of aspects and pointcuts at runtime, even over the same instance of the target class. This is done through the `addBinding()` and `removeBinding()` methods of the API.

AspectC

- Aspects into C programs

AspectC provides full ANSI-C compliance and gcc source-compatibility, so aspects can be applied to any C application with the restriction of recompiling the application from scratch due to the weaving is performed at compile-time.

AspectC++

- Aspects into C++ programs

As in AspectC, AspectC++ needs to recompile the application from scratch. Aspects are woven before compiling sources. C++ programs can be woven with the ac++ tool first and then compiled with the g++ compiler, or can also be woven and compiled with the ag++ tool. There are some limitations to use ac++: (1) woven code does not compile on a different platform, and (2) ac++ does not support weaving in templates yet. These potential join-points are silently ignored by the match mechanism.

FeatureC++

- Aspects into C++ programs

As in the previous proposals, weaving is at compile-time and application must be recompiled to apply aspects. However, due to FeatureC++ is an FOP language and not a pure AOP language, aspects do not exist as a real concept, and there is not any explicit join point model or pointcuts specification to define the points of the code that we want to intercept. All refines classes are applied when compiling the code.

Integrating Java and .NET

Java applications with the aspects woven at compile-time using AspectJ can also be converted from .jar to .exe in order to execute them in the .NET framework. To do that, it is necessary to convert the `aspectjrt.jar` library within the application. For example:

```
ikvmc myApp.jar aspectjrt.jar
```

4.3 Selection of the aspect weaver

In this section we discuss the selection of the AOP frameworks that are suitable for the project. In order to sustain this discussion a summary of the analysis presented in previous section can be found in Table 4.4.

Basically, the main results of the analysis are the following:

1. **The existing runtime weavers have limitations.** Basically, for three reasons:
   - They offer a reduced expressivity and versatility compared with compile-time and load-time weavers because the pointcut language is normally less expressive in dynamic approaches. That means that the number of join points that can be intercepted is usually reduced and in many cases limited to only method call interceptions.
   - They usually have lower performance. An exhaustive study of the performance of the different AOP weavers have not been performed, but it has been previously stated in the AOP community that dynamic weaving has always less performance than static ones.
   - They impose important restrictions on the execution environment because they require the use of their own application server, or additional code to implement the dynamic weaving outside the application server. Two examples are: (1) the dynamic weaving in Spring AOP that works only on top of Spring IoC mechanism; aspects cannot be applied directly to standalone applications, and (2) in order to use dynamic weaving in JBoss AOP, without the JBoss AS, we need to implement additional code that uses a JBoss AOP API to load/unload the aspects and to perform the dynamic changes.
2. **There are different degrees of dynamicity to be considered**. At this state of the project it is still soon to know the degree of dynamicity that will be needed in different parts of the INTER-TRUST framework and also in the different use cases. Anyway, the use of a load-time weaver + the enable/disable of aspects at runtime it is an option that must be considered. This mechanism is offered by AspectJ and the advantages here are that: (1) AspectJ is the most mature AOP framework that exists nowadays; (2) using this mechanism pointcuts/advice can be added at runtime by "enabling" them and can be deleted at runtime by "disabling" them. The shortcomings would be that the list of advices that can be woven at any point of the execution, and the list of pointcuts where the aspects can be woven, must be known during the deployment of the application.
3. **It is not feasible by now to select just a "one-for-all" framework.** Due to technical impediments (e.g. the analysis indicate that the same AOP weaver cannot be used to inject aspects for applications implemented in more than one programming language, such as Java and C/C++), and to the lack of information (the required degree of dynamicity is still not clear, the join points to be captured are still not clear and more details about the negotiation of security policies and the software architecture are needed). As discussed in the first part of this deliverable this is not a limitation of the use of AOP in the project, since the use of a particular AOP weaver is only an implementation detail of one possible instantiation of the INTER-TRUST framework. The software architecture of the framework will be designed

independently of the selected AOP weaver and thus more than one weaver is possible if the INTER-TRUST framework is instantiated in different programming/execution environments.

Which are then the possibilities?

As shown in Table 4.4 the most expressive, versatile and mature approach is AspectJ. This means that if the programming language is Java and the degree of required dynamicity is medium (e.g. load-time weaving + enabling/disabling of aspects) our first option would be always to use AspectJ. Moreover, AspectJ can be used in combination with Spring AOP to provide dynamic weaving at runtime (with the already discussed limitation of having to use the Spring framework and its IoC mechanism).

A second option when the programming is Java, a high degree of dynamicity is required and the use of the Spring framework is not an option, is the use of JBoss AOP. The only shortcoming of using this approach is that this project is not currently on evolution and thus the last version of 2008 should be used.

Finally, if the programming language is C we will need to use AspectC and if the programming language is C++ our selection would be to use AspectC++, because is the most mature and complete AOP language for C++. However, FeatureC++ will be also considered due to it is based on a different approach: Feature Oriented Programming (FOP). For the .NET world Spring .NET AOP could be used, with the same limitations of Spring AOP for Java.

Table 4.4. Summary of fulfilment of the INTER-TRUST requirements by the proposals.

(✓ = Yes, ! = partly, ✗ = No, - means not applicable).

	AW-Req												
Proposal	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
AspectJ	✓	!	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	!
Spring AOP	!	✓	!	!	✓	✓	✓	✓	✓	✗	!	!	-
Spring AOP + AspectJ	✓	!	✓	✓	✓	✓	✓	✓	✓	✗	!	!	-
JBossAOP	✓	✓	✓	!	✓	✗	✓	✓	✓	✗	✓	-	-
AspectC	✓	✗	✓	✓	✓	✓	✓	!	✗	✓	✗	-	-
AspectC++	✓	✗	✓	✓	✓	✓	✓	!	✗	✓	✗	-	-
FeatureC++	✗	✗	✗	!	✗	✓	✓	!	✗	✓	✗	-	-
Spring.NET AOP	✓	✓	!	!	✓	✓	✓	✓	✗	✗	✗	-	-

4.4 Custom-made aspect weaver

For those situations in which any of the analysed frameworks would suite the necessities of a part of a project, the solution would be to implement our own custom-made aspect weaver. Although this is a viable possibility, as the extensive list of research projects developing their own aspect weavers demonstrates (see section 3 of this document to consult this list), this is a very costly and error-prone solution that should be avoided, if possible, in the context of this project where the main goal is 'using' an existing AOP framework that suits the AOP necessities of the project and not the 'construction' of new AOP frameworks.

5 Conclusions

In this deliverable an exhaustive analysis of existing AOP frameworks, and their corresponding AO weavers, have been performed. After including a complete list of both academic and industrial proposals we have thoroughly analysed those proposals that we consider can fit best with the necessity of this project. In order to do that, we have identified five general selection criteria, and thirteen more specific requirements, that have to be satisfied by the selected aspect weaver. All the tests that have been performed have been included as appendix of the document, with the problems we have encountered but also with the solutions to those problems. Finally, the discussions about the results of the analysis and the best AOP frameworks that can be used in the project have been performed.

6 REFERENCES

6.1 Bibliography

[1] N. Loughran, et al., A domain analysis of key concerns - known and new candidates, AOSD-Europe Deliverable D43, http://www.aosdeurope.net/deliverables/d43.pdf

[2] Kung Chen, Ching-Wei Lin: An Aspect-Oriented Approach to Declarative Access Control for Web Applications. APWeb 2006.

[3] D. Xu and V. Goel. An aspect-oriented approach to mobile agent access control. International Conference on Information Technology: Coding and Computing (ITCC), 2005.

[4] Craig Walls and Ryan Breidenbach. 2007. *Spring in Action*. Manning Publications Co., Greenwich, CT, USA.

[5] Klass van den Berg, José María Conejero, Ruzanna Chitchyan. *AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation.* Deliverable 9, IST-2-004349-NOE AOSD-Europe project. 2005.

[6] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA. 2003.

[7] Pascal Bruegger. *Authentication and Authorization*. In Mini-Proceedings of the Master Seminar Advanced Software Engineering Topics: Aspect Oriented Programming, organized by the Software Engineering Group Department of Informatics University of Fribourg (Switzerland). 2006.

[8] Kang, K.C., Cohen, S.G., Hess, J.A. Novak W.E., Peterson, A.S: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, 1990.

[9] Lee, K., Kang, K., Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering, LNCS, vol. 2319, pp.62--77. Springer, Heidelberg, 2002.

[10] Batory, D.S.: Feature models, grammars, and propositional formulas. In Obbink, J.H., Pohl, K., eds.: Proc. of the 9th I.C. on Software Product Lines. LNCS, Vol. 3714, 7-20, 2005.

[11] Thüm, T., Batory, D., Kästner, C.: Reasoning about edits to feature models. In Proceedings of the 31th International Conference on Software Engineering, 2009.

[14] Gilani, W., Spinczyk, O.: Dynamic Aspect Weaver Family for Family-based Adaptable Systems. In Proceedings of NODe/GSEM, 94-109, 2005.

[15] Greenwood, P., Blair, L.: Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System, Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS Technical Report 04.01, March, 2004, Lancaster, UK, 2003.

[28] Badii, A. User-Intimate Requirements Hierarchy Resolution Framework (UI-REF): Methodology for Capturing Ambient Assisted Living Needs, Proceedings of the Research Workshop, Int. Ambient Intelligence Systems Conference (AmI'08), Nuremberg, Germany November 2008.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. *An Overview of AspectJ*. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01), Jørgen Lindskov Knudsen (Ed.). Springer-Verlag, London, UK, UK, 327-353.

6.2 Netography

[12] *http://www.aspyct.org/aspyct/* Accessed on March 2013.

[13] http*://www.hpi.uni-potsdam.de/hirschfeld/projects/aop/index.html* Accessed on March 2013.

[16] AOSD: *http://www.aosd.net/* Accessed on March 2013.

[17] *http://eclipse.org/aspectj/* Accessed on March 2013.

[18] *http://docs.jboss.org/jbossaop/docs/index.html* Accessed on March 2013.

[19] *http://www.postsharp.net/* Accessed on March 2013.

[20] *http://static.springsource.org/spring/docs/3.0.x/reference/aop.html* Accessed on March 2013.

[21] *http://www.springframework.net/doc/reference/html/aop-quickstart.html* Accessed on March 2013.

[22] *http://jac.ow2.org/* Accessed on March 2013.

[23] *http://www.aspectc.org/* Accessed on March 2013.

[24] *http://www.castleproject.org/AspectSharp* Accessed on March 2013.

[25] http://www.aspectjs.com/ Accessed on March 2013.

[26] *http://aspectwerkz.codehaus.org/index.html* Accessed on March 2013.

[27] *http://docs.castleproject.org/Windsor.Introduction-to-AOP-With-Castle.ashx?HL=aspect* Accessed on March 2013.

Appendix A. Aspect weaver questionnaire

A.1. Questionnaire[7] for the Deliverable 4.1

We need to clarify some requirements of the AOP framework before choosing the most appropriate one for the INTER-TRUST framework. In order to do that, UMA has prepared the following questionnaire. This questionnaire is to be filled specially by the end user partners, although inputs by all partners are welcomed.

Question 1. For which of the project case studies are you going to provide the answers to the following questions?

Question 2. Which programming language/languages are used to implement the different parts of the case study? *During the KoM partners mentioned different languages and we need to clarify this point in order to know if more than one AOP weaver needs to be used in the project.*

Question 3. Do you think that the aspects of the Inter-Trust framework will need to be implemented in all the above mentioned languages? *It is possible that you still do not have a clear idea about where the aspects are going to be weaved. Thus, we will complete the information provided to this question with the case studies requirements specification.*

Question 4. Which kinds of devices are used in the case study? Which operating systems? *Only desktop computers?, both desktop computers and mobile devices?, mobile devices with Android?, mobile devices with different OS?*

Question 5. Which is the programming model of the applications to be weaved with the aspects? *POJO classes?, beans?, applets for the client side?, others?*

Question 6. At which *points* of your case study will the aspects be weaved? In an aspect-oriented way the question would be "which are the join-points that need to be intercepted"? *All the join-points supported by AspectJ?, a subset of them? Which subset?*

UMA will answer to this question before selecting one specific AOP weaver, but it is interesting for us to have, if possible, a few scenarios of the case study reflecting how and where you think that the aspects need to be weaved in your case study. As said for question 3, we will complete the information provided to this question with the case studies requirements specification.

[7] A summary of the answers to this questionnaire by the end-users can be found in Table 4.1 of Section 4.1.1 where the inter-trust case studies requirements were described.

Appendix B. AOP frameworks details

B.1. AspectJ

AspectsJ Joint Point Model

Table B.1.1. Overview of join point categories exposed by AspectJ.

Category	Exposed join point	Code it represents
Method	Execution	Method body
Method	Call	Method invocation
Constructor	Execution	Execution of an object's creation logic
Constructor	Call	Invocation of an object's creation logic
Field access	Read access	Access to read an object's or a class's field
Field access	Write access	Access to write an object's or a class's field
Exception processing	Handler	Catch block to handle an exception
Initialization	Class initialization	Class loading
Initialization	Object initialization	Object initialization in a constructor
Initialization	Object pre-initialization	Object pre-initialization in a constructor
Advice	Execution	Advice execution

AspectsJ Pointcut designators

Table B.1.2. Mapping of exposed join points to pointcut designators.

Join point category	Pointcut syntax
Method execution	`execution(MethodSignature)`
Method call	`call(MethodSignature)`
Constructor execution	`execution(ConstructorSignature)`
Constructor call	`call(ConstructorSignature)`
Class initialization	`staticinitialization(TypeSignature)`
Field read access	`get(FieldSignature)`
Field write access	`set(FieldSignature)`
Exception handler execution	`handler(TypeSignature)`
Object initialization	`initialization(ConstructorSignature)`
Object pre-initialization	`preinitialization(ConstructorSignature)`
Advice execution	`adviceexecution()`

B.2. Spring AOP

Typical full-fledged Spring web application

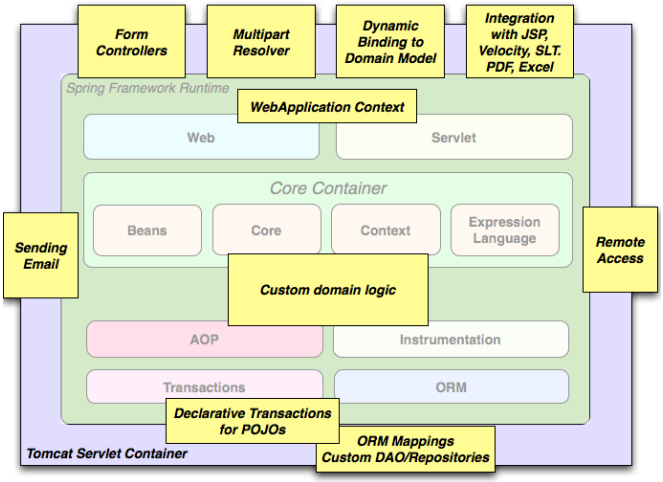

Figure B.2.1. Typical full-fledged Spring web application

Spring AOP pointcut designators

Table B.2.1. Spring leverages AspectJ's pointcut expression language for defining Spring aspects ([4]).

AspectJ designator	Description
`args()`	Limits join point matches to the execution of methods whose arguments are instances of the given types
`@args()`	Limits join point matches to the execution of methods whose arguments are annotated with the given annotation types
`execution()`	Matches join points that are method executions
`this()`	Limits join point matches to those where the bean reference of the AOP proxy is of a given type
`target()`	Limits join point matches to those where the target object is of a given type
`@target()`	Limits matching to join points where the class of the executing object has an annotation of the given type
`within()`	Limits matching to join points within certain types
`@within()`	Limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
`@annotation`	Limits join point matches to those where the subject of the join point has the given annotation

B.3. JBoss AOP

JBoss AOP join point model and pointcut constructs

Joinpoint	Pointcut Construct	Bean		ContextValues			
		Invocation	JoinpointBean	Target	Caller	Arguments	Return Value
field read	`read, field, all`	`FieldReadInvocation`	`FieldAccess`	Yes	No	No	Yes
field write	`write, field, all`	`FieldWriteInvocation`	`FieldAccess`	Yes	No	Yes	No
method execution	`execution, all`	`MethodInvocation`	`MethodExecution`	Yes	No	Yes	Yes
constructor execution	`execution`	`ConstructorInvocation`	`ConstructorExecution`	No	No	Yes	Yes
construction	`construction`	`ConstructionInvocation`	`ConstructorExecution`	Yes	No	Yes	No
method call	`call, within, withincode`	`CallerInvocation, MethodCalledByConstructorInvocation, MethodCalledByMethodInvocation`	`MethodCall, MethodCallByConstructor, MethodCallByMethod`	Yes	Yes	Yes	Yes
constructor call	`call, within, withincode`	`CallerInvocation, ConstructorCalledByConstructorInvocation, ConstructorCalledByMethodInvocation`	`ConstructorCall, ConstructorCallByConstructor, ConstructorCallByMethod`	Yes	Yes	Yes	Yes

Figure B.3.1. Joinpoint types

B.4. AspectC

AspectC Keywords

Table B.4.1. AspectC keywords.

Keyword	Description
`After`	An advice type declarator, it represents the advice code that should be run after the matched join point(s).
`Args`	A pointcut type declarator, it represents the join points whose argument types are matched by the pointcut specified.
`Around`	An advice type declarator, it represents the advice code that should be run and the matched join point(s), that in the absence of `proceed()`, are skipped.
`before`	An advice type declarator, it represents the advice code that should be run before the matched join point(s).
`call`	A pointcut type declarator, it represents the join points of calling a function whose prototype is matched by the pointcut specified.
`cflow`	A pointcut type declarator, it represents all the join points under the control flow of the specified pointcut.
`execution`	A pointcut type declarator, it represents the join points of executing a function matched by the pointcut specified.
`infile`	A pointcut type declarator, it represents the join points which exist in the specified file.
`infunc`	A pointcut type declarator, it represents the join points which exist in the specified function.
`pointcut`	Associates a name to a pointcut definition. The name can then be used in advice declarations to refer to the named pointcut declaration.
`proceed`	Used inside an around advice function, where it identifies that the original join point should be executed.
`result`	A pointcut type declarator, it represents the join points whose return type is matched by the pointcut specified.
`this`	Is used inside an advice functions. It is a pointer to a struct and allows advice code to access context information of the matched join points. Through it, advice code can access the function name via "this->funcName" and the join point type via "this->kind".

AspectC joinpoint model

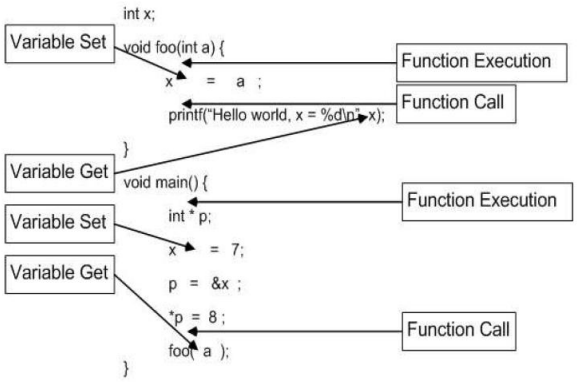

Figure B.4.1. Joint point model of AspectC.

B.5. AspectC++

AspectC++ Join point model

types:	
`Result`	result type
`That`	object type
`Target`	target type
`AC::Type`	encoded type of an object
`AC::JPType`	join point types
static methods:	
`int args()`	number of arguments
`AC::Type type()`	typ of the function or attribute
`AC::Type argtype(int)`	types of the arguments
`const char *signature()`	signature of the function or attribute
`unsigned id()`	identification of the join point
`AC::Type resulttype()`	result type
`AC::JPType jptype()`	type of join point
non-static methods:	
`void *arg(int)`	actual argument
`Result *result()`	result value
`That *that()`	object refered to by `this`
`Target *target()`	target object of a call
`void proceed()`	execute join point code
`AC::Action &action()`	`Action` structure

Figure B.5.1. API of the join point model of AspectC++.

Appendix C. How to

C.1. AspectJ

C.1.1 How to use compile-time weaving

Aspects can be compiled and woven at source time using the AspectJ compiler (`ajc`):

```
ajc aspect.aj BaseProgram.java
```

C.1.2 How to use LTW

- Compile aspect/s and generate *.jar* files for LTW:

  ```
  ajc aspect.aj –outxml –outjar aspect.jar
  ```

  Aspect can be defined in classic syntax (`.aj` file type) or @AspectJ annotations (`.java` file type). A `jar` file can include one or more aspects.

  Option `–outxml` automatically generate the `aop.xml` file within the `jar` file.

- The `aop.xml` file:

  Aspects must be defined in one or more `aop.xml` files. This file can be named `aop.xml` or `aop-ajc.xml`, others names will be ignored by the weaver. The file must be located in `META-INF/` directory (inside or outside of *.jar* files) and accessible to the *classloader*.

  Figure C.1.1 shows an `aop.xml` example file. A complete description of this file can be found in http://www.eclipse.org/aspectj/doc/released/devguide/ltw-configuration.html#enabling-load-time-weaving.

```
<aspectj>
    <!-- declare existing aspects to the weaver -->
    <aspects>
        <aspect name="Aspect1"/>
        <aspect name="Aspect2"/>
        ...
    </aspects>

    <!-- weaver options -->
    <weaver options="-verbose -showWeaverInfo -debug">
        <!-- Weave types that are within the "bar1" package -->
        <include within="bar1.*"/>
        ...
        <!-- Do not weave types within the "bar2" pakage -->
        <exclude within="bar2.*"/>
    </weaver>
</aspectj>
```

Figure C.1.1. Example of the aop.xml file.

- Enabling LTW:

  To run any application with LTW, the option `-javaagent:pathto/aspectjweaver.jar` must be specified to the JVM. Moreover, `aspectjrt.jar` library, *.jar* files with the aspects, and `aop.xml` files must be located in the *classpath*.

C.1.3 How to use LTW in Java Applets

In order to use LTW in Java Applets, aspects must be specified directly in the AspectJ weaving classloader through the `WeavingURLClassLoader` class. To do that, we can wrap the original Applet with an Applet that defines the *urls* of the aspects and loads the original Applet. Figure C.1.2 shows an example of how to perform these actions:

```
public class MyAppletWrapper extends JApplet {

    public void init() {
        URL source = new URL(getCodeBase() + "MyApplet.jar");       // base program (original Applet)
        URL aspectjrt = new URL(getCodeBase() + "aspectjrt.jar");   // aspectjrt.jar library
        URL aspect = new URL(getCodeBase() + "myAspect.jar");       // aspects

        URL[] urlList = {source, aspectjrt, aspect};
        URL[] aspectList = {aspect};

        WeavingURLClassLoader weavingClassLoader = new WeavingURLClassLoader(urlList, aspectList,
                                                        Thread.currentThread().getContextClassLoader());
        Thread.currentThread().setContextClassLoader(weavingClassLoader);

        Class mainApplet = weavingClassLoader.loadClass("MyApplet");
        Object applet = mainApplet.newInstance();
        if (mainApplet != null) {
            Class[] args = null;
            Method init = mainApplet.getMethod("init", args);
            init.invoke(applet, (Object[])null);
            this.add((Component)applet);
        }
    }
}
```

Figure C.1.2. Applet using the WeavingURLClassLoader.

To specify the `-javaagent:pathto/aspectjweaver.jar` option to the JVM in which the Applet runs, we can use an `.jnlp` file and the Java Web Start technology. In the `.jnlp` file we must also define all resources used by the Applet (including the `aspectjweaver.jar` library). Figure C.1.3 shows an example of the `.jnlp` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="" href="">

    <information>
        <title>Title</title>
        <vendor>Vendor</vendor>
    </information>

    <resources>
        <!-- Application Resources -->
        <j2se version="1.6+"
              href="http://java.sun.com/products/autodl/j2se"
              java-vm-args="-javaagent:aspectjweaver.jar"/>
        <jar href="MyAppletWrapper.jar" main="true"/>
        <jar href="aspectjrt.jar" />
        <jar href="aspectjweaver.jar"/>
        <jar href="myAspect.jar" />
    </resources>

    <applet-desc
         name="Applet name"
         main-class="MyAppletWrapper"
         width="300"
         height="300">
     </applet-desc>
     <update check="background"/>

</jnlp>
```

Figure C.1.3. Example of .jnlp file.

All *jar* files that the Applet uses must be signed. These are, among others: the original Applet, the Applet that wraps the original Applet, the *jar* files of the aspects, the `aspectjrt.jar` and `aspectjweaver.jar` libraries.

C.1.4 How to enable/disable aspects at runtime

Each aspect or even each pointcut of an aspect can be enable/disable at runtime using the `if()` pointcut that AspectJ provides to define a conditional pointcut expression which will be evaluated at runtime for each candidate join point. To do that, each aspect that we want to enable/disable at runtime must have an "enabled" property. That variable must be static and can be set at runtime. Moreover, we can have more than one "enabled" property in each aspect, for example one per pointcut to control which pointcut must be enabled or disabled. The following code (Figure C.1.4) is an example of using enable/disable capability. `if(…)` statement can include any complex condition, but the involved variables must be defined as static.

```
public aspect MyAspect {

    static boolean enabled = true;

    pointcut methodsOfInterest():
        if(enabled) && execution(* *(..));

    ...

    public static boolean isEnabled() {
        return enabled;
    }

    public static void setEnabled(boolean newState) {
        enabled = newState;
    }
}
```

Figure C.1.4. Enable/disable capability at runtime.

The `if()` pointcut is also available with the @AspectJ style, as we shown in Figure C.1.5. In this case, the condition cannot be defined inside de `if()` statement, and the method that represents the pointcut will return the *boolean* value of the `if()` condition.

```
@Aspect
public class MyAspect {

    static boolean enabled = true;

    @Pointcut("if() && execution(* *(..))")
    public static boolean methodsOfInterest() {
        return enabled;
    }

    ...

    public static boolean isEnabled() {
        return enabled;
    }

    public static void setEnabled(boolean newState) {
        enabled = newState;
    }
}
```

Figure C.1.5. Enable/disable capability at runtime using @AspectJ style.

C.1.5 How to use AspectJ in Android O.S.

As Dex relies on `.class` files, AspectJ can be used in Android performing the following steps:

1. Let Java compile java to `.class` files.
2. Let AspectJ compiler inject point cuts and advices to the `.class` files.
3. Let Dex take these new class files and create `.dex` files.

More information can be found in http://code.google.com/p/android-aspectj/

C.2. Spring AOP

C.2.1 How to use runtime weaving

- Required libraries:

  In order to use Spring and Spring AOP, there are many required libraries; the most important are: `antl-runtime`, `aopalliance`, `asm`, `cglib`, `commons-logging`, `spring`, `spring-aop`, `spring-aspects`, `spring-beans`, `spring-context`, `spring-context-support`, `spring-core`, `spring-expression`, `spring-tomcat-weaver`, and `spring-web`.

To use Spring proxy, you need to add CGLIB2 library

- Defining advices:

  Advices are defined as Java classes that implement the appropriate interface: `MethodBeforeAdvice` (Before advice), `AfterReturningAdvice` (After returning advice), `ThrowsAdvice` (After throwing advice), or `MethodInterceptor` (Around advice). Figure C.2.1 shows an example of a before advice:

```
1  import java.lang.reflect.Method;
2  import org.springframework.aop.MethodBeforeAdvice;
3
4  public class MyBeforeMethod implements MethodBeforeAdvice {
5      @Override
6      public void before(Method method, Object[] args, Object target) throws Throwable {
7          // do something
8      }
9  }
```

Figure C.2.1. Example of Before advice.

- The bean configuration file:

  Classes that will be woven and advices that will be injected can be defined as beans in one or more .xml files. A proxy object of the class that will be woven must be created. This can be done explicitly or automatically.

  An example of how to manually create a proxy bean is shown in Figure C.2.2; `target` property defines which bean will be intercepted, and `interceptorNames` property defines which classes (advices) will be applied on that proxy/target object. This aspect automatically intercepts all the methods of the target class.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="myBeanID" class="Myclass">

    </bean>

    <bean id="myBeanAdviceID" class="MyAdviceClass" />

    <bean id="MyBeanProxyID" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="myBeanID" />
        <property name="interceptorNames">
            <list>
                <value>myBeanAdviceID</value>
            </list>
        </property>
    </bean>
</beans>
```

Figure C.2.2. Example of bean configuration file.

- Defining pointcuts:

Pointcuts are defined in a bean configuration file indicating which method should be intercept, by method name or regular expression pattern. An 'advisor' must be created to associate both advices and pointcuts. An advisor groups advices and pointcuts into a single unit that is passed to a proxy factory object. Figure C.2.3 shows an example of this. The property `mappedName` indicates the name of the method that will be intercepted.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="myBeanID" class="MyClass">

    </bean>

    <bean id="myBeanAdviceID" class="MyAdviceClass" />

    <bean id="myBeanProxyID" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="myBeanID" />
        <property name="interceptorNames">
            <list>
                <value>myAdvisorID</value>
            </list>
        </property>
    </bean>

    <bean id="myPointcutID" class="org.springframework.aop.support.NameMatchMethodPointcut">
        <property name="mappedName" value="myMethodName" />
    </bean>

    <bean id="myAdvisorID" class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <property name="pointcut" ref="myPointcutID" />
        <property name="advice" ref="myBeanAdviceID" />
    </bean>
</beans>
```

Figure C.2.3. Example of a pointcut in a bean configuration file.

Regular expression pointcut can also be used. Figure C.2.4 shows an example of a pointcut that intercepts the methods which has the word "`my`" within the method name.

```
<bean id="myAdvisorID" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="patterns">
        <list>
            <value>.*my.*</value>
        </list>
    </property>

    <property name="advice" ref="myBeanAdviceID" />
</bean>
```

Figure C.2.4. Example of regular expression pointcut.

- Creating proxies automatically (autoproxying):

  Spring comes with two auto proxy creators (`BeanNameAutoProxyCreator` and `DefaultAdvisorAutoProxyCreator`) to create proxies for the beans automatically, instead of creating many proxy factory beans manually. `BeanNameAutoProxyCreator` allows including all the beans (via bean name, or regular expression name) and advisor into a single unit. `DefaultAdvisorAutoProxyCreator` makes transparent the use of proxies, if any of the beans is matched by an advisor, Spring will create a proxy for it automatically. An example of this latter proxy creator is shown in Figure C.2.5.

```
<bean id="myAdvisorID" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="mappedName" value="myMethodName" />
    <property name="advice" ref="myBeanAdviceID" />
</bean>

<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />
```

Figure C.2.5. Example of DefaultAdvisorAutoProxyCreator.

- Getting beans:

  Do to Spring is a beans container, we must get the instances of our beans through the `ApplicationContext` class as Figure C.2.6 shows.

```
ApplicationContext appContext = new ClassPathXmlApplicationContext(new String[] { "Spring.xml" });
    MyClass obj = (MyClass) appContext.getBean("myBeanProxyID");
```

Figure C.2.6. Getting instances of the proxies.

If we use the proxy creator, we can get the original bean and Spring will create the proxy automatically (Figure C.2.7).

```
ApplicationContext appContext = new ClassPathXmlApplicationContext(new String[]{ "Spring.xml" });
MyClass obj = (MyClass) appContext.getBean("myBeanID");
```

Figure C.2.7. Getting instances of beans.

C.2.2 How to use Spring AOP and AspectJ

To use @AspectJ aspects in a Spring configuration it is need to enable Spring support for configuring Spring AOP based on @AspectJ aspects, and autoproxying beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.

The @AspectJ support is enabled by including the following element inside the spring configuration file:

```
<aop:aspectj-autoproxy/>
```

It will also need two AspectJ libraries on the classpath of the application: `aspectjweaver.jar` and `aspectjrt.jar`.

With the @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the @Aspect annotation) will be automatically detected by Spring and used to configure Spring AOP. Figure C.2.8 shows how to declare a bean for the aspect in the same way we have done for Spring AOP advices in the previous section.

```
1  <bean id="myAspectID" class="org.xyz.MyAspectClass">
2      <!-- configure properties of aspect here as normal -->
3  </bean>
```

Figure C.2.8. Declaring the bean of the aspect in the Spring configuration file.

Aspects (classes annotated with @Aspect) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations. Figure C.2.9 shows an example of an annotated class with @AspectJ.

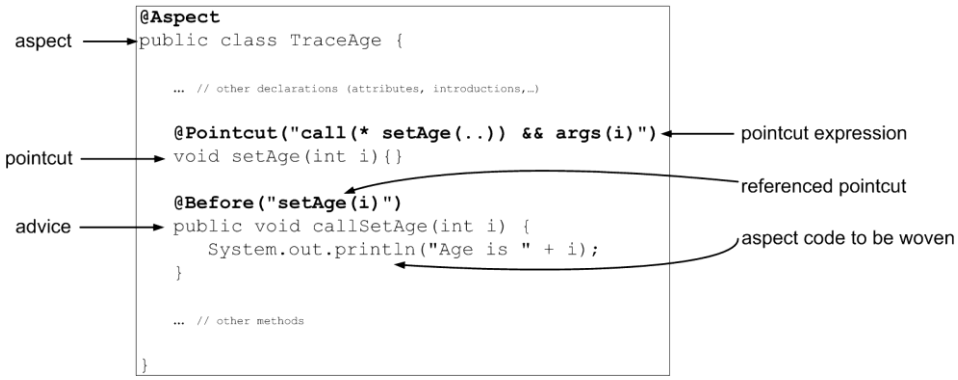

Figure C.2.9. @AspectJ class definition.

The complete @AspectJ support for Spring AOP can be found in
http://static.springsource.org/spring/docs/2.0.8/reference/aop.html

C.2.3 How to use Spring AOP and AspectJ load-time weaving

- Required libraries:

  At a minimum you will need the following libraries to use the Spring Framework's support for AspectJ LTW: `spring.jar` (version 2.5 or later), `aspectjrt.jar` (version 1.5 or later), and `aspectjweaver.jar` (version 1.5 or later). In addition, if you are using the Spring-provided agent to enable instrumentation, you will also need the `spring-agent.jar` library.

- Aspects:

  The aspects that you use in LTW have to be AspectJ aspects. They can be written in either the AspectJ language itself or you can write your aspects in the @AspectJ-style. The latter option is, of course, only an option if you are using Java 5+, but it does mean that your aspects are

then both valid AspectJ and Spring AOP aspects. Furthermore, the compiled aspect classes need to be available on the *classpath*.

As in LTW in AspectJ, it also need to create an `META-INF/aop.xml` file to inform the AspectJ weaver which aspects we want to weave into the classes.

- Configuring LTW:

  It needs to specify a `LoadTimeWeaver` in the beans configuration file of Spring:

  ```
  <context:load-time-weaver/>
  ```

- Running LTW:

  In this case it is also need to specify a java agent to the JVM. This is the `spring-agent.jar` file. The option is as follows:

  ```
  -javaagent:<path-to>/spring-agent.jar
  ```

C.3. JBoss AOP

C.3.1 How to use runtime weaving with the API

- Required libraries:

  `jboss-aop-jdk50.jar` is the only required library, that includes the API.

- Preparing classes:

  To indicate which joinpoints must be instrumented for dynamic AOP, the classes must be prepared using “prepare” expressions. A prepare expression can be declared either in an `.xml` file or as an annotation. For instance, the following examples prepare the entire class POJO for dynamic AOP:

jboss-aop.xml	**Annotations**

`<aop>`
`  <prepare expr="all(POJO)"/>`
`</aop>`
	`@Prepare`
	`public class POJO {`
	`  ...`
	`}`

- Defining advices:

  Advices can be implemented as a classic Java class or as an interceptor. Figure C.3.1 and Figure C.3.2 show examples of advices and interceptors.

```
public class MyAspect {

  public void advice1() {
    …
  }

  public Object advice2(Invocation invocation)
      throws Throwable {
    …
    return invocation.invokeNext();
  }
}
```

Figure C.3.1. Aspect as a Java class.

```
public class SimpleInterceptor
    implements Interceptor {

  public String getName() {
    return "SimpleInterceptor";
  }

  public Object invoke(Invocation invocation)
      throws Throwable {
    …
    return invocation.invokeNext();
  }
}
```

Figure C.3.2. Interceptor.

- Using the API:

  Advices and interceptors are useless unless they are bound to a pointcut expression, indicating when they should be called. Figure C.3.3 shows how to use the API for the previous advice defined in Figure C.3.1, while Figure C.3.4 shows how to use the API for the interceptor defined in Figure C.3.2.

```
// declares the aspect class
AspectFactory aspectFactory = new GenericAspectFactory(MyAspect.class, null);
AspectDefinition aspectDefinition = new AspectDefinition("myAspect",
                                                         Scope.PER_INSTANCE,
                                                         aspectFactory);
AspectManager manager = AspectManager.instance();

// deploys the aspect
AspectManager.instance().addAspectDefinition(aspectDefinition);

// creates the advice factory
AdviceFactory adviceFactory = new AdviceFactory(aspectDefinition,
                                                "advice1",
                                                AdviceType.BEFORE);
// creates the binding
AdviceBinding binding = new AdviceBinding("execution(POJO->new(..))", null);
binding.addInterceptorFactory(adviceFactory);

// adds the binding
AspectManager.instance().addBinding(binding);
```

Figure C.3.3. Using the JBoss AOP API.

```
1  // creates the binding
2  AdviceBinding binding =
3    new AdviceBinding("execution(POJO->new(..))",
4                      null);
5  binding.addInterceptor(SimpleInterceptor.class);
6
7  // adds the binding
8  AspectManager.instance().addBinding(binding);
```

Figure C.3.4. Using the JBoss AOP API with interceptors.

The API provides methods to add and remove any binding.

In both cases, the binding can be also defined in a `jboss-aop.xml` file, as Figure C.3.5 shows for the advice or as Figure C.3.6 shows for the interceptor.

```
1  <aop>
2    <aspect name="myAspect" class="MyAspect" />
3    <binding pointcut="execution(POJO->new(..))">
4      <before aspect="myAspect" name="advice1" />
5    </binding>
6  </aop>
```

Figure C.3.5.Defining the binding in the jboss-aop.xml file.

```
1  <aop>
2    <binding pointcut=" execution(POJO->new(..))">
3      <interceptor class="MyInterceptor.class"/>
4    </binding>
5  </aop>
```

Figure C.3.6. Defining the binding of the interceptor in the jboss-aop.xml file.

- Compiling and running:

  To run any application that uses the API, the `jboss-aop-jdk50.jar` library must be located in the `jboss.aop.classpath` and the java agent provided by the `jboss-aop-jdk50.jar` library must be specified as an option to the JVM as follows:

  `-javaagent:jboss-aop-jdk50.jar`

C.4. AspectC

C.4.1 How to use AspectC

- Defining aspects:

  Aspects are defined in `.acc` files. These files contain only the code of the advices in an AspectJ similar syntax. The following examples (Figure C.4.1) show three different kinds of advices in an `.acc` file. Pointcuts can also be named as in AspectJ.

```
1  pointcut executionMain(): execution(int main()) ;
2
3  before(): executionMain() {
4      printf("before ... ");
5  }
6
7  after(): executionMain() {
8      printf("after ... ");
9  }
10
11 void around(): call(void foo(char *)) {
12     printf("in around advice: start\n");
13     proceed();
14     printf("in around advice: end\n");
15 }
```

Figure C.4.1. An .acc aspect file.

- Weaving compilation:

  The ACC compiler tools primarily consist of two utilities: `accmake` and `tacc`, extensions to the GNU make utility and the `gcc` compiler respectively. `accmake` is to be used in place of the `make` utility, and `tacc` to be that of the `cc` compiler. The basic usage of `tacc` tool is:

  `tacc foo.c bar.c aspect1.acc` (generates *a.out* with aspect functionalities in *aspect.acc*)

  `tacc -o bin foo.c bar.c aspect.acc` (generates *bin* with aspect functionalities in *aspect.acc*)

  `tacc -o bin *.c *.acc` (a short-hand form generates *bin* with aspect functionalities)

C.5. AspectC++

- Defining aspects:

  Aspects are defined in `.ah` files in a similar syntax of AspectJ language. Figure C.5.1 shows an example of a reusable tracing aspect.

```
aspect Trace {
   pointcut virtual functions() = 0;
   advice execution(functions()) : around() {
      cout << "before " << JoinPoint::signature() << "(";
      for (unsigned i = 0; i < JoinPoint::ARGS; i++)
         cout << (i ? ", " : "") << JoinPoint::argtype(i);
      cout << ")" << endl;
      tjp->proceed();
      cout << "after" << endl;
   }
};
```

Figure C.5.1. Reusable Trace aspect.

This aspect can be derived to redefine the pointcut functions to apply the aspect to the desired set of functions, as Figure C.5.2 shows.

```
aspect TraceMain : public Trace {
   pointcut functions() = "% main(...)";
};
```

Figure C.5.2. Derived Trace aspect.

- Weaving compilation:

  The usage of `ag++` is mainly influenced by the usage of the GNU `g++` compiler and the sinopsis is like.

  ```
  ag++ [options] [files…]
  ```

  For example to weave and compile a single file you simply invoke

  ```
  ag++ -c main.cc
  ```

C.6. FeatureC++

- Defining aspects:

  In FeatureC++, aspects are classes that refine classes and override the methods that we want to intercept. A simple example is shown in Figure C.6.1 where a class `Hello` defines the method `run()`. Figure C.6.2 shows the aspect: the class Hello with the keyword “refines” and the overridden method `run()`. This method now invokes the original method of the super class and then does anything in the same way that in an after advice.

```
1  class Hello {
2  public:
3          int run() {
4          printf("Hello");
5          return 0;
6      }
7  };
8
9  int main() {
10     //create instance of "test"
11     Hello myHello;
12
13     //run Hello::run as entry-point of the app
14     return myHello.run();
15 }
```

Figure C.6.1. Example base class.

```
1  refines class Hello {
2    public:
3      int run() {
4        //invoke refined method
5          int res = super::run();
6          if (res!=0)
7              return res;
8
9          printf(" World!");
10         return 0;
11     }
12 };
```

Figure C.6.2. Example of refined class (aspect).

- Preparing the files:

  Both classes need to be defined in different files (both with the same name, `Hello.h` in the above example) and placed in different folders (`Base` folder and `Hello` folder for example).

  Moreover, an additional file is required in the root folder with the following content:

```
Base
World
```

This file has the `.equation` extension (e.g., `HelloWorld.equation` in the above example).

- Weaving compilation:

  Weaving and compiling are two different steps in FeatureC++. First, first the `fc++` has to be executed with the previous defined `.equation` file:

  ```
  gc++ -gpp HelloWorld.equation
  ```

  This process generates a new folder: `.output.HelloWorld` containing the source files. Then we have to compile these sources files with the classic `g++` compiler:

  ```
  g++ -o bin .output.HelloWorld/*.cpp
  ```