Management of Aspect Weaving

By

Mohamed Hassan, M.Sc.

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfillment of

The requirements for the degree of

Master of Applied Science in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

January 2008

© Copyright 2008, Mohamed Hassan

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 978-0-494-36824-4

Our file Notre référence

ISBN: 978-0-494-36824-4

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

The undersigned recommends to the Faculty of Graduate Studies and Research
acceptance of the thesis

Management of Aspect Weaving

Submitted by
Mohamed Hassan

In partial fulfillment of the requirements for the
Degree of Master of Applied Science in Electrical Engineering

Thesis Supervisor
Prof. Samuel A. Ajila

Chair, Department of Systems and Computer Engineering
Prof. Victor C. Aitken

Carleton University
January 2008

ABSTRACT

Dynamic weaving in Aspect-Oriented Programming (AOP) systems has received growing interests in both the academia and the industry. It can be classified into two levels in terms of aspect weaving: a class-level and an object-level weaving. Current weaving techniques have a significant performance overhead which is considered a deficiency.

In order to provide an efficient dynamic weaving technique at both the class-level and the object-level, these performance drawbacks have to be reconsidered. In this thesis, new techniques are proposed at each weaving level to resolve them.

The first technique is an efficient dynamic aspect weaving for both the class-level and the object-level. The technique checks the availability and the location of the aspect in the production (compiled) or the source (un-compiled) library, and dynamically weaves it into the system. It seeks to improve the performance of weaving an aspect, executing an advice code, and the normal execution. The second technique is a novel concept to check if the aspect was previously weaved to the running system from the source library and weaves only the modified part of the aspect.

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Samuel A. Ajila, for providing amazing guidance and feedback and showing me the ropes in the world of academic research. His promptness and continual advices were highly appreciated.

I am indebted to Rasha, my wife, for her extreme patience and understanding at stressful times. Completion of this work would have been difficult without her sincere love and encouragement.

Mohamed Hassan

Table of Contents

Chapter 1: Introduction.....	1
1.1 Weaving Aspect.....	4
1.2 Problem Statement.....	5
1.3 Contribution.....	6
1.4 Thesis Organization	7
Chapter 2: Weaving Aspects.....	9
2.1 Aspect-Oriented Programming.....	9
2.2 Aspect-Oriented Design.....	11
2.2.1 Types of Crosscutting Interfaces	11
2.2.2 Implementation of Crosscutting.....	12
2.2.3 Composition Rules.....	13
2.2.4 The Relationship between Aspect and Object	15
2.3 Static Weaving in AOP.....	16
2.4 Dynamic Weaving in AOP	17
2.4.1 Examples of Dynamic Weaving	18
2.4.2 Dynamic Weaving Techniques	19
2.4.2.1 Static Code Translation.....	19
2.4.2.2 Dynamic Code Translation	20
2.5 Related Works.....	21
2.5.1 Support AOP Features	25
2.6 Problem with Ordinary Weaving Techniques	25

2.7 Summary	26
Chapter 3: Management of Aspect Weaving	28
3.1 The Weave History Technique	29
3.1.1 An Overview of the Weave History Technique.....	29
3.1.2 Aspect in Weave History Technique	32
3.1.3 Abstract Aspect.....	34
3.1.4 Advice	35
3.1.4.1 Pointcut	35
3.1.4.2 Advice Body	36
3.1.5 The Method <i>sourceLib()</i>	39
3.2 Comparing Two Aspects	40
3.3 Controlling the Weaver.....	43
3.4 Identifying Joinpoints	44
3.5 Summary.....	45
Chapter 4: Implementation of the Weave History Technique	46
4.1 Case Study – Auction System.....	46
4.1.1 Description.....	46
4.1.2 Using the English Auction Service.....	47
4.2 Checking the Version of a Running System.....	49
4.2.1 Insert Hooks as Breakpoints	50
4.2.2 Call the Advice	50
4.3 Access Control According to the Connection Type	51
4.3.1 Insert Hooks as Breakpoints	51

4.3.2 Call the Advice	52
4.4 Weave History and Other Dynamic AO Approaches	52
Chapter 5: Weave History Technique Experimental Results	54
5.1 Two Kinds of Weaving.....	55
5.2 Testing the Aspect Codes	56
5.3 Weave History Technique Measurements	58
5.4 Summary.....	61
Chapter 6: Conclusion.....	63
6.1 Contributions	63
6.2 Limitations	64
6.3 Future Directions	65
Bibliography	66
Appendix A: A Sample from the Weave History Program.....	72

List of Figures

Figure 1.1. Crosscutting Concerns.....	2
Figure 1.2. An Information Technology Map Published by Japan Information..... Technology Services Industry Association (JISA) [32], 2004.....	3
Figure 2.1. An Aspect in AspectJ.....	10
Figure 2.2. Pointcut Representations.....	14
Figure 2.3. One-to-one Relation- An Aspect per Object.....	15
Figure 2.4 One-to-many Relation- An Aspect is Shared by a Group of Objects.....	16
Figure 2.5. Static Aspect Oriented Programming.....	17
Figure 3.1. Weave History Technique Architecture.....	30
Figure 3.2. Activity Diagram for an Advice Body.....	34
Figure 3.3. Abstract Pointcut Defined in the Subaspect.....	36
Figure 3.4. Elements Interaction within the Weave History Technique Architecture.....	38
Figure 3.5. Handling the Source and the Previously Weaved Libraries.....	40
Figure 3.6. The Method getClass() Compares between Two Classes.....	42
Figure 3.7. The Method getMethod() Confirms that the Requested..... Method is Modified.....	43
Figure 4.1. An Auction Service.....	48
Figure 4.2. Abstract Pointcuts Related to the Two Versions of Auction System.....	49
Figure 4.3. Replace Client Rules Depending on Connection Type.....	51
Figure 5.1. Login Menu before Using Weave History Technique.....	57
Figure 5.2. Override Original Login Menu Using around Joinpoint.....	58

Figure 5.3. Elapsed Time (ms) for Translating New and Modified Classes from.....	
Source Library to the Running System Using Weave History.	59
Figure 5.4. Elapsed Time (ms) for Static and Dynamic_ of New Classes.....	60

List of Tables

Table 2.1. Comparison between Different Dynamic AOP Approaches.....	25
Table 2.2. A Comparison between Class-level and Object-level Dynamic Weaving.	27
Table 3.1. Weave History Technique is a Hybrid of the Two Techniques.....	32
Table 5.1. The Average Elapsed Time for Weave History Technique.....	55
Table 5.2. Comparison between Weave History, and Static Weaving, <i>Wool</i> , <i>JAC</i> , <i>Prose</i>	61

Chapter 1

Introduction

The concept of *Separation of Concerns (SoC)* [9] is proposed by Dijkstra. He claimed that a complex problem could be broken down into loosely-coupled, easy to solve, sub-problems. These sub-problems or concerns can be relatively easier to solve than the original problem.

The separation of concerns enables us to focus on the aspect of the subprogram without caring about other irrelevant aspects. Modularizing systems enhance their qualities in terms of understandability, reusability and maintainability. Such modularized subprograms are called functions, methods or classes depending on the programming language.

However some concerns are difficult to be modularized because they spread across the system, for example security concern. This type of concern is called the crosscutting concern, as shown in figure 1.1. Crosscutting concerns lead to interdependencies between modules and increase the complexity of the code. They prevent reuse and increase the difficulty in determining the effects arising from changing a line of code [23].

Figure 1.1. Crosscutting Concerns

Examples of crosscutting concerns [31]:

- Logging calls are usually *scattered* and *tangled* throughout system code.
- Software layers are often violated to meet their performance requirements.
- Methods or variables are sometimes introduced to an object to make the object acceptable in another context, though they may be unrelated to the object's original purpose.

Scattered code is found in places where it does not belong. The code necessary to fulfill one concern is spread over the classes needed to fulfill another concern. Collecting these codes into a common location seems to be a worthy improvement.

There has been a number of programming languages aimed to distinguish separation of concerns such as Aspect-Oriented Programming (AOP) [16]. Figure 1.2 shows that in the industry domain, AOP is actually becoming an essential technology in developing software systems.

Figure 1.2. An Information Technology Map Published by Japan Information Technology Services Industry Association (JISA) [32], 2004

Sato et al. [32] described a number of enterprise systems support AOP functionality in different forms (for example pointcut/ advice and XML/ interceptor) to provide a different range of functionalities. These systems contain various crosscutting concerns such as logging and debugging information about particular modules and testing them. These functionalities normally scatter a group of tangled codes concerning each function, and crosscut the boundary between the modules.

Separating crosscutting concerns in AOP can give quality modules that are independent from the rest of modules and thus can easily be reused for other purposes. Elrad et al. [10] defined the Aspect-Oriented approach as a technique that resolves crosscutting concerns where each concern is encapsulated in a modular unit, called *Aspect*. The Aspects are localized to provide the benefits of modularity. The interfaces

between the aspect and the rest of the system are defined. These interfaces provide the possibility to deal with different isolated concerns.

1.1 Weaving Aspect

AOP supports two methods to weave (compose) an aspect into a system:

- Static weaving: It allows weaving an aspect into the system only at compiling time. We consider this as a disadvantage since the aspects have to be recompiled whenever the weaving description changes. Another disadvantage is that the running system is not completely separated from the aspect's functionality. The aspect code is in-lined into the system code which makes it difficult to adapt or replace the aspect during runtime. In our view this leads to a stronger coupling between an aspect and particular system.
- Dynamic weaving: It allows dynamic weaving and unweaving (decomposing) an aspect into and from the running system. Therefore, *pointcuts* and *advice* are changeable during runtime.

Dynamic AO weaving techniques [31] are classified at two levels:

- Class-level weaving: A class-level dynamic weaving technique affects the behavior of the whole system. It is mainly used during system development. Most of class-level dynamic weaving implementations are based on static code translation. They allow advices to be activated or deactivated during runtime. Hooks are statically inserted into all the joinpoints. These hooks determine at runtime whether or not there is associated advice to be activated at each joinpoint.

These runtime checks for the status of the advice in all the joinpoints perform a significant overhead.

- Object-level weaving: An object-level dynamic weaving technique changes the behavior of a part of the system at runtime. It can change the behavior of an instance in the running system to follow specific requirements. Object-level weaving is provided by most of today's Java-based applications such as J2EE servers. However, this implementation technique assumes that the aspect is available (in the production library) to be weaved to the running system. In order to load object into the running system, debugger interface is used to insert a hook as a pointcut. This technique imposes heavy cost on the execution time and drawbacks in system performance.

1.2 Problem Statement

In current dynamic AOP techniques, programmers can weave or remove aspect from the running system without recompiling the system each time.

When an event occurs, the programmer weaves the aspect into the running system using either a debugger interface or a compiled aspect available in the production library. The programmer searches for the proper aspect in the production library. If the aspect is not available in the production library, the programmer will search for it in the source library, compile it and move the aspect to the production library. During these manual steps, the running system will be under threat from any malicious activity. Moreover, these manual steps have serious drawbacks in the running system performance.

The implementation of current dynamic AOP techniques assumes that the programmer is always available to weave the aspect or to insert codes into the system

which is not practical. Today, with the increasing number of internet system attacks, these techniques do not have the flexibility to overcome threats efficiently

1.3 Contribution

In order to provide an efficient dynamic weaving technique at the class-level and the object-level by addressing each problem mentioned above, this thesis proposes two techniques to manage the weaver. Both of them are implemented in Java and provide efficient class-level and object-level aspect weaving.

- The first one is an integrated dynamic weaving technique for class-level and object-level from both the production (compiled) and source (un-compiled) libraries. This requires bi-directional transformations of rules between the aspect and the running system. The proposed technique seeks to improve the dynamic weaving of an aspect and the execution of an advice code. As a consequence, the performance is significantly improved.
- The second proposed technique in this thesis is a concept designed to check if the class was previously weaved to the running system from the source library and weaves only the modified part of the class. This technique seeks to reduce the overhead by using incremental weaving of a class from the source library.

The proposed techniques for the class-level and the object-level aspect weaving are a dynamic aspect weaver for Java called *Weave History Technique*.

Using *Weave History Technique*, the running system controls the selection of weaving aspect. When an event occurs, the Technique transfers the event to a request for a class or an object within a certain class. The technique can verify the availability and the location of the aspect. If the aspect is available in the production library, it weaves the

aspect to the running system. Otherwise, the technique calls the aspect from the source library, loads it into the production library and weaves it into the running system.

The *Weave History Technique* inserts hooks as pointcuts into loaded classes (or objects) at runtime. The definition of the hooks can dynamically change according to the system requirements. This seeks to reduce the checking overhead at unnecessary hooks in the running system.

This thesis provides ample techniques that take advantage of the runtime flexibility of the static AO languages without forgetting the research results of the dynamic AOP implantation. The AO programming languages have trade-offs between runtime flexibility and costs. Static AOP has performance advantages compared to dynamic AOP since the necessary runtime type checks are statically done. Due to that property, static AOP has however less runtime flexibility than dynamic AOP.

In the research domain of AOP, researchers try to add dynamic runtime rules to the static AOP as much as dynamic AOP. However, most of these systems do not require that much runtime power, and this causes non-negligible runtime overheads.

In industry, developers claim that static AOP is too strict and not suitable for developing modern application software [32] which requires dynamic plugged components such as J2EE.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2: Dynamic Aspect Weaver

This chapter describes AOP and its different design constraints. We will review the benefits of a dynamic weaving in AOP environment. Then, we will note how dynamic weaving features are helpful in both development and operation phases. Two dynamic weaving techniques are introduced: static code translation and dynamic code translation. Finally, the current dynamic weaving techniques are reviewed.

Chapter 3: Management of Aspect Weaving

In order to address the performance problem discussed in the previous chapter, we first propose a dynamic aspect weaver called *Weave History Technique*. We explain the design and the execution process of the Weave History.

Chapter 4: Implementation of the Weave History Technique

We present a case study and implementation of the Weave History. We highlight the pros and cons of the proposed technique. A comparison between the proposed technique and the other related techniques will be discussed.

Chapter 5: Weave History Technique Experimental Results

Measurement experiments based on dynamically weaving classes and objects to a running system are presented. The evaluation of the technique is reported. A comparison with the other AOP implementation approaches is presented.

Chapter 6: Conclusion

This chapter presents the contributions and the limitations of the thesis and the different future directions to expand this work.

Chapter 2

Weaving Aspects

Object-oriented software systems that are developed using aspect-oriented programming techniques consist of classes and aspects. Classes implement the functionality of the system. On the other hand, aspects capture technical concerns such as persistence, failure handling, or process synchronization. Aspect languages cannot be processed by today's compilers for OO programming language.

Aspects are merged with classes in order to produce an executable program. The process of merging is called weaving. There are two ways in which classes and aspects can be woven: static or dynamic.

In the rest of this chapter, Aspect-Oriented Programming (AOP) techniques are described and the different design constraints are presented. The static and dynamic weaving techniques are described in more detail. Then, we show examples that require a more flexible, *dynamic*, AOP system. Existing techniques for enabling dynamic weaving are discussed and their problems are specified.

2.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) offers a new set of language features to increase modularity and separation of concerns. The most popular implementation of AOP is AspectJ [6, 24], an extension of Java, figure 2.1. There are three main components for Aspect in AOP:

1. Joinpoint: A defined point in the program that is executed where behaviour will be added. These points may be in a class, methods, etc...
2. Pointcut: Group of joinpoints, at least one, based on a certain criteria.
3. Advice: Responsible for adding behaviours that will be executed when a point is captured in the program. An advice can capture more than one pointcut.

Advice component can catch code at three different moments:

- *Before*: The advice is executed before the joinpoint that trigger it.
- *After*: The advice is executed after the joinpoint.
- *Around*: The advice is executed around the joinpoint that trigger it. It has the same functionality as *Before* but allows cancellation or replacing of the class or the method execution.

Figure 2.1. An Aspect in AspectJ.

In Figure 2.1 a logging aspect in AspectJ is presented. The pointcut represents any call to a method in the *MyClass* class and subclasses. In AspectJ pointcuts can be named. In this example, it is associated with the name *LogPoints*. The description of the advice contains the keywords *after throwing*, which refines the exact moment when the advice should be executed. In this case, the advice will be executed after an exception was thrown inside a method, matching the definition of the pointcut. When this happens, the captured error is written to a log file.

2.2 Aspect-Oriented Design

In this section, different AO design constrains dealing with weaving aspects to the system will be presented. Incorrect aspect weaving order produces a woven model that does not meet the design goals and may cause security problem. The order of weaving and the methods of integrating the aspects with the running system are presented.

2.2.1 Types of Crosscutting Interfaces

There are three main crosscutting interfaces:

1. *Evaluation interface.*

Pinto et al. [26] proposed a technique that combines the benefits of both Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD). When a component sends a message, Dynamic Aspect-Oriented Platform (DAOP) interrupts it and evaluates the corresponding aspect. If all aspects are evaluated successfully, the message is received by the components. If an error occurs, the other aspects are not evaluated and the message does not reach the target component.

2. *Pointcut labels.*

Barais et al. [2] proposed Transform Software Architecture Technologies (TranSAT) as a framework for designing a software architecture based on AOSD principles. In TranSAT, the pointcut definition consists of hook identification for target architecture. The pointcut definition is generalized with pointcut label definition. The pointcut label serves as a reference to rules specification. It is associated with a pointcut mask that defines a set of structural, behavioral and architectural constraints on the software architecture model.

3. *Crosscutting interfaces.*

Kulesza et al. [23] developed an aspect-oriented generative approach for the domain of multi-agent systems. In the proposed approach, an aspectual component represents crosscutting features at the architectural level. It may crosscut other aspectual or normal components using its crosscutting interfaces. The *crosscutting interface* may add new state or behavior in the other components or modify the existence behavior of the components.

2.2.2 Implementation of Crosscutting

Stein et al. [39] recommended that the implementation of crosscutting takes place at the *link* that is used to communicate the message. Objects can interact only if there is a link between them (if the sender-object has reference to the receiver-object). Control is passed from one instance to another using communication links. Links represent main points in the dynamic execution of a system. They are instances of the association

relationships that connect two (or more) classifiers. As an example, two objects can interact using communication link to create, override or destroy an instance.

2.2.3 Composition Rules

Composition rules describe the sequence in which the aspects must be composed with other objects.

Seinturier et al. [34] and Stein et al. [38, 39] described a UML notation to design Aspect-Oriented Modeling (AOM) from the concepts defined in the AspectJ language. Both of them identify three types of composition:

- **Overlap:** The aspect is applied before or after the object.
- **Override:** The behaviour defined by the aspect will replace the behaviour of the object.
- **Wrap:** The behaviour of the object encapsulated by the behaviour of the aspect.

They extended UML to support AOM by defining the aspect as a class with special stereotype called `<<aspect>>`, as shown in figure 2.2. As the standard UML class, aspect serves as container and namespace for various features such as attributes and operations.

Seinturier et al. [34] represented the pointcut as a link between the aspect and different points in the program as shown in figure 2.2-b. The aspect-method is defined as an operation in the aspect class with special stereotype such as `<<before>>`, `<<after>>`, `<<around>>`, `<<replace>>` and `<<role>>`.

Stein et al. [38, 39] represented pointcut as an operation with special stereotype named `<<pointcut>>` in the aspect class; in figure 2.2-a. They defined also advice operation with special stereotype `<<advice>>`. *Advice* is responsible to define the

executable code whenever a pointcut of a particular set of joinpoints is reached. The control passes through each pointcut twice [19], first, when the communication is dispatched and the second, when it has been fulfilled. Before executing the communication, we must decide when the *Advice* must be woven (i.e., *Before*, *After*, or *Around*).

a. A pointcut as an operation. b. A pointcut as a link

Figure 2.2. Pointcut Representations.

Mussbacher et al. [25] extended the existing User Requirements Notation (URN) to describe Aspect-oriented User Requirements Notation (AoURN). They go beyond Seinturier et al. [34] and Stein et al. [38, 39] that allowed only standard rules: *before*, *after* and *around*.

The URN contains two complementary modeling languages for goals and scenarios, the Goal-oriented Requirements Language (GRL) and the Use Case Maps (UCMs) respectively. AoURN defines a joinpoint model for GRL and UCMs. Nodes of GRL and UCMs are joinpoints that can be matched by pointcut expressions. Pointcut expressions are defined on pointcut graphs and maps. The behaviour and structure of aspects are defined on advice graphs and maps which are loosely coupled to pointcut

graphs and maps. Flexible composition rules defined on advice graphs and maps are described with URN and not restricted by the capabilities of any particular pointcut language.

2.2.4 The Relationship between Aspect and Object

The pointcut links aspect-method to an object as shown in figure 2.3, or to a group of objects as in figure 2.4 [34].

Figure 2.3. One-to-one Relation- An Aspect per Object.

When an aspect is instantiated, a joinpoint or a set of joinpoints is instantiated between the aspect and the base object instance. Crosscut points between aspect and objects can be implemented as one-to-one, figure 2.3, or one-to-many, figure 2.4. Through the crosscut line, the aspect-method of the pointcut is applied to the base objects so that their behaviors are extended to implement the aspect's concern.

Figure 2.4. One-to-many Relation- An Aspect is Shared by a Group of Objects.

2.3 Static Weaving in AOP

Static weaving [14] means adding aspect-specific statements at joinpoints to the classes statically at compile time as shown in figure 2.5. It produces a well-formed system. The result is highly optimized woven code, whose execution speed is comparable to the code written without using AOP. However, a major disadvantage is that adding and removing aspect requires recompiling the entire system. Certain environments need the ability to change the global policies implemented through aspects during runtime. Static weaving makes it difficult to identify aspect-specific statements in woven code later.

Figure 2.5. Static Aspect Oriented Programming.

2.4 Dynamic weaving in AOP

Dynamic weaving [14] means that aspects can be applied or removed at any time during runtime. It allows the integration between components and the aspects at runtime. This results in a system which is more adaptable and extensible. In dynamic weaving, aspects can be added to the system on the fly. This can help avoiding the need to recompile and restart the system.

There are two levels of approaches in dynamic weaving [31], *class-level* and *object-level*.

Each of these approaches has a different benefit:

- A class-level dynamic weaving allows composing an aspect with a set of *classes*. Since it affects all present and future objects, it can dynamically change the behavior of whole application software globally and continuously. It is useful for an application that requires major changes in it.

- An object-level dynamic weaving allows composing an aspect with a particular set of *objects*. This kind of dynamic systems can change the behavior of a part of the software application locally and temporarily at runtime. It is useful for an application, in which various crosscutting concerns are switched to follow the specific requirements.

2.4.1 Examples of Dynamic Weaving

- Fixing bugs during software development: Changing an executing system using debugging during runtime without recompiling the system again. This functionality provided by class-level systems is useful for rapid prototyping in the development process. This feature can improve the efficiency of development.
- Measuring software performance: Dynamic weaving mechanism is recognized as a non-functional concern that can be modularized using AOP. The code for collecting information to measure performance is usually spread over the whole system. It should be modularized into an aspect.
- Adaptation of services: Services provided by a system will be adapted dynamically in response to changes in the environment and requirements of clients. For example, a security concern on which generality leads to easily broken points scatter around the system. An aspect including various levels of security could provide the system with dynamically security policies depending on the client. Both class-level and object-level dynamic weaving mechanisms are useful for modularizing these security policies for runtime system.

However, dynamic weaving can sometimes have drawbacks. Using dynamic weaving to replace aspects during runtime can be time consuming or not possible. For example, a load balancing aspect [4] could replace the previous woven load distribution strategy woven with a better one. The load balance replacement will depend on the current load of the managed servers.

2.4.2 Dynamic Weaving Techniques

Typical implementations of AOP systems, including both static and dynamic AOP, insert *hooks* at a number of joinpoints such as method calls. If the program execution reaches joinpoints identified by pointcuts, the inserted hook captures it and executes the advice code included in these set of joinpoints.

For dynamic weaving, all the joinpoints are dependent on the execution context, since the set of joinpoints is specified at runtime. Furthermore, as the set of joinpoints changes dynamically, the execution must continue after the joinpoint has been specified. This is to find out whether or not the system should execute the advice.

In this section, the two dynamic weaving techniques [31] are explained:

- Static code translation.
- Dynamic code translation.

2.4.2.1 Static Code Translation

Static code translation approach enables every joinpoint in a class or an object to be checked at runtime. It does not cause much performance penalty in advice execution;

however it involves some overhead in normal operations with no woven aspect. The execution of advice is fast since an inserted hook is represented as just a method call.

However, if no aspects are woven, all checks are performed whether or not the system executes an advice. These result in unnecessary method calls or verbose indirections of object references, which involve overhead in normal operations that cannot be ignored.

For example, *JAC* (Java Aspect Components) [35] supports the dynamic weaving mechanism using a static code translation approach. Using *JAC*, the compiler inserts minimal hooks for all the potential joinpoints. They weave the code of a program to a version with inserted hooks. Most AOP systems use static code weaver because most intercepted joinpoints are identified statically.

Prose [28] has implemented a Just-In-Time (JIT) aspect. *Prose* JIT compiler inserts hooks at all potential joinpoints only at the time of compilation. Thus, their work can be regarded as the static code translation approach mentioned above. They avoided adding options to the JIT compiler that could recompile byte-code since that would increase the complexity of the JIT compiler support.

2.4.2.2 Dynamic Code Translation

Most of the application servers and AOP environments provide a kind of dynamic weaving functionality. In the application server domain, hot deployment is used to plug or unplug software components of the JVM without restarting the application. There are two types of hot deployment [31]:

- Hot Deployment for Class-level Weaving.

To enable dynamic aspect weaving at the class-level, the class loaders can modify an application component by composing an aspect (class) with the application program during loading or compiling time.

- Hot Deployment for Object-level Weaving.

The class loader is used as a factory of the objects that are composed with an aspect.

2.5 Related Works

Researchers have suggested different aspect-oriented approaches for the problem of defining and weaving aspects to a system dynamically. In this section, we will briefly discuss related aspect-oriented approaches.

Sato et al. [31] proposed a Java-based dynamic AOP system called *Wool* which is a hybrid of two approaches. When a new aspect is woven in, *Wool* allows the programmers to select from two implementation techniques the best one for each joinpoint. The first one is to insert the hooks as breakpoints handled through the debugger interface of the Java virtual machine (JVM). The other one is to produce a program in which the hooks are embedded as method calls. Then, it reloads that new program into the JVM.

Wool is implemented as a Java library that provides dynamic AOP functionality, consisting of APIs to write aspects, a weaver to compose aspects with programs, and a subsystem for accepting a request for weaving from the outside of the running program. The weaver instance is connected to the subsystem of *Wool*.

Wool allows the aspect to be woven either locally, from within an application running on the same JVM, or remotely when sent to the subsystem of *Wool*. In the local woven case, the aspect instance is created in the running program. In the remotely woven case,

the aspect instance is created and recomposed outside of the JVM in which it will be woven. It is then serialized and sent over the network to the subsystem of *Wool* in the target JVM.

Wool supports only method calls, field accesses, object instantiation, and exception handlers. However it does not allow introductions which are used to crosscut the structure of the classes, since the “Hot Deployment” feature does not allow reloading a class file to which a new method or field is appended.

Seinturier et al. [35] developed Java Aspect Components (*JAC*), a dynamic weaver suitable for constrained environments such as application servers. The proposed approach checks authentication rules between the user and the application server. Unlike AspectJ, which provides a language to separate concerns at a syntactical level, *JAC* provides a set of properties to the distributed programs and adaptable containers. *JAC* provides a configuration interface that allows the programmer to adapt and integrate new aspects which are written in pure Java. Programmer defines a new aspect component by subclassing the *JAC* class. This *JAC* class provides default primitives to implement technical requirements that interface a set of functional objects in a Symantec way. It is the implementation unit that defines extra characteristics which crosscut a set of base-objects. *JAC* defines three main additional concepts:

- First, provide classification means for distributed entities expressing the fact that heterogeneous objects belong to the same context-space.
- Second, define pointcuts in another entity available at run-time which make aspects more reusable.

- Finally, aspect-classes are denoted by the pointcut relations in the program. They are implemented using the crosscutting points.

Unlike *Wool*, *JAC* changes the byte code of classes of an application when it is loaded into the JVM. *JAC* automatically distributes the aspects on distant servers. This allows *JAC*'s aspects to handle concerns like deployment, data consistency, fault tolerance, or scalability. Coupled with dynamic adaptability, *JAC*'s distribution capabilities let programmers build very flexible architectural schemes.

Prose [27, 28] is based on locating support for weaving and unweaving of aspects directly in the JVM. It is implemented by extending the VM's Just-In-Time (JIT) compiler. It is a JVM extension which can interrupt calls at run-time. *Prose* provides a new API within the JVM for weaving aspects at runtime called Java virtual machine aspect interface (JVMAI) which is designed as a plug-in to JVM. Thus applications must run with *Prose* specific JVM. The Just-In-Time compiler and the joinpoint API represent the AOP system (the AOP engine). The AOP engine can be treated as an exchangeable module to achieve the following:

- Run-time weaving must be secure: one should be able to control what local resources can or cannot be used by advice code originating from foreign hosts. When hooks are woven through the base code, the actual advice code is kept separated from the base application. This separation controls the access to system resources based on the origin of entire classes or archives.
- Atomic weaving: The weaving operation should appear to the application as one atomic step. Every time when a joinpoint is reached, a hook method is called and the

actual advice is executed. Weaving atomicity can be achieved by blocking advice execution in the hook method until the weaving operation completes.

- Advice weaver: The Just-In-Time (JIT) compiler weaves the actual advice code. Under normal operations (no woven aspects), the dynamic AOP system should not lead to significant performance degradation. Weaving advice code makes JIT necessary once an aspect is removed or exchanged.

-

Prose does not support crosscuts that add new objects to a given class in the original code (introductions), because its implementation cannot change the source-code or byte-code of the original systems.

Axon [1] is based on an interpreter extension approach and makes use of the debugging interface of JVM. It does not modify the source-code or the byte-code but uses the JVM for monitoring. Pointcuts are described in terms of events (joinpoints) and conditions, while advice is described as an action associated with events and conditions.

Axon is implemented as a plug-in to Sun's JVM. Events are generated by the JVM. The conditions are checked by *Axon* using run-time inspection with the JVM debugging interface. The actions are advices which are implemented using Java methods. The *around* advice is not feasible because it is based on interception at joinpoints. The advice can not replace the called method. *Axon* supports limited joinpoints such as method entry/exit, exception throw/catch and field read/write.

2.5.1 Support AOP features

Dynamic AO weaving approaches differ from each other in a number of ways.

Table 2.1 contains a comparison between the different approaches in terms of supporting AOP features.

	<i>Wool</i>	<i>JAC</i>	<i>Axon</i>	<i>Prose</i>
Extension approach	×	×	√	√
Around advice	×	√	×	√
Hand-coded	√	×	√	√
Change byte-code	√	√	×	×
Use debugger	√	×	√	×

Table 2.1. A Comparison between Different Dynamic AOP Approaches.

2.6 Problem with Ordinary Weaving Techniques

This section explains problems of ordinary techniques for implementing dynamic weaving in the AOP system.

Dynamic AOP in Java using *JAC* and *Prose* needs aspects to be implemented as Java classes which include their own weaving descriptions. We consider this a disadvantage since this class has to be recompiled whenever the weaving description changes. This means that first a compiler has to be available and the AOP system must reload the recompiled class if it wants to apply weaving changes at run-time.

Hooks in *JAC* are woven at all potential pointcuts. They check all hooks whether or not the system executes an advice. Another disadvantage in *Prose* is that the weaving description is not completely separated from the aspect's functionality. In our view this leads to a strong coupling between an aspect and particular software. In our case, *Weave History Technique* was designed to overcome these disadvantages.

Axon and *Wool* approaches are based on interception at joinpoints. *Axon* weave is described as sequences of API calls and not as classes. Weaving descriptions do not require recompilation, and changes can be made dynamically. However, using sequences of calls have their drawbacks. For example, AspectJ's *around* advice, which replaces the called method with advice functionality, is not easily implemented. When an approach, such as *Axon* is chosen, it will only be based on interception at joinpoints.

Wool allows the programmer manually selects between two techniques: First, insert the hooks as breakpoints handled through the debugger interface. The other one is to weave the program using hooks embedded in the system.

Applying these dynamic AOP techniques, programmers can weave or remove aspect from the running system without recompiling the system each time. This can be done using either a debugger interface or a compiled aspect available in the production library. During these manual steps, the running system will be under threat from any malicious activity. The running system cannot control the woven aspect. It cannot verify the availability and the location of the requested aspect.

2.7 Summary

This chapter illustrates practical benefits of dynamic weaving in AOP and currently known techniques for enabling them. We have presented two techniques, static code translation and dynamic code translation.

Through this chapter, we have seen that the class-level and the object-level dynamic weaving mechanisms are effectively used for different purposes. The class-level dynamic weaving mechanism is useful for a system that requires several runtime changes spreading among that system. The object-level dynamic weaving mechanism is used for a

system that contains various crosscutting concerns to handle the requirements of runtime environment.

Table 2.2 summarizes a comparison between the two techniques. Both of them allow implementing the static code and the dynamic code translation.

	Class-level dynamic weaving	Object-level dynamic weaving
Static code Translation	With regular execution and no woven aspect, there will be significant performance penalties.	A more serious performance penalty, due to the heavy weight hooks involving look-ups of appropriate objects.
Dynamic code translation	Class loaders are costly but clear approach.	Complicate method while adopting different software versions

Table 2.2. A Comparison between Class-level and Object-level Dynamic Weaving

In the following chapter, we propose new mechanisms for addressing efficient dynamic weaving mechanisms at both the class-level and the object-level. The mechanism allows the system to check if the object has been weaved before and weave only the modified object.

Chapter 3

Management of Aspect Weaving

This chapter presents the *Weave History Technique* which manages aspect weaving. The previous implementations of dynamic weaving techniques suffered from performance problems while composing classes or objects from the source library. The proposed *Weave History Technique* is an efficient class or object Java-based dynamic weaving technique addressing the performance problem mentioned in chapter 2. The technique has two approaches. First, the running system can request a class or a method from either the production or the source library. The proposed technique weaves the new (or the modified) class or object into a pointcut which is hooked in the running system. Secondly, hooks are inserted as breakpoints in the system dynamically.

Dynamic weaving, unlike static weaving in AOP, allows weaving and unweaving of aspect into/from a system. Advice body and joinpoints can be changed during runtime. These dynamic features extend the application domains of aspect-oriented programming. In dynamic weaving, aspects can adapt the behavior of the system to follow the changes and requirements of the runtime environment.

The *Weave History Technique* implementation is based on advice insertion. The running system requests an advice from the production or the source library. The requested advice will be composed and activated at the pointcuts.

The rest of this chapter is organized as follows. Section 3.1 presents our new implementation technique for class-level and object-level dynamic weaving. It also describes the *Weave History Technique* modules. In section 3.2, weave history methods are discussed to compare two objects. Section 3.3 shows the different types of weaving. Section 3.4 presents how to control the weaving. Section 3.5 presents how to identify joinpoints. We conclude this chapter in section 3.6.

3.1 The Weave History Technique

We developed a *Weave History Technique*, in which the system calls advice (class or method) from the production or the source library and hooks on demand, in Java. Since the advice could previously be inserted in the system, *Weave History Technique* inserts only the modified objects.

This section presents the details of our new dynamic AOP system *Weave History Technique* and shows how it enables efficient class-level and object-level dynamic weaving.

3.1.1 An Overview of the Weave History Technique

Weave History Technique is implemented in Java language because it provides dynamic weaving functionality. This weaving functionality consists of abstract aspect which contains generic reusable pointcuts, subaspects for accepting weave requests from the running system, a weaver to weave classes or methods from the production library, and a loader to load files from the source library to the production library.

Weave History Technique searches for the requested classes or methods to be woven either in the production library within running system, or in the source library. The following figure 3.1 shows the *Weave History Technique* architecture.

Figure 3.1. Weave History Technique Architecture.

Two methods handle the relationships between the *Weave History* and the production and the source libraries: *productionLib()* and *sourceLib()*.

The running system calls a class or a method from the production library using the method *productionLib()*. The method weaves the class or method dynamically into the system and runs immediately after the joinpoint is called.

If the requested class or method is not available in the production library, the method *sourceLib()* gets the file containing the requested class (or method) from the source library and compiles it. It loads the file to the production library where it will be woven dynamically to the system using method *productionLib()*. During the loading procedures, the dynamic weaving stops the system execution for a short time. After weaving the class (or method), the control continued and the class (or method) is hooked in the running system.

The strategy for deciding at the hooked joinpoint whether the advice is executed from a production library or called from a source library into the running system is simple. All the hooks are represented as breakpoints. At the hooked joinpoint, the running system sends a request which is equivalent to a call for a class or a method. There are two alternative ways: if the class or method exists in the production library, it will be weaved to the running system otherwise it will be called from source library.

A comparison between the implementation techniques of dynamic weaving mechanisms is shown in Table 3.1. *Weave History Technique* is a hybrid of the two techniques. The joinpoint could be defined as static or dynamic code translation depending on how frequently the advice will be executed.

	Static code Translation	Dynamic code Translation
Rarely executed advice	Not preferred (more overheads)	Preferred for small number of executed advice
Frequently executed advice	Preferred	Not preferred (more overheads)
Weave History Technique	Preferred in frequently executed advice	Preferred in rarely executed advice

Table 3.1. Weave History Technique is a Hybrid of the Two Techniques.

Dynamic code translation is not efficient under certain circumstances. Dynamic weaving causes only a single instant to hook into the running system. The cost of the translation and the insertion (or replacement) performed for every crosscut class or object, is relatively high if the advice is frequently executed (Column 2 in Table 3.1). In this case, dynamic code translation is not suitable. In order to reduce the overhead caused by frequently executed advices, the static code translation approach could be used as described in section 2.4.2.1.

3.1.2 Aspect in Weave History Technique

Using the *Weave History Technique*, an aspect can be instantiated in AspectJ program. Therefore, aspects can be composed and changed by the system dynamically which means that pointcuts and advices can be reconstructed while the target system runs. Furthermore, the system can also get a class or a method within a certain file from

the source library, compile and load it to the production library. The following program is a sample profiling aspect described in AspectJ within the *Weave History Technique*:

```
public aspect Session extends BaseAspect {
    public pointcut Login() :call (void EnglishSystem.Check_Login(int) );
    before (int value) : Login_Menu(value) {
        //
        // Joinpoint received a request “value” from the system
        System.out.println("----- Request :"+value);
        //
        // The advice body transfer the request to an equivalent
        // class or method name and gets its the location and status
        String searchResult= aspectSearch(fileName);
        //
        // If the aspect is available in the production library,
        // weave the class or the method into the system
        int weaveResult = productionLib(fileName, aspectName);
        //
        // If the aspect is not available in production library: Get the aspect
        // from the source library and load it into the production library
        int loadResult = sourceLib(fileName, aspectName, aspectStatus);
        •
        •
    }
}
```

In the above example, the subaspect *Session* is inherited from *BaseAspect*. It specifies the *before* joinpoint, *Login()* and the *before* advice body. The advice body calls a class or a method at the time an aspect is actually requested by the running system and weaves it using the method *productionLib()*.

The subaspect *Session* uses *before* joinpoint to check the user behavior. Depending on the user behavior, the subaspect *Session* will weave a new class or method into the running system using a set of methods. The method *aspectSearch()* searches in the *Aspect Register List* for the required class or method and weaves it into the running

system using the method *productionLib()*. If the required class or method is not available in the production library, the method *sourceLib()* gets it from the source library, compiles and loads it to the production library. Figure 3.2 describes the main activities for the advice body.

Figure 3.2. Activity Diagram for an Advice Body.

3.1.3 Abstract Aspect

The first step of the *Weave History Technique* is to create a *BaseAspect* object representing an abstract aspect and make it accessible from the running system. *BaseAspect* has no advice or internal declaration. It contains only generic reusable pointcut logic. The abstract aspect *BaseAspect* is inherited into subspects to reuse the declared pointcuts.

In the following example, abstract aspect *BaseAspect* contains logical pointcut definition *Login()*. This logical pointcut is defined in subaspect *Session*, inherited from *BaseAspect*. This feature is useful because pointcuts can be defined and reused throughout the running system without repeating the generic pointcut logic again. Using this technique, the aspect can be used by more than one system and can be loaded into libraries of reusable aspects.

```
public abstract aspect BaseAspect {  
  
    // abstract pointcut placeholder  
    public abstract pointcut Login();  
    // Calling advice whenever the joinpoint picked by Login  
    // Login is specified in a subaspect  
    pointcut Login_Menu(int value): Login() && !within(BaseAspect +)  
    && args(value);  
}
```

3.1.4 Advice

Advice in AOP consists of a pointcut and an advice body. In the *Weave History Technique*, the definition of the pointcut logic and the advice body are implemented in subaspect which is inherited from the abstract aspect.

3.1.4.1 Pointcut

The logical definition of the pointcut identifies a set of joinpoints in the subaspect inherited from the abstract aspect. Pointcuts can be hooked to any class or method in the system depending on their identifications. For example, in figure 3.3 the logical pointcut *Login_Menu()* in the abstract aspect *BaseAspect* identifies a joinpoint *Login()* in the subaspect *Session*.

Figure 3.3. Abstract Pointcut Defined in the Subaspect.

3.1.4.2 Advice Body

Advice body in the *Weave History Technique* is represented in subaspects which are inherited from the abstract aspect. This consists of four items: *Aspects Register List*, *aspectSearch()*, *productionLib()* and *sourceLib()*. These items can be called from any advice body.

The Aspects Register List. It is specified as follows:

- The *aspectName*: The method name.
- The *fileName*: The name of the file containing the aspect. It is the same class name.
- The *aspectVer*: It consists of two digits which represent the version of the last woven aspect. They are located in the last two digits in the file name.
- The *aspectStatus*: The location of the class (or the method)
 - The Production Library, the status equals *P*

- The Source Library, the status equals *S* for new aspect or *M* for modified version.

The method *aspectSearch()*. The *Weave History Technique* transfers the running system request into a parameter, *fileName*. The method *aspectSearch()* used this parameter to get the location and the status of the requested aspect from the *Aspects Register List*.

The method *productionLib()*. This method uses the parameters *fileName* and *aspectName* to weave the class or the method from the production library into the running system.

The method *sourceLib()*. This method uses the parameters *fileName*, *aspectName* and *aspectStatus* to get the file containing the required class or method from the source library. If the class had been modified, the method *sourceLib()* gets the modified part only. Finally, the method compiles and loads the class or method into the production library.

The definitions of the above advice body methods are independent of the running system. Therefore, the programmer can reconstruct the advice body methods without re-compiling the system.

Figure 3.4. Elements Interaction within the *Weave History Technique* Architecture.

Figure 3.4 shows how different elements interact within the *Weave History Technique*. When an event occurs in the running system, the system sends a request to a subaspect inherited from abstract aspect. The advice body in the subaspect will transfer the request into a call for a class or method. Using the *Aspects Register list*, the method *aspectSearch()* gets the aspect status, the method name and the name of the file containing the class or method.

If the requested class or method is available in the production library and has not been modified, the *Status* parameter will be equal to *P*. The method *productionLib()* will weave the class or method to the running system. Otherwise, the request will be passed to

the method *sourceLib()*. The method gets the requested new class (or method) or the modified parts from the source library and loads them into the production library where they will be woven to the running system using the method *productionLib()*.

In the next section, we will describe the flow process for the method *sourceLib()* and its activities.

3.1.5 The Method *sourceLib()*

The method *sourceLib()* process can be described using the UML Activity Diagram, figure 3.5. The input parameters for the method *sourceLib()* are the file name, the method name and the aspect Status. During the flow process, two activities are executed:

First, checks *the Status* for the requested aspect:

- The aspect *Status* is equal to *S*: The aspect has never been loaded to the production library. This means that the running system requests a new aspect from the source library.
- The aspect *Status* is equal to *M*: The aspect had previously been loaded to the production library and a new version is available in the source library. The method *getClass()* or the method *getMethod()* is called and produces an aspect contains only the modified parts.

Second, the method *compileFile()* compiles the new or modified aspect. The new or modified aspect is moved to the previously weaved library using the method *movePrevious()* and the compiled aspect is loaded to the production library using the method *moveProduction()*.

The name of the new or modified aspect is same aspect name in the source library plus two digits. The *Weave History Technique* uses these two digits as an aspect version number. The last two digits in the loaded aspect name are equal to the parameter *aspectVer* in the *Aspects Register List* plus one. Finally, the parameter *Status* will be recorded as *P* instead of *S* or *M* and the *Aspects Registration List* will be updated.

Figure 3.5. Handling the Source and the Previously Weaved Libraries.

3.2 Comparing Two Aspects

The method *sourceLib()* checks the parameter *aspectName* which contains the requested method name. If the parameter is equal to null, this means that the running system is requesting a class available the source library and the method *getClass()* is

called. Otherwise, the running system is requesting a method available in the source library and the method *getMethod()* is called.

The method *getClass()*, figure 3.6, searches for the differences between two classes and produces a file containing a class with the new or modified methods. The method has two inputs files: The source file and the previously weaved file. First, the method Checks if the required class is available in the both input files. Then, it compares each method within the source class with the equivalent method in the previously weaved class. If the method in the previously weaved class is different or not found, the source method is written in the output file. Finally, the results are written in a file that has the same name of the source file plus two digits for the version number.

Figure 3.6. The Method *getClass()* Compares Between Two Classes.

The method *getMethod()*, figure 3.7, searches for the differences between two methods and consequently produces a file containing a class with the modified method. The method has two inputs files: The source file and the previously weaved file. First, the method checks if the required method is available in the both input files then, it compares

the two methods. If the method in the previously weaved file is different, the method in the source file is written into the output file. Finally, the results are written in a file which has the same name of the source file plus two digits for the version number.

Figure 3.7. The Method *getMethod()* Confirms that the Requested Method is Modified.

3.3 Controlling the Weaver

The *Weave History Technique* provides functions to control the behavior of a weaver. The system passes a request for an aspect to the *Weave History Technique*. The

technique can verify the availability and the location of the aspect. If the aspect is available in the production library, it composes the aspect to the running system. Otherwise, the technique calls the aspect from the source library. If the aspect is previously weaved, the technique weaves only the modified parts.

Weaved classes or methods can override the original classes or methods and the effect appears in the running system. These functions operate at the time when the system catches the joinpoint. The methods of inserting hooks to the system can also be selected as described in detail in the next section.

3.4 Identifying Joinpoints

There are two possibilities for joinpoints [14]:

- First: The joinpoints are known in advance. When the system starts running, the joinpoints are already hooked into the system and will be affected by the aspects. If the system is extended (for example, more classes are loaded), the joinpoints can not hook to these additional classes. Aspects will not have any effect on these additional loaded classes.
- Second: There is no advanced knowledge of the joinpoints. If the system is extended, the joinpoints can be hooked to the additional classes and aspects will affect the loaded classes. As an example, when different versions of security policies need to be applied to a running system. The system is extended and the joinpoints can be hooked to these additional classes. Aspects can affect the loaded classes.

In the proposed *Weave History Technique*, the joinpoints may not be known in advance. The joinpoints can be added to the running system or the loaded classes. The aspects will have effect on both the system and the additional loaded classes.

3.5 Summary

This chapter has presented a new dynamic aspect weaver called *Weave History Technique*, which makes it possible to implement, class-level and object-level, dynamic weaving mechanisms. It integrates aspect weaving from both the production library and the source library, which allows us to load or replace classes or methods on running system. When the running system requested an aspect, the *Weave History Technique* checks the availability and the location of the aspect and dynamically weaves it to the joinpoint. Furthermore, it checks if the class was previously weaved and weaves only the modified part.

In the following chapters, we present a case study and the implementation of the *Weave History Technique*. The pros and cons of our implementation will be highlighted and a comparison with the other related works will be discussed.

Chapter 4

Implementation of the Weave

History Technique

This chapter presents the *Weave History Technique* implementation by describing the details of weaving step-by-step in two situations. First in section 4.1 a case study for an auction system is described. Second 4.2, the version of the running system is checked and different versions of security policies are applied and implemented as aspects in the running system. Third in section 4.3, depending on the connection type, the weave history will replace the login menu of the running system. Then in section 4.4, we compare the *Weave History Technique* with related dynamic AOP approaches.

4.1 Case Study – Auction System

The case study used throughout this chapter is based on auction system [20]. It is found on various internet sites such as www.ebay.com.

4.1.1 Description

The auction system runs on a set of computers connected via internet. Users access the auction service from one of these computers. It allows the users to buy and sell

items on auctions. There are different types of auctions. The most famous types are: *English auctions*, *Dutch auctions*, *1st Price auctions*, and *2nd Price auctions*.

The case study is based on the English Auction which is the most known form of auctions. The goods for sale are put up for auction starting at a low price and the duration of the bid is advertised before starting, for example 10 days. Bidders place their bids until the auction is closed. The number of bidder is not known before starting the bid. Therefore, auction service must be dynamic. A new bidder must be able to join the auction at any time.

4.1.2 Using the English Auction Service

Consider an online English auction service as shown in figure 4.1, and a user wants to use the service. The following steps are executed:

At the user-side:

- The user starts the internet service through a remote application.

At the server-side:

- The auction service opens login form, user-interface (UI) service, which allows the user to enter the username and password.
- The auction service starts a session.
- Loop
 - The user replaced a bid on an item and continues to increase his bid.
- The user may win /lose bid (s).
- The total bid amounts will be deposited from the user account.
- The user closed the auction service.

- The auction service is disconnected

Figure 4.1. An Auction Service.

In the auction system implementation, the *Weave History Technique* hooks pointcuts in two situations:

The first situation is, *before* establishing the connection. The technique checks the system version, being used by the client. Depending on the version, a class or a method is weaved into the running system. The second situation is, *around* the login menu. Depending on the connection type, the technique replaces the login menu if it is required. In both situations, the technique checks the availability and the location of the required class or method.

In the following sections, we will present the two pointcut situations. The pointcuts are defined in subaspects and can be hooked to any class or method in the auction system

depending on their identifications. They can be added or removed without recompiling the running system as described in section 3.1.4.

4.2 Checking the Version of a Running System

Before the client establishes a connection to the auction system, the aspect will check which system version is being used by the client. Depending on the system version, different versions of security policies need to be applied. The aspect will apply a series of appropriate actions before the client continues using the auction service.

In figure 4.2, the declaration and the references of the pointcuts are placed in the abstract aspect *BaseAspect* while the definition of the pointcuts logics is placed in specialized subaspects *Session_v1* and *Session_v2*. Each subaspect is responsible to handle a certain version of the auction system. The subaspects will cover any modification or actions that are required before the client continues using the service.

Figure 4.2. Abstract Pointcuts Related to the Two Versions of Auction System.

4.2.1 Insert Hooks as Breakpoints

In order to apply a series of appropriate security policies before the client uses the system, the *Weave History Technique* sets breakpoint *before* the method *Check_Login()* by using the subaspect *Session_v1* or *Session_v2* included in the *auction* package. At the same time, an advice body represented as an object associated with the joinpoint is loaded through *Session_v1* or *Session_v2*.

4.2.2 Call the Advice

The two advices in figure 4.2 are used to check the version for two auction systems and an appropriate action is taken depending on the system version. When a connection is established to auction system and the execution reaches the joinpoint hooked *before* the method *Check_Login()*, the *Weave History Technique* intercepts the execution using either the logical pointcut *Login_Menu1* or the logical pointcut *Login_Menu1* in the aspect *BaseAspect* depending on the system version. These logical pointcuts dynamically identify either the joinpoint *Login1()* or the joinpoint *Login2()* in the subaspect *Session_v1* or *Session_v2* respectively.

At the same time, The advice body in each subaspect dynamically gives information about the joinpoint using *getSignature()* and executes the method *aspectSearch()* to search in the *Aspect Register List* for the availability and the location of the required method. The method *productionLib()* weaves the required method into the running system. Finally, the control flow is given back to the *before* joinpoint *Login_Menu1* or *Login_Menu2*.

4.3 Access Control According to the Connection Type

After the connection to the auction system has been established, and depending on the connection type, the method *Connection()* will display the proper login menu. For example, a client connected from a known system will use a login menu with username and password. While the client connected through the internet will use a login menu with more rules such as username, password and account number.

In figure 4.3, the declaration and the reference of the pointcut are placed in the abstract aspect *BaseAspect* while the definition of the pointcut logic is placed in a specialized subaspect *Connect*. The subaspect will cover any modification or actions that are required before the client continues using the service. It will replace the login menu depending on the connection type if required.

Figure 4.3. Replace Client Rules Depending on Connection Type.

4.3.1 Insert Hooks as Breakpoints

The *Weave History Technique* sets breakpoint to specific method *Connection()* by using the subaspect *Connect* included in the *auction* package that is implemented using

the abstract aspect *BaseAspect*. At the same time, an advice represented as an object associated with the joinpoint is loaded through the subaspect *Connect*.

4.3.2 Call the Advice

When the system execution reaches the joinpoint hooked *around* the method *Connection()*, it is intercepted by the *Weave History Technique*. The advice body *Connect_type*, in figure 4.3, dynamically gives information about the joinpoint using *getSignature()* and checks connection rules related to the running system before executing the method *Connectio()*. Depending on the request received from the system, the advice may decide to replace the running system login menu with another one with new rules and conditions. The method *aspectSearch()* will be called to search in the *Aspect Register List* for the availability and the location of the required method. If the required method is not available in the production library, the method *sourceLib()* loads it from the source library. Then, the method *productionLib()* weaves the required method into the running system and a new login menu will be displayed to the client. Finally, the advice will deliver directly an interrupted message to the running system and the control flow is given back to the *around* joinpoint.

4.4 Weave History and Other Dynamic AO Approaches

In this section, we discuss some AOP implementations related to our work, and compare them to the *Weave History Technique*. Most current AOP implementations are based on inserting hooks at all potential pointcuts or using debugger interface to insert hooks as pointcuts.

Two dynamic weaving approaches were presented in section 2.5. First, hooks consist of the all-breakpoints such as *JAC* [35] or they use debugger interface such as *Wool* [31]. Hooks in *JAC* are woven at all potential pointcuts. The *History Weaving Technique* avoids this by defining pointcuts in subaspects. Pointcuts are woven according to the running system requirement and not at the compile time.

Our work is similar to *Wool* [31] except that *Wool* uses the debugger interface to insert a hook as a breakpoint. It supports only object reload and does not allow class reloads. Sato et al. [31] reported that their approach had imposed heavy cost in execution time due to the debugger interface. In *Weave History Technique*, the pointcuts are defined in subaspects and can be added or removed without recompiling the running system as described in section 3.1.4.

Aspects in *JAC* [35] and *Prose* [28] are implemented as Java classes and the running system must reload and recompile the classes if it wants to apply weaving changes at run-time. The *Weave History Technique* was designed to overcome this disadvantage. Using *Weave History*, both classes and objects can be reloaded if requested by the running system. If aspect is previously weaved, only the modified part is recompiled and reloaded to the running system as described in section 3.1.5.

In the following chapter, we present the experimental results for the proposed aspect-oriented techniques. The pros and cons of our implementation will be highlighted.

Chapter 5

Weave History Technique

Experimental Results

We need to confirm that the *Weave History Technique* improves the performance of the system by only weaving the modified aspect. In order to do this, we will conduct measurement experiments based on dynamic weaving classes and methods to a running system. Section 5.1 presents some results of our experiments. Section 5.2 presents a test for the crosscutting behaviour implemented with aspects. We validate the fundamental of the *Weave History Technique* based on dynamic weaving of classes from the production and the source libraries. Section 5.3 reports the result of our technique and compares the Weave History with the other AOP implementation approaches.

We performed all the experiments on Microsoft Windows XP Professional using the following software:

- Java Eclipse SDK Release 3.2.0, Last revised June 8, 2006
- Eclipse AspectJ Development Tools Version: 1.4.0
- AspectJ version: 1.5.2
- J2SE Runtime Environment 1.5.0_06-b05

Our *Weave History Techniques* do not depend on features that are unique to Microsoft window and could be implemented in other systems that support Aspect-Oriented programming. Our implementation is simple and structured to allow easy transformation to other systems.

5.1 Two Kinds of Weaving

In order to demonstrate the differences between the two kinds of advice weaving, from the production library and the source library, we compared the performance of weaving class in a joinpoint from both libraries using *Weave History Technique*.

In these measurements, the joinpoint was hooked *before* the method called in the running system, and the advice was subaspect inherited from abstract aspect. These measurements involved 100 iterations. The results of these micro-measurements are shown in Table 5.1.

Operation Weaving a class to the system	Average elapsed time (ms)	Minimum (ms)	Maximum (ms)
The class is available in the production library	1,422	1,297	1,547
New class is available in the source library	32,297	31,000	33,594
Modified class is available in the source library	31,812	30,515	33,109

Table 5.1. The Average Elapsed Time for *Weave History Technique*.

Table 5.1 shows that weaving a class from production library is 22 times faster than loading from the source library and then weaving it. Loading a modified class from source library is 1.5% faster than loading a new class from source library. Although it is not very a very significant but improvement is there.

5.2 Testing the Aspect Codes

We present a technique for testing crosscutting behaviour implemented with aspects to build confidence in our *Weave History Technique*. Both the functionality and the specification (the advice and the pointcut) will be checked. We will highlight whether the test pattern verifies the crosscutting functionality, the specification, or both. We will write a test against the auction system. The system is not compiled with the aspect. We used the abstract aspect *BaseAspect* that had been presented in section 4.3 in our test. The abstract aspect *BaseAspect* defines *around* joinpoint in a specialized subaspect *Connect*. The subaspect *Connect* captures the value of the joinpoint and replaces the login menu with another one depending on the connection type.

Verifying the aspect

The presented testing process allows testing the woven source code and the crosscutting functionalities, implemented as aspects, in separate steps.

Verify the aspect specification.

The method *getSignature()* used in the advice body to get information about the joinpoint signature. The return information verifies that the aspect is crosscutting the right class or method in the system.

Verify the aspect functionality.

- The method *sourceLib()* uses the method *getClass()* and the method *getMethod()* to verify that there is a difference between the files in the source library and the previously weaved library.
- The method *aspectSearch()* used to check if the requested aspect exists in the different libraries. The method uses the *Aspects Register list* to check if the

requested aspect is available in the production or source library. If the aspect is available in the list, the method gets its status and the name of the file containing the aspect. If the requested aspect is available in the production library and has not been modified, the *Status* parameter will be equal to *P*. Otherwise the *Status* parameter could be equal to *S* for new aspect or *M* for modified aspect.

Validating the parameters passed to the Weave History Technique methods

We validate the parameters passed from the auction system to the Weave History methods. Using *Weave History Technique*, the login service and the authentication concern can be dynamically modified depending on the type of connection and the client behaviours. In figure 5.1, the original login menu for the client may need to be replaced with another one depending on the connection type.

Figure 5.1. Login Menu before Using Weave History Technique.

The connection type is translated to a request for a class (new login menu). The *Weave History Technique* checks the availability of the requested class using method *aspectSearch()*. If the class is available, the original login menu will be replaced using method *productionLib()*, figure 5.2. Finally, the advice body will return the new result to the auction system.

Figure 5.2. Override Original Login Menu Using *around* Joinpoint.

The test shows that the *Weave History Technique* supports a bi-directional transformation of rules between the aspect and the running system. The technique demonstrated the integration between loading an aspect from the source library and weaving it to the system from the production library.

5.3 Weave History Technique Measurements

In order to demonstrate the effectiveness of the proposed Weave History dynamic aspect translation, we compared the overhead of *Weave History Technique* dynamic code translation with static weaving and related AOP approaches: *Wool* [31], *JAC* and *Prose* [41]. We measured the execution time of a running system to request a class from production and source libraries. We provided a *before* advice codes and let them be woven into different public method bodies in the running system.

The woven advice codes are over 124 lines and called 100 times. For comparison, we measured the execution time of the system with the advice woven by varying the underlying system with the static weaving and dynamic code translation stated in Chapter 2 using *Weave History Technique*.

Figure 5.3 illustrates the results of the average elapsed time between dynamic translation of a new and modified class from production library and source library using *Weave History Technique*. At 100 iterations, translating a class from production library is

22 times faster than translating a class from source library. This is because the time required for compiling the class, loading it to the production library and finally weaving the class to the running system.

Figure 5.3. Elapsed Time (ms) for Translating New and Modified Class to the Running System Using Weave History.

Figure 5.4 and table 5.3 illustrate the differences in elapsed time (ms) between weaving a class statically and dynamically using the *Weave History Technique*.

These results show that the *Weave History Technique* is 42 - 50 times slower than static weaving for dynamic translation of a class as following:

- Translation from production library: The static weaving is 42 times faster than the dynamic code translation.
- Translation from source library: The static weaving is 49-50 times faster than the dynamic code translation.

This is because the Weave History dynamically translates the class or method when the running system reaches the hooked pointcut whereas static weaving does not, unless an aspect is explicitly required. In this case, the system must be recompiled again.

Figure 5.4. Elapsed Time (ms) for Static and Dynamic Weaving of New Classes.

Table 5.2 illustrates Weave History dynamic weaving compared to related AOP approaches [32, 41]: *Wool*, *JAC* and *Prose* based on the static weaving approach. The benchmark execution results show that the Weave History dynamic code translating is outperformed the other related AOP techniques. Weave History dynamic code translating is about 4.3-4.6% faster than *Wool*. This is because Weave History weaves classes and methods directly to the system and not using the debugger interface of the JVM as in *Wool*. We observe that *JAC* and *Prose* have a rather big overhead in comparison to other AOP approaches. Weave History is about 90-99% faster than *JAC* and *Prose*. *JAC* checks all hooks whether or not the system executes an advice. For high number of empty

hooks, this performs an overhead to the system cannot be ignored [14]. For *Prose*, this big overhead can probably be implied to the lack of an efficient implementation as it is unable to catch which aspects are applied on which specific joinpoints [41].

Weave History Operation	Average elapsed Time (ms)	Comparison (Percentage)			
		Static weaving	Wool	JAC 0.11	Prose 1.1.2
Dynamic code translation from production library	1,328	97.6 %	(4.3%)	(92.29%)	(99.86%)
Dynamic code translation for new class from source library	1,578	98.1 %	(4.6%)	(90.82%)	(99.84%)
Dynamic code translation for a modified class from source library	1,515	97.05%	(4.51%)	(91.19%)	(99.84%)

Table 5.2. Comparison between Weave History, and Static Weaving, *Wool*, *JAC*, *Prose*.

5.4 Summary

In this chapter a process to verify the proposed dynamic aspect weaver, *Weave History Technique* was presented. This process verifies that it is possible to efficiently implement the *Weave History Technique* in the class-level and the object-level. *Weave History Technique* was implemented in Java without modifying the existing runtime system. It integrates two techniques to weave classes or methods from both production and source libraries.

Our experiments showed that the Weave History dynamic code translation is faster than related AOP dynamic approaches: About 4.3-4.6% times faster than *Wool* and

90.82-99.84% faster than *JAC* and *Prose*. This is because the *Weave History Technique* avoids using debugger interface to insert hooks as joinpoints and does not execute empty hooks. This ratio needs to be improved in the future by reducing the *Weave History Technique* overhead.

The experiment shows the bi-directional transformation of rules between the aspect and the running system. It also shows that the *Weave History Technique* can dynamically weave classes and methods from both source and production libraries. Weaving class from production library is about 95.6% faster than source library. Moreover, the *Weave History Technique* checks if the class is previously weaved from the source library and weave only the modified part. Weaving the modified class from source library is 1.5% faster than weaving a new class. We note a moderate improvement.

Chapter 6

Conclusion

This thesis has presented dynamic AOP weaving techniques, focusing on efficiency and granularity. It takes into account the lessons learned from the past and present cases and realizes dynamic AOP with deep attention on the runtime costs. The techniques presented in this thesis enable efficient class-level and object-level dynamic weaving. Thus, it will propel AOP to different domains including telecommunication environment and database systems.

6.1 Contributions

The contributions by this thesis are summarized as following:

This thesis has proposed dynamic aspect weaving techniques from both the production (compiled) and source (un-compiled) libraries to the running system. Using *Weave History Technique*, the running system controls the selection of weaving aspect instead of the intervention of the programmer.

The technique defines the hooks in specialized subaspects and allows insertion of suitable hooks into the running system. This achieves a high degree of independence between the running system and the aspects and makes the aspects more reusable in different contexts. This technique can be used to extend the specification of the running system and it opens up the possibilities for reusable pointcuts library.

Furthermore, this thesis has proposed a technique to improve the performance of weaving aspects. The *Weave History Technique* checks if the aspect was previously woven to the running system and weaves only the modified part. Weaving only the modified part of the aspect reduces the overhead of the weaving mechanism compared to the weaving of the complete aspect.

6.2 Limitations

The limitations of this thesis are summarized as following:

Order of weaving

The dynamic weaving process starts after an event or a threat targets a running system. Suppose that the system is trying to compose multiple aspects in parallel, the order of the woven aspects will depend on certain scheduler. After multiple weaving processes, it will be difficult to compose aspects following universal conventions. This will produce a system that does not meet the design goals and may present a security problem. Therefore, dynamic AOP systems including *Weave History Technique* need to provide a monitor method to control the weaver of the target system.

Aspect Declaration

Aspects in AspectJ can declare objects such as fields, methods, and constructors. They can also implement new interfaces or extend a new class. This can be done from the advice code. *Weave History Technique* uses these declarations to compose aspects (or modified aspect) dynamically into the running system from source library. This causes performance penalties, since we are implementing this technique using AspectJ language that does not support this feature.

6.3 Future Directions

We are interested in extending this thesis in three directions as following:

Implementing Weave History on large scale environment

Dynamic aspect using Weave History has not been implemented yet on large scale environment. We did not assume that an enormous number of clients are co-exists at the same time. On the other hand, *Weave History Technique* should be implemented and tested to be sure that it has proper scalability in practice.

Applying the technique into the real world

The techniques proposed by this thesis need to be applied to production environment and tested in the real world. Weave History aspect weaver is implemented on the Eclipse JVM. Most of the VMs are equipped with various kinds of optimization mechanisms such as memory management. Our *Weave History Technique* must not forbid those optimizations. We should confirm it by exposing them to the real world such as the BEA's JRockit VM.

Optimizing the aspect weaving

In our first version of Weave History, the system requests an aspect from the libraries. This selection produces good results. However, the system can not figure out the best aspect to be called. In future, we will implement a better profiler to automatically select the most appropriate aspects.

Bibliography

- [1] Swen Ausmann, and Michael Haupt, “Axon – Dynamic AOP through Runtime Inspection and Monitoring”, ECOOP'03 First workshop on advancing the state-of-the-art in Run-time inspection (ASARTI), July 21-25, Darmstadt, Germany, 2003.
- [2] Olivier Barais, Laurence Duchien, and Anne-Francoise Le Meur, “A Framework to Specify Incremental Software Architecture Transformations”, Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications, Volume 00, p.62-69, ISBN:0-7695-2431-1, Norway, 2005.
- [3] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann, “Virtual Machine Support for Dynamic Join Points”, In International Conference on Aspect-Oriented Software Development, p.83–92, ISBN:1-58113-842-3, Lancaster, UK, 2004.
- [4] Kai Böllert, “On Weaving Aspects”, International Workshop on Aspect-Oriented Programming at ECOOP99, 1999.
- [5] Jonas Boner, Alexandre Vasseur, “AspectWerkz - Plain Java AOP”, Available at: <http://aspectwerkz.codehaus.org/>, 2002.
- [6] Adrian Colyer, Andy Clement, George Harly and Matthew Webster, “Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and Eclipse AspectJ Development Tools”, Addison-Wesley, 2005.
- [7] Bart De Win, Wouter Joosen, and Frank Piessens, “AOSD & Security: a practical assessment”, Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT03), p.1-6, February 2003.

- [8] Martin Deubler, Michael Meisinger, Sabine Rittmann, and Ingolf Krüger, "Modeling Crosscutting Services with UML Sequence Diagrams", Model Driven Engineering Languages and Systems", 8th International Conference, MoDELS 2005, p.522-536, ISBN 3-540-29010-9 , Montego Bay, Jamaica, October 2-7, 2005.
- [9] Edsger Wybe Dijkstra, "A Discipline of Programming", Prentice Hall (Sd), 1976.
- [10] Tzilla Elrad, Moderator Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher, "Discussing Aspects of AOP", Communication of the ACM, Vol. 44, No. 10, p.33- 38, October 2001.
- [11] Elke Franz and Christoph Pohl, "Towards a unified treatment of security and other non-functional properties", Proc. Workshop on AOSD Technology for Application-Level Security (AOSDSEC'04), p.11-16, Lancaster, UK, 2004.
- [12] Shu Gao, Yi Deng, Huiqun Yu, Xudong He, Konstantin Beznosov, and Kendra Cooper, "Applying aspect-orientation in designing security systems: A case study", Proceedings of International Conference of Software Engineering and Knowledge Engineering (SEKE'04), p.360-365, Banff, Alberta, Canada, 2004.
- [13] Geri Georg, Indrakshi Ray, and Robert France, "Using aspects to design a secure system", Proceedings of the 8th IEEE International Conference on Engineering Complex Computing Systems (ICECCS), ACM Press, p.117 , ISBN:0-7695-1757-9, Greenbelt, MD, December 2002.
- [14] Wasif Gilani and Olaf Spinczyk, "A Family of Dynamic Weavers", Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS Technical Report 04.01, Lancaster, UK, March 2004.

- [15] Robert Grimm and Brian N. Bershad, “Providing policy-neutral and transparent access control in extensible systems”, Technical Report UW-CSE-98-02-02, University of Washington, 1998.
- [16] Object Management Group (OMG), Unified Modeling Language (UML) specification, Version 2.0, 2004.
- [17] Erik Hillsdale and Gregor. Kiczales, “Tutorial slides: Aspect-oriented programming with AspectJ”, FSE 2001, September 2001.
- [18] W. L. Hürsch and C. Lopes, Videira, “Separation of concerns”, Technical Report NUCCS-95-03, Northeastern University, Boston, Mass, February 1995.
- [19] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, “An Overview of AspectJ”, In Proc. of ECOOP'01, LNCS 2072, p. 327- 252, Budapest, Hungary, June 2001.
- [20] Jörg Kienzle, Alfred Strohmeier and Alexander Romanovsky, “Auction System Design Using Open Multithreaded Transactions”, Proceedings of the Seventh IEEE International Workshop of Object-Oriented Real-Time Dependable Systems (WORDS 2002), p.95-104, ISSN:1530-1443, San Diego, CA, USA, January 7-9, 2002.
- [21] Khaled M. Khan and Jun Han. “Composing security-aware software”, IEEE Software, 19(1), p.34 –41, 2002.
- [22] Ivan Krechetov, Bedir Tekinerdogan, Alessandro Garcia, Christina Chavez, and Uirá Kulesza, “Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design”, 5th International Conference on Aspect-Oriented Software Development (AOSD'06), Bonn, Germany, March 20-24, 2006.

- [23] Uira Kulesza, Alessandro Garcia, and Carlos Lucena, "Towards a Method for the Development of Aspect-Oriented Generative Approaches", Workshop on Early Aspects, OOPSLA'04, Vancouver, Canada, November 2004.
- [24] Russ Miles, "AspectJ Cookbook", O'REILLY, 2005.
- [25] Gunter Mussbacher, Daniel Amyot, and Michael Weiss, "Visualizing Aspect-Oriented Requirements Scenarios with Use Case Maps", International Workshop on Requirements Engineering Visualization (REV 2006), p.4–4, Digital Object Identifier 10.1109/REV.2006.9, Minneapolis, St. Paul, USA, September 2006.
- [26] Monica Pinto, Lidia Fuentes, and Jose M. Troya, "A Dynamic Component and Aspect Platform", The Computer Journal, Vol. 48, No 4, p.401-420, 2005.
- [27] Andrei Popovici, Thomas Gross, and Gustavo Alonso, "Dynamic Weaving for Aspect Oriented Programming", In 1st international conference on Aspect-Oriented Software Development (AOSD), p.141–147, ISBN:1-58113-469-X, Enschede, The Netherlands, April 2002.
- [28] Andrei Popovici, Thomas Gross, and Gustavo Alonso, "Just-In-Time Aspects: Efficient Dynamic Weaving for Java", Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD), p.100-109, Boston, Massachusetts, 2003.
- [29] Awasi Rashid, "Aspect-Oriented Database Systems", ISBN-3-540-00948-5, Springer-Verlag, 2004.
- [30] Indrakshi Ray, Robert France, Na Li, and Geri Georg, "An aspect-based approach to modeling access control concerns", Information and Software Technology, 40(9), p.575-587, 2004.

- [31] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori, “A Selective, Just-In-Time Aspect Weaver”, *Generative Programming And Component Engineering*; Vol. 48, p.189-208, 2003.
- [32] Yoshiki Sato, “A Study of Dynamic Weaving for Aspect-Oriented Programming”, Ph.D. thesis, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, December 2005.
- [33] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer, and Gerti Kappel, “A Survey on Aspect-Oriented Modeling Approaches”, DRAFT 30.01.2006, funded by the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF), 2006.
- [34] Lionel Seinturier and Laurent Martelli, “A UML Notation for Aspect-Oriented Software Design”, In *Aspect-Oriented Modeling with UML Workshop at AOSD’02*, Enschede, The Netherlands, April 22-26, 2002.
- [35] Lionel Seinturier, Renaud Pawlak, Laurence Duchien, Fabrice Legond-Aubry, Gerard Florin and Laurent Martelli, “JAC: An Aspect-based Distributed Dynamic Framework”, *Software Practice and Experience*, Volume 34 , Issue 12, p.1119–1148, ISSN:0038-0644, October 2004.
- [36] Viren Shah and Frank Hill, “An aspect-oriented security framework: Lessons learned”, *Proc. Workshop on AOSD Technology for Application-Level Security (AOSDSEC’04)*, p.143-145, Lancaster, UK, 2004.
- [37] Geórgia Sousa, Sérgio Soares, Paulo Borba and Jaelson Castro, “Separation of crosscutting concerns from requirements to design: Adapting a use case driven

approach”, Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design. Workshop at AOSD’ 04, Lancaster, UK, March 22-26, 2004.

[38] Dominik Stein, Stefan Hanenberg, and Rainer Unland, “An UML-based Aspect-Oriented Design Notation for AspectJ”, To appear in Proc. of AOSD’02, ACM, p.106–112, ISBN: 1-58113-469-X, Enschede, The Netherlands, April 2002.

[39] Dominik Stein, Stefan Hanenberg, and Rainer Unland, “Designing aspect-oriented crosscutting in UML”, In Proceedings of Aspect-Oriented Modeling with UML, As part of the 1st International Conference on Aspect-Oriented Software Development, p.301, Enschede, The Netherlands, April 22-26, 2002.

[40] Klaas van den Berg, Jose Maria Conejero, and Ruzanna Chitchyan, “Aosd ontology 1.0 - public ontology of aspect-orientation. Technical Report”, AOSD-Europe-UT-01, AOSD-Europe, May 2005.

[41] Wim Vanderperren and Davy Suvée, “Optimizing JAsCo dynamic AOP through HotSwap and Jutta”, Proceedings of the 2004 Dynamic Aspects Workshop (DAW04), p.120-134, Lancaster, England, March 2004.

[42] Dianxiang Xu and Kendall Nygard, “A threat-driven approach to modeling and verifying secure software”, Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, p.342–346, Long Beach, CA, USA, 2005.

[43] Gefei Zhang, Hubert Baumeister, Nora Koch, and Alexander Knapp, “Aspect Oriented Modeling of Access Control in Web Applications”, In 6th Int. Workshop Aspect Oriented Modeling (AOM’05), Chicago, USA, 2005.

Appendix A

A Sample from the Weave History Program

Abstract Aspect

```
package auction;

public abstract aspect BaseAspect {

    // abstract pointcut placeholder
    public abstract pointcut Login();
    // Calling advice whenever the joinpoint picked by Login
    // Login is specified in a subaspect
    pointcut Login_Menu(int value): Login() && !within(BaseAspect +)
    && args(value);

}
```

Subaspect

```
package auction;

import org.aspectj.lang.*;
import java.io.*;
import java.lang.*;

public aspect Session extends BaseAspect {
    public pointcut Login() :call (void EnglishSystem.Check_Login(int) );
    before (int value) : Login_Menu(value) {
        //
        // Joinpoint received a request “value” from the system
        System.out.println("----- Request :"+value);
        //
        // The advice body transfer the request to an equivalent
        // class or method name and gets its the location and status
        String searchResult= aspectSearch(fileName);
        //
        // If the aspect is available in the production library,
        // weave the class or the method into the system
        int weaveResult = productionLib(fileName, aspectName);
        //
    }
}
```

```

// from the source library and load it into the production library
int loadResult = sourceLib(fileName, aspectName, aspectStatus);

    }

    // Search for an aspect
    //
    private String aspectSearch (String fileName) {

    }

    //      Weave method from the production library to the running system
    //
    private int productionLib (String fileName, String aspectName) {

    }

}

```

An Aspect in the Production Library

```

package auction;

public class WeaveFile {

    public static int New_method(){
        int Weave_result = 0;
        System.out.println("-----");
        System.out.println(" Weaved method ");
        System.out.println("-----");
        //System.out.getClass().getName();
        Weave_result = 2;
        return Weave_result;
    }

    public static void New_Login_Menu(){
        System.out.println("-----");
        System.out.println(" Weave a new login menu ");
        System.out.println("-----");
    }

}

```