Generative Aspect-oriented Component Adaptation

Yankui Feng

Submitted in partial fulfilment of
the requirements of Napier University
for the Degree of
Doctor of Philosophy

School of Computing

April 2008

Abstract

Due to the availability of components and the diversity of target applications, mismatches between pre-qualified existing components and the particular reuse context in applications are often inevitable and have been a major hurdle of component reusability and successful composition. Although component adaptation has acted as a key solution for eliminating these mismatches, existing practices are either only capable for adaptation at the interface level, or require too much intervention from software engineers. Another weakness of existing approaches is the lack of reuse of component adaptation knowledge.

Aspect Oriented Programming (AOP) is a new methodology that provides separation of crosscutting concerns by introducing a new unit of modularization - an Aspect that crosscuts other modules. In this way, all the associated complexity of the crosscutting concerns is isolated into the Aspects, hence the final system becomes easier to design, implement and maintain. The nature of AOP makes it particularly suitable for addressing non-functional mismatches with component-based systems. However, current AOP techniques are not powerful enough for efficient component adaptation due to the weaknesses they have, including the limited reusability of Aspects, platform specific Aspects, and naive weaving processes. Therefore, existing AOP technology needs to be expanded before it can be used for efficient component adaptation.

This thesis presents a highly automated approach to component adaptation through product line based Generative Aspect Oriented Component adaptation. In the approach, the adaptation knowledge is captured in Aspects and aims to be reusable in various adaptation circumstances.

Automatic generation of adaptation Aspects is developed as a key technology to improve the level of automation of the approach and the reusability of adaptation knowledge. This generation is realised by developing a two dimensional Aspect model, which incorporates the technologies of software product line and generative programming. The adaptability and automation of the approach is achieved in an Aspect-oriented component adaptation framework by generating and then applying the adaptation Aspects under a designed weaving process according to specific adaptation requirements. To expand the adaptation power of AOP, advanced Aspect weaving processes have been developed with the support of an enhanced aspect weaver. To promote the reusability of adaptation Aspects, an expandable repository of reusable adaptation Aspects has been developed based on the proposed two-dimensional Aspect model.

A prototype tool is built as a leverage of the approach and automates the adaptation process. Case studies have been done to illustrate and evaluate the approach, in terms of its capability of building highly reusable Aspects across various AOP platforms and providing advanced weaving process.

In summary, the proposed approach applies Generative Aspect Oriented Adaptation to targeted components to correct the mismatch problem so that the components can be integrated into a target application easily. The automation of the adaptation process, the deep level of the adaptation, and the reusability of adaptation knowledge are the advantages of the approach.

Acknowledgments

I would like to thank my supervisors Dr. Xiaodong Liu and Professor Jon Kerridge, my PhD panel chair Kathy Buckner for all their help, support, expertise and understanding throughout my period of PhD study. I would also like to thank all staff at the School of Computing at Napier University, especially the members at the Centre for Information and Software Systems group, for providing me valuable feedback and suggestions during my PhD study.

I am most indebted to my beloved wife, Lei, who was supportive and understanding during the period of my PhD study. Finally, great appreciation and thanks to my mum, dad and my parents-in-law. Although they are in China, they always encourage me on the phone and give me consistent spiritual support. I am proud of them and appreciate what they contribute to my life.

Publications from the PhD Work

Journal articles:

[1] Liu, X., Feng, Y., & Kerridge, J. "Generative Aspect-Oriented Component Adaptation", *IET Software*, Volume 2, No. 2, ISSN: 1751-8806, April 2008.

[2] Liu, X., Feng, Y., & Kerridge, J. "Automated Responsive Web Service Evolution through Generative Aspect-Oriented Component Adaptation". *International Journal of Computer Applications in Technology (IJCAT), Special Issue on: "Agent and Aspect-Oriented Software Development and Evolution"*, Volume 31, Issue 1/2, 2008.

[3] Feng, Y., Liu, X., & Kerridge, J. "An Aspect-Oriented Component-Based Approach to Seamless Web Service Composition", *System and Information Sciences Notes*, SIWN-The Systemics and Informatics World Network, 1(2), ISSN: 1753-2310 (Print), July 2007.

Conference papers:

[4] Feng, Y., Liu, X., and Kerridge, J., Smooth Quality Oriented Component Integration through Product Line Based Aspect-Oriented Component Adaptation. *Proceedings of International Conference on Software Engineering and Knowledge Engineering* (SEKE'2007), Boston, USA, pp. 71-76, July 9-11, 2007.

[5] Feng, Y., Liu, X., & Kerridge, J. "A product line based aspect-oriented generative unit testing approach to building quality components", *Proceedings of The IEEE International Workshop Quality-Oriented Reuse of Software* (IEEE QUORS'07), Beijing, China, July 24-26, 2007.

[6] Liu, X., Feng, Y., & Kerridge, J. "Achieving Dependable Component-Based Systems Through Generative Aspect Oriented Component

Adaptation", *Proceedings of 30th Annual International Computer Software and Applications Conference*, Chicago, pp. September 18-21, 2006.

[7] Feng, Y., Liu, X., & Kerridge, J. "Achieving Smooth Component Integration with Generative Aspects and Component Adaptation", *Proceedings of 9th International Conference on Software Reuse*, Torino, Italy, LNCS 4039, pp. 260-272, June 11-15, 2006.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Problem Statement

The idea underlying the paradigm of Component Based Software Development (CBSD) is to develop software systems not from scratch but by assembling pre-existing parts, as is common in other engineering disciplines. CBSD focuses on building large software systems by integrating previously-existing software components. By enhancing the flexibility and maintainability of systems, this approach can potentially be used to reduce software development costs, assemble systems rapidly, and reduce the maintenance burden associated with the support and upgrade of large systems. At the foundation of this approach is the assumption that certain parts of large software systems reappear with sufficient regularity that common parts should be written once, rather than many times, and that common systems should be assembled through reuse rather than rewritten again and again [1][132].

However, the reality is that CBSD has not been as widely adopted as it should be. From a technical perspective, the reason is largely due to the difficulty of locating suitable components in a library and adapting these components to meet the specific needs of the user. Ideally, previously-existing components can be assembled by simply plugging perfectly compatible components together to build component based systems [9]. In practice, due to the availability of components and the diversity of target applications, in many cases mismatches[1] between pre-qualified available components and the specific reuse context of particular applications are inevitable and have been a major hurdle for wider component reusability and

1. Component mismatch refers to the situation that the selected component does not work well for or is not suitable for the target application. Full details are discussed in section 2.1.4.1.

component composition. These mismatches may occur in a range of issues of system quality, e.g. dependability and safety, and degrade the quality of the target component-based system severely. Consequently, the component marketplace will only exist when software developers can adapt software components to work within the applications [55].

Therefore, component adaptation is recognized as an unavoidable, crucial task in CBSD and has been researched over years as a key solution to the above mismatch problem [9][55][90][104][154][155]. The possibility for application developers to easily adapt COTS (Commercial-Off-The-Shelf) components to work properly within their application is a must for the creation of component based systems [17]. Until now, however, due to the complex nature of the mismatch problem, available approaches still have their disadvantages:

- Some approaches are only capable for adaptation at simple levels such as wrappers [155]. In this type of adaptation, only interface level adaptation can be performed, e.g. changing the number of parameters for methods.
- Some other approaches [104][154] are inefficient to use as a result of lack of automation in their adaptation process, which limits the wide use of the approaches. Too much user intervention is required to provide the necessary information for adaptation.
- Current approaches such as [155] are not proficient in Quality-of-Service (QoS) related adaptation. While fulfilling typical functional adaptation requirements during the adaptation process, these approaches are not capable of improving the QoS of the target components.

To assure the quality of target component-based software, more efficient and automated adaptation mechanisms are still needed to eliminate the above mismatches.

Aspect Oriented Programming (AOP) is a new methodology that provides separation of crosscutting concerns[1] by introducing a new unit of modularization - an Aspect that crosscuts other modules [91][113][128][137][147]. In this way, all the associated complexity of the crosscutting concerns is isolated into the Aspects. It is asserted [13][127] [151] that the final system becomes easier to design, implement and maintain. The nature of AOP makes it particularly suitable for addressing non-functional issues with component-based systems. However, there are still some unaddressed issues associated with current AOP techniques, which limit the wide adoption of AOP in software development:

- The reusability of Aspects. Currently, the research communities focus on the implementation of different aspect-oriented programming languages. However, the reusability issues of Aspects have not been addressed properly, which restricts the wider use of AOP.
- Platform specific Aspects. All current AOP platforms are bound to specific programming languages, e.g. AspectJ for Java, Aspect C/C++ for C/C++, and aoPHP for PHP. For generic reuse, semantically equivalent Aspects may be needed in different heterogeneous systems which are implemented by various programming languages. As a result, these semantically equivalent Aspects have to be re-implemented again and again for application in different systems.
- Naive weaving process[2]. To use Aspects in a component based system, an advanced weaving process, e.g. to fulfil the complex adaptation requirements, complex flow control such as a switch structure, is needed. However, the weaving mechanisms in current AOP projects only support individual Aspect weaving, rather than put them into a flexible and advanced weaving process.

1. Crosscutting concerns are aspects of a program which affect (crosscut) other concerns, full details are provided in section 2.4.1.1.
2. Weaving in AOP means compiling the Aspects with the affected software modules, full details are provided in section 2.4.1.4.

1.2 Aim and Objectives of the Research

To solve the problems of component based systems and aspect-oriented programming mentioned in Section 1.1, a Generative Aspect-oriented component adaptatIoN (GAIN) approach is proposed to achieve high adaptability, and therefore high reusability of components with Aspect-oriented component adaptation technologies.

The objectives of the research are as follows:

- To define a framework to support generative aspect oriented component adaptation. The key elements are defined in the framework, e.g. Reusable Aspect model, advanced weaving process.
- To develop a novel component adaptation approach within the defined framework. Based on existing fields of software product lines, generative programming, component adaptation, and AOP techniques, to develop a concrete component adaptation approach to solve the problems associated with existing AOP and component adaptation approaches. The approach applies aspect-oriented generative adaptation to targeted components to correct the mismatch problem so that the components can be integrated into the target application easily. Automation, deep level adaptation[1] and reusable adaptation knowledge should be the benefits of the approach.
- To build a prototype tool to illustrate and scale up the approach. As the implementation of the approach, a prototype tool is developed to support the automation of component adaptation in the approach and to demonstrate its scalability.
- To do case studies to evaluate the approach. Thereby evaluating the usability and correctness of the approach; case studies in various programming languages and platforms are performed.

1. Deep level component adaptation refers to the adaptation that changes the component functionalities.

1.3 Contribution to Knowledge

The proposed approach applies aspect-oriented generative adaptation to targeted components to correct component mismatches so that the components can be integrated into the target application more easily. Automation and deep level adaptation, reuse of adaptation knowledge, QoS correction, and flexible adaptation process support are the benefits of the approach, which are achieved with the following key techniques in an aspect-oriented product line based component reuse framework:

- **Product line based reusable adaptation aspect model**. In the approach, the adaptation knowledge is captured in Aspects and aims to be reusable in various adaptation circumstances. To achieve product line based automatic generation of the adaptation Aspects, a two dimensional aspect model is developed.
- **Highly reusable and AOP platform independent adaptation Aspects**. With the support from the product line based adaptation Aspect model, the adaptation knowledge is reusable via adaptation Aspects. Currently, the approach supports four AOP platform independent adaptation Aspects, namely Logging, Authentication, Database connection pool, and Policy enforcement. These Aspects are mapped to specific AOP platforms by automatically generating platform-specific Aspects via selecting and applying corresponding Semantic Interpreters.
- **Aspect repository for adaptation Aspect reuse and automatic generation**. As an embodiment of the product line based reusable adaptation Aspect model, the Aspect repository was developed as a key element to structure the three layers of an Aspect, to realize reusable Aspects in different layers in the approach, and to provide a mechanism for incremental reuse of Aspects.
- **Advanced adaptation Aspect weaving process**. The enhanced Aspect weaver supports the advanced weaving processes, e.g.

sequence and switch structure in a weaving process. The advanced weaving processes may be also added into the Aspect repository for further reuse.

1.4 The Structure of the Thesis

The thesis is organized as follows:

Chapter 1 gives the introduction of the research, including the problem statement, the aim and objectives of the research, and the contributions to knowledge.

The literature review is presented in Chapter 2, which includes the current state of software reuse, component based systems, software product lines, aspect oriented programming, and model driven architecture.

Chapter 3 summarises the related research projects in component adaptation, AOP, and software product lines, and describes typical research projects in detail. In addition, these projects are critically analysed and a conclusion is drawn, which gives the motivation of the research.

Chapter 4 presents a formal description of the approach, which includes the product line aspect model, the approach framework, the process based component adaptation, and the Aspect repository.

Chapter 5 describes the multi-layered reusable Aspect structure and the Aspect repository.

Chapter 6 demonstrates the process based component adaptation in detail.

The prototype tool, including its architecture and implementation is described in chapter 7.

In chapter 8, three case studies are used to illustrate and evaluate the approach and the tool.

Chapter 9 presents the conclusions of the research and future work.

Chapter 2 Literature Review

This chapter conducts a broad survey of many techniques that have been found useful for developing reusable component based systems such as Component Based Software Development, Web Services, Software Reuse technique, Generative Reuse technique, Aspect Oriented Programming, and Software Product Lines. These techniques are the foundation of the development of the proposed approach.

2.1 Current State of Component Based System

2.1.1 Component Based Technology

In past decades, object-oriented software development has achieved great success in software development. However, it has not achieved extensive reuse since individual classes are too detailed, specific, and bound to application domains. As a result, Component Based Software Development (CBSD) emerged in the late 1990s as a reuse-based approach to software systems development [89][122][132].

CBSD focuses on building large scale software systems by integrating existing software components. By enhancing the flexibility and maintainability of systems, the CBSD approach can potentially be used to reduce software development costs, assemble systems rapidly, and reduce the maintenance burden associated with the support and upgrade of large systems [17][79][122][132][139]. Under the methodology of CBSD, both Commercial-Off The Shelf (COTS) [79] components and in-house components can be integrated to build a range of target applications,

including traditional systems and most modern applications such as web services in a service-oriented architecture [20][34][45][46][94][95].

CBSD is being increasingly adopted as a major approach to software engineering even though reusable components are not always available. Components are independent and their implementation details are hidden, hence components do not interfere with each other. Without affecting the rest of the system, components can be easily replaced or upgraded by others which provide additional functionalities reflecting new customers' requirements. Components communicate with each other by using well-defined interfaces. In addition, the cost of software development is reduced by adopting existing, mature components [132].

CBSD is concerned with the assembly of pre-existing software components into larger pieces of software. Underlying this process is the idea that software components are written in such a way that they provide functions common to many different systems. Borrowing ideas from hardware components, the goal of CBSD is to allow parts (components) of a software system to be replaced by newer, functionally equivalent components.

The idea is not new. Componentizing software had been suggested by McIlorys [110] as a way of tackling the software crisis, yet only in the last decade or so has the idea of component-based software development taken off. Nowadays there is an increasing market place for COTS components [79], embodying a "buy, not build" approach to software development. The promise of CBSD is a reduction in development costs: component systems are flexible and easy to maintain due to the intended plug-and-play nature of components.

Commercial off-the-shelf component is a term for software components that are ready-made and available for sale, lease, or license to the general public. They are often used as alternatives to in-house developments. The

use of COTS is being mandated across many business programs, as they may offer significant savings in costs and time. However, since COTS software specifications are written by external sources, end-users are sometimes worried about using of these products because they fear that future changes to the product will not be in their control.

Components can be categorised into two groups [132]:

- Specific components: they are the components bound to an application domain.
- General components: they are general purpose components such as user interface and database connection components.

The widely used component models include:

- Common Object Request Broker Architecture (CORBA) [72]. CORBA is a distributed object standard developed by the Object Management Group (OMG). CORBA is a mechanism for normalizing the method-call semantics between application objects that reside either in the same or remote address. CORBA uses an interface definition language (IDL) to specify the interfaces that objects will present to the outside world. CORBA then specifies a “mapping” from IDL to a specific implementation language like C++ or Java. Standard mappings exist for Ada, C, C++, Lisp, Smalltalk, Java, COBOL, PL/I and Python. There are also non-standard mappings for Perl, Visual Basic, Ruby, Erlang, and Tcl implemented by Object Request Brokers (ORBs) written for those languages.
- Common Object Model (COM), COM+ and Distributed Common Object Model (DCOM) [76]. Microsoft COM (Component Object Model) technology is used to support communications between components and dynamic object creation in any programming language that supports the technology. COM is used by developers to create reusable software components, build applications by linking

components together. For example COM OLE (Object Linking and Embedding) technology allows Word documents to dynamically link to data / diagrams in Excel spreadsheets. Microsoft also provides COM interfaces for many Windows application services such as Microsoft Active Directory (AD) and Microsoft Message Queuing (MSMQ).

"COM+ is the name of the COM-based services and technologies first released in Windows 2000. COM+ brought together the technology of COM components and the application host of Microsoft Transaction Server (MTS)."[76]. Difficult programming tasks such as resource pooling, and event publication and subscription are automatically handled by COM+.

DCOM[76] is a technology enabling software components distributed across several networked computers to communicate with each other. DCOM extends Microsoft's COM, and provides the communication substrate under Microsoft's COM+ application server infrastructure. It has been deprecated in favour of Microsoft .NET.

COM is expected to be replaced to at least some extent by the Microsoft .NET framework [77]. However, Microsoft claims that COM objects can still be used with all .NET languages.

- Microsoft .NET Framework [77]. "The .NET Framework is Microsoft's managed code programming model for building applications on Windows clients, servers, and mobile or embedded devices." The pre-coded solutions that form the framework's class library cover a large range of programming needs in areas including: user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications. The Microsoft .NET Framework is a software component that can be added to or is included with Microsoft Windows operating system. It provides pre-coded solutions to common program requirements, and manages the execution of programs written specifically for the framework. The .NET Framework is a key Microsoft offering, and is intended to be used by most new applications created for the

Windows platform. The functions of the class library are used by programmers who combine them with their own code to develop applications.

- JavaBeans / Enterprise JavaBeans (EJB) [63] are the components model developed by SUN. JavaBeans are classes written in the Java programming language conforming to a particular convention. They are used to encapsulate many objects into a single object (the bean), so that the bean can be passed around rather than the individual objects. The EJB specification is one of the several Java APIs on the Java Platform Enterprise Edition (J2EE). EJB is a server-side component that encapsulates the business logic of an application. The EJB specification intends to provide a standard way to implement the back-end business logic code typically found in enterprise applications (as opposed to front-end user-interface code). Such code was frequently found to solve the same problems, and it was found that solutions to these problems are often repeatedly re-implemented by software developers. Enterprise Java Beans were intended to handle such common concerns such as persistence, transactional integrity, and security in a standard way, leaving programmers free to concentrate on the business logic.
- Web services [81]: Web services are Web based applications that use open, XML-based standards and transport protocols to exchange data with clients. A Web service is described via WSDL (Web Service Description Language) and is capable of being accessed via standard network protocols such as but not limited to SOAP (Simple Object Access Protocol) over HTTP (Hyper Text Transfer Protocol). There are some simple mechanisms for interested parties to locate the service and locate its public interface. The most prominent directory of Web services is currently available via UDDI (Universal Description, Discovery, and Integration).

2.1.2 Component Qualification

Qualification is the process of discovering and determining the suitability of a component for use within the intended system [1]. Reusable components are normally identified by the characteristics of their interfaces. However, the interface does not provide a complete picture of the degree to which the component will fit the architecture design and requirements. The software engineer must use a process of discovery and analysis to select the most suitable components.

Selection is dependent on the condition that measures exist for comparing one component against another and evaluating the fitness of use of components. During this activity, the issues of trust and certification arise. The process of certification is two-fold [1]:

1. To establish facts about a component and to determine that the properties a component possesses are also conformant with its published specification; and
2. To establish trust in the validity of these facts, perhaps by having a trusted third-party organisation check the truth of this conformance and to provide a certificate to verify this.

The motivation for component certification is that there is a causal link between a component's certified properties and the properties of the final system. If enough information is known about the certified components selected for assembly then it may be possible to predict the properties of the final assembled system. For many of components in the marketplace prediction is difficult because of a lack of information about a component's capabilities and a lack of trust in this information.

2.1.3 Component Composition

Component composition is the process of integrating components to form a working system if reusable components are available. In component based

software development, most systems will be constructed by composing these reusable components together [132].

There are a number of types of component composition [132], for example:

- Sequential composition: this occurs when the components are executed in sequence.
- Hierarchical composition: this occurs when one component uses functionalities provided by another component.
- Additive composition: this occurs when the interfaces of several components are put together to create a new component. The interfaces of the composite component are created by integrating all of the interfaces of the related components, and removing duplicate operations if necessary.

All types of component composition may be used when creating a system. In component composition, 'glue code' is used to link components. For example, in sequential composition, the output of component C1 may become the input to component C2. Intermediate statements are needed to call component C1, get the result and then call component C2 with that result as a parameter [132].

2.1.4 Component Adaptation

2.1.4.1 Component Mismatch Problem

In CBSD, while integrating various existing components to build a system, side effects may occur. Component mismatch problems arise while those side effects clash. The typical mismatch problems include risks, dependability, safety, incompatibilities, inconsistencies, and functional unsuitability. These mismatches may degrade the quality of the target component-based system severely [38] [50].

For example, the incompatibility problem [10] [122] [146] [159] during component composition is likely caused by the interface of the component. In some cases, where components are developed independently for reuse, developers will often be faced with interface incompatibilities where the interfaces of the components that they wish to compose are not the same. The component incompatibility problem can be compared to the incompatible problems to electrical appliances. The plugs are standardized, and therefore can be used by different electrical appliances. However, the standardization of plugs is often limited within a country. The electrical appliances are usually not plug compatible in other countries (for composition). In this situation, adaptors are needed to bridge the different interfaces (for composition) [122].

In practice, due to the availability of components and the diversity of target applications, in many cases mismatches between pre-qualified components and the specific reuse context of particular applications are inevitable and have been a major hurdle for wider component reusability and component composition[50][103].

Many of the existing approaches classify component mismatch to syntactic, semantic, and pragmatic mismatches and connect them to various issues of the component like functionality, architecture, and quality [9][10][159].

However, if component mismatches do happen in a component-based system, software developers need to fix these mismatches by using various component adaptation techniques (introduced in section 2.1.7 and section 3.1) in different circumstances. For example, mismatches may be solved by writing an adaptor component that reconciles the interfaces of the components being reused. Usually, an adaptor component converts one interface to another. The precise form of the adaptor depends on the type of composition. In some cases, the adaptor simply takes a result from one

component and converts it into a form where it can be used as an input to another.

The mismatch can be determined by various methods. Firstly, the interface of the component should be checked against the required interface in the target system. If they have different interfaces, it means that some mismatches exist. Secondly, the documentation of the component can also be used as a reference to find potential mismatch problems because the original component may require specific conditions such as operating system, hardware/software environments to work properly. Last but not least, the verification process should be used to determine whether the properties a component possesses are conformant with its published specification.

2.1.4.2 Component Adaptation

Despite the success of component-based reuse, the mismatches between available pre-qualified components and the specific reuse context in individual applications continue to be a major factor hindering component composition and therefore reusability. From a technical perspective, the reason is largely due to the difficulty of locating suitable components in a library (retrieval) and adapting these components to meet the specific needs of the user. As how to locate a suitable component in a library is beyond the scope of the research during the study, it is discussed in the future work (Section 9.3.3). The research focused on component adaptation.

Many researchers [54][122][132][155][160] have identified that “as-is” reuse is very unlikely to occur and that in the majority of the cases, a reused component has to be adapted in some way to match the application’s requirements. The reason for this is that individual components are written to meet different requirements, each one making certain assumptions about the context in which it is deployed.

Therefore, components often must be adapted when used in a new system. The process of changing the component for use in a particular application is often referred to as component adaptation. The purpose of component adaptation is to ensure that mismatches among components are minimised. Component adaptation is recognized as an unavoidable, crucial task in CBSD and has been researched over the years as a key solution to the mismatch problem (section 2.1.4.1) [9][55][90][104][154][155].

Component adaptation is the sequence of steps performed whenever a software component is changed in order to comply with new requirements emerging from end users. Such changes can be performed at different stages during the software development life cycle. Therefore, component adaptation can be distinguished as requirement adaptation, design-time adaptation, and run-time adaptation [9][21]:

- Requirement adaptation is used to react to changes during requirements engineering, especially when new requirements are emerging in the application domain.
- Design-time adaptation is applied during architectural design whenever an analysis of the system architecture indicates a mismatch between two constituent components.
- Run-time adaptation takes place when the system offers different behaviour depending on the context the parts are running in.

2.1.4.3 Functional and Non-functional Requirements

In software engineering, a functional requirement defines a function of a software system or its sub systems. A function can be described as a set of inputs, the behaviour or the processing, and outputs. Non-functional requirements are requirements which specify criteria that can be used to judge the operation of a system, rather than specific behaviours. In general,

functional requirements define what a system is supposed to do whereas non-functional requirements define how a system is supposed to be [61].

2.1.5 Classification of Component Adaptation Techniques

Components must be adapted based on rules that ensure mismatches among components are minimized. Adapting a component can be achieved in several ways, but traditional techniques can be categorized into white-box, e.g. inheritance and copy-paste, grey-box, e.g. own extension language in components, and black-box, e.g. wrapping, depending on the accessibility of the internal structure of a component [122]:

However, in practice, the above classification is not absolutely appropriate. For example, wrapping a component may require more understanding of the component, rather than its interface specification.

2.1.5.1 White-box Adaptation

White-box adaptation techniques require the software engineer to adapt a reused component either by changing its internal specification or by overriding and excluding parts of the internal specification. An adapted component is a component together with the glue code necessary for the original component to plug into the component system.

The main disadvantage of the white-box adaptation approach is that the modification to source code requires additional testing and can result in serious maintenance and evolution concerns in the long term. A new component derived by modifications to an existing component must be regarded as a new component and thoroughly tested. Additionally, the new component requires separate maintenance [122].

2.1.5.2 Grey-box Adaptation

In Grey-box component adaptation, a component provides its own extension language or application programming interface (API). Therefore, the end-user can adapt the component by using the extension language or API, instead of changing the source code of the component.

However, using an extension language or API is not an easy job for end-users, which limits the use of grey-box adaptation.

2.1.5.3 Black-box Adaptation

In Black-box adaptation, only a binary executable form of the component is available and there is no extension language or API. Therefore, the component can be either reused as it is, or adapted at the interface level of the component. Software engineers need to have the knowledge about the interface of the component, rather than the internal specifications.

2.1.5.4 Simple Level Adaptation and Deep Level Adaptation

Simple level adaptation refers to the adaptation that does not change the component itself. Normally, simple level adaptation is performed at component interface level, for example, Wrapper (refer to section 2.1.7.3 for details) is a typical simple level adaptation.

Deep level adaptation refers to the adaptation that changes the component functionalities, such as adding a new service, removing or modifying an existing service of the component, or altering quality features of the component.

Black/white box and simple/deep level adaptation are the different views of component adaptation technologies. Black/white box focus on whether the source code of original component is available. Simple/deep level

adaptations focus on the effect of the adaptation e.g. whether the original component is changed. For example, deep level adaptation is not necessarily white box adaptation as some black box / grey box adaptations can adapt the component without knowing the internal details of the component.

2.1.5.5 Source Code Adaptation and Binary Code Adaptation

Source code adaptation is an adaptation technique using source code level analysis, and adaptation to modify the source code directly. Rather than modifying source code manually in normal software development, developers use an existing source code adaptation tool to adapt the source code automatically. For example, in some UML (Unified Modelling Language) modelling tools [75][83], when developers change the UML model in the graphic view, the source code will be changed accordingly and automatically.

However, there are some criticisms to source code adaptation. First of all, a typical criticism to source code adaptation is the effort to glue-code development [4][58]. “Although the cost of glue-code development in a component based system accounts for less than half of the total cost, the effort per line of glue-code is about three times the effort per line of application code” [4]. In addition, the glue code layer is often fragile [4][58], and can break if either one of the modules it is gluing together is changed. Therefore, it requires making sure that the glue layer is kept up to date with any changes in either module.

Secondly, as the original application is modified during source code adaptation, recovering the original application will become difficult, which introduces the common version control problems for software systems [122].

Binary level adaptation is a component adaptation technique that modifies binary level components without knowing the source code of those

components. It is appropriate to use binary level adaptation when the source code of the components is not available. Two types of binary level adaptation are recognized: static binary adaptation and dynamic binary adaptation. Static binary adaptation applies adaptation to components before loading components into the run-time environment while dynamic binary adaptation applies adaptation to components at run-time, for example, the Java Virtual Machine (JVM) is often modified to support additional adaptations in some Java based binary adaptation projects [90].

2.1.6 Requirements for Component Adaptation Techniques

The following shows the requirements (R1-R8) for component adaptation techniques that have been compiled from various papers [56][90][155]. These requirements can be used to evaluate component adaptation techniques. It may not be possible for an adaptation technique to fulfil all requirements because there are conflicts between these requirements such as black-box vs. deep level. Moreover, there is no clear indication which requirement has priority.

R1. Black-box

Ideally, the adaptation of a component and the component itself should be two separate entities. In other words, the adaptation mechanism requires no access to the internal details of the component, only the interface level of the component is accessed. Therefore, the developer adapting the component only needs to understand the interface to the component.

However, in practice, black-box adaptation is not always feasible for most types of adaptation because insufficient information about a component is available. Sometimes the internal information of a component is required because software developers have to understand the design details of the component prior to performing the adaptation.

R2. Transparent

Transparent can be understood as that both the end-user of the adapted component and the component itself should be unaware of the adaptation between them. In addition, the functionalities of the component that do not need to be adapted should be accessible directly without the help from the adaptation.

R3. Composable

The composable requirement deals with the inter-relationships between adapted components, or between adaptations themselves. A composable adaptation provides recombinant adapted components and adaptation themselves that can be selected and assembled in various combinations to satisfy specific user requirements.

The composable requirement has three relevant aspects [56]:

First, the adaptation process should be easily applied to the original component.

Second, the adapted component should be able to integrate with other components as it was without the adaptation.

Finally, the adaptation should be able to integrate with other adaptations, which means the adaptation can be applied to other adapted components.

R4. Reusable

Reusable means that the code performing an adaptation can be used again and again in other adaptation scenarios. The purpose of component adaptation is to use the existing components repeatedly in different situations. Therefore, it would be highly desirable if the adaptation process is

repeatable because the adaptation process may be required in similar adaptations.

R5. Configurable

The adaptation mechanism should be able to apply the same adaptation to a set of target components with different settings.

R6. Automatic

Ideally, while adapting components, the adaptation process should work without user intervention. With the tool support, developers should be released from the heavy-coded tasks as the adaptation knowledge should be saved in the adaptation framework, e.g. a repository.

R7. Deep level adaptation

Component adaptation mechanisms should be able to deal with different level of adaptations. For example, some adaptation techniques deal with interface level adaptation such as adding parameters to a method. On the other hand, some other adaptation mechanisms can change the functionalities of a component, such as adding a new method or modifying the behaviour of an existing method.

R8. Language independence

As all components are implemented in many different programming languages, the mechanism for adaptation should not depend on any language-specific feature. The benefit of language independence is that the same adaptation mechanism can be applied to different components.

2.1.7 Conventional Component Adaptation Techniques

When using a conventional object-oriented language, the software engineer has three component adaptation techniques that can be used to modify a reused component, i.e. copy-paste, inheritance and wrapping. In the

following sections, each technique is described and subsequently evaluated with respect to the identified requirements.

2.1.7.1 Copy-paste

When a previously used component has some similarity with a new component, the most straightforward way is to copy the code of that part of the component that is suitable to be reused in the new component under development. Software engineers usually need to make changes to the code copied from existing components to fit the requirements for the new component.

However, Copy-paste has many disadvantages. First of all, software engineers must fully understand the code to be copied. Otherwise, the code may be used inappropriately, which may introduce errors to the new component. Secondly, if the code to be copied has a bug, then after Copy-paste, the bug will spread over the new component. Last but not least, if the same code is copied and pasted everywhere in the new component, it is very difficult to maintain the code in the new component when the original code is changed. Therefore, Copy-paste is also called a “quick and dirty” approach for reuse.

According to the aforementioned requirements, the evaluation of the Copy-paste technique can be summarised as follows:

- **Black-box**: Since Copy-paste is a simple source code level operation, there is no adaptation code at all. Therefore, the black-box requirement is not fulfilled.
- **Transparent**: Since the reused code and the modification to it are mixed together to build a new component, the end-user is unaware of the change to source code. The transparency requirement is fulfilled.

- **Composable**: Due to the manual based code modification, composability of adaptation functionality with the reused component is very low. When software engineers want to compose several types of adaptation behaviour, they have to do it manually.
- **Reusable**: Since the adaptation code is mixed with the code of the reused component, there is no reuse of for the component and the adaptation behaviour at all, except through the same copy-paste behaviour.
- **Configurable**: Adapting a component through copy-paste does not represent the adaptation behaviour as a first class entity, thus no configurability is available.
- **Automatic**: Due to the manual based code modification, there is no automation during the whole process. The automation requirement is not fulfilled.
- **Deep level adaptation**: Since software developers can write any code they want to modify the previously existing source code, the deep level adaptation requirement is fulfilled.
- **Language independence**: As a source code level adaptation technique, Copy-paste is programming language specific.

2.1.7.2 Inheritance

In object-oriented programming languages, inheritance [65] is provided to support the reuse of components. Depending on the language model, all or part of internal aspects are available to the reusing component. For example, in Java, while using private, protected and public keywords, none, or part of, or all methods/attributes can be accessed in a sub class. The advantage of using Inheritance is that the reusable code only exists in the superclass. As a result, better maintainability can be achieved by using Inheritance.

However, as a typical white box adaptation technique, Inheritance inevitably has its disadvantages. For example, software developers have to fully

understand the details of the superclass when they want to define a subclass.

With respect to the requirements, the evaluation of Inheritance can be summarised as follows:

- **Black-box**: Whether inheritance is black-box, depends primarily on its implementation in the language model. For example, in Java / C++, if an attribute or method is declared as private in a superclass, the superclass is treated as a black box because the adaptation behaviour is separated from the original component.
- **Transparent**: Since the end-user of the subclass is unaware of the adaptation between superclass and subclass, the transparent requirement is fulfilled.
- **Composable**: Even the adaptation behaviour in inheritance is specified in the subclass, and therefore separated from the original component, it is still difficult to apply the same adaptation behaviour to different components or compose multiple adaptation behaviours together. Therefore, the composable requirement is not fulfilled.
- **Reusable**: Despite the fact that inheritance improves the reusability of original component, the adaptation behaviour itself is still not reusable because the adapted behaviour is bound to the specific requirements of the subclass. Therefore, the reusable requirement is not fulfilled.
- **Configurable**: Although the adaptation behaviour is represented as a subclass, inheritance provides no means to configure the specific part of the adaptation behaviour. As a result, the configurable requirement is not fulfilled.
- **Automatic**: As a manual based coding process, inheritance does not support automation.
- **Deep level adaptation**: Since software developers can write code to modify the super class, the deep level adaptation requirement is fulfilled.
- **Language independence**: As a source code level adaptation technique, inheritance is programming language specific.

2.1.7.3 Wrapping

Wrapping [122][155] is a typical black-box component adaptation technique. A wrapper is a container that wholly encapsulates a component and provides an interface that can either restrict or extend a component's functionality. For example, the wrapper may be used to adapt the interface of the component, forwarding all calls to the wrapper to appropriate corresponding methods of the component.

Wrapping components is a very common solution in solving interoperation problems [122]. For example, text-based components can be reused in a Graphical User Interface application after being wrapped. In this way, the direct modification of the component can be avoided.

The main disadvantages of wrappers [58] are:

- Excessive amount of adaptation code: if wrappers are applied frequently, then an excessive amount of adaptation code will be required.
- Performance: as all the calls to the component will be received by the wrapper and then forwarded to the component, more processing time is required, which result in performance overhead.

The evaluation of wrapping with respect to the requirements is the following:

- **Black-box**: Since only the interface of wrapped components is available to the wrapper, the wrapping technique is black-box. The wrapper has no way to access the internals of the original component.
- **Transparent**: As the wrapper completely encapsulates the adapted component, all messages from the clients are intercepted by wrapper first. Therefore, the transparent requirement is not fulfilled.
- **Composable**: Since a wrapper and its wrapped components can be wrapped by another wrapper, the composable requirement is fulfilled.

- **Reusable**: Since the design of wrapper highly depends on its wrapped components, a new wrapper is required when underlying components are changed. Therefore, the reusable requirement is not fulfilled.
- **Configurable**: As a hard coded technique, wrapper does not support flexible configurations while being changed. Developers have to write corresponding code to support the change to the wrapper.
- **Automatic**: Since software developers need to understand the interface of underlying component(s) and write the code manually, the automatic requirement is not fulfilled.
- **Deep level adaptation**: Without knowing the internal specification, only simple interface level adaptation can be performed by a wrapper. Therefore, the deep level adaptation requirement is not fulfilled.
- **Language independence**: Since developers need to write language specific wrappers to wrap different underlying components, the language independence requirement is not fulfilled. However, if the underlying components are published as web services and the wrapper works on existing web services, the wrapper can be language independent.

2.1.7.4 Evaluating Conventional Techniques

In Table 2.1, an overview of the conventional adaptation techniques including Copy-paste, Inheritance, and Wrapping is presented that indicates how well each technique fulfils the specified requirements.

Adaptation techniques	R1	R2	R3	R4	R5	R6	R7	R8
Copy-paste	-	+	-	-	-	-	+	-
Inheritance	-	+	-	-	-	-	+	-
Wrapping	+	-	+	-	-	-	-	+/-

R1: Black-box

R2: Transparent

R3: Composable

R4: Reusable

R5: Configurable

R6: Automatic

R7: Deep level adaptation

R8: Language independence

+: fulfilled

-: not fulfilled

+/-: fulfilled or not fulfilled depends on the adaptation circumstance, details can be found in section 2.1.7.3

Table 2.1 Evaluation of conventional component adaptation techniques

2.2 Current State of Web Services and Web Service Composition

2.2.1 Introduction

An XML web service is a self-described and self-contained software component developed for the integration of web based loosely-coupled distributed systems [19][20][34][95][126]. A web service is the end result of research into the problems with distributed applications based on binary protocols, like DCOM, CORBA and EJB [93][115] [157]. The fast adoption of web protocols was one of the factors that made XML based web services possible. An XML web service is based on the XML standard, as the name implies, but there are a number of other standard protocols, including Hyper-Text Transfer Protocol (HTTP) and Simple Object Access Protocol (SOAP) that are instrumental in making XML web services functional. In addition,

web services could be described by Web Service Description Language (WSDL).

Based on standard description languages and protocols, web services can be used as a common mechanism to wrap up enterprise software applications for integration beyond the enterprise boundary across heterogeneous platforms in a distributed environment [22][24].

XML web services provide a model for design and implementation in which an application is built up from a number of smaller web services which can inter-operate regardless of how they are implemented and where they are hosted. This style of application development relies on three essential ingredients [94][121][145]:

- A protocol (SOAP) to allow communication between heterogeneous components on heterogeneous computers in heterogeneous networks;
- A means of agreeing the interface (WSDL) between service providers (servers) and service users (clients);
- A means for service users to find the service provider(s) who can satisfy their requirements.

Based on XML web services, Service Oriented Architecture (SOA)[15] is the latest evolution in distributed computing. The key differences between SOA and Object Oriented Programming (OOP) are listed below [94][126]:

OOP	SOA
Invoke	Find-bind-use
Synchronous	Asynchronous
Stateful	Stateless

A SOA consists of three primary parts [109][121]:

- The service provider: provides web services.
- The service requester: consumes web services.

- The service agency: provides interfaces of different web services. The service requester finds appropriate web services from the service agency and then consumes web services from the service provider.

2.2.2 Web Service Composition

In SOA, business process languages are used to describe the workflow, e.g. Business Process Execution Language for Web Services (BPEL4WS) [73][97]. Developers focus on the composition of different web services.

Web service composition does not involve the physical integration of all web services: the basic web services that participate in the composition remain separated from the composite web service. The main goal of web service composition is to specify which operations need to be invoked, in what order, and how to handle exceptional situations [26][37].

Composition of web services can be analysed from two standpoints[18] [37][141]:

- Composition in the "part-of" sense (granularity), i.e. larger part encapsulates web services and exposes itself as a web service.
- Composition in the "sequencing" sense, i.e. definition of the invocation order of web services.

Web service composition can be broadly classified into three categories [34][84][109][120][126][130][131]: manual composition, semi-automated composition and automated composition.

In manual composition, users have to generate workflow scripts either graphically or through a text editor, which are then submitted to a workflow execution engine. For instance, Triana [108] provides a graphical user interface, allowing the user to select the service from a toolbox and drag and drop onto a canvas. Then, the composed graph can be executed over a peer-to-peer or Grid network. BPEL4WS [73] allows the user to compose

web services at XML level. The composed web services are then submitted to an underlying execution engine. However, these systems have several drawbacks [53][156]. Firstly, the discovery and selection of web services is impossible with the increasing number of services. Secondly, users have to get some low-level knowledge, e.g. in the case of BPEL4WS, users are expected to build a workflow at the XML level. Although Triana provides a graphical interface, it is not suitable for a large workflow. Third, if the service is unavailable, the execution will fail.

Semi-automated composition techniques [120][129] make semantic suggestions during service composition process. Users still have to select appropriate services from a shortlist and link them together. Although these systems solve some problems of manual composition frameworks, they are still un-scalable as the filtering process may provide too many services for a user to select from [53].

Automated composition techniques [11][23][53][109][120][126][140][144] use smart software with embedded artificial intelligence which automatically detects what users need, finds out appropriate web services on the Internet, composes them in the right order and execute users' requests.

Currently, web service composition is still a highly complex task, and it is already beyond human capability to deal with the whole process manually. The complexity, in general, comes from the following sources. First, the number of services available over the web has increased dramatically during recent years, and one can expect to have a huge web service repository to be searched. Second, web services can be created and updated on the fly, thus the composition system needs to detect the updating at runtime and the decision should be made based on the up to date information. Therefore, most researchers work in the realm of workflow composition or AI planning [120][132][134].

2.3 Current State of Software Reuse

Software reuse is the use of existing software, or software knowledge, to build new software [52][86][119]. Ad hoc reuse has been widely practiced from the earliest days of programming. Programmers have always reused sections of code, templates, functions, and procedures. However, as early as 1968, Douglas McIlroy[110] proposed that the software industry should be based on reusable components, software reuse is recognized as an area of study of software engineering.

A very common example of software reuse is the technique of using a software library. Many common operations, such as accessing network resources, manipulating database systems, designing graphical user interfaces, etc. are needed by many different software systems. Developers of new systems can use the code in a software library to perform these tasks, instead of “re-inventing the wheel”, by writing fully new code directly in a program to perform an operation.

Reuse-based software engineering is an approach to development that tries to maximise the reuse of existing software [132]. Although the benefits of reuse have been recognised for many years [110] , it is only in the past 10 years that there has been a gradual transition from traditional software development to reuse-based development. The change to reuse-based development has been in response to demands for lower software production and maintenance costs, faster delivery of systems and increased software quality [100]. More and more companies regard their software as a valuable asset and are promoting reuse to increase their return on software investments [132].

In practice, different requirements come from different contexts and hence software components are reusable if these components can be adapted to these contexts to conform to the different requirements [13]. Therefore,

adaptability is the key to reusability: software components are reusable only when these components can be adapted to various situations.

Object oriented technology offers more sophisticated adaptation mechanisms such as inheritance, object delegation, object composition, and object aggregation. However, these mechanisms only cover functional adaptability. The non-functional issues of the adaptability of an application are difficult or impossible to model and implement [132].

The software units that are reused may have different sizes [132]. For example:

- Application system reuse. The whole of an application system may be reused by customizing the application for different users or by developing application families that have the same architecture but are tailored for specific users.
- Component reuse. All components of an application may be reused. For example, GUI components developed as part of a word-processing system may be reused in a spreadsheet system.
- Object and function reuse. Software components that implement single functions, such as a database connection or a class, may be reused. This type of reuse, based on function libraries or class libraries, has been commonly used for the past 40 years. Many libraries of functions and classes for different types of application and platform are available. These can be easily used by invoking them with other application code.

Software systems and components are specific reusable entities, but sometimes it is expensive to modify them for a new system. A complementary type of reuse is concept reuse, which is more abstract and is designed to be configured and adapted for a range of circumstances.

Concept reuse can be embodied in approaches such as design patterns, configurable system products and program generators [132].

“An obvious advantage of software reuse is that overall development cost is reduced.” However, there are still some problems associated with reuse. For example, the cost of understanding whether a component is suitable for a particular reuse circumstance and testing that component is still high. The benefits of using reuse may decrease because of these additional costs [132] (p417).

2.3.1 Design Patterns

The pattern is a description of the problem and the essence of its solution, so that the pattern can be reused in different situations. Gamma et al. [49] define the four essential elements of design patterns:

- A name of the pattern.
- A description of the problem area that explains when the pattern may be used.
- A solution description, which is the template of a concrete solution. In other words, the solution description can be instantiated in different ways.
- A statement of the consequences – the results of applying the pattern.

The design patterns can be divided into three types [32][49]:

- **Creational patterns**: create objects for you, so you do not have to instantiate objects directly. Your program gains more flexibility in deciding which objects need to be created for a given case. For example, the factory pattern, the factory method pattern, the abstract factory pattern, and the singleton pattern.
- **Structural patterns**: help you to organize groups of objects, such as a complex user interface. For example, the facade pattern.

- **Behavioural patterns**: help you to define the communication between objects in your system and how the flow is controlled in a complex system. For example, the Observer pattern.

2.3.2 Generative Reuse

Generative reuse is a black-box reuse technique. In generator based reuse, domain knowledge and relevant system building knowledge is embedded into a domain specific application generator [7][12][47][122][132]. In such a system, the input for a program generator is the application specification, which provides the parameters to the generator. With these parameters and domain knowledge, the generator translates the specification into code for the new system in a selected language. The generation process may be automated, or may require manual intervention.

Generator based reuse has been particularly successful for business application systems. These generators may generate complete applications or part of the applications. The generator based approach is also used in other areas [29][51][123][132], for example:

- Parser generators for language processing. The input to the generator is a grammar describing the language to be parsed, and the output is a language parser. For example, JavaCC (Java Compiler Compiler) [67] is an open source parser generator for the Java programming language.
- Code generators in CASE tools. The input to code generators is the software design and the output is an implemented system. For example, in UML CASE tools, e.g. IBM Rational Rose [83] and MagicDraw [75], based on UML models, CASE tool generates either a complete program or a code skeleton. The software developer then adds detail to complete the code.

Generator-based reuse is a cost-effective approach for application development. It is much easier for end-users to develop programs using generators compared to other component-based approaches [132].

2.3.3 Application Frameworks

In the early stages of Object Oriented Programming (OOP), objects were regarded as the most appropriate abstraction for reuse. However, experience has shown that objects are often too fine-grain and too specialised to a particular requirement. The larger-grain abstractions called Frameworks provide better solution for object-oriented reuse [132].

An Application Framework is a system built by a collection of various classes and interfaces between them [158]. Applications are often constructed by integrating a number of different Frameworks with various functionalities.

There are three classes of Framework [40][114]:

- System infrastructure Frameworks. These Frameworks are used to develop the essential system infrastructures such as communications, user interfaces and compilers.
- Middleware integration Frameworks. These Frameworks are used to support component communications. These Frameworks include CORBA, Microsoft's COM+, and Enterprise Java Beans.
- Enterprise application Frameworks. These Frameworks focus on specific application domains such as global travel information, telecommunications or financial systems. These Frameworks encapsulate application domain knowledge as standard APIs and support the development of end-user applications.

Applications developed by Frameworks have the great potential for further reuse through software product line technologies. Consequently, the

maintenance of these systems such as modifying family members to create new family members is simplified [132].

2.3.4 COTS Product Reuse

A commercial-off-the-shelf (COTS) [2][3][16][79][132] product is a software system that can be used directly by its buyer without any modifications. Typical desktop software and a wide variety of server products are COTS software. As COTS software is developed for general purpose, such as word-processing, database management, etc., it usually has many features that can be reused in many different applications. Although there can be problems with this approach to system construction [143], COTS is widely used across government and enterprises because they offer significant savings, in terms of costs and development time.

Some types of COTS components have been very popular for many years, such as database management systems and GUI components. Very few developers want to implement their own database system. However, until the mid-1990s, integrating these large systems and making them work together was a big challenge because most large systems were designed as standalone systems [118][132][142].

At present, well-defined Application Programming Interfaces (APIs) that allow program access to system functions is always available in COTS systems. Consequently, constructing a large system by integrating a range of COTS systems is a popular approach. This way, the costs of development and delivery times are reduced. Furthermore, risk may be reduced as the mature COTS systems are already available.

2.3.5 Component Based Software Reuse

There are various component characteristics that lead to reusability [122][132]:

- The component should reflect stable domain abstractions. Stable domain abstractions are fundamental concepts in the application domain that change slowly.
- The component should hide the way in which its state is represented and should provide operations that allow the state to be accessed and updated.
- The component should be as independent as possible. Ideally, a component should be stand-alone so that it does not need any other components to operate. In practice, this is only possible for very simple components, and more complex components will inevitably have some dependencies on other components.
- All exceptions should be part of the component interface. Components should not handle exceptions themselves as different applications will have different requirements for exception handling. Rather, the component should define what exceptions can arise and should publish these as part of the interface.

As more and more systems are built from existing components, it is important to identify the major challenges of component based software reuse. Sommerville[132] proposes three critical requirements for software design and development with reuse:

- It must be possible to find appropriate reusable components.
- The reuser of the components must be confident that the components will behave as specified and will be reliable.
- The components must have associated documentation to help the reuser understand them and adapt them to a new application.

2.4 Current state of Aspect Oriented Programming (AOP)

2.4.1 Introduction to AOP

2.4.1.1 Crosscutting Concerns

The separation of concerns is an important principle of software design and implementation [132]. The basic idea of this principle is that each element (class, method, procedure, etc.) should do one thing and one thing only. Separation of concerns breaks down a program into distinct parts that overlap in functionality as little as possible. Consequently, developers can focus on each element without knowing other elements in the system. When developers need to modify their system later, they are only required to understand and modify a small number of elements. All programming methodologies such as procedural programming and object-oriented programming support some separation and encapsulation of concerns into single programming elements. For example, procedures in procedural programming, packages, interfaces, classes, and methods in OOP all help programmers encapsulate concerns into single elements.

However, as shown in Figure 2.1, in many situations, some concerns such as logging, performance optimization, and policy enforcement defy these forms of encapsulation. Software developers call these crosscutting concerns [91][98] [127][132], because they cut across many modules in a program. For example, if software developers want to keep track of the usage of each system module by each system user; they have a logging concern that has to be associated with all components. The specific logging that is carried out needs context information from the system function that is being monitored. Therefore, the related code spreads all over the system, which makes the system difficult to maintain and upgrade since the addition of new crosscutting features and even certain modifications to the existing crosscutting functionality require modifying the relevant core modules.

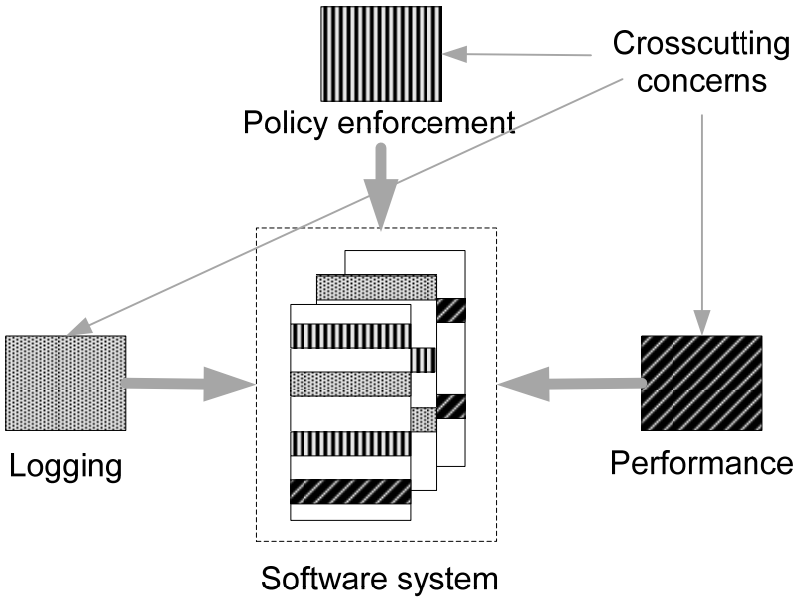


Figure 2.1 Crosscutting concerns

As shown in Figure 2.2, “Code tangling is caused when a module is implemented that handles multiple concerns simultaneously” [98](p15). Code tangling is the result of the implementation of crosscutting concerns in a traditional OOP system. Cross-cutting concerns must be implemented by modifying many methods in many classes. This approach prevents modularization and is error-prone when requirements affecting crosscutting concerns are changed or added. Software developers often need to consider crosscutting concerns such as Logging, Performance, Policy enforcement, etc in each component across the system. As a result, code tangling is inevitable.

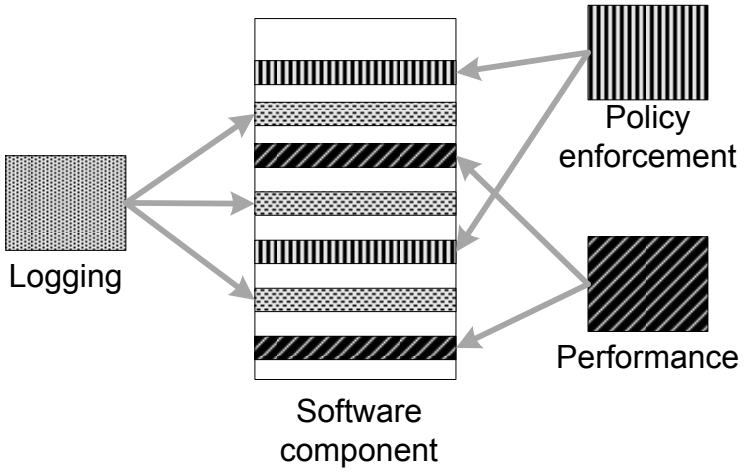


Figure 2.2 Code tangling

“Code scattering is caused when a single issue is implemented in multiple modules” [98](p16). As crosscutting concerns are spread over different modules, the implementation of those concerns are also scattered over all those modules. For example, as shown in Figure 2.3, in a software system, Logging concerns may affect all the modules accessing the Logging module.

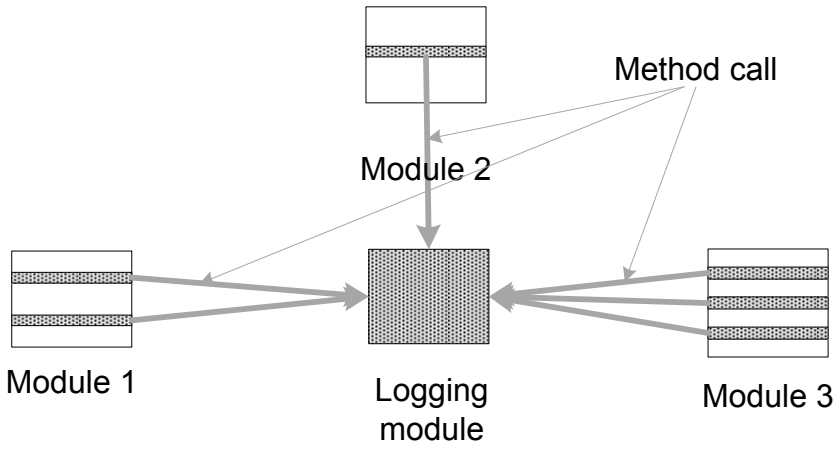


Figure 2.3 Code scattering

Figure 2.3 shows how a software system would implement Logging using traditional approaches. Each client in module1, module 2, and module 3 needs the code to call the related Logging method. Consequently, the effect is an undesired code scattering between all the modules needing Logging and the Logging module itself.

2.4.1.2 Introduction to AOP

“Aspect Oriented Programming (AOP) is a methodology that provides separation of crosscutting concerns by introducing a new unit of modularization - an Aspect that crosscuts other modules.”[98](p4). This is a “divide and conquer” strategy. As shown in Figure 2.4, developers implement crosscutting concerns in Aspects (e.g. Logging, Transaction and Persistence) instead of putting them into core modules. AOP allows crosscutting concerns to be developed independently. In this way, all the

associated complexity of the crosscutting concerns is isolated into the Aspects [98][112][127], hence the final system becomes easier to design, implement and maintain. Software developers do not need to think about the crosscutting concerns throughout the whole development process – what they need to think about are these crosscutting concerns only at design stage.

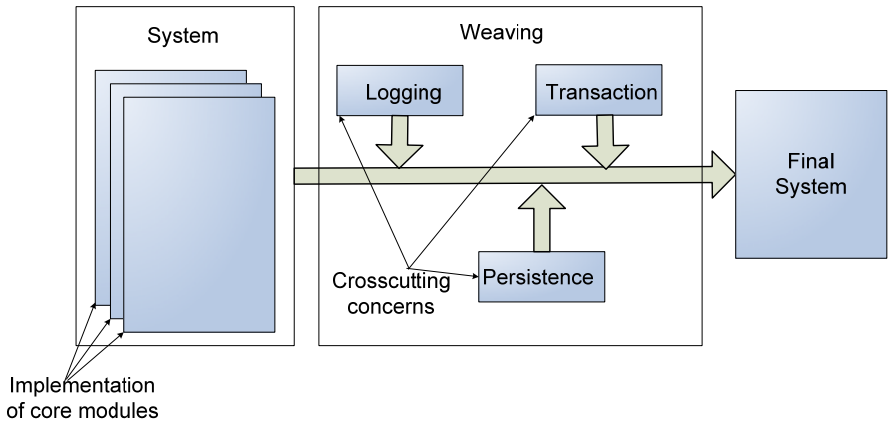


Figure 2.4 Aspect Oriented Programming

“An aspect weaver, which is a compiler-like entity, composes the final system by combining the core and crosscutting modules through a process called weaving.” [98](p4).The Aspect Weaver will weave these concerns into the system automatically and in the maintenance stage, software developers only need to update these concerns and re-weave them into the system. Therefore, crosscutting concerns are logically separated from the system in the design stage.

AOP builds on top of existing methodologies such as Object Oriented Programming (OOP) and procedural programming, equipping them with new concepts such as Aspect, Advice, etc. in order to modularize crosscutting concerns. With AOP, the core concerns are implemented by using the chosen base methodology. For example, in OOP, developers implement core concerns as classes. All crosscutting concerns can be put into Aspects, which support how the different modules in the system need to be woven together to form the final system [98]. AOP is the appropriate technology for

addressing non-functional issues that could be resolved by introducing extra process, operations and resources, e.g., Monitoring, Policy enforcement, Persistence, Optimization, Authentication, Authorization, Transaction Management, and implementing business rules. However, AOP can not be used to address all non-functional issues during software development process, e.g., platform compatibility, documentation, budget, and deadlines.

The key benefit of AOP is that it addresses the problem associated with crosscutting concerns in an elegant way by supporting the separation of concerns. Separating concerns into other elements rather than including different concerns in the core business modules is good software engineering practice [132]. By describing cross-cutting concerns as Aspects, these concerns can be used and reused independently. For example, if Logging is described as an Aspect that logs necessary information to a file system. This Aspect can be automatically woven into the system wherever Logging is required. In addition, as shown in Figure 2.5, with the help from AOP, software evolution and maintenance become easier [28][116].

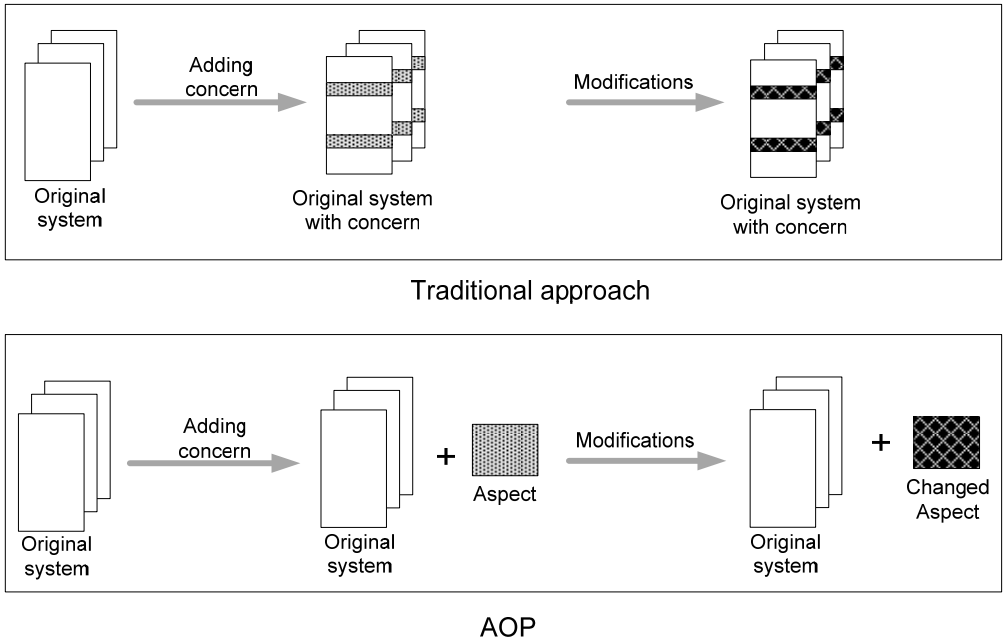


Figure 2.5 The difference between the conventional approach and AOP in software development

Aspect Oriented Software Development (AOSD) is rapidly developing as an important, new software engineering technique [39][91][151]. Research and development in AOSD has primarily focused on aspect oriented programming [91]. Aspect-oriented programming languages such as AspectJ [60] have been developed to extend object-oriented programming in supporting separation of concerns. Some companies, such as IBM, are starting to use aspect-oriented programming in their software production processes [30]. However, it has now been recognised that cross-cutting concerns are equally problematic at other stages of the software development process. Researchers [112][116][152] are now investigating how to utilise aspect-orientation in system requirements engineering and system design and how to test and verify aspect-oriented programs.

2.4.1.3 Terminologies in AOP

In AOP, as shown in Figure 2.6, to address the problem associated with crosscutting concerns, the basic idea is to encapsulate these crosscutting concerns in an Aspect. An important characteristic of Aspects is that they include a definition of where they should be included in a program, as well as the code implementation of the cross-cutting concerns. Thus, developers can specify when the crosscutting code should be called, such as before or after a specific method call or when an attribute is accessed. For example, the end user asks developers that user authentication is required before any change to customer details is made in a database. This requirement can be fulfilled by declaring an Aspect, which specifies that the authentication code should be included before each call to methods that update customer details. By this way, developers can apply this authentication Aspect to all database updates, which can be easily implemented by modifying the Aspect through changing the definition of where the authentication code is to be woven into the system. Developers do not have to search the whole system to find all occurrences of these methods. In addition, fewer mistakes

might be made and the possibility of introducing security vulnerabilities into the program can be reduced [91][116][127].

Within the Aspects, developers need to define where an Aspect is associated, which is called a join point. “A join point is an identifiable point in the execution of a program. It could be a call to a method or an assignment to a member of an object. ” [98](p35)

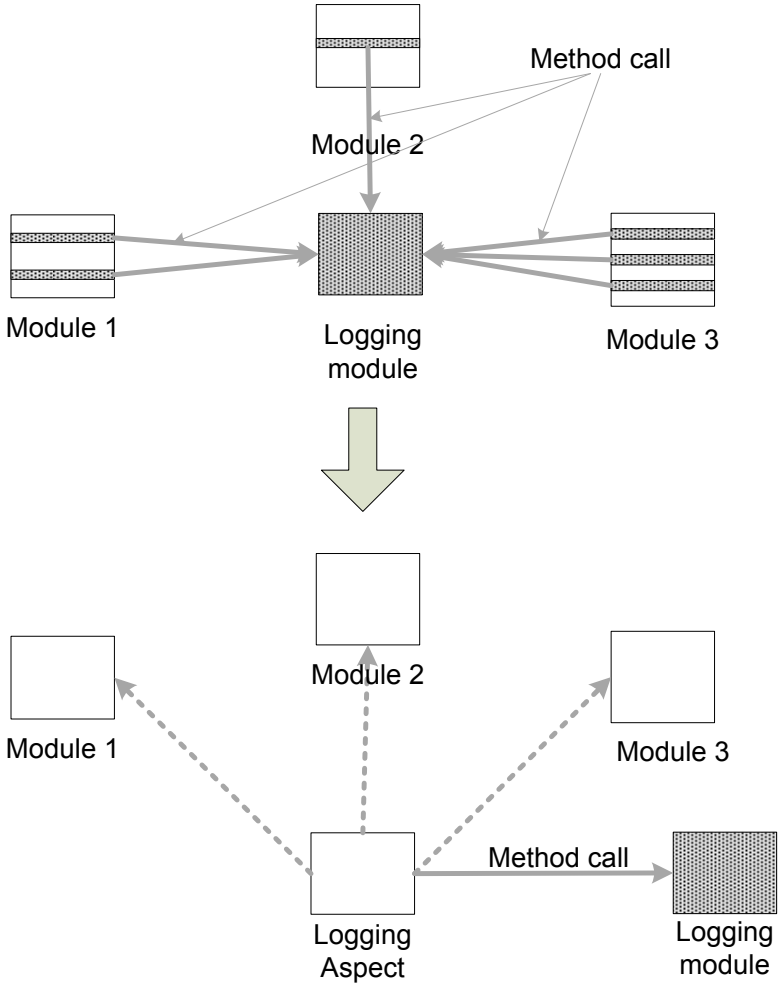


Figure 2.6 Aspect in AOP

Aspects provide two constructs to specify the new behaviour and where it should apply: advice and pointcuts.

“A pointcut is a program construct that selects join points and collects context at those points.” [98](p35). These pointcuts indicate where advice should be executed. For example, a pointcut can select a join point that calls a method, and it could also capture the method’s context, such as the target object on which the method was called and the method’s parameters.

“Advice is the code to be executed at a join point that has been selected by a pointcut.” [98](p35). Advice can execute before, after, or around the join point. Before/After advice can be used to do some operations before/after executing the code at certain join points that are spread across several modules. Around advice can modify the execution of the code that is at the join point, such as replacing or bypassing the code at the join point. The content of advice looks like a method as it encapsulates the logic executed at a join point.

Often the term interception is used when implementing AOP techniques. Interception is a technique which captures method calls and presents the call to some pre- or post-code for additional processing. It can be realized by the decorator pattern [32] [124] but is often part of component runtime environments. For example, the Java 2 Enterprise Edition (J2EE) container technology uses interception to add advanced functionality to components during deployment like container managed persistency or security.

2.4.1.4 Weaving Process

In AOP, Aspects are developed separately; then, in a pre-compilation step called Aspect weaving, they are linked to the join points. Aspect weaving is a form of program generation – the output from the weaver is a program where the Aspect code has been integrated. Finally, an Aspect weaver combines Aspect functionality with the original system to produce an executable system.

Aspect weaving can be seen as a source code transformation process and the Aspect oriented language can be seen as a sort of meta-language that specifies the code transformation [14]. The Aspect weaver works like a compiler that reads the Aspect program and uses it to modify the original code and automatically generate new code modified to implement the desired Aspect.

Some AOP frameworks perform static weaving, where invocation of the advice is statically compiled in at each join point. Others are capable of dynamic weaving, where the advice is connected to its join points at the time code is loaded or even at run time.

One problem associated with the weaving process of current AOP platforms is that only simple weaving processes are supported. Multiple Aspects might be woven to the same join points but only in sequential order, or even worse, the order of the Aspects to be woven is not guaranteed. However, in a complex Aspect-oriented system, an advanced weaving process is desired such as determining the exact execution order and dependencies among the Aspects [117].

2.4.1.5 AOP Development Stages

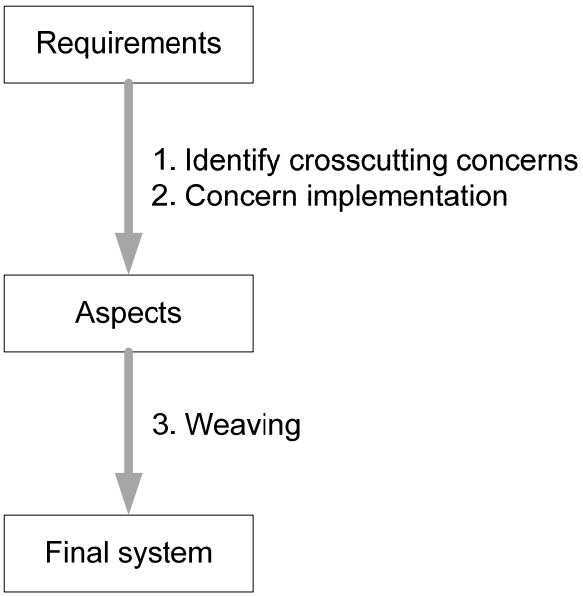


Figure 2.7 AOP development stages

As shown in Figure 2.7, aspect oriented software development includes the following stages:

- Identify the concerns: based on the requirements from the users; developers need to identify the crosscutting concerns from the requirements. This stage is also called Aspectual decomposition.
- Implementation of concerns: the crosscutting concerns then need to be implemented in appropriate AOP platform(s) as Aspects.
- Form the final system by weaving Aspects into the original system: after gathering all crosscutting concerns, developers weave those Aspects into the original system. This stage is also called Aspectual re-composition because at the binary code level, all crosscutting concerns are woven into the original system to form the final system.

2.4.1.6 The Pitfalls of AOP

Although AOP solves crosscutting concern problems in software development, it has not been widely used in the software industry because there are still problems associated with this approach, or even worse, AOP can be dangerous when not used properly [31]. Its drawbacks are shown as follows:

First of all, today's mainstream Aspect oriented languages suffer from pointcut languages because pointcut declarations result in a high coupling between Aspects and the original system [27]. Some researchers [31] identified that the action of AOP is very similar to the "goto" statement and is as harmful as the "goto" statement. While looking at an original class in an AOP environment, you have to find all Aspects impacting the original class. Although some tools are available to help, it's much more difficult to indicate program flow in an AOP environment as opposed to a standard non-AOP program.

Secondly, the pointcuts in AOP are fragile because further changes to the Aspect's source code by other developers may break pointcut semantics easily [31].

Last but not least, the incorrect view of AOP is that AOP is just a crutch to allow developers to quickly and easily add new functions that they forget to specify at requirements design stage [36]. AOP could even be used to 'patch' programs without doing the necessary design to properly install the missing function.

2.4.2 Classification of Current AOP Frameworks

Current AOP research projects can be classified into two categories: heavyweight AOP and lightweight AOP. Heavyweight AOP is usually characterised where the actual language itself has AO concepts built in, such as AspectJ [60]. Lightweight AOP uses existing OO or other methods to implement Aspects with minimal disruption to the existing language or approach. Lightweight AOP is often found in Enterprise frameworks such as Spring [80], JBoss [74] or even within standalone development like AspectWerkz [59] and is usually based on configuration files defining the Aspects that are to be woven into an unchanged language such as Java or C#.

2.4.2.1 Popular Heavyweight AOP Frameworks

AspectJ

AspectJ [60][92] is the most popular and mature aspect-oriented Java implementation, which was developed specifically to popularize the idea of aspect oriented programming in the Java community. AspectJ is now

available as an Eclipse Foundation open-source project [62], both stand-alone and integrated into Eclipse.

AspectJ is a heavyweight AOP implementation. It uses Java-like syntax and has included IDE integrations for displaying crosscutting structure since its initial public release in 2001. AspectJ introduces new keywords to the Java language for defining Aspects and join points. Tightly integrated with Java, the major advantage of AspectJ is the expressiveness of its “pointcut” language, which can be used to describe the pointcuts and advices clearly. The pointcut language describes the condition on which the corresponding Aspect advice is executed. This means that the developer must learn additional language syntax and use the AspectJ compiler in order to build any code written with AspectJ. In AspectJ, developers do not need to modify the client code because all Aspects, pointcuts, advices, etc. are saved in separate source files.

AspectJ uses static weaving at compile time, although the most recent version of AspectJ does provide some initial support for weaving to be performed at class load time. In all cases, the AspectJ program is transformed into a valid standard Java program running in a Java virtual machine. All classes affected by Aspects are binary-compatible with the unaffected classes.

However, AspectJ also has its weakness [99][137][152] as shown below:

- High coupling: most of the current pointcut designators explicitly specify their target locations by naming these classes/methods. These explicit references obviously introduce a high coupling between the original system and the Aspects, making Aspect reuse harder. AspectJ does not allow the specification for a crosscutting concern to be written as a separate entity from the Aspect itself, therefore the developer must have a full understanding of the Aspect

code and thus cannot use or reuse the Aspect in a black box manner [107]. AspectJ offers wild-cards to reduce coupling. However, this introduces a new problem as the wild-cards are not checked by the compiler. As a result, programmers have to be very careful with their pointcuts to avoid wrong or missed matches.

- Complicated syntax: although very powerful, the language is now full of complicated, semantically challenged constructions that are added to standard Java syntax.
- Debugging: one of the greatest problems for AspectJ is debugging. While at the syntactic level AspectJ program code appears separate, at run-time it is not. Therefore, after weaving, the execution of the final system can become unpredictable.

AspectC++

AspectC++ [71][133] extends the AspectJ approach to C/C++. It is a set of C++ language extensions to facilitate aspect-oriented programming with C/C++. While being strongly influenced by the AspectJ language model, AspectC++ claims that it supports all additional concepts that are unique to the C++ domain. This ranges from global functions, operators, and multiple inheritance up to weaving in template code and join point-specific instantiation of templates [105].

aoPHP

aoPHP [70] is an addition to PHP that allows the use of Aspect-Oriented programming in web based applications.

aoPHP was originally developed in Java 1.5. It relied on a PHP script and Apache's mod_rewrite module to properly redirect incoming calls to PHP Scripts. The calls to PHP Scripts were redirected to the aoPHP weaver where Aspects were woven at run-time on the server. The resulting code,

which was the output of aoPHP weaver, was then passed to Apache PHP engine to be executed and displayed.

The current versions of aoPHP (version: 4.0) works in a very similar way to the original version. It still relies on a PHP script to call the weaver. However the weaver is now written in C++ with full support for Regular Expressions. Therefore, the new weaver provides a good improvement in both performance and speed of the parser and weaver.

2.4.2.2 Popular Lightweight AOP Frameworks

AspectWerkz

AspectWerkz [59] is a lightweight AOP framework without modifying the Java language in any way. This means that there is no new syntax for a developer to learn. It is capable of compile time, load-time, and run-time weaving.

AspectWerkz also supports annotations. That is, join points can be marked through Java annotations. This is a powerful mechanism that can alleviate some of the concerns caused by having the Aspects completely separate from the implementation. Now, Aspect oriented features can be declared as annotations.

Spring AOP

Spring AOP [80] is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a J2EE web container or application server.

Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs.

Spring AOP's approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation; therefore Spring AOP does not compete with AspectJ to provide a comprehensive AOP solution. Spring AOP tries to provide a close integration between AOP implementation and to help solve common problems in enterprise applications.

AOP in JBoss

The JBoss AOP is an alternative to the AspectJ Java implementation. JBoss AOP [68] is a pure Java Aspect Oriented Framework. JBoss AOP is not only a framework, but also a pre-packaged set of Aspects that are applied via annotations, pointcut expressions, or dynamically at runtime such as caching, transactions, security, etc. The JBoss AOP Framework, although available as a separate library, is also heavily used by the latest version of the JBoss application server.

2.5 Current State of Software Product Line

One of the most effective approaches to software reuse is building software product lines or application families. “A product line is a set of applications with a common application-specific architecture. Each specific application is specialised in some way.” [6][35][47][132](p432).

The common core of the application family is reused each time and variations are implemented in different ways when a new application is developed. The development of new applications may involve specific

component configuration, implementing new components and adapting some of the existing components to meet new requirements.

"Software product lines are designed to be reconfigured." While reconfiguring the new application, developers may need to add or remove components from the system, define parameters and constraints for system components, and include knowledge of business processes [8][132](p432).

There are various types of specialisation of an application family that may be developed [35][132]:

- Platform specialisation, which means that different versions of the application need to be developed for different platforms. In this case, the core functionality of the application is normally unchanged; only those components that relate to various hardware and operating systems are modified. For example, a Global Positioning System (GPS) system may have different versions depending on the type of Personal Digital Assistant (PDA) device used.
- Environment specialisation, where different versions of the application are created to deal with different operating environments and peripheral devices. In this case, the functionality may vary to reflect the functionality of various environment and peripherals, and components that interface with peripherals must be modified.
- Functional specialisation, where different versions of the application are developed for customers with different requirements. In this case, components implementing the functionality may be modified to fulfil the different requirements.
- Process specialisation, where the system is modified to handle different business processes. For example, a Human Resource management system may be adapted to deal with a centralised recruitment process in a local company and a distributed recruitment process in another multi-national enterprise.

The steps involved in adapting an application family to create a new application [132]:

- Elicit stakeholder requirements.
- Choose most appropriate family member.
- Renegotiate requirements.
- Adapt existing family member.
- Deliver new family member and add it to the product family.

The software product line technology has been used in the approach because the nature of product line makes it a suitable ingredient of the proposed approach. The idea of product line such as separating variations from core assets and related technology such as generative programming can be developed in the approach to improve the reusability of Aspects such as implementing platform-independent Aspects. Functional variations, parameter variations, and platform variations are allowed by applying software product line technology to the approach.

2.6 Summary

Based on the literature review, the following conclusions have been reached:

- Component adaptation techniques are the key solution to address the mismatch problem in component based systems. However, due to the complex nature of the mismatch between reuse requirements and components, available component adaptation approaches are either only capable of adaptation at simple levels such as wrappers, or inefficient to use as a result of lack of automation in their adaptation process. Deep-level, automatic component adaptation is still needed.
- Generative based reuse technology is a cost-effective approach with reasonable automation for application development. The nature of this technology makes it a potentially suitable ingredient of the proposed component adaptation approach to support the automation.

- AOP introduces a new unit of modularization - an Aspect that crosscuts other modules. The nature of AOP makes it particularly suitable for addressing non-functional mismatches with component-based systems. Therefore, AOP may be employed in achieving adaptable components by imposing needed effects on the components [33]. Hence, the mismatch problem can be addressed and the quality of target components can be improved by using AOP.
- Existing AOP technology is inefficient to be used in the component adaptation because AOP-based system implementation still has its disadvantages, e.g., reduced readability and maintainability of the final system, poor reusability of Aspect assets. Some AOP based frameworks have been developed to achieve reusable Aspects. However, an AOP platform independent framework is still desired in a heterogeneous distributed environment to solve crosscutting problem since a common model for AOP is still missing. Furthermore, current AOP techniques only support weaving Aspects sequentially. To cope with complex adaptation, it often requires weaving Aspects in more sophisticated control flow, e.g. dynamically deciding whether to invoke a particular Aspect.
- A software product line is an effective approach to software reuse. Common core concerns and variations are the two parts of software product lines. As the Aspects in AOP also have their common parts and variations, software product line techniques may be applied to AOP to achieve highly reusable Aspects.
- It is promising to develop a novel method to conduct Generative Aspect-oriented Component Adaptation by integrating software product line, AOP and generative component adaptation to correct the above weakness of component adaptation techniques and AOP. Such an approach is expected to be highly automatic, feasible at deep level, and supports reuse of adaptation assets.

Chapter 3 Related Work

This chapter conducts a review of many approaches that have been found useful for component adaptation, generative programming, and AOP related projects. The problems associated with these approaches are the motivation of the development of the proposed approach.

3.1 Component Adaptation Approaches

3.1.1 Superimposition

Superimposition [155] is a black-box adaptation technique. In Superimposition, software developers are able to apply a number of predefined, but configurable types of functionality to reusable components. The principle behind superimposition is that a component and the functionality adapting the component should be decoupled from each other. There are three categories of typical adaptation types: component interface, component composition and component monitoring.

The notion of superimposition has been implemented in the Layered Object Model (LayOM), an extensible component object language model. The advantage of layers over traditional wrappers is that layers are transparent and provide reuse and customizability of adaptation behaviour.

Superimposition uses nested component adaptation types to compose multiple adaptation behaviours for a single component. However, due to lack of component information, modification is limited to a simple level, such as conversion of parameters and refinement of operations. Moreover, with more layers of code imposed on original code, the overhead of the adapted component increases heavily, which degrades system efficiency.

3.1.2 Binary Component Adaptation (BCA)

Binary Component Adaptation (BCA) [90] is an adaptation technique that applies adaptations on-the-fly (during program loading) to component binaries without requiring any source code access. BCA rewrites component binaries before (or while) they are loaded.

The BCA system is currently implemented to work with Java. An application builder wishing to adapt a Java component constructs a delta file specification containing information about the desired changes to a class; this includes adding or renaming an interface, method, field, or method reference. One can even alter the super class for a component. A Delta File Compiler (DFC) creates a binary delta file containing the necessary byte code adjustments to the component being adapted. Once a component is adapted, other classes that refer to the adapted component must be recompiled using a modified Java compiler. The class Loader for the Java compiler merges byte code streams from the original Component class file and the extra byte code stored in the binary delta file.

However, together with the binary code adaptation, especially with “online” (on-the-fly) adaptations, extra processing time is required because BCA needs to rewrite class files while they are loaded. As a result, the load-time overhead is a major problem. Consequently, when more adaptation processes are required, the load-time will be the bottleneck of the system performance.

3.1.3 Customizable Components

Customizable components [96], as part of the COMPOSE project, is an environment for building customizable software components, it is an

approach to expressing customization properties of components. The declarations enable the developer to focus on what to customize in a component, as opposed to how to customize it. Customization transformations are automatically determined by compiling both the declarations and the component code; this process produces a customizable component. Such a component is then ready to be custom-fitted to any application.

In this work, the customized components generated for various usage contexts have exhibited performance comparable to, or better than manually customized code, however, component adaptation is limited to pre-defined optional customization, and deeper adaptation is not supported.

3.1.4 SAGA

Scenario-based dynamic component Adaptation and GenerAtion (SAGA) [104][153][154] developed a deep level component adaptation approach through XML-based component specification, interrelated adaptation scenarios and corresponding component adaptation and generation.

In the SAGA project, a Component Definition Language (CDL) is used to record the design configuration of components reused in specific applications. Scenarios are an XML document of a series of adaptation actions. Scenarios may be adjusted, composed or associated interactively to cope with complex reuse cases. Scenarios are used by the adaptor and the generator to perform adaptation and generation automatically.

The SAGA project mainly focused on generative component adaptation at binary code level, i.e., the adapted part of the component will be generated as new blocks of binary code and these blocks will then be composed with other unchanged blocks of code to form a new adapted component. The SAGA project achieved deep adaptation with little code overhead in the adapted component.

However, automation is a challenge in the SAGA approach because it is always complex to generate blocks of code according to scenarios and the original component code. To reach a high level of automation, a large set of adaptation rules and domain knowledge have to be developed to support the process, and probably the application domain has to be restricted as well.

Based on the above features, the SAGA approach is more suitable for the development of traditional component-based systems where developers have more access to the internal design of components and can impose more intervention on the adaptation and integration process.

3.1.5 A Non-Invasive Approach to Dynamic Web Service Provisioning

Based on .NET Common Language Runtime, a dynamic web service provision approach [22] has been proposed. In this approach, the runtime code manipulation at the Intermediate Language (IL) level with a repository of adaptation aspects is used to support service provisioning. The main idea of this approach is to intercept the web service call at IL level before it is compiled.

However, the invasive change of web service code will pollute the original application such that recovering the original application will become difficult, which introduces the common version control problems for software systems. In addition, the change to Common Language Runtime (CLR) brings potential stability issues to the system and its performance will decrease because web service calls are intercepted by a profiler first. Last but not least, most web services are available only in binary form, rather than in source code or intermediate language. This approach is not applicable to web services in binary form.

3.1.6 Interface Level "wrapper" for Web Service Adaptation

In SOA, it is not cost effective to require each base web service to individually handle the variety of messages than can come its way. Therefore, there is the need for an adaptation layer in an SOA to deal with all the aspects of message handling.

In Fuchs's approach [48], web service adaptation can be achieved by altering the WSDL contract. They proposed a framework to wrap the underlying web services with an adaptation layer to organize meta-information about operation behaviour. The adaptation layer mediates between the underlying services and the service consumers. 'Virtual' APIs are provided in the adaptation layer to change the name and parameters of operations.

However, working as a wrapper, this approach only deals with the simple adaptation requirements because adaptations only occur at the interface level.

3.1.7 Evaluation of Component Adaptation Techniques

In table 3.1, the approaches that adapt a software component to create a new component are listed. All these approaches are evaluated on how well each technique fulfils the component adaptation requirements specified in section 2.1.6.

Adaptation techniques	R1	R2	R3	R4	R5	R6	R7	R8
SAGA	-	+	-	+	-	-	+	-
Superimposition	+	-	+	+	+	-	-	-
BCA	-	+	-	-	-	-	+	-
Customizable Components	-	-	+	+	+	+	-	-
Non-Invasive approach to WS	+	+	+	+	-	+/-	+	+
Wrapper for WS adaptation	+	-	+	-	-	-	-	+

R1: Black-box

R2: Transparent

R3: Composable

R4: Reusable

R5: Configurable

R6: Automatic

R7: Deep level adaptation

R8: Language independence

+: fulfilled.

-: not fulfilled.

+/-: The automation of a particular adaptation case highly depends on whether an adaptation aspect is available in the repository. If the related aspects are available, the profiler works automatically.

Table 3.1 Requirements for component adaptation techniques

3.2 Generative Programming Related Approaches

3.2.1 XVCL

XVCL (XML-based Variant Configuration Language) [87][138] is a general-purpose mark-up language for configuring variants in programs and other types of documents.

XVCL is capable of injecting changes, according to pre-defined plans, into programs represented as a hierarchy of highly parameterized meta-components or templates. Meta-component parameters may be as integer or string values, or complex as a hierarchy of other meta-components. XVCL uses a “composition with adaptation” mechanism to instantiate parameters and to generate concrete programs from generic meta-component architectures. Many template engines have been proposed to tackle specific problems, in specific domains. XVCL is a language, problem and domain independent template based generative engine.

However, in each frame in XVCL, the code templates and variables to support software product lines are mixed together, which can make the code difficult to read, understand and therefore maintain.

3.3 AOP Related Projects

3.3.1 Aspectual Component

To achieve reusable Aspects, Karl Lieberherr et al. introduce the concept of Aspectual Components [99]. In Aspectual Components, Aspects are specified independently as a set of abstract join points. They believe that aspect-oriented programming means expressing each Aspect separately, in terms of its own modular structure. With this model, an Aspect is described as a set of abstract join points which are used when an Aspect is combined with the base-modules of a software system. The Aspect-behaviour is kept separate from the core components, even at run-time. Explicit connectors are then used to bind these abstract joinpoints with concrete joinpoints in the target application.

Aspectual components propose a new type of interface that allows components to describe adaptations independent of the concrete components that will be adapted.

Aspectual components distinguish between components that enhance and cross-cut other components and components that only provide new behaviour. An Aspectual component has a provided and a required interface. The required interface includes a participant graph that describes an ideal structure for formulating the behaviour of the component. The purpose of a component is to add data members and function members to other components and to modify function members of other components. The provided interface of a component includes both new function members and modified function members. Connectors connect the provided and required interfaces of other components. The connection process starts with level-zero components consisting of very simple class definitions.

However, since Java interfaces are rather an implementation mechanism than an Aspect-description mechanism, this approach violates the separation of component description and implementation [125].

3.3.2 JAsCo

JAsCo [137] is an Aspect based research project for component based development, in particular, the Java Beans component model. JAsCo combines the expressive power of AspectJ with the Aspect independency idea of Aspectual Component.

The JAsCo language introduces two concepts: Aspect beans and connectors. An Aspect bean is used to define Aspects independently from a specific context, which interferes with the execution of a component by using a special kind of inner class, called a hook. Hooks are generic and reusable entities and can be considered as a combination of an abstract pointcut and advice [88][137]. Because Aspect beans are described independently from a specific context, they can be reused and applied upon a variety of components. Connectors are used to connect Aspect beans to specific components. Connectors have two main purposes [88]: instantiating the

abstract Aspect beans onto a concrete context and thereby binding the abstract pointcuts with concrete pointcuts. In addition, a connector allows specifying precedence and combination strategies between the Aspects and components.

However, JAsCo is not very suitable for specific modification requirements since it does not provide a mechanism for conducting users' requirements. In addition, the application of Aspects on target components or systems is based on traditional AOP processes, and therefore, may result in lower readability, maintainability and performance. Moreover, the current implementation of JAsCo has been bound to Java, which means it can not be used in a heterogeneous system including different programming language implementations.

3.3.3 Shared Join Points Model

In AOP, it is possible that several units of Aspectual behaviour need to be woven at the same join point. In these cases, Aspects are said to 'share' the same join point. Such shared join points may have problems such as determining the exact execution order and dependencies among the Aspects. Shared Join Points Model [117] is a general and declarative model for defining constraints upon the possible compositions of Aspects at a shared join point. Currently, Shared Join Points Model is integrated with AspectJ.

Most AOP platforms provide reflective information about the current join point. For example, "thisJoinPoint" in AspectJ provides the context information within an advice. In this way, "thisJoinPoint" acts as a communication channel among the aspect instances that are sharing the same join point. Thus, aspect instances being applied at the same join point can exchange information placed in the "thisJoinPoint" and conditional execution of aspects can be achieved.

However, Shared Join Points Model highly depends on the language features of the existing AOP platform, which limits the wider use of the model.

3.3.4 Framed Aspects

Since current AOP does not support configuration mechanisms and generalisation capabilities that are required to realise variability, the potential for Aspects to be reused in different contexts is limited [106][107].

Frame technology [5] provides mechanisms to support configuration and generalisation capabilities. Loughran & Rashid [107] introduced an approach to support reusable Aspects that combines the respective strengths of AOP and frame technology.

In Framed Aspects, parameterisation is supported for AOP, which enables Aspects to be customised for a particular scenario, and therefore increases the reusability of the Aspects. Meanwhile, conditional compilation allows for optional and alternative variant features of an Aspect module to be included or excluded, thus resulting in optimal usage of code.

However, templates are not supported within some mainstream languages such as Java and C# and therefore, the approach is limited to a small range of AOP platforms.

3.3.5 Critical Analysis of AOP technologies

AOP developers claim that compared with traditional software development approaches, Aspect oriented approaches make it easier to develop and maintain certain kinds of application code which has crosscutting concerns. To assess these claims, the following criteria are proposed to assess the

technical requirements (T1-T7) for Aspect oriented approaches. These requirements can be used to evaluate AOP approaches. It may not be possible for an adaptation technique to fulfil all requirements.

T1: Short Learning Curve

Since various features are supported to solve crosscutting concerns in AOP approaches, the syntax to support these features normally is not easy to learn, which restricts the usability of these AOP technology. Therefore, the ease of learning an AOP approach is an important requirement.

T2: Reusable

The code of each Aspect should be used repeatedly in other Aspect oriented development. The purpose of AOP is to reduce the complexity of software development process. Therefore, it is highly desired that mature Aspects can be used repeatedly, i.e. to be applied to recurring systems.

T3: Light Weight

As discussed in section 2.4.2, current AOP frameworks may be heavy weight or light weight. Rather than mixing Aspects with standard programming code in heavy weight AOP, light weight AOP is preferred because based on existing programming code, only extra configuration files are needed to describe the basic information of Aspects.

T4: Configurable

Since the same Aspects may be reused in similar situations, the AOP weaver should be able to apply the same Aspects to a set of target components with different settings.

T5: Advanced Weaving Process

While applying Aspects to existing component based systems, multiple Aspects may be needed in the weaving process. Sequential weaving is not powerful enough to handle sophisticated adaptation (section 2.4.1.4), hence more advanced weaving process, such as a switch structure is needed. With advanced weaving process, Aspects can be used flexibly and widely to address complex crosscutting problems.

T6: AOP Programming Language Independence

As AOSD is a general approach in software development, the AOP idea is independent of any programming language. However, current AOP platforms and their implementations are bound to a specific programming language, which restricts the wide use of current AOP platforms. Therefore, an AOP platform will be more reusable if a programming language independent mechanism is included.

T7: Generative Aspects

After identifying the crosscutting concerns, developers need to implement related Aspects. Due to the complicated syntax of existing AOP languages, the automatic generation of Aspects is desired in AOP development. Generative requirement is used to describe whether an approach supports automatic Aspect generation.

In table 3.2, an overview of discussed AOP approaches including Aspectual Component, JAsCo, Shared Join Points Model, and Framed Aspects is presented that indicates how well each approach fulfils the specified requirements.

AOP approaches	T1	T2	T3	T4	T5	T6	T7
Aspectual Component	-	+	-	-	-	-	-
JAsCo	-	+	-	+	-	-	-
Shared Join Points Model	-	-	-	-	+	-	-
Framed Aspects	-	+	+	+	-	+	+

T1: Short learning curve
T2: Reusable
T3: Light weight
T4: Configurable
T5: Advanced weaving process
T6: Language independence
T7: Generative Aspects
+: fulfilled
-: not fulfilled

Table 3.2 Evaluation of current AOP approaches

3.4 Summary and Conclusions

To summarise, available component adaptation approaches are either only capable for adaptation at simple levels such as wrappers, or inefficient to use as a result of lack of automation in their adaptation process. Some AOP based frameworks have been developed to achieve reusable aspects. However, current AOP techniques only support weaving aspects sequentially. To cope with complex adaptation, it often requires weaving aspects in more sophisticated control flow, e.g. dynamically deciding whether to invoke a particular aspect, and synchronizing in multi-thread applications.

In conclusion, to address these problems and develop a novel component adaptation approach, the following requirements need to be considered:

3.4.1 Requirement 1: Deep level component adaptation with high automation

Due to the complex nature of the mismatches between existing components, available component adaptation approaches are either only capable of

adaptation at simple levels, or are inefficient to use as a result of lack of automation in their adaptation process. Therefore, a deep level component adaptation technique with high automation is desirable.

The nature of AOP makes it particularly suitable for addressing quality-oriented issues with component-based systems. Generative programming is an ideal technology to increase the degree of automation of the proposed adaptation approach. Therefore, AOP and Generative programming can be used in the proposed approach to fulfil this requirement.

3.4.2 Requirement 2: Highly reusable Aspects in AOP platforms

Some AOP based frameworks have been developed to achieve reusable Aspects. However, current approaches do not support Aspect reusability efficiently. For example, in some approaches, reusable Aspects are not supported at all. In other approaches, the reusable Aspects are bound to a specific programming language. As a result, the Aspects developed for a specific programming language are not reusable in the other languages. In other words, in current available AOP platforms, semantically equivalent Aspects still crosscut in different AOP platforms.

Consequently, an AOP platform independent framework is still desired in a heterogeneous distributed environment to solve crosscutting problem since a common model for AOP is still missing [111]. The design of AOP platform independent framework to avoid repetition is the heart of and prerequisite to achieving highly reusable Aspect. Failure to design an AOP platform independent framework leads to low level reuse and high maintenance costs.

3.4.3 Requirement 3: Advanced weaving process support in AOP platform

Current AOP platforms only support a simple weaving process. Individual Aspects can be weaved to target components individually. A single join point may have more than one Aspect associated with that join point. However, when these Aspects are woven to a target component, the sequence of the execution of these Aspects is not guaranteed.

In aspect oriented systems, it is often the case that Aspects at the same join point may require a specific execution order or conditional execution. Therefore, to cope with complex aspect oriented component adaptation, it often requires weaving Aspects in more sophisticated control flow, e.g. dynamically deciding whether to invoke a particular Aspect.

3.4.4 Requirement 4: Short learning curve of AOP platform

As the learning curve of current AOP platforms is steep, an easy-to-use approach is desirable to promote AOP being more widely used. Generative techniques are cost effective approaches to generate source code. Therefore, Generative techniques can be used to generate Aspects automatically. As the AOP platform specific knowledge can be hidden in the generator, the learning curve will be reasonably short.

3.4.5 Conclusions

In conclusion, based on the investigation of current techniques on component adaptation, generative programming, and AOP, a new approach is required to eliminate the problems associated with these techniques, and therefore achieve deep level component adaptation with high automation. The requirements from Section 3.4.1 to Section 3.4.4 must be fulfilled.

Chapter 4 The Approach

This chapter introduces the approach in detail including the two dimensional Aspect model, the approach, the Aspect generation process, and the aspect oriented adaptation process.

4.1 Introduction

To achieve automated component adaptation at a deep level, particularly aiming at eliminating mismatches in non-functional issues such as system performance, dependability and safety, a Generative Aspect-oriented component adaptatIoN (GAIN) approach is proposed. The problems associated with current AOP frameworks such as the reusability of Aspects, programming language independent Aspect support, and advanced weaving process support are also addressed in the approach. In addition, as the generative programming is used in the approach, the AOP specific knowledge is hidden which results in the short learning curve of the approach.

In the GAIN approach, component adaptation is carried out within an aspect-oriented component adaptation framework by generating and then applying the adaptation Aspects to original component(s) under a designed weaving process according to specific adaptation requirements.

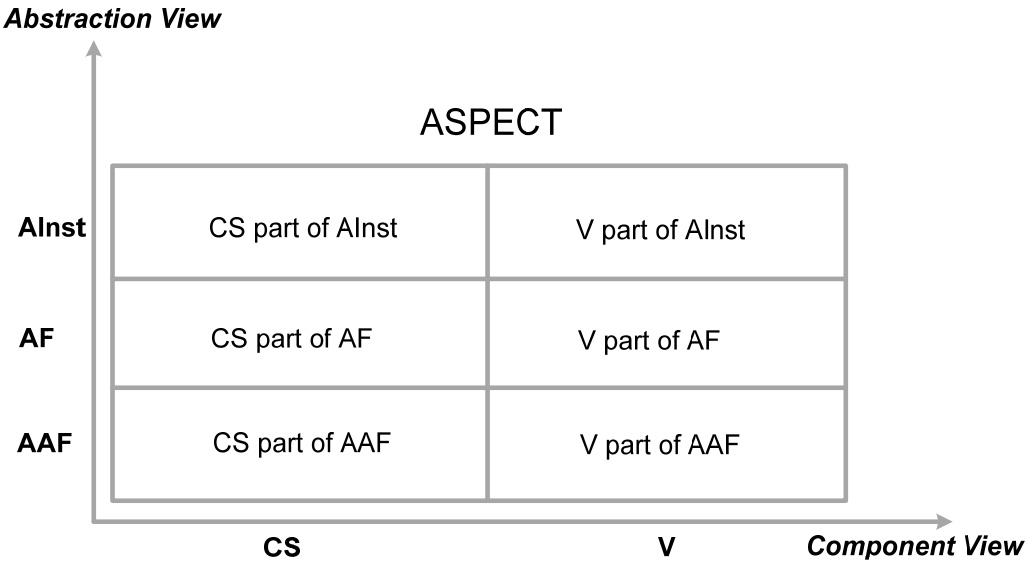
The approach is based on the successful points in a few technologies, i.e., Aspect Oriented Programming, Software Product Line, Generative Programming, and Component Adaptation. First of all, Generative Programming and AOP techniques are applied to component adaptation; hence deep level adaptation is achieved with high automation. Secondly, based on the software product line concepts and generative component

adaptation techniques, a two dimensional Aspect model (refer to section 4.2) has been developed to support highly reusable and AOP platform independent Aspects. To facilitate the reusability of adaptation knowledge, an expandable library of reusable adaptation Aspects is required. Thirdly, compared with traditional AOP, the weaving process of Aspects in the approach supports more complex control flow, i.e., not only sequence, but also switches, to make the adaptation more accurate and efficient for components reused in more complicated applications. Last but not least, the approach absorbs the variation concept of a software product line and Generative Programming techniques. All the specific knowledge of AOP is hidden in the generator; hence the learning curve is short.

4.2 Product Line Based Aspect Model

In the approach, the adaptation knowledge is captured in Aspects and aims to be reusable in various adaptation circumstances. To achieve product line based automatic generation of the adaptation Aspects and to enlarge the reusability of Aspects, a two dimensional Aspect model is developed and in practice an Aspect repository is built as an embodiment of the above model. Different abstract levels of Aspects is supported in the Aspect repository.

Figure 4.1 presents the views of an Aspect from two different dimensions: component view and abstraction view. From component view, the CS (Common Structure) and V (Variations) of each component are separated from each other to support various product family members. The different levels of abstraction for each Aspect are described in abstraction view as AAF (Abstract Aspect Frame), AF (Aspect Frame), and AInst (Aspect Instance). With the two-dimensional Aspect model, reusable Aspects are achieved and AOP language independent Aspects are implemented.



CS: Common Structure

V: Variations

AAF: Abstract Aspect Frame

AF: Aspect Frame

AInst: Aspect Instance

Figure 4.1 A two dimensional view of an Aspect

As shown in figure 4.1, during the whole Aspect oriented adaptation process, from the designing of different Aspects in AAFs, to the implementation of AOP platform independent Aspects in AFs, to the implementation of concrete AOP platform specific Aspects in AInsts, all Aspects are presented in two parts: CS and V, no matter which abstract level they are, such as AAF, AF, or AInst. This mechanism maximise the reusability of Aspects by separating variations from the common core assets in the two dimensional model.

4.2.1 Component View

In software product lines, a family of applications are derived from one base architecture. Software product lines have two parts: core assets, which can be used in different product family members, and variations, which show the difference between different product family members. A challenge with the

product line approach is to model the variability between the core assets and the applications.

In the approach, each Aspect consists of two parts, namely CS (Common Structure) and V (Variations) to realize the main idea of a software product line approach and reflect the core assets and the variability of software product family members.

The commonness of various Aspects is summarized in CS which can be reused in all similar Aspects in various Aspect oriented systems. All Aspects in the approach have the same structure in CS because only common information for each Aspect such as the details of pointcut and advice, is kept in the CS part. The CS parts of different Aspects are produced by filling detailed content information into the basic CS structure.

Each Aspect varies in its V part, which includes the implementation details of various functionalities, and the configuration information of a particular Aspect. Together with the basic information in CS, distinctive information reflecting the variations is used to define different Aspects. For example, for logging Aspects, the output device such as a hard drive and a file name need to be specified. On the other hand, for a database connection pool Aspect, the capacity of the pool, and the expiry time of each connection are needed to instantiate a specific Aspect.

In the abstraction dimension, the CS and V are presented in different forms with different abstraction details at the three abstraction levels. For example, XML Schema files for different types of Aspects at AAF level, XML files for programming language independent Aspects at AF level, and program source code for executable Aspects at AInst level. Full details are described in section 4.2.2.

In summary, the component view presents the structural elements inside an Aspect: CS and V. Therefore, in the component view, an Aspect is defined as follows in the proposed model:

ASPECT = (CS, V), where

CS = (P, A), where

P = Declaration of Pointcut = (D, O), where

D = Details of the pointcut

O = Object that the pointcut applies to

A= Declaration of Advice

V = (C, SI), where

C = Content of variations

SI= Semantic Interpreters

- CS is the Common Structure of Aspects. CS defines the common structural elements that any Aspect will have despite its functionality or implementation platform. CS consists of two sub-elements: Pointcut (P) and Advice (A). All Aspects have the same CS at AAF level (refer to section 4.2.2 for details) no matter how different these Aspects are in functionality and implementation platform.
- P is the specification of the pointcut, which includes two parts: D and O. D defines the details of the pointcut, such as the name of the pointcut, the applying time of the pointcut, etc. and O is the object that the pointcut applies to, including the component name and method signatures.
- A is the declaration of advice, which contains the basic information of the advice, e.g. when the joint point will be inserted into the target component.
- V is Variations, which defines the variations of different Aspects. V consists of two parts: the content of the variations (C) and the Semantic Interpreters (SI).

- C refers to the contents of the variations. For example, in a logging Aspect, C includes the logging message to be saved, and the file name used to save the message, whilst in a database connection pool Aspect, the capacity of the pool is part of its variation parameter.
- SI is the Semantic Interpreters of the Aspect, which specifies the actions to be carried out when the Aspect is applied. SI is unique when the type of Aspect (refer to section 4.2.2) and the target AOP platform are specified.

4.2.2 Abstraction View

In the proposed Aspect model, to achieve high reusability, an Aspect is defined at three abstraction levels, which constitutes the abstraction view. Therefore, an Aspect can be described as follows in the abstraction view:

$\mathcal{ASPECT} ::= \{\mathcal{AAF} \mid \mathcal{AF} \mid \mathcal{AInst}\}$, where

- AAF stands for Abstract Aspect Frame, which defines the structure of the Aspect.

  AAF is the fundamental and the most abstract level of the Aspect definition in the model. As an XML schema file, AAF is used to define the structure of different types of Aspects. According to the functionality, AAF forms a hierarchical structure that reflects functional variations of different adaptation Aspects. Adaptation Aspects are modelled into different types (refer to section 5.2.2 for the implementation of AAF), for example, logging, caching, authentications, etc. Each AAF consists of CS and V. All Aspects have exactly the same CS part because they all have identical definition in a XML schema for their CS. On the other hand, different Aspects have unique V part definitions in AAF to reflect the variations.

- AF stands for Aspect Frame, which is an AOP platform independent instantiation of AAF in a specific adaptation circumstance.

  Each AAF may have many Aspect Frames (AFs), depending on the usage of the AAF in various adaptation scenarios. AF is the second abstraction layer in an Aspect definition. An AF is an instance of the related AAF in a specific adaptation circumstance. Compared with its AAF, an AF has the details of a concrete Aspect populated into it by assigning values to the parameters. User interaction is required to create an AF from an AAF. Defined in XML format, AF is independent from concrete AOP platforms. From the component view, the AFs generated from the same AAF have the same structure but may have different data in both CS and V parts.

- AInst stands for Aspect Instance, which is an instantiation of AF on a specific AOP platform.

  An AF is not executable until it is mapped onto a concrete AOP platform. The result of this mapping is a family of Aspect Instances (AInst) based on various AOP platforms. An Aspect Instance is executable and specific to a concrete AOP platform, and it reflects platform variations of an Aspect on different AOP platforms. The template based transformation mechanism to generate AInst from their AF is called a Semantic Interpreter, which is Aspect and AOP platform specific. The generation process is fully automatic. The AInsts generated from the same AF may have different CS and V parts because they may be mapped into different AOP platforms.

4.3 The Approach

4.3.1 The Framework

The general process of the approach is given in Figure 4.2 as a framework. It is presumed that a component has been found with suitable potential to be used in a component-based application, however, based on reading of the related design documents, the application developer identified some mismatches of the component and wishes to have it adapted.

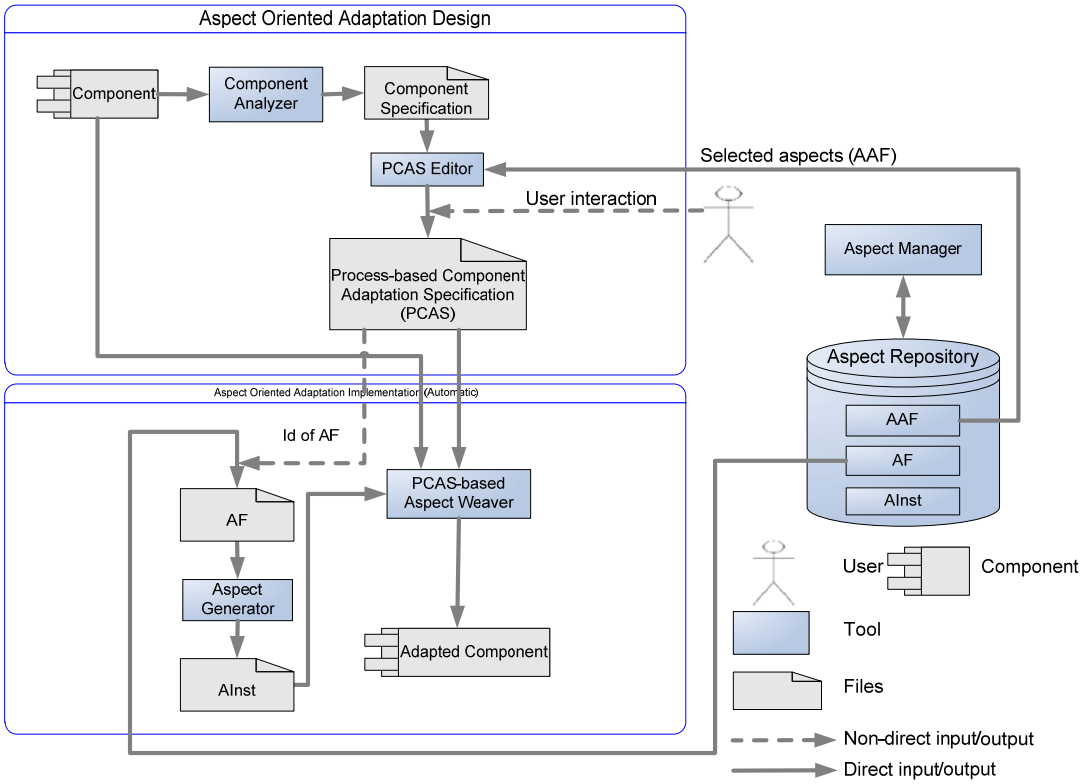


Figure 4.2 The Generative Aspect-oriented component adaptatIoN (GAIN) framework

The mismatches will be eliminated by applying Aspect-oriented adaptation to the original component. The whole adaptation process includes Aspect-oriented adaptation design and Aspect-oriented adaptation implementation.

As shown in Figure 4.2, in the Aspect oriented adaptation design stage, the adaptation requirements are captured and as a result, the Process-based

Component Adaptation Specification (PCAS) is produced at the end of the design stage. In the implementation stage, with tool support, the component adaptation process is performed automatically.

4.3.2 Aspect Oriented Adaptation Design

The purpose of this phase is to define the detailed component adaptation specification. To achieve this goal, two types of information need to be identified first:

- **Component specification**: Initially, the original component is analyzed with a tool called Component Analyzer, which performs simple source code analysis to the component and extracts basic information of the component including class names and method signatures. The information gathered is referred as component specification and will be used to build the PCAS and perform component adaptation.
- **Available Aspects**: During the adaptation process, available Aspects are retrieved from the Aspect Repository to be used in the adaptation. The Aspect repository supports highly and incrementally reusable Aspects. Reusable Aspects are defined at different abstraction levels and kept in the repository as AAF, AF, and AInst. The reusable assets in the repository include both primitive Aspects and Aspect Frameworks, which come from the adaptation process in PCAS. The saved Aspects, particularly Aspect Frameworks are potentially reusable for component adaptations in other applications with similar scenarios. While the framework is used, the repository will be populated with more and more Aspects incrementally.

Then based on the adaptation requirements, a Process-based Component Adaptation Specification (PCAS) will be composed by selecting Aspects defined at the abstraction level of Abstract Aspect Frames (AAF). The

selection of Aspects is actually the process to determine functional variations of a specific adaptation. An AAF is considered as a template to produce specific Aspects. The composition of Aspects in PCAS is supported by an interactive IDE called PCAS Editor, which supports both a graphical view and an XML source view of the PCAS.

A PCAS is an XML formatted document, which includes the details of component adaptation, such as the target component, the weaving process, and the Aspects to be applied. In a PCAS, sequence and switch structure are supported to achieve flexible adaptation on components. In PCAS, the adaptation process is depicted with only the ID of the selected Aspects. Full details of all related Aspects are kept in the Aspect Repository.

4.3.3 Aspect Oriented Adaptation Implementation

In the Aspect oriented adaptation implementation, the whole process is fully automatic with the support from the Aspect Generator and PCAS based Aspect Weaver. The aim of this stage is to perform component adaptation according to the pre-defined PCAS in the design stage.

Based on PCAS and the detailed Aspect definition, namely Aspect Frame (AF) in the Aspect repository, executable Aspect instances (AInsts) are generated by the Aspect Generator according to different AOP implementation specifications. As a result, platform variations are supported during Aspect generation. The input for the Aspect Generator is AF and the output is AInst.

The generated executable Aspects (AInsts) are finally weaved to the component by the PCAS based Aspect Weaver. A new adapted version of the component is then created through Aspect weaving. As existing AOP platforms that do not support complicated flow control such as switch in the weaving process, a pre-process is developed to enable process-based advanced weaving in the framework. Basically, during the pre-weaving

process, the Aspect weaver takes PCAS and the individual Aspects in AF format as input, and then generates the AInsts for the selected AOP platform with complex flow control implemented.

4.4 Aspect Generation Process

In the proposed Aspect model, the generation process of an Aspect can be viewed as the refinement of an AAF into an AF and finally into an AInst. If we define $\ni$ as refinement, then

$$\mathcal{AAF} \ni \mathcal{AF} \ni \mathcal{AInst}$$

During this process of refinement, the Aspects are refined from the most abstract form of Aspect, namely AAF, to the less abstract form of Aspect, namely AF, and finally to the concrete and executable form of Aspect, namely AInst. There are different variations such as functional variations, parameterization variations, and platform variations among these three forms of Aspect.

AAF is defined by a set of XML Schemas, which specify the structure of an Aspect. AF is an XML file with the above schemas populated with data. User interaction is needed when refining an AAF to an AF. AInst is the final concrete programming source code of the Aspect. The transformation from an AF to an AInst is done by the Aspect Generator which takes AF and Semantic Interpreter (SI) as input. The process is automatic.

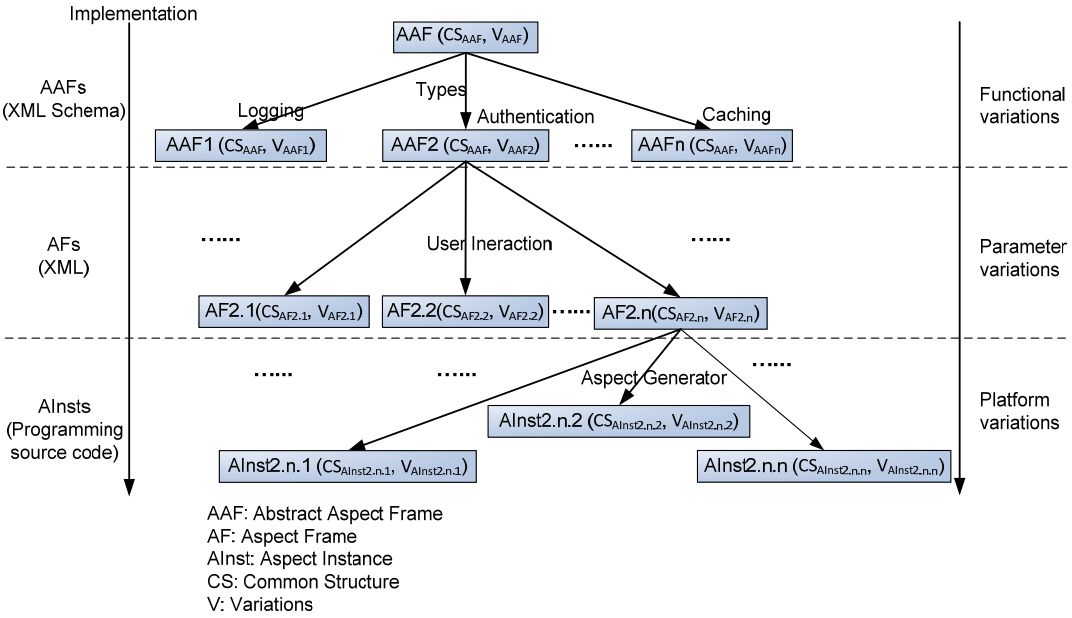


Figure 4.3 Product line based Aspect generation

As shown in Figure 4.3, a product family of the adaptation Aspects is designed to achieve high reusability. All Aspects are defined at three abstraction levels: AAF, AF, and AInst. These three abstraction levels of Aspects facilitate the reusability of adaptation Aspects as they realize different variations of these Aspects:

- Functional variations are achieved by a hierarchical classification of Aspects at AAF level. Aspects are classified into different types. With each type, an XML schema is used to define the structure of the selected Aspect.
- Parameter variations reflect the refinement of Aspects in specific adaptation circumstances, which is achieved by the determination of the values of CS and V in AF. Typically, user interaction is required to get parameter values. A software tool was developed to conduct the user interaction.
- Platform variations reflect the implementation of Aspects on particular AOP platforms, which is done by mapping the Aspect in AF format to AInsts in particular AOP platforms. The mapping process is fully automatic by using AOP platform and Aspect related Semantic

Interpreters. For each AOP platform, a set of corresponding interpreters need to be developed to deal with various types of Aspects in the specific AOP platform. The adaptation framework is extensible to new AOP platforms by developing appropriate interpreters.

With the basic definition in Figure 4.1, the tuple (CS, V) is refined across the three abstraction layers in both format and contents.

At AAF level, the tuple is an XML schema definition. CS consists of the elements to describe the common structure of an Aspect. V consists of the elements to declare the parameters of variations of the Aspect. All Aspects have the same CS at AAF level.

At AF level, conforming to the structure definition in AAF, the tuple is filled with the XML format data specific to an adaptation circumstance. CS will have the data such as the Aspect name, the signature of the particular component and method(s) on which the Aspect is to be applied. V will be filled with parameter values which determine the specific adaptation circumstances in which the Aspect is applied. Aspects at AF level are bound to specific Aspect types but are still AOP platform independent.

At AInst level, a concrete executable Aspect is generated automatically by the Aspect Generator. The common structure (CS), and variations (V) are finally mapped by a Semantic Interpreter into executable program code.

We define $\mathcal{T}$ as the transformation process between AAF, AF, and AInst. We define $\mathcal{S}_{\mathcal{T}}$ as the three states during the transformation. We define $\mathcal{A}_{\mathcal{T}}$ as the actions during the transformation. Then the transformation is defined as follows:

$\mathcal{T} = <S_T, \mathcal{A}_T>$, *where*

$S_T = \{\mathcal{AAF}, \mathcal{AF}, \mathcal{AInst}\}$

$\mathcal{A}_T = \{a1, a2\}$, *where*

$\overset{a1}{\rightarrow} = \{<\mathcal{AAF}, \mathcal{AF}>\}$,

$\overset{a2}{\rightarrow} = \{<\mathcal{AF}, \mathcal{AInst}>\}$

Hence,

$\mathcal{AAF} \overset{a1}{\rightarrow} \mathcal{AF} \overset{a2}{\rightarrow} \mathcal{AInst}$

The process of a1 is described in Figure 4.4:

```
1.  GET AAF FROM ASPECT REPOSITORY
2.  CHECK WHETHER AAF IS WELL-FORMED
3.  VERIFY CS PART OF AAF
4.  VERIFY V PART OF AAF
5.  IF AAF IS INVALID
6.      RETURN Ø
7.  END IF
8.  READ POINTCUT DEFINITION INFORMATION FROM USER
9.  VALIDATE POINTCUT DATA AGAINST ITS AAF DEFINITION
10. IF POINTCUT DATA IS INVALID
11.     RETURN Ø
12. END IF
13. READ ADVICE DEFINITION INFORMATION FROM USER
14. VALIDATE ADVICE DATA AGAINST ITS AAF DEFINITION
15. IF ADVICE DATA IS INVALID
16.     RETURN Ø
17. END IF
18. READ VARIATIONS DATA FROM USER
19. VALIDATE VARIATIONS DATA AGAINST ITS AAF DEFINITION
20. IF VARIATIONS DATA IS INVALID
21.     RETURN Ø
22. END IF
23. CREATE A NEW AF
24. FILL POINTCUT DATA INTO AF
25. FILL ADVICE DATA INTO AF
26. FILL VARIATIONS DATA INTO AF
27. RETURN AF
```

Figure 4.4 The transformation between AAF and AF

The process of a2 is described in Figure 4.5:

```
1.  GET AF FROM ASPECT REPOSITORY
2.  CHECK WHETHER AF IS WELL-FORMED
3.  VERIFY AF AGAINST ITS AAF DEFINITION
4.  IF AF IS INVALID
5.      RETURN Ø
6.  END IF
7.  READ TARGET AOP PLATFORM FROM USER
8.  CHECK SI IN REPOSITORY
9.  IF SI IS UNAVAILABLE
10.     RETURN Ø
11. END IF
12. GET POINTCUT DATA FROM AF
13. GET ADVICE DATA FROM AF
14. GET VARIATIONS DATA FROM AF
15. FILL POINTCUT DATA INTO PRE-DEFINED TEMPLATE IN SI
16. FILL ADVICE DATA INTO PRE-DEFINED TEMPLATE IN SI
17. FILL VARIATIONS DATA INTO PRE-DEFINED TEMPLATE IN SI
18. GENERATE AInst
19. RETURN AInst
```

Figure 4.5 The transformation between AF and AInst

4.5 The Adaptation Process

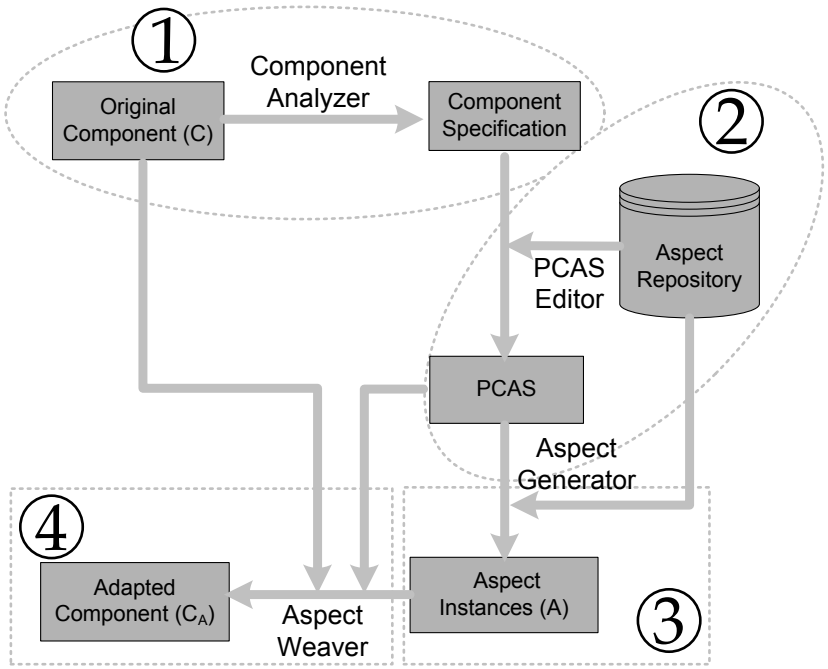


Figure 4.6 Adaptation process

Figure 4.6 describes the process of the proposed Aspect-oriented generative component adaptation, which involves the creation and application of suitable Aspects with the Aspect model defined in section 4.2. It is presumed that in a component based system, a component has been found potentially suitable to be reused in a new system and some mismatches still exist. Prior to performing component adaptation in GAIN to make the component reusable in a new system, the following criteria must be considered:

- Suitability of component for target application. According to the component interface and related documentation, the component is examined whether it is potentially suitable for the target system.
- Some mismatches still exist. If the component can potentially be used in the target system, the details of the mismatch problem must be provided as the foundation of the adaptation.
- Available Aspects. Whether the approach can be used to perform the adaptation also depends on the availability of the Aspects in the Aspect Repository.

The mismatch will be eliminated by applying Aspect-oriented adaptation to the original component. Initially, the component is analyzed with the Component Analyzer (refer to section 7.2.1), which analyzes the source of the component and extracts component specification information, e.g. class names and method signatures. The component specification will be used during the component adaptation.

Then based on the adaptation requirements, a PCAS will be created by selecting appropriate Aspects defined at the abstraction level of Aspect Frames (AF) and define the weaving process of these Aspects in the support tool called PCAS Editor (refer to section 7.2.2). The selection of Aspects is actually the process to determine functional variation of a specific adaptation. The composition of PCAS is supported by an interactive IDE called PCAS Editor, which supports both a graphical and XML source view of the PCAS.

A PCAS is an XML formatted document, which includes the details of component adaptation, such as the target component, the weaving process, and the AAFs to be applied. In a PCAS, sequence and switch structure are supported to achieve flexible adaptation on components. All Aspects used in the adaptation process are depicted with their unique ID. Full details of the Aspects can be retrieved from the Aspect Repository by using their ID as a key.

Based on PCAS and the details of the Aspect definition, namely Aspect Frame (AF) in the Aspect repository, executable Aspect instances (AInsts) are generated by the Aspect Generator according to different AOP implementation specifications. As a result, platform variation is achieved during Aspect generation. The inputs for the Aspect Generator are PCAS which tells 'which Aspects are needed' and AF which tells 'the details of each needed Aspects', and the output is an AInst.

The Aspect Repository is an embodiment of the proposed product line based Aspect model. Reusable Aspects are defined at three abstraction levels and kept in the repository as AAF, AF, and AInst. The reusable assets in the repository include both primitive Aspects and Aspect Frameworks (refer to section 6.4), which comes from the adaptation process in PCAS.

The generated executable Aspects are finally applied to the component by the Aspect Weaver. A new adapted version of the component is then created through Aspect weaving. Since current AOP platforms do not support complicated flow control such as switch in weaving process, pre-processing (refer to 6.5) is applied to enable process-based weaving in an existing AOP platform.

If we define the original component as C, the individual AInsts as A, and the adapted Component as C_A, then the weaving process is described in Figure 4.7 as below:

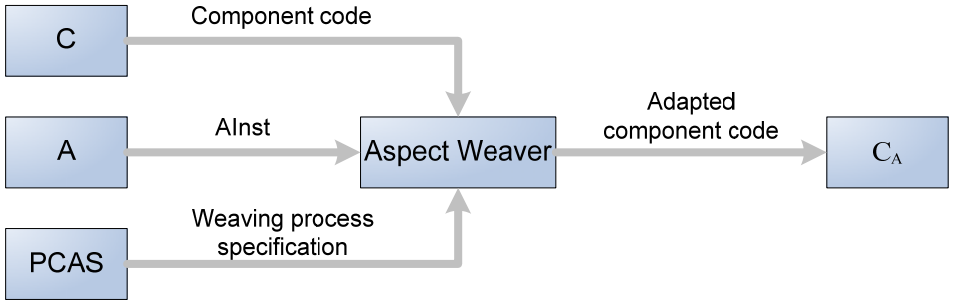


Figure 4.7 The Weaving process

The Aspect Weaver takes C, A, and PCAS as input and generates C_A as output. Depending on the target AOP platforms, various Aspect Weavers can be employed to perform the weaving process. For example, by default, the embedded Aspect Weaver will adapt C and generate C_A as standard programming source code. However, if the target AOP platform is an existing AOP platform, the pre-weaver is used to add advanced weaving processes to Aspects in the existing AOP platform. The details of pre-weaving process are introduced in section 6.5.

The adapted component must be tested before being deployed into the target system. All the test cases applied to the original component should also be applied to the adapted component to assure the correctness of the adapted component. Additionally, the adapted component should also be tested on its new features. The standard testing process including unit testing, integration testing, and system testing should be used to test the new features of an adapted component.

Chapter 5 Aspect Repository

This chapter describes the multi-layered reusable Aspect structure and the Aspect repository in detail.

5.1 Introduction

5.1.1 Reusable Aspects and Platform Independence

AOP is designed to deal with the crosscutting concerns, e.g. logging, authentication in software systems and write elegant code to avoid code tangling and scattering.

However, current AOP platforms are bound to specific programming languages, e.g. AspectJ for Java, AspectC++ for C/C++, aoPHP for PHP. Consequently, these AOP platforms produce a new crosscutting problem in AOP platforms while solving traditional crosscutting problems, particularly in a heterogeneous system. For example, as shown in Figure 5.1, in a heterogeneous distributed system including a Java based subsystem, a C# based subsystem, logging Aspects are required in all subsystems. In existing AOP platforms, developers have to develop the same logging Aspects in different AOP platforms, e.g. logging in AspectJ for Java based subsystem 1, logging in Aspect C# in C# based subsystem 2. Therefore, a new crosscutting concern - logging spreads over different subsystems.

In the GAIN framework, platform independent Aspects namely the Abstract Aspect Frame (AAF) and the Aspect Frame (AF) are developed to address the above problem. In other words, a new crosscutting concern – platform variation concern is considered and addressed in higher abstraction levels. For example, as shown in Figure 5.1, the platform independent abstract

logging Aspect can be designed in AAF and the platform independent concrete logging Aspect can be designed in AF. Platform specific Aspect code can be generated from AF by employing a set of Semantic Interpreters.

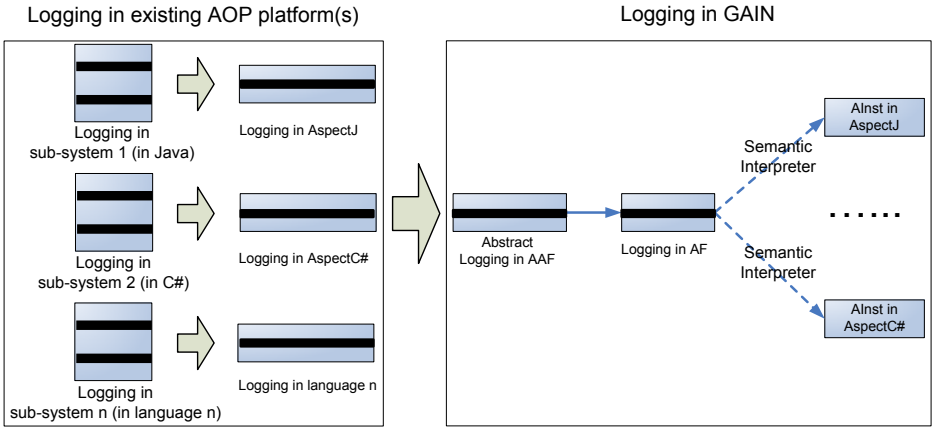


Figure 5.1 A comparison between existing AOP methods and GAIN

5.1.2 Aspect Repository

To achieve highly reusable and platform independent Aspects, a two-dimensional multiple-abstraction level Aspect Repository is developed. The reusability is improved via the following:

- The meta data of Aspects are saved in the Aspect Repository. The definition of Aspects is stored in the Aspect Repository as AAF. A type system is built by declaring hierarchical AAFs. AAF can be reused in different applications, e.g. new AFs can be created by providing detailed content information for each Aspect.
- The platform independent Aspects are saved as AFs in the Aspect Repository. The XML format concrete Aspects are saved in the Aspect Repository for further reuse. For each AF, different programming source code is generated for different AOP platforms by using different Semantic Interpreters.
- The platform specific Aspects can be generated automatically as AInsts by Aspect Generator and Semantic Interpreters.

With the support of three abstract levels of Aspects and the Aspect Framework in the Aspect Repository, the reusability of the framework is increased incrementally. First, all currently available AAFs and AFs in Aspect Repository are reusable in similar adaptation scenarios, e.g. AAFs can be reused directly without modification since they define Aspect types. AFs are reusable with the pre-defined structure populated by new content. Also, all newly designed AAFs and AFs required by a new adaptation scenario are kept automatically in the repository for further reuse. Moreover, the typical combination of various Aspects can also be saved in the repository as an Aspect Framework for further reuse as well. In summary, the more the framework is used, the more Aspects are available in the framework, the more reusable the framework becomes. The approach is able to address the non-functional requirements that need extra process, operations and resources to correct, e.g., performance and security. Currently, four Aspects have been added into Aspect Repository: Authentication, Logging, Database Connection Pool, Policy enforcement for new object creation, and Policy enforcement for the restriction to standard output methods.

5.2 Two-dimensional Multiple Abstraction Level Aspect Model

5.2.1 The Architecture

In the approach, the content of each Aspect such as Aspect name, Aspect type, which component(s) to be adapted, when to adapt, and how to adapt are saved in Aspects in three different layers. The adaptation knowledge (what) is captured in the Aspects and PCAS (Details are discussed in Chapter 6) and the adaptation is implemented via related tools such as Aspect Generator (Section 7.3.2) and PCAS-based Aspect Weaver (Section 7.3.3).

As shown in Figure 5.2, to achieve automated and precise adaptation, these Aspects are defined at three abstraction levels, i.e., Abstract Aspect Frame (AAF), Aspect Frame (AF), and Aspect Instance (AInst).

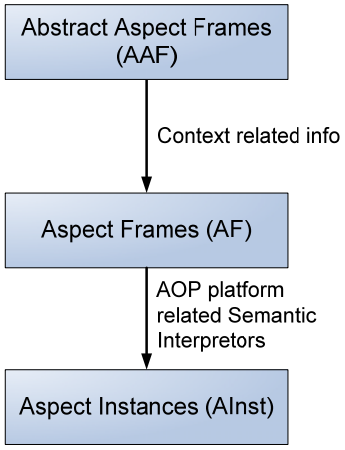


Figure 5.2 Three abstraction layers of Aspects

As shown in Figure 5.3, the three abstraction levels of Aspects facilitate the reusability of adaptation Aspects as they show different variations of these Aspects, including functional variations, parameter variations and platform variations.

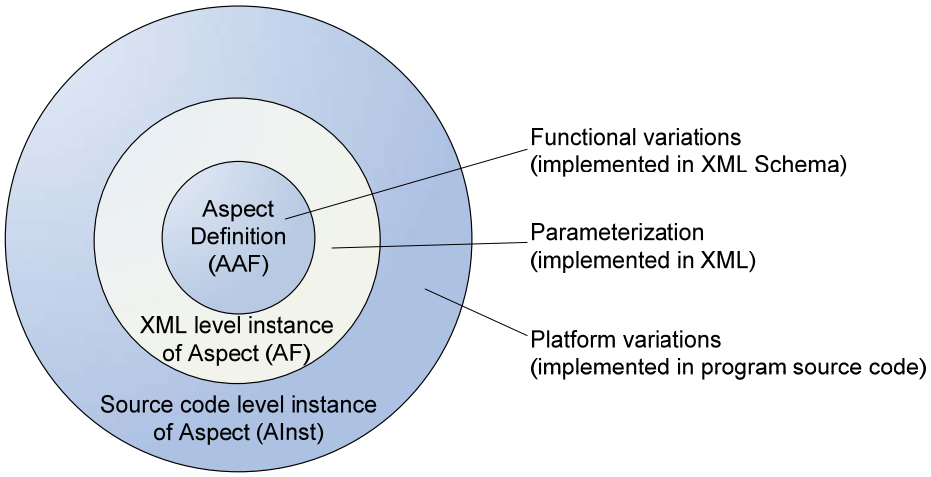


Figure 5.3 Variations in Aspects

Only AAFs and AFs are kept in the Aspect repository because all AInsts are generated from AFs.

At each level, a pair, namely (CS, V) is used to describe the Common Structure (CS) and the Variations (V). Common core assets are defined in CS and product specific variations are defined in V.

CS provides the basic information of an Aspect, e.g. which component to be adapted (target component), pointcut name. All Aspects have the same CS at AAF level no matter how different these Aspects are in functionality and AOP implementation platform because all Aspects have the same structure to define their basic characters.

V provides the information of the variations of different Aspects of the same or different Aspect types. For example, for an Aspect of logging type, an output file name must be provided; similarly an authentication Aspect must be supplied with an authentication type.

5.2.2 Abstract Aspect Frames

Abstract Aspect Frames are the most abstract level of the Aspect Repository. As XML schema files, an AAF is used to define the structure of different Aspects. According to the functionality, the AAF forms a hierarchical structure that shows functional variations of different Aspects. Adaptation Aspects are modelled into different types, for example, logging, caching, authentications, etc. Each Aspect type can be refined into a group of sub-types. For example, Aspects of authentication may consist of operating-system-based authentication and database-based authentication.

AAF is a hierarchical Aspect type system defined in XML schema format. This type hierarchy includes many levels of Aspect types and sub-types, which capture various functionalities of the adaptation Aspects. The Aspect Repository, assisted with the Aspect Manager, can adjust its Aspect type structure to accommodate Aspects with different functionalities as long as they are defined in the required AAF formats.

An example of database connection pool Aspect in AAF is shown in Figure 5.4. As discussed in section 4.2, each Aspect has two parts: CS and V, which are declared as CommonStructure and Variation elements in AAF. In AAF level, all Aspects are identical in their CommonStructure parts, which consist of PointCut and Advice elements. PointCut and Advice are used to describe the basic information of an Aspect such as pointcut and advice (refer to section 2.4.1.3 for the definition of pointcut and advice). For example, the name of the pointcut (Name element in AAF), when and where the pointcut happens (When, ReturnType, ClassName, MethodName, and Parameters element), when the advice code is applied (When element). On the other hand, Variation defines the unique elements for each Aspect. For example, Figure 5.4 shows the definition of Capacity, ExpireTime, CheckPoint, MaxIdleTime elements for database connection pool Aspect.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <!-- Each Aspect has two parts: CommonStructure and Variation -->
 <xs:element name="Aspect">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="CommonStructure" />
    <xs:element ref="Variation" />
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>
 <!-- Following is the definition of CommonStructure part -->
 <xs:element name="CommonStructure">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="PointCut" />
    <xs:element ref="Advice" />
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="PointCut">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="Name" />
    <xs:element ref="When" />
    <xs:element ref="ReturnType" />
    <xs:element ref="ClassName" />
```

```
      <xs:element ref="MethodName" />
      <xs:element ref="Parameters" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Advice">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="When" />
      <xs:element ref="PointCutName" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Name">
  <xs:complexType mixed="true" />
</xs:element>
<xs:element name="When">
  <xs:complexType mixed="true" />
</xs:element>
<xs:element name="ReturnType">
  <xs:complexType mixed="true" />
</xs:element>
<xs:element name="ClassName">
  <xs:complexType mixed="true" />
</xs:element>
<xs:element name="MethodName">
  <xs:complexType mixed="true" />
</xs:element>
<xs:element name="Parameters">
  <xs:complexType mixed="true" />
</xs:element>
<xs:element name="PointCutName">
  <xs:complexType>
    <xs:attribute name="ref" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>

<!-- Following is the elements in Variations part of AAF definition -->
  <xs:element name="Variation">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Capacity" />
      <xs:element ref="ExpireTime" />
    </xs:sequence>
    <xs:attribute name="type" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
<xs:element name="Capacity">
  <xs:complexType mixed="true" />
</xs:element>
<xs:element name="ExpireTime">
  <xs:complexType>
```

```
      <xs:sequence>
        <xs:element ref="CheckPoint" />
        <xs:element ref="MaxIdleTime" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="CheckPoint">
    <xs:complexType mixed="true" />
  </xs:element>
  <xs:element name="MaxIdleTime">
    <xs:complexType mixed="true" />
  </xs:element>
</xs:schema>
```

Figure 5.4 An example of AAF

5.2.3 Aspect Frames

Each AAF may have many Aspect Frames. As the instances of related AAFs, AFs are the second abstraction layer in an Aspect definition. AAFs are parameterized to simplify customization for particular applications. Compared with its AAF, an AF has the details of a concrete Aspect populated into it by assigning a value to the parameters. User interaction is required in the tool to provide necessary information for creating an AF from an AAF. All information gathered from the tool will be described in (CS, V) pair. Defined in XML format, AFs are independent from concrete AOP platforms.

An example of database connection pool Aspect in AF is shown in Figure 5.5. Within the CommonStructure tag, the common information of Aspects is provided, such as pointcut name (Name tag), when and where the pointcut happens (When, ReturnType, ClassName, MethodName, and Parameters tag), and when the advice code will be injected. On the other hand, the database connection pool Aspect specific variations are provided in related tags, e.g. Capacity, CheckPoint, and MaxIdleTime tag.

```
<?xml version="1.0" ?>
<Aspect name="OnlineTestingDBPoolAspect">
  <!-- Common Structure -->
  <CommonStructure>
    <PointCut>
      <Name>connectionOpen</Name>
      <When>call</When>
```

```
        <ReturnType>java.sql.Connection</ReturnType>
        <ClassName>java.sql.DriverManager</ClassName>
        <MethodName>getConnection</MethodName>
        <Parameters>String url,String username,String password</Parameters>
    </PointCut>
    <Advice>
      <When>around</When>
      <PointCutName ref="connectionOpen" />
    </Advice>
  </CommonStructure>
  <!-- Variations -->
  <Variation type="DBConnectionPoolOpen">
      <Capacity>50</Capacity>
      <ExpireTime>
         <CheckPoint>02:00:00</CheckPoint>
         <MaxIdleTime>86400</MaxIdleTime>
       </ExpireTime>
  </Variation>
</Aspect>
```

Figure 5.5 An example of AF

5.2.4 Aspect Instances

An AF is not executable until it is mapped onto a concrete AOP platform. The result of this mapping is a family of Aspect Instances based on various AOP platforms.

An AInst is executable and specific to a concrete AOP platform, and it reflects platform variations of an Aspect on different AOP platforms. The program to generate Aspect Instances from their AF is called Aspect Generator (section 7.3.2). The generation process is fully automatic.

By default, the AInst is generated in the same programming source language as the original component used. However, if a user elects to use an existing AOP platform as target platform such as AspectJ, GAIN also provides Semantic Interpreters to generate corresponding AInsts in the target platform.

An example of AInst in AspectJ is shown in Figure 5.6. Line 3 declares an DBPooling Aspect. The pointcut and advice are declared in line 7 and 9 respectively. The variation information such as the capacity of the pool and the maximum idle time for each connection are transformed from AF defined in Figure 5.5 to AInst in line 5. The detail of advice code is shown from line 10 to 16.

```
01 import java.sql.*;
02
03 public aspect DBPoolingAspect_1{
04
05 DBConnectionPool dbcp = new DBConnectionPool(50 ,"02:00:00",86400);
06
07 pointcut connectionOpen(String url, String username, String password) :
execution(java.sql.Connection java.sql.DriverManager.getConnection(String,String,String))
&& args(url, username, password);
08
09 java.sql.Connection around(String url, String username, String password) throws
SQLException : connectionOpen (url, username, password) {
10   Connection connection = dbcp.getConnection(url, username, password);
11
12   if(connection == null) {
13      connection = proceed(url, username, password);
14      dbcp.registerConnection(connection, url, username, password);
15   }
16   return connection;
17  }
18   public boolean isReachMaxCapicity() {
19      return dbcp.isReachMaxCapicity();
20   }
21 }
```

Figure 5.6 An example of AInst in AspectJ

5.2.5 Validation of new Aspects

The Aspects Repository can be extended by adding new Aspects. There are various ways to validate the newly added Aspects depending on the abstract level of each Aspect.

Validation of AAF

As the AAF reflects the design of a new Aspect type, one way to valiate the correctness of the AAF is to review the related documentation such as adaptation requirements, the functional / non-functional description of the Aspect. In addition, like the standard software development process, the testing result of AF/AIns is the other means to validate the correctness of AAF.

Validation of AF

Since all AFs are XML files, they can be validated against their schemas defined in AAFs. Also, as an XML document, AF files must be well-formed.

Validation of AInst

As AInsts are generated by the Aspect Generator automatically, the validation result of Aspect Generator also shows whether the AInsts are correct. The detailed testing of the tool is given in Section 7.4.

Chapter 6 Process Based Aspect Oriented Component Adaptation

This chapter introduces the process based aspect oriented component adaptation specification, the Aspect Framework, Aspect generation, and the Aspect weaving process.

6.1 Introduction

Current AOP platforms focus on applying the AOP idea to various programming languages, in other words, on the implementations of AOP languages, e.g. AspectJ for Java, AspectC++ for C/C++, aoPHP for PHP. In these platforms, the individual Aspects can only be weaved into target components one by one in sequential order.

However, in Aspect oriented software development, more advanced weaving process support is desirable. For example, in a software system, a developer wants to apply database connection pool Aspect to all components invoking database connection APIs. Moreover, for performance tuning purposes, the user also wants to log the usage of the database connection pool to monitor whether the maximum capacity of the pool has been reached. In this case, different Logging Aspects need to be used in different circumstances depending on the execution outcome of database connection pool Aspect. Then based on the logging information, the user can adjust the parameters of the connection pool to achieve the most appropriate performance with minimum resource usage. Another example is authentication in a student record system. As there are different types of roles in the system, e.g. system administrator, course tutor, and student, different operations are performed when users login in different roles. Also, the system needs to log various information when users login successfully or

fail to login. In this case, the system needs to log different information depending on the result of authentication Aspect.

Satisfying the above adaptation requirements often requires performing complex adaptations to component(s) with a set of generated Aspects applied to these components under a specially designed adaptation process containing flexible flow controls. However, implementing conditional execution of various aspects is not trivial since existing AOP platforms do not provide explicit language mechanisms for this purpose [117]. Therefore, Process-based Component Adaptation Specification (PCAS) is designed to fulfil these complex adaptation requirements.

6.2 Basic Entities of PCAS

Process-based Component Adaptation Specification (PCAS) is developed to describe the complicated Aspect-oriented adaptation details. Finally implemented in XML, PCAS is defined with the following tuple.

$\mathcal{PCAS} = (\mathcal{C}, \mathcal{A}, \mathcal{P})$, where

$\mathcal{C} = Component(s)$

$\mathcal{A} = Aspect(s)$

$\mathcal{P} = Process$

- **C** defines the component(s) which Aspects apply to. One or more component details are defined in C, e.g. component name, and the methods in component that will be involved in the Aspect weaving process.
- **A** defines the Aspects to be applied in the process, including the necessary information of the Aspects, such as the Aspect id, and Aspect type, so the full details of each Aspect can be retrieved from the Aspect repository when required during the weaving process.
- **P** defines weaving process control, including execution mode such as “Switch”, “Case”, and the guard condition(s).

6.3 Process-based Component Adaptation Specification (PCAS)

PCAS was developed to apply a set of Aspects to target component(s) under the designed advanced adaptation process. The structure of PCAS, defined in XML schema, is given in Figure 6.1.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="AOP-Process">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="AspectFramework" />
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>
 <xs:element name="AspectFramework">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="Apply-aspect" maxOccurs="unbounded" />
    <xs:element ref="Switch" maxOccurs="unbounded" />
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required" />
   <xs:attribute name="awhen" type="xs:string" use="required" />
   <xs:attribute name="returntype" type="xs:string" use="required" />
   <xs:attribute name="when" type="xs:string" use="required" />
   <xs:attribute name="joinpointcomponent" type="xs:string" use="required" />
   <xs:attribute name="joinpointmethod" type="xs:string" use="required" />
   <xs:attribute name="parameters" type="xs:string" use="required" />
   <xs:attribute name="sourcefile" type="xs:string" use="required" />
   <xs:attribute name="path" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>
 <xs:element name="Apply-aspect">
  <xs:complexType>
   <xs:attribute name="method" type="xs:string" use="required" />
   <xs:attribute name="class" type="xs:string" use="required" />
   <xs:attribute name="synchronized" type="xs:string" use="required" />
   <xs:attribute name="comment" type="xs:string" use="required" />
   <xs:attribute name="af_name" type="xs:string" use="required" />
   <xs:attribute name="aspect_level" type="xs:string" use="required" />
   <xs:attribute name="af_id" type="xs:string" use="required" />
   <xs:attribute name="aspect_type" type="xs:string" use="required" />
   <xs:attribute name="aspect_id" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>
 <xs:element name="Switch">
  <xs:complexType>
   <xs:sequence>
```

```
      <xs:element ref="case" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="when" type="xs:string" use="required" />
    <xs:attribute name="expr" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>
 <xs:element name="case">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="Apply-aspect" maxOccurs="unbounded" />
   </xs:sequence>
   <xs:attribute name="value" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 6.1 XML Schema of PCAS

The tags used in PCAS include:

- AOP-Process. The basic data of the weaving process is given in AOP-Process, such as the name and the namespace of the weaving process.
- AspectFramework. As various Aspects are used in the advanced weaving process, the combination of all Aspects and flow controls can be regarded as an “Aspect Framework”, which can be reused as a whole in similar situations. All the Aspects within the AspectFramework are applied to the same join point. The basic information of this join point is provided by a set of attributes including name, awhen, returntype, joinpointcomponent, joinpointmethod, parameters, sourcefile, and path. The details of “Aspect Framework” are given in section 6.4.
- Switch and case. Switch and case are used to support the switch structure in the weaving process. Working like a switch structure in many programming languages, an expression is needed in switch tag and depends on the value of this expression, different Aspects are applied in different case tags.

- Apply-aspect. This tag is used to add an individual Aspect to the weaving process. To perform this task, the following attributes are needed:
    - class and method: provide the basic information of the target component that will be adapted.
    - aspect_level: is used to describe whether an aspect is primitive Aspect or Aspect Framework.
    - aspect_type: is the type of the Aspect, e.g. Logging, Authentication, and Performance.
    - af_id: is the unique ID for AF. The full detail of each AF can be retrieved from Aspect repository.
    - af_name: is the name of each AF.
    - comment: is the descriptive text to explain the current Aspect.

With the support of PCAS and PCAS based Aspect Weaver (refer to section 7.3.3), a complex and flexible adaptation process can be achieved and more adaptation requirements can be fulfilled. As a result, the reusability of target component based systems is increased. The typical structure of PCAS definition is given in Figure 6.2 with the data detail omitted, while a full example of the definition is given in Chapter 8.

```
<?xml version="1.0"?>
<AOP-Process          name="xxx"
                      xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
<AspectFramework   name="xxx" …>
<Apply-aspect         class="xxx"
                      method="xxx"
                      aspect_id="xxx"
                      aspect_level="xxx"
                      aspect_type="xxx"
                      af_id="xxx"
                      af_name="xxx"
                      comment="xxx"/>
<Switch expr=”xxx” when=”xxx”>
<case value="xxx">
        <Apply-aspect …/>
</case>
<case value="xxx">
        <Apply-aspect …/>
```

```
</case>
</Switch>
</AspectFramework>
</AOP-Process>
```

Figure 6.2 Process based component adaptation specification

6.4 Aspect Framework

As similar combinations of Aspects are often used in different adaptation scenarios, Aspect Frameworks are developed in GAIN to improve further reuse. The reason for supporting Aspect Frameworks is that relatively fixed combination of various Aspects and the control flow connecting these Aspects to perform a specific adaptation task can be reused in the similar adaptation situations. For example, when a developer wants to optimize the use of database connection techniques from the simple JDBC / ODBC / OLEDB API invocations to a database connection pool, a set of Aspects is needed to get a satisfactory performance with minimum resource occupancy e.g. DBPooling Aspect and logging Aspects, which are organized in PCAS to perform this task. Therefore, the combination of a DBPooling Aspect, Logging Aspects, and the flow control can be saved in the Aspect Repository as an Aspect Framework, and then developers can reuse the same combination of Aspects in similar situations.

An example of an Aspect Framework is shown in Figure 6.3. The adaptation process described inside the “AspectFramework” tag can be regarded as an Aspect Framework, which is the combination of Authentication Aspect, Logging Aspects and Exit Aspect. Within the AspectFramework tag, the Authentication Aspect is applied first, then based on the value of expression “StudentSysAuth.getAuthenticationStatus()” defined in Switch tag in run-time, different Aspects will be applied. In this case, if the value is true, a Logging Aspect is applied, otherwise, the other Logging Aspect and an Exit Aspect are applied because that means the authentication failed. As the situation of

applying different Logging Aspects and Exit Aspect according to the result of Authentication is common, the Aspect Frameworks can be reused in different application systems.

```
<AspectFramework name="Auth_loggingOnStudentinfo"
                 sourcefile="Student.java"
                 path="d:\My_doc\Thesis\GAIN\Gain\Work\"
                 joinpointcomponent="Student"
                 joinpointmethod="launchApp"
                 when="call"
                 returntype="*"
                 parameters=".."
                 awhen="before">
<Apply-aspect    class="Student"
                 method="launchApp"
                 aspect_id="02"
                 aspect_level="Primitive"
                 aspect_type="Authentication"
                 af_id="3"
                 af_name="StudentSysAuth"
                 synchronized="false"
                 comment="check user name and password"/>
<Switch expr="StudentSysAuth.getAuthenticationStatus()" when="before">
<case value="true">
<Apply-aspect    class="Student"
                 method="launchApp"
                 aspect_id="01"
                 aspect_level="Primitive"
                 aspect_type="Logging"
                 af_id="1"
                 af_name="StudentSysLogging1"
                 synchronized="true"
                 comment="Log the access to the system"/>
</case>
<case value="false">
<Apply-aspect
                 class="Student"
                 method="launchApp"
                 aspect_id="01"
                 aspect_level="primitive"
                 aspect_type="Logging"
                 af_id="2"
                 af_name="StudentSysLogging2"
                 synchronized="true"
                 comment="log the rejection of access to the system"/>
<Apply-aspect
                 class="Student"
                 method="launchApp"
                 aspect_id="08"
                 aspect_level="Primitive"
                 aspect_type="Exit"
                 af_id="16"
```

```
                    af_name="StudentSysExit"
                    synchronized="false"
                    comment="Exit"/>
</case>
</Switch>
</AspectFramework>
```

Figure 6.3 An example of Aspect Framework

In summary, the use of Aspect Frameworks enhances the incremental reusability of the GAIN framework. The more usage of combinations of Aspects is saved in the repository as Aspect Frameworks, the higher reusability can be achieved.

6.5 Aspect Generation and the Weaving Process

After PCAS is gathered in the Aspect oriented design phase (refer to section 4.3.2), the weaving process is performed. As shown in Figure 6.4, first, the basic information for each Aspect can be retrieved from PCAS, and then the details of each Aspect can be accessed from the Aspect Repository. Then, depending on which target AOP platform is selected, the appropriate Semantic Interpreters are selected to generate the AOP platform specific AInsts. By default, the AInsts are generated in the same programming language as the original component used. However, if a developer specifies an existing AOP platform as the target platform, as current AOP platforms do not support advanced weaving process such as switch structure defined in PCAS, a pre-weaving process is applied to support the advanced weaving process. In this case, the AInsts are generated in the selected AOP platform with advanced weaving process support.

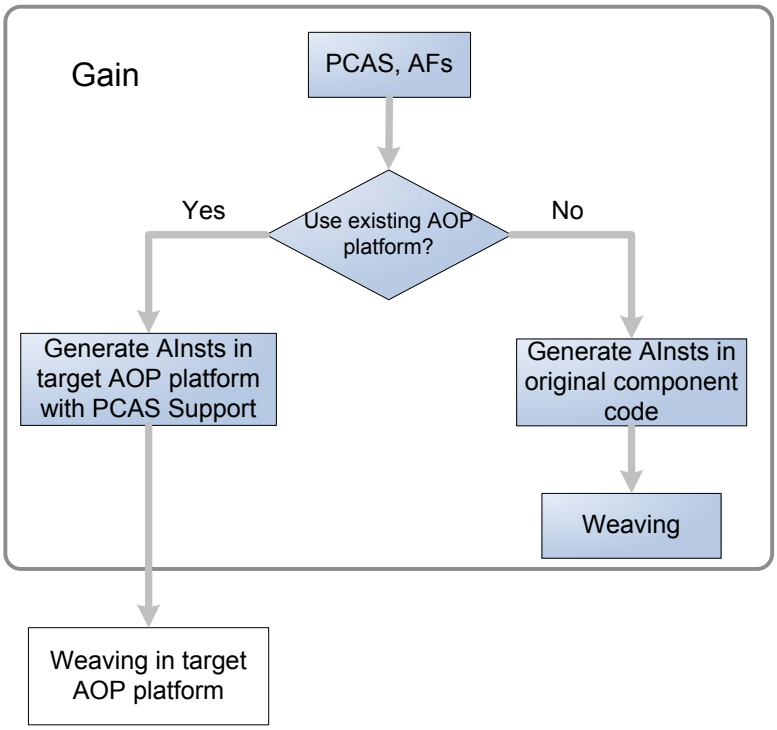


Figure 6.4 The generic weaving process

We define C as original component, $\mathcal{A}$ as required individual Aspects in AF level during the adaptation, $C_{\mathcal{A}}$ as the adapted component, $\mathcal{W}$ as the weaving process between $\{PCAS, C, \mathcal{A}\}$ and $C_{\mathcal{A}}$. We define $S_{\mathcal{W}}$ as the two states during the transition. We define $\mathcal{A}_{\mathcal{W}}$ as the action during the transition. Then the weaving process is defined as following:

$\mathcal{W} = <S_{\mathcal{W}}, \mathcal{A}_{\mathcal{W}}>$, *where*

$S_{\mathcal{W}} = \{\{PCAS, C, \mathcal{A}\}, C_{\mathcal{A}}\}$, *where*

$PCAS$ *is the process based component adaptation specification,*

C *is the component to be adapted,*

$\mathcal{A}$ *is the required individual Aspects in AF level during the adaptation,*

$C_{\mathcal{A}}$ *is the adapted component*

$\mathcal{A}_{\mathcal{W}} = \{a\}$, *where*

$\xrightarrow{a} = \{<\{PCAS, C, \mathcal{A}\}, C_{\mathcal{A}}>\}$,

Hence,

$\{PCAS, C, \mathcal{A}\} \xrightarrow{a} C_{\mathcal{A}}$

The process of a is described in Figure 6.5:

```
1.  READ PCAS
2.  VALIDATE PCAS
3.  IF PCAS IS INVALID
4.     RETURN Ø
5.  END IF
6.  READ TARGET AOP PLATFORM FROM USER
7.  GET ALL INDIVIDUAL  AFs FROM PCAS
8.  GENERATE AInsts from AFs IN GAIN
9.  IF TARGET AOP PLATFORM IS GAIN
10.    GENERATE INJECTION CODE IN GAIN ACCORDING TO PCAS
11.    ADD INJECTION CODE INTO ORIGINAL COMPONENT
12.    COMPILE ORIGINAL COMPONENT
13. ELSE
14.    GENERATE INJECTION CODE IN SELECTED PLATFORM  ACCORDING TO
    PCAS
15.    ADD INJECTION CODE INTO RELATED AInsts
16.    WEAVING ASPECTS INTO COMPONENT BY USING EXISTING ASPECT
    WEAVER
17.  ENDIF
```

Figure 6.5 The weaving process

PCAS is supported by the GAIN framework directly. Therefore, if the target AOP platform is not specified, the Aspect Generator generates AInsts in the same programming language as the original component used. Then the Aspect Weaver weaves the Aspects to original components directly according to the adaptation specification in PCAS. All structures and features supported in PCAS are reflected in the adapted components.

If a user selects an existing AOP platform such as AspectJ which does not support an advanced weaving process as a target platform, a pre-weaving technique is developed to support flexible flow control in PCAS during the weaving process.

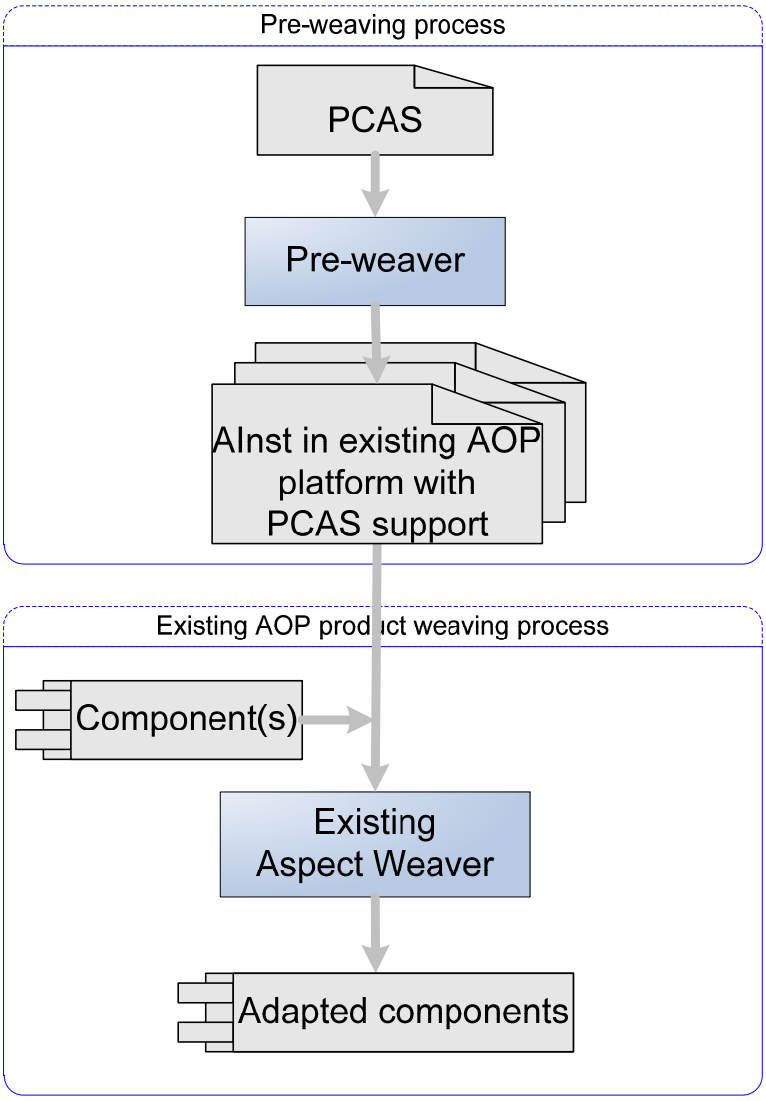


Figure 6.6 Weaving process in existing AOP platform in the approach

As shown in Figure 6.6, two steps are needed during the whole weaving process:

First, a pre-weaving process is developed to support the advanced weaving process. The pre-weaver takes PCAS as input, and then generate AInsts in target AOP platform with advanced weaving process support.

Then, as well as platform specific Aspects are generated, the Aspect Weaver in the existing AOP platform is used to weave these Aspects into the original components. For example, if AspectJ is selected as target AOP

platform, AspectJ weaver can be used to weave Aspects into the original components. As a result, the original components are adapted.

Chapter 7 The CASE Tool

This chapter introduces the system architecture, the tool in aspect oriented adaptation design/implementation, and the testing of the tool.

7.1 System Architecture

A CASE tool has been developed to facilitate the proposed approach and help developers performing automatic component adaptation. With this tool, firstly, component developers use the Component Analyzer to analyze component information and use the PCAS Editor to define the Aspect weaving process in a graphical interface. Secondly, they select candidate Aspects and fill in necessary details of CS and V for each Aspect with support from the Aspect Manager. Thirdly, the Aspect Generator and Semantic Interpreters generate AInsts for each Aspect automatically. Finally, according to the defined PCAS, Aspect Weaver will complete the Aspect weaving and generate final adapted components.

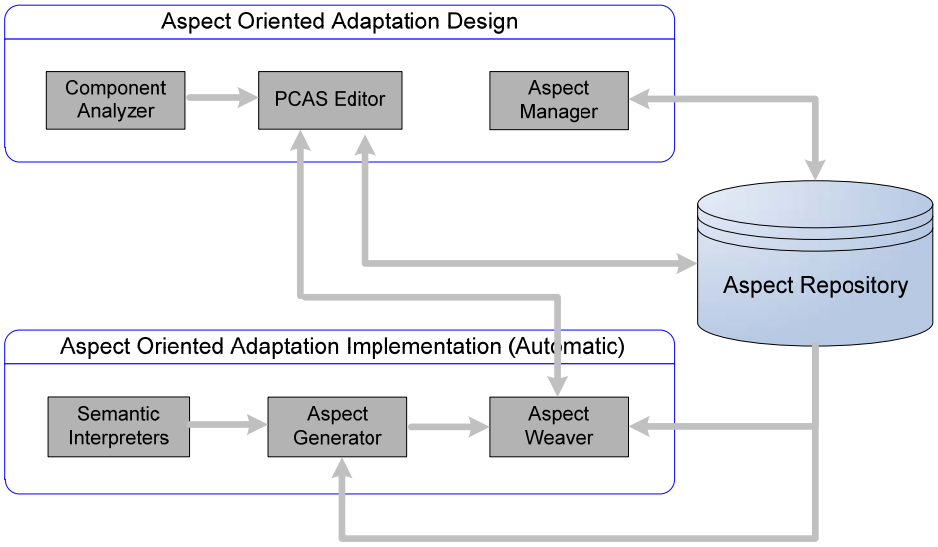


Figure 7.1 CASE tool in the framework

As given in Figure 7.1, the tool includes two parts: the Aspect Oriented Adaptation Design and the Aspect Oriented Adaptation Implementation respectively. In the design phase, the aim is to help developers generate a component adaptation specification – PCAS file. In the implementation phase, the aim is to perform Aspect generation and weave generated Aspects into original component. The tool consists of the following parts:

- **Component Analyzer**, which analyzes component and gets necessary information such as the class names and method signatures, for component adaptation.
- **PCAS Editor**, which provides an edit environment for PCAS both in graphical interface and at source code level.
- **Aspect Manager**, which supports the management of reusable Aspects in the Aspect Repository and the graphical view of different levels of Aspects.
- **Aspect Generator**: based on AFs and related Semantic Interpreters, concrete Aspect Instances are generated by Aspect Generator.
- **Semantic Interpreters**, which translate AFs to AInsts based on selected specific AOP platform and the type of Aspect(s).
- **Aspect Weaver**, which is used to weave Aspects generated by the Aspect Generator to original component according to the adaptation definition in PCAS.

7.2 Aspect Oriented Adaptation Design Phase

7.2.1 Component Analyzer

Before performing component adaptation, the required details of original components such as class name, method signatures must be provided. Therefore, a Component Analyzer was developed in the framework to analyze the original component(s) and get necessary information such as

the class names and method signatures. All this information is provided to the PCAS Editor during the creation of PCAS. Text based source code analysis and regular expressions are used to pick up the required information from the component source code.

Another thought is to use reflection mechanisms [136] of popular languages such as in Java [66], and in C# [69] to gather basic information of the component from its binary format. These techniques are essential while implementing the tool to support binary code adaptation in the future work (section 9.3.1).

7.2.2 PCAS Editor

The PCAS Editor provides an edit environment for PCAS both in a graphical interface and at XML level. In the PCAS Editor, the Aspect weaving process is defined by selecting various Aspects and putting these Aspects in sequence or switch structure in a graphical interface.

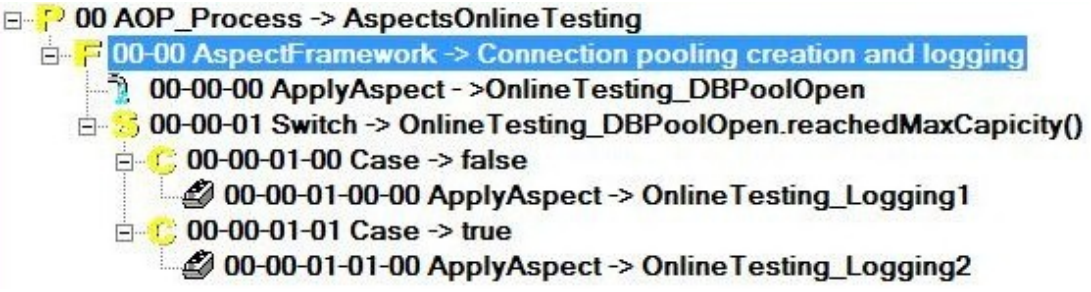


Figure 7.2 PCAS Editor in graphics view

As shown in Figure 7.2, all selected Aspects and flow controls are organised to build a PCAS. Software developers can add Aspects one by one into Aspect Framework. Then flexible control flows can be used to connect these selected Aspects, e.g. sequence or switch control flow. At the same time, the corresponding source code of the PCAS is generated automatically in the right part of the screen, as shown in Figure 7.3.

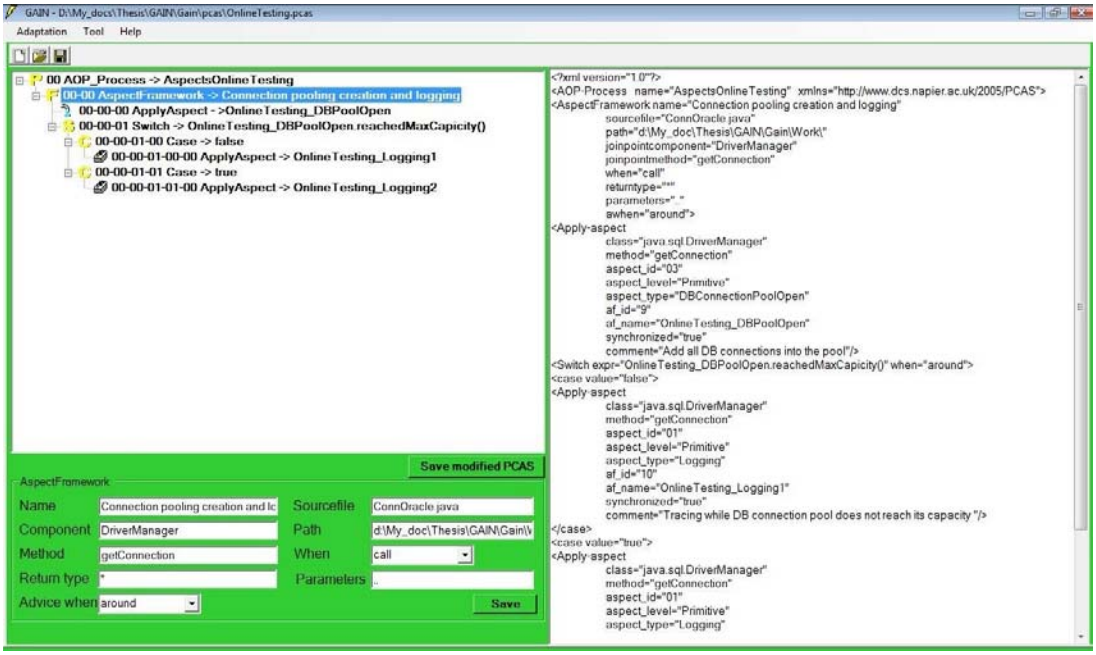


Figure 7.3 Main interface

On the other hand, experienced developers also can start creating PCAS in the source code view, and the corresponding graphical representation is also updated automatically.

For example, the PCAS in Figure 7.2 shows a PCAS that uses three Aspects: one database connection pool Aspect and two logging Aspects. Based on the status of database connection pool Aspect, one of these two logging Aspects will be selected.

In addition, as shown in Figure 7.4, the combination of primitive Aspects and control structures can be saved into the Aspect repository as an Aspect Framework. As shown in Figure 7.5, the Aspect Framework(s) can be re-loaded from Aspect repository to support further reuse.

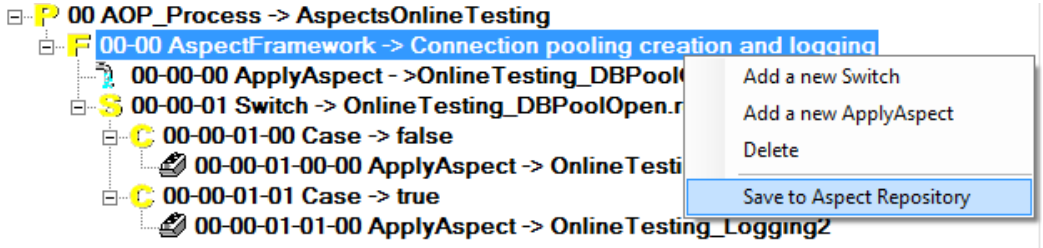


Figure 7.4 Save Aspect Framework to Aspect repository

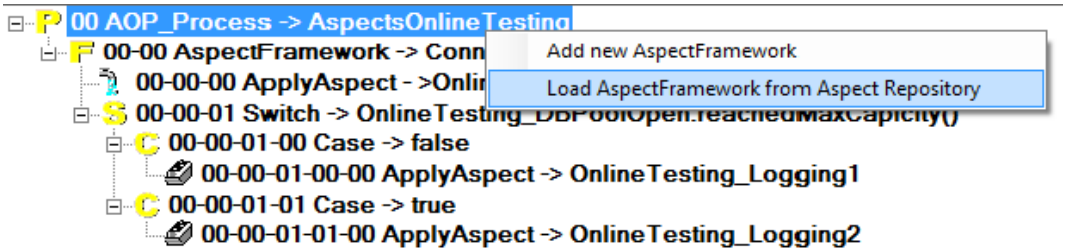


Figure 7.5 Load Aspect Framework from Aspect repository

7.2.3 Aspect Manager

The Aspect Manager supports the management of reusable Aspects in the Aspect Repository and the graphical view of different levels of Aspects. Aspects at two different levels, namely AAF, and AF can be created, removed, and edited in the Aspect Manager, either in the graphical user interface, or at XML level source code view.

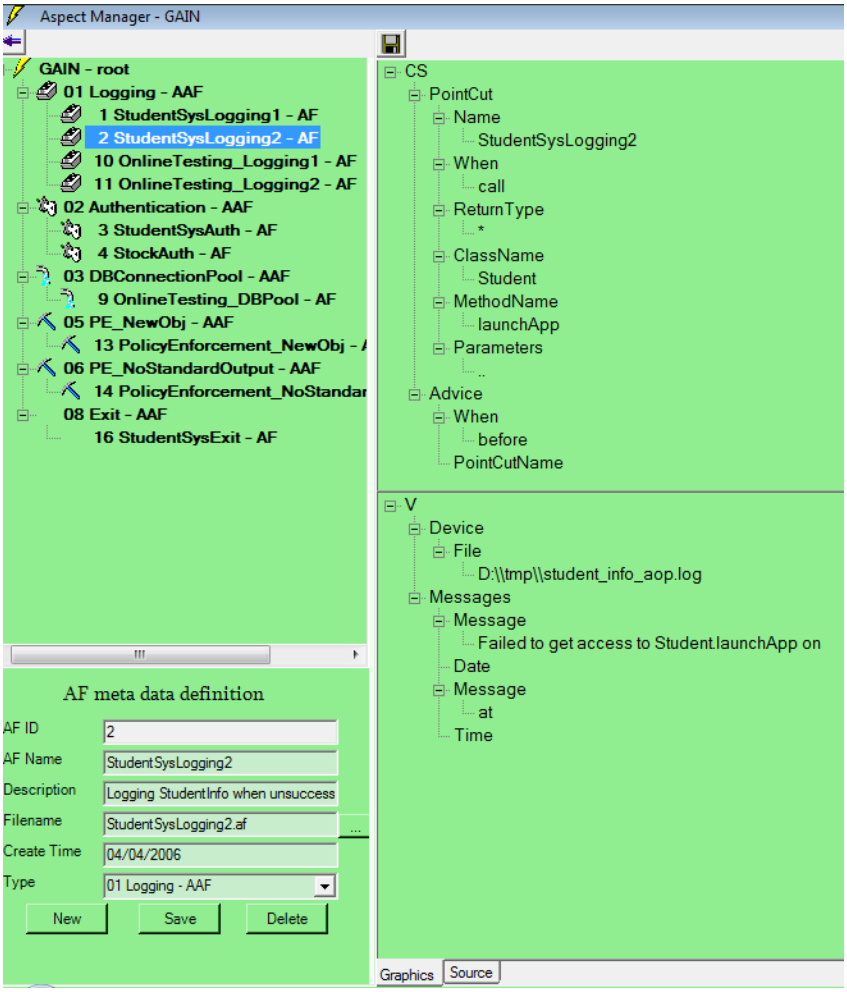


Figure 7.6 Aspect Manager

As shown in the top left of Figure 7.6, the hierarchy of various AAF and AF in Aspect Repository is represented as a tree in Aspect Manager. For example, there are six AAFs in Figure 7.6, e.g. Logging, Authentication, DBConnectionPool, PE_NewObj, PE_NoStandardOutput, and Exit. Zero, one or more AFs are listed in the tree as the XML level instance of each AAF. For example, in Figure 7.6, there are four AF level instances of Logging AAF, and two AF level instance of Authentication Aspect.

In the left bottom of Figure 7.6, AAF/AF meta data Edit window is designed to add new AFF/AF, edit the meta data of AAF/AF, and delete existing AAF/AF. For example, as shown in Figure 7.7, for each AAF, the description, the filename, icon, and the create time of AAF can be edited, and for each AF, the name, description, filename, create time, and type can be edited.

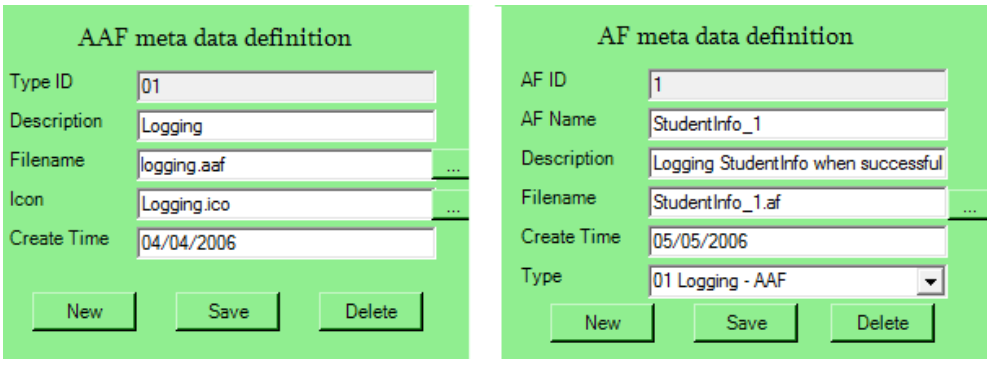


Figure 7.7 AAF/AF meta data definition window

For any AAF/AF, the full definitions are saved in XML schema / XML files, which can be edited by double-clicking related AAF/AF in the tree in left top window. The file for each AF is represented both by graphical and source code view. For example, in Figure 7.6 and Figure 7.8, the StudentSysLogging2 AF is shown both in graphical view and in source code view. On the other hand, the file content for each AAF is editable in source code view.

In both graphical and source code view, two parts are used to support software product line:

- Common Structure (CS): the core concerns of each AAF/AF are shown in this part.
- Variations (V): the variations of each AAF/AF are shown in this part.

Each AF can be used as a basis to generate a set of AInsts for different AOP platforms by employing Aspect Generator and appropriate Semantic Interpreters. The generation process is performed in a new Aspect generation window.

Figure 7.8 Source view of AF

As shown in Figure 7.9, all AOP platforms supported by the framework need to be selected from a listbox for AInst generating and the generating process is fully automatic. For example, the AspectJ code for logging Aspect from StudentSysLogging2 AF is given in Figure 7.9. The resulting AspectJ code will not be saved into Aspect repository because the AInsts can be re-generated automatically as required.

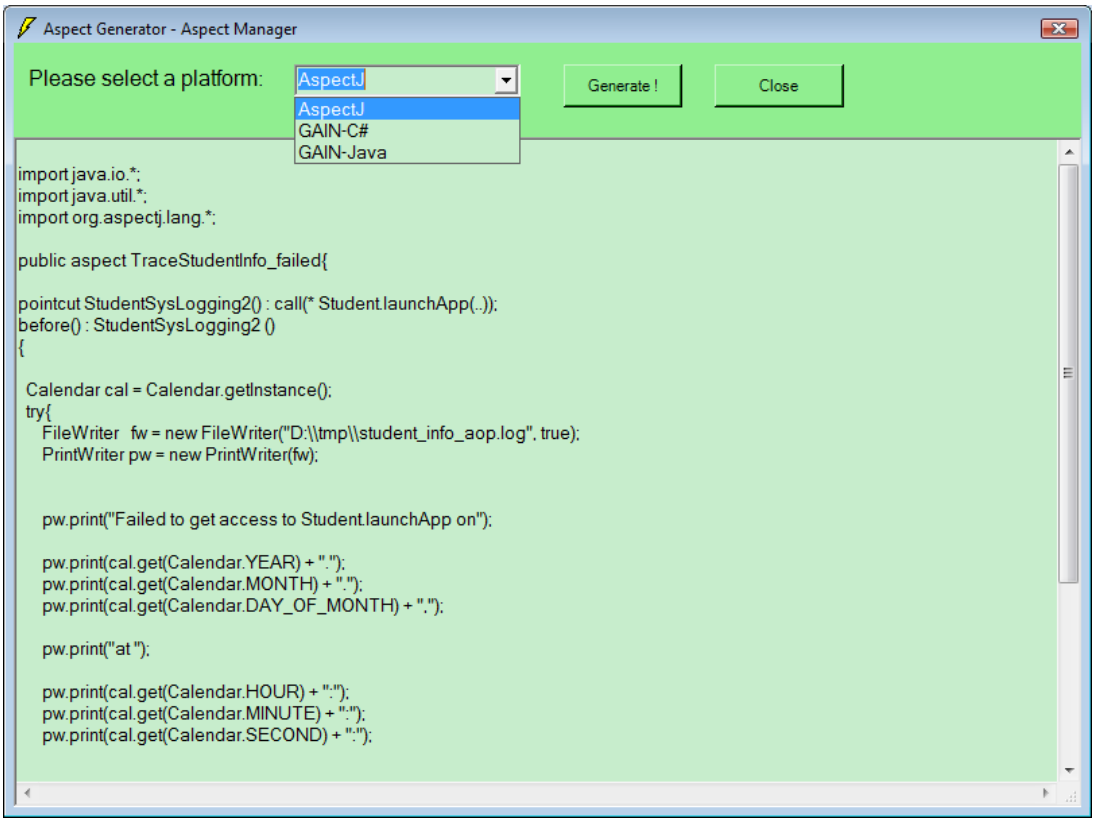


Figure 7.9 Aspect generation

7.3 Aspect Oriented Adaptation Implementation Phase

7.3.1 Semantic Interpreters

A set of XSLT (Extensible Stylesheet Language Transformations) [82] based Semantic Interpreters (SI) is developed to support the automatic code generation from AF to AInst based on selected specific AOP platform and the type of the Aspect. Semantic Interpreters allow the GAIN framework to perform concrete Aspect generation tasks automatically, and in a more reliable manner than this could be done manually. At the same time, developers can focus on the creation of platform independent Aspects (in

AF) during the adaptation process, rather than the specific syntax in a selected AOP platform.

XSLT is an XML-based language used for the transformation of XML documents into other XML, plain text, programming source code, or other format documents. The original document is not changed and a new document is created based on the content of the original one by an XSLT processor. XSLT is most often used to convert data between different XML schemas or to convert XML data into other format documents. The XSLT processor builds a source tree from the input XML document. It then starts by processing the source tree's root node, finding in the stylesheet the best-matching template for that node, and evaluating the template's contents.

In the GAIN framework, as shown in Figure 7.10, the basic idea of XSLT based code generation is to fill all content related information in AF and define processing logic in SI. Then the XSLT processor transforms the AF to AInst by filling contents in AF to pre-defined templates in SI. The XSLT processor reads all tags in AF and tries to find a matched template for each tag in the related SI. If the matched template is found in SI, then the code associated with that template is transformed to AInst. Having interpreted AF, SI outputs an AInst with content information declared in AF.

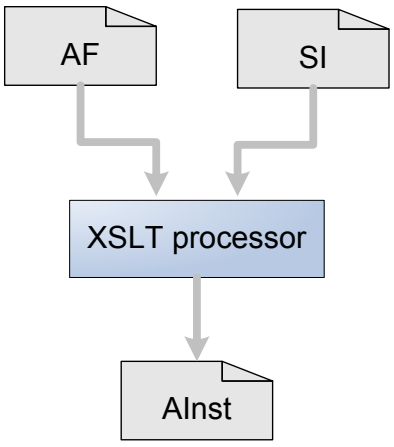


Figure 7.10 Semantic Interpreters and XSLT processing

The SI is called by the Aspect Generator during code generation. If there are m different supported AOP platforms and n different types of Aspect in the framework, there will be m × n different interpreters. The tool provides an interface to manipulate the Semantic Interpreters in the framework, e.g. viewing, editing, and deleting existing Semantic Interpreters, or adding a new SI into the framework to support more AOP platforms and Aspect types. Currently, the Semantic Interpreters are implemented to support Logging, Authentication, DB Connection pool, Policy enforcement, and Exit Aspects in AspectJ, Java, and C# respectively.

7.3.2 Aspect Generator

Based on AF and corresponding Semantic Interpreters, executable Aspect instances will be generated by Aspect Generator. Aspect Generator highly relies upon Semantic Interpreters to provide the transformations between AFs and AInsts. The delegation pattern [32] is used to implement the Aspect Generator. The class diagram is shown in Figure 7.11.

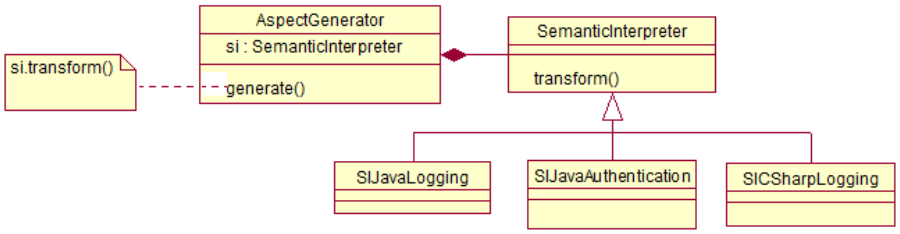


Figure 7.11 The implementation of Aspect Generator

The work flow of Aspect generation process is shown in Figure 7.12 below.

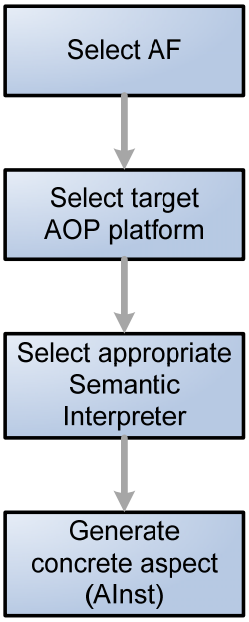


Figure 7.12 Flow chart of Aspect generator

After gathering PCAS in PCAS Editor, all related Aspects can be retrieved from PCAS as candidate Aspects for Aspect generation. Then after selecting the target AOP platform, AInsts can be generated accordingly.

7.3.3 Aspect Weaver

Aspect weaver is used to support the weaving process and generate final adapted components to fulfil the adaptation requirements defined in PCAS. The inputs for Aspect Weaver are PCAS and the original component and the output is the adapted component.

As discussed in section 6.5, if the target AOP platform is a traditional AOP platform, a Pre-weaver is also required to implement the advance flow control defined in PCAS. The input for Pre-weaver is PCAS and the outputs of Pre-weaver are AInsts with PCAS support enabled.

The Aspect Weaver is implemented by using the following technologies:

- Text based source code analysis. As the Aspect Weaver modifies the original component source code, the first task is to find the method(s) in the component to be adapted. In the Aspect Weaver, source code analysis is applied first to find the exact place to insert the AOP related code.
- Source code adaptation. According to the adaptation specification defined in PCAS, the source code modifications are performed.

The weaving process is fully automatic in the tool. Only PCAS and target platform need to be specified prior to starting the weaving process. After the execution of the weaving process, the selected component in PCAS is adapted.

7.4 Testing of the tool

To evaluate the correctness of the tool, unit testing, integration testing, and system testing were conducted respectively. The test result has shown that as the realization of the approach, the CASE tool is capable of performing non-functional component adaptation tasks in a semi-automatic manner.

Unit testing

Unit testing was performed to test the functionalities of each individual module of the tool before the code was added to the version control system (Microsoft Visual Sourcesafe). Unit testing consists of verifying the interfaces allow data to properly flow into and out of the object and that the underlying data structures are proper and sound for storing their intended data. The details are shown below:

- Component Analyzer was tested on its capability of retrieving basic information from components.
- PCAS Editor was tested on its capability of building PCAS either in graphical or in source code view. The saving and loading of Aspect Frameworks were also tested.

- Aspect Manager was tested on its capability of manipulating Aspects at AAF or AF level.
- Semantic Interpreters was tested with various Aspects such as Logging, Authentication, Database connection pool, Policy enforcer in AspectJ.
- Aspect Generator was tested on how it can generate AInst in AspectJ, standard Java and C#.
- PCAS based Aspect Weaver was tested on its capability of weaving and pre-weaving in AspectJ, standard Java, and C# respectively.

The detailed sample unit test cases are listed in Appendix E.1. As a demonstration, a unit test case is shown below:

Test case 1 (Testing getMethodList and getComponentname method of Component Analyzer)
Description: Test whether Component Analyzer provides basic information of a component.
Input: The file name of a component (ConnOracle.java).
Steps:
(1) Create a new object of ComponentAnalyzer class by passing a component file name (ConnOracle.java) as parameter to the constructor.
(2) Invoke getComponentName and display the return value.
(3) Invoke getMethodList method and display the return value.
Expected result:
Basic component information, e.g. component name(ConnOracle), method signatures (ConnOracle, executeQuery, executeUpdate).
Real result:
Classname: ConnOracle
MethodList: ConnOracle
executeQuery
executeUpdate
Status: passed.

Integration testing

Following unit testing and prior to the beginning of system testing, groups of individual modules are fully tested. As the goal of integration testing is to verify whether the modules can work together correctly and adequately, the Integration test cases focused on scenarios where one component is being called from another. In addition, the overall application functionality was also

tested to make sure the tool worked when the different components were brought together.

The detailed sample integration test cases are listed in Appendix E.2. As a demonstration, a test case is shown below:

Test case 1: Test integration between Aspect Manager and Aspect Generator.

Desired Functionality: The selected AF in Aspect Manager can be passed to Aspect Generator and based on this, Aspect Generator can generate an AInst from the selected AF.

Steps:
(1) Launch Aspect Manager
(2) Select an AF ("StudentSysLogging1") by clicking it
(3) Click on "Generate AInst" to launch Aspect Generator
(4) Select "AspectJ" as target AOP platform from the listbox
(5) Click on "Generate !"
(6) The source code of generated AInst should be shown in the textbox

Status: passed.

System testing

System testing was performed by doing three case studies to show the whole ability of the tool to perform component adaptation from aspect oriented design to aspect oriented implementation. The goal of system testing is to verify that the functions are carried out correctly. The sample test cases are presented in Appendix E.3.

Chapter 8 Case Studies

8.1 Introduction

Case studies have been undertaken to illustrate and evaluate the approach, in terms of its capability of building highly reusable Aspects across various AOP platforms and providing an advanced flow control of weaving process.

The aim of the case studies is to prove that the proposed approach and tool can deal with component adaptation in various software development architectures, such as client-server, browser-server architecture and Microsoft .NET framework. Various component-oriented platforms and programming languages are considered, such as JavaBeans, .NET components, and PHP.

8.2 Case Study 1: Student Record Management System

In this section, a case study has been undertaken to demonstrate that the approach and its tool are capable of performing Aspect-oriented component adaptation to desktop applications under client / server (C/S) architecture. The case study also illustrates how PCAS works and how to generate an executable Aspect by mapping through the different abstraction views of the Aspect in the framework.

8.2.1 Background

The approach has been applied to the construction of a student record system which was a coursework in the School of Software, Harbin Institute

of Technology [135] as a case study to test its correctness and capability. In the case study, a component is found from a previous system providing access to student information, which is shown in Figure 8.1. The component user has found the component is potentially suitable for the new application and wishes to integrate it into the new system. However, the component user wants to restrict the access to the student information only to the approved users, and wishes to monitor the access by logging the usage time.

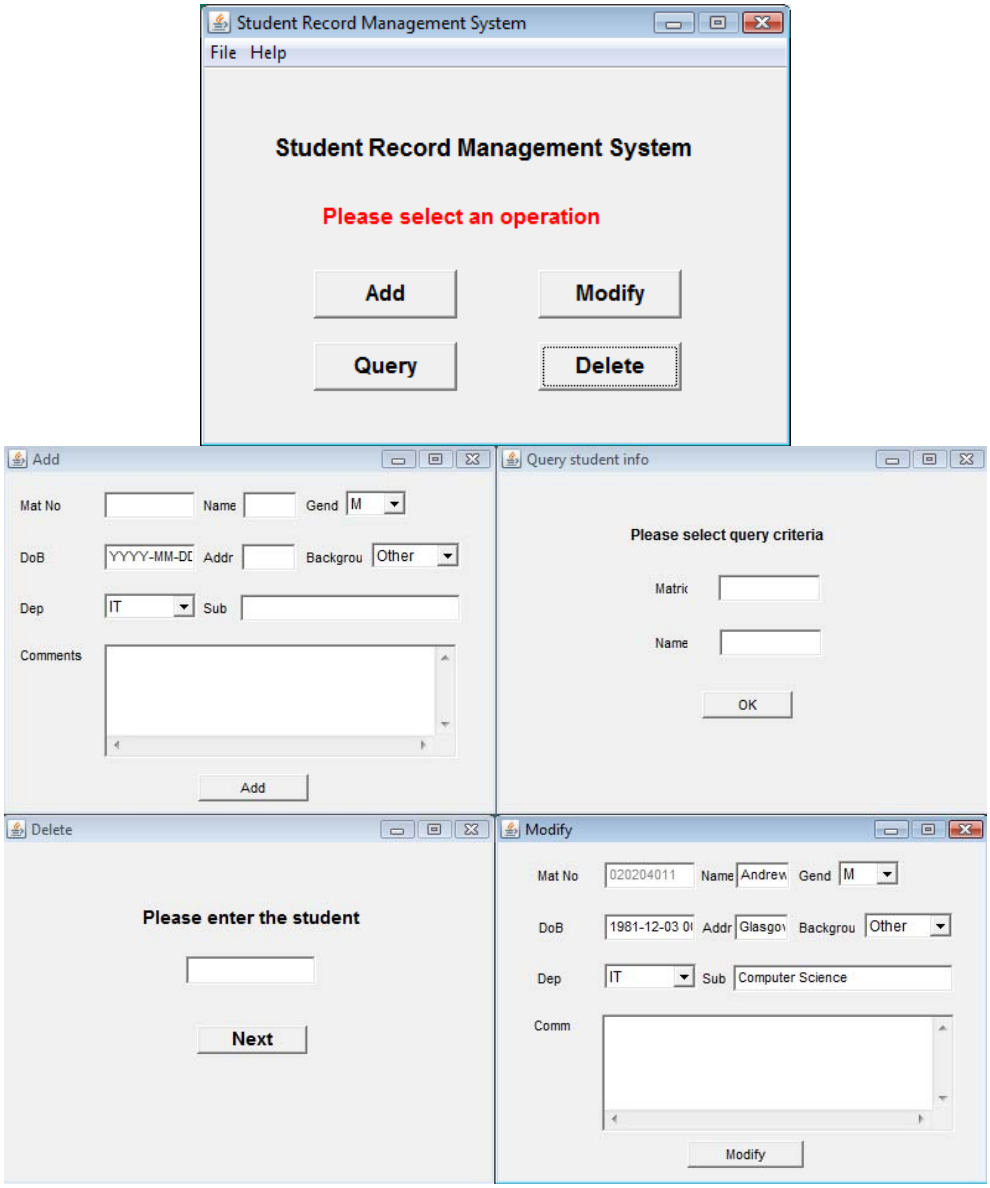


Figure 8.1 Student record management system

8.2.2 Solution

To respond to the above need, the component user plans to add authentication to this component prior to using it. According to the result of authentication, the detail of access activity to the component will be recorded.

An authentication Aspect is applied to this component first, followed by the application of corresponding logging Aspects depending on the result of authentication Aspect.

8.2.3 PCAS

The adaptation actions are then described in a PCAS shown in Figure 8.2. Four Aspects including an Authentication Aspect, two Logging Aspects, and an Exit Aspect and the flow control are given to provide a solution to the problem described in section 8.2.1. As the combination of these Aspects are applied to the same joinpoint, the basic information of the joinpoint and the component is provided in the attributes of AspectFramework tag. Within AspectFramework tag, an Authentication Aspect is applied to the joinpoint, then based on the outcome of the Authentication Aspect, different combinations of Aspects are applied. If the authentication is successful, a Logging Aspect is applied. Otherwise, another Logging Aspect and an Exit Aspect are applied.

```
<?xml version="1.0"?>
<AOP-Process   name="Aspects_on_StudentSys"
                    xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
<AspectFramework name="Auth_loggingOnStudentinfo"
                        sourcefile="Student.java"
                        path="d:\My_doc\Thesis\GAIN\Gain\Work\"
                        joinpointcomponent="Student"
                        joinpointmethod="launchApp"
                        when="call"
```

```
                    returntype="*"
                    parameters=".."
                    awhen="before">
<Apply-aspect       class="Student"
                    method="launchApp"
                    aspect_id="02"
                    aspect_level="Primitive"
                    aspect_type="Authentication"
                    af_id="3"
                    af_name="StudentSysAuth"
                    synchronized="false"
                    comment="check user name and password"/>
<Switch expr="StudentSysAuth.getAuthenticationStatus()" when="before">
<case value="true">
<Apply-aspect       class="Student"
                    method="launchApp"
                    aspect_id="01"
                    aspect_level="Primitive"
                    aspect_type="Logging"
                    af_id="1"
                    af_name="StudentSysLogging1"
                    synchronized="true"
                    comment="Log the access to the system"/>
</case>
<case value="false">
<Apply-aspect
                    class="Student"
                    method="launchApp"
                    aspect_id="01"
                    aspect_level="Primitive"
                    aspect_type="Logging"
                    af_id="2"
                    af_name="StudentSysLogging2"
                    synchronized="true"
                    comment="Log the rejection of access to the system"/>
<Apply-aspect       class="Student"
                    method="launchApp"
                    aspect_id="08"
                    aspect_level="Primitive"
                    aspect_type="Exit"
                    af_id="16"
                    af_name="StudentSysExit"
                    synchronized="false"
                    comment="Exit"/>
</case>
</Switch>
</AspectFramework>
</AOP-Process>
```

Figure 8.2 The PCAS for student record system

As shown in Figure 8.3, the specification is created with the PCAS Editor by finding appropriate AAFs, and putting these AAFs into an adaptation process. Functional variation of adaptation is implemented through the composition of various AFs.

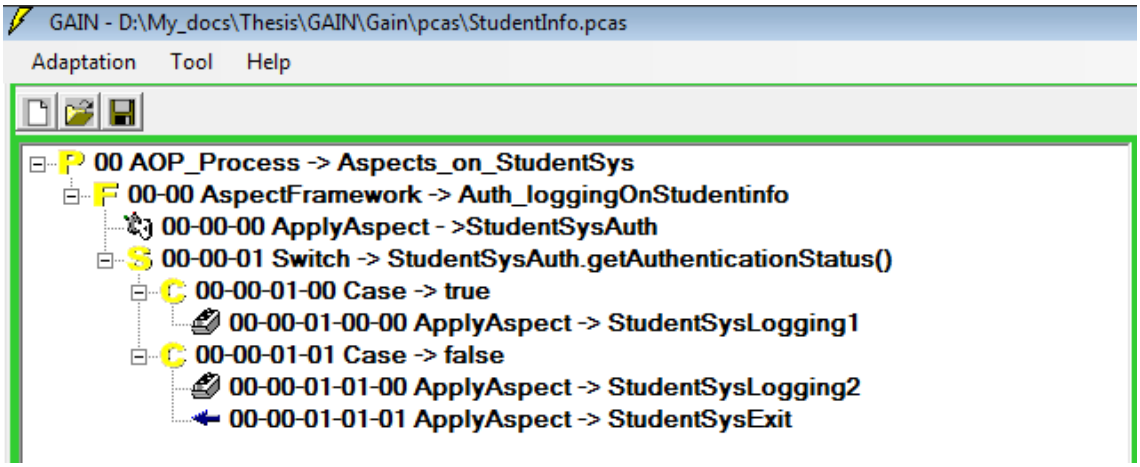


Figure 8.3 The PCAS for student record system

8.2.4 Aspects

The specification in PCAS is at an overview level and does not contain the details of individual Aspects. Developers need to provide parameter values for each Aspect. Common AFs can be saved into the Aspect Repository for further reuse. In this example, four AFs will be generated: AF for authentication, AF for logging if authenticated successfully, AF for logging if authenticated unsuccessfully, and AF for Exit. Due to the structural similarity of AFs of different Aspects, only the AF for logging if authenticated successfully is given in Figure 8.4 as an example. The basic information of the Logging Aspect such as Name of the pointcut is provided within “CommonStructure” tag and the specific information about the Logging Aspect is given within “Variation” tag.

```
<?xml version="1.0" ?>
<Aspect name="TraceStudentInfo_Successful">
 <!-- Core asset -->
 <CommonStructure>
  <PointCut>
    <Name>StudentSysLogging1</Name>
    <When>call</When>
    <ReturnType>*</ReturnType>
    <ClassName>Student</ClassName>
```

```
      <MethodName>launchApp</MethodName>
      <Parameters>..</Parameters>
    </PointCut>
    <Advice>
      <When>before</When>
      <PointCutName ref="StudentSysLogging1" />
    </Advice>
  </CommonStructure>

  <!-- Variations -->
  <Variation type="Logging">
    <Device>
      <File>D:\\tmp\\student_info_aop.log</File>
    </Device>
    <Messages>
      <Message>Succeed to get access to Student.launchApp on</Message>
      <Date/>
      <Message>at </Message>
      <Time/>
    </Messages>
  </Variation>
</Aspect>
```

Figure 8.4 Logging Aspect in AF level in student record system

From AFs, the Aspect Generator generates Aspect instances (AInsts) that are specific to a selected AOP platform. The generated AInst of the AF in Figure 8.4 is given in Figure 8.5. The code from line 1 to 9 and line 22 to 26 corresponds to “CommonStructure” part in Figure 8.4. The code from line 10 to line 21 corresponds to “Variation” part in Figure 8.4.

```
01 import java.io.*;
02 import java.util.*;
03 import org.aspectj.lang.*;
04 public aspect TraceStudentInfo_Successful{
05 pointcut StudentSysLogging1() : call(* Student.launchApp(..));
06 before() : StudentSysLogging1 ()
07 {
08  Calendar cal = Calendar.getInstance();
09  try{
10      FileWriter   fw = new FileWriter("D:\\tmp\\student_info_aop.log", true);
11      PrintWriter pw = new PrintWriter(fw);
12      pw.print("Succeed to get access to Student.launchApp on");
13      pw.print(cal.get(Calendar.YEAR) + ".");
14      pw.print(cal.get(Calendar.MONTH) + ".");
15      pw.print(cal.get(Calendar.DAY_OF_MONTH) + ",");
16      pw.print("at ");
17      pw.print(cal.get(Calendar.HOUR) + ":");
18      pw.print(cal.get(Calendar.MINUTE) + ":");
```

```
19      pw.print(cal.get(Calendar.SECOND) + ":");
20      pw.println();
21      pw.close();
22  }catch(Exception e) {
23      System.out.println("Error occured: " + e);
24  }
25 }
26}
```

Figure 8.5 AInst for Logging Aspect

The Aspect Weaver weaves the generated AInsts into the original component according to the PCAS. The final adapted component source code is invisible to the developer. By deploying the adapted component, the new application is built and released to the targeted user.

8.2.5 Summary

In conclusion, this case study has shown that the approach and the related tool are capable of performing Aspect-oriented component adaptation to desktop applications under a client / server architecture. Also, PCAS has been used to describe the adaptation requirements and then the adaptation actions have been taken according to the adaptation specification defined in PCAS.

8.3 Case Study 2: On-line Testing System

This case study applies the approach to a typical browser / server (B/S) architecture application – Online-testing system implemented in Java 2 Enterprise Edition (J2EE). The aim of this case study is to illustrate the ability of the approach to perform Aspect-oriented component adaptation to B/S applications.

8.3.1 Background

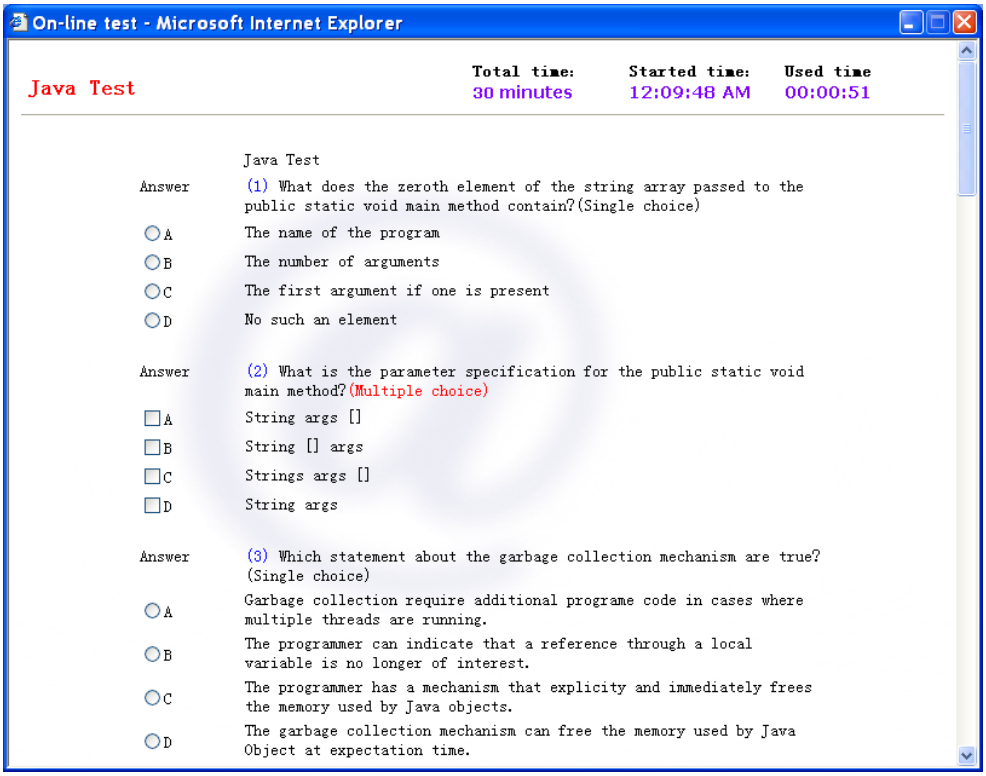


Figure 8.6 On-line testing component

The case study deals with an on-line testing component, which was developed by Oriental Standard [78], a software company. This web-based testing system is based on J2EE technology. It has four major components: the testing preparation system, the testing system, the auto-marking system, and the administration system.

- The testing preparation system is used to manage question storage and randomly generate testing papers. The question repository contains test questions, possible answers, the question types such as single choice or multiple choices, and the topics, etc. The repository is open to teachers, allowing them to add questions and answers. All testing papers are generated randomly by the system administrator or teachers.

- As shown in Figure 8.6, the testing system has a web-based testing interface for students, which include a client side interface and whole testing control, e.g. time control.
- The auto-marking system is designed to mark the test result automatically.
- The administration system is used to register students with the system, and manage login accounts.

The IT department of a university planned to build their browser / client based online assessment system and bought the component for integration as part of the system as a core part of the students' performance evaluation activities. However, they identified that the large student numbers would impose a heavy access load and make the system performance poor.

8.3.2 Solution

To meet the above reuse requirements, the development team decided that prior to integration of the online testing component three actions should be done to adapt the component:

First, a database connection pool is to be introduced to the online testing system to improve system performance.

Second, logging is used to monitor the usage and status of the connection pool.

Finally, based on the logging information, the connection pool is tuned to achieve the best performance with reasonable resource cost such as memory consumption, by constantly adjusting the parameters, including the capacity of connection pool and the expire time of each connection instance.

8.3.3 PCAS

To implement the above adaptation actions, the following three Aspects are applied to the component, namely database connection pool, logging if connection pool reaches its maximum capacity, and logging if connection pool does not reach its maximum capacity.

These adaptation actions are then described in a PCAS shown in Figure 8.7. The specification is created with the PCAS Editor by finding appropriate AFs, i.e., either primitive types or Aspect Frameworks. Finally, these AFs are put into an adaptation process. Functional variation of adaptation is implemented through the composition of Aspects in PCAS. Within Figure 8.7, the join point related information is provided by a set of attributes of AspectFramework tag. Then within AspectFramework tag, a DBConnectionPool Aspect is applied to improve the performance of target component. And then different Logging Aspects are applied depending on whether the maximum capacity of the pool is reached.

```
<?xml version="1.0"?>
<AOP-Process   name="AspectsOnlineTesting"
                    xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
<AspectFramework name="Connection pooling creation and logging"
          sourcefile="ConnOracle.java"
          path="d:\My_doc\Thesis\GAIN\Gain\Work\"
          joinpointcomponent="DriverManager"
          joinpointmethod="getConnection"
          when="call"
          returntype="*"
          parameters=".."
          awhen="around">
<Apply-aspect
          class="java.sql.DriverManager"
          method="getConnection"
          aspect_id="03"
          aspect_level="Primitive"
          aspect_type="DBConnectionPool"
          af_id="9"
          af_name="OnlineTesting_DBPool"
          comment="Add all DB connections into the pool"/>
<Switch expr="OnlineTesting_DBPool.reachedMaxCapicity()" when="around">
<case value="false">
<Apply-aspect
```

```
            class="java.sql.DriverManager"
            method="getConnection"
            aspect_id="01"
            aspect_level="Primitive"
            aspect_type="Logging"
            af_id="10"
            af_name="OnlineTesting_Logging1"
            comment="Tracing while DB connection pool does not reach its capacity "/>
</case>
<case value="true">
<Apply-aspect
            class="java.sql.DriverManager"
            method="getConnection"
            aspect_id="01"
            aspect_level="Primitive"
            aspect_type="Logging"
            af_id="11"
            af_name="OnlineTesting_Logging2"
            comment=" Tracing while DB connection pool reaches its capacity "/>
</case>
</Switch>
</AspectFramework>
</AOP-Process>
```

Figure 8.7 The PCAS for On-line Testing system

8.3.4 Aspects

The specification in PCAS is at an overview level and does not contain the details of individual Aspects. Developers need to provide parameter values for each Aspect. Common AFs can be saved into Aspect Repository for further reuse. In this example, three AFs will be generated for each of the above Aspects accordingly. Due to the structural similarity of AFs of different Aspects, only the AF for the DB connection pool Aspect is given in Figure 8.8 as an example. Within Figure 8.8, the basic information of pointcut and advice is given within CommonStructure tag and the DB connection pool Aspect specific information, such as the capacity, the checkpoint and the maximum idle time of the pool are given within Variation tag.

```
<?xml version="1.0" ?>
<Aspect name="OnlineTestingDBPoolAspect">
 <!-- Common Structure -->
 <CommonStructure>
   <PointCut>
     <Name>connection</Name>
     <When>call</When>
```

```
        <ReturnType>java.sql.Connection</ReturnType>
        <ClassName>java.sql.DriverManager</ClassName>
        <MethodName>getConnection</MethodName>
        <Parameters>String url,String username,String password</Parameters>
    </PointCut>
    <Advice>
      <When>around</When>
      <PointCutName ref="connection" />
    </Advice>
  </CommonStructure>

  <!-- Variations -->
  <Variation type="DBConnectionPool">
      <Capacity>50</Capacity>
      <ExpireTime>
         <CheckPoint>02:00:00</CheckPoint>
         <MaxIdleTime>86400</MaxIdleTime>
       </ExpireTime>
  </Variation>
</Aspect>
```

Figure 8.8 An AF of DB connection pool

From the above AF in Figure 8.8, Aspect Generator generates an AInst that is specific to a selected AOP platform. The generated AInst in AspectJ of the AF in Figure 8.8 is given in Figure 8.9. In Figure 8.9, the AOP platform specific Aspect instance is generated by putting the parameters giving in Figure 8.8 into the pre-defined template in Semantic Interpreter. For example, the capacity, checkpoint, and maxidletime within Variation tag in Figure 8.8 is transformed to line 3 in Figure 8.9. The basic information of pointcut and advice giving in CommonStructure tag in Figure 8.8 is transformed to line 4 to 7 in Figure 8.9.

```
01 import java.sql.*;
02 public aspect OnlineTestingDBPoolAspect{
03   DBConnectionPool dbcp = new DBConnectionPool(50 ,"02:00:00" ,86400);
04   pointcut connection () : call( java.sql.Connection
05       java.sql.DriverManager.getConnection(String url,String username,String password));
06 java.sql.Connection around(String url,String username,String password) :
07      connection (String url,String username,String password) throws SQLException
08 {
09   Connection connection = dbcp.getConnection(url, username, password);
10   if(connection == null) {
11         connection = proceed(url, username, password);
12         DBConnectionPool.registerConnection(connection, url, username, password);
13   }
14   return connection;
```

```
15    }
16 }
```

Figure 8.9 AInst of DB connection pool Aspect

The Aspect Weaver weaves the generated Aspect instances into the original component according to the PCAS. The final adapted component source code is invisible to the developer. By deploying the adapted component, the targeted users' requirements regarding to system performance is fulfilled.

8.3.5 Summary

In conclusion, this case study has shown that the approach and its tool are capable of adapting enterprise level applications under B/S architecture. This case study has also illustrated that the advanced flow control of weaving process can be carried out and the various abstraction views of Aspects can be implemented in the proposed framework. In addition, since AF is platform-independent, based on the other case study, the AF in Figure 8.8 can be used in another AOP platform, such as aoPHP [70], by employing appropriate Semantic Interpreters.

8.4 Case Study 3: Company Policy Enforcement

This case study applies the approach to various software applications crossing different programming languages and platforms. The aim of this case study is to illustrate the ability of the approach to develop highly reusable platform independent Aspects in AF format, to transform these Aspects to platform specific Aspects in AInst format automatically, and to apply them to various platforms and programming languages to respond to the policy enforcement requirement of a corporation.

8.4.1 Background

"Policy enforcement is a mechanism for ensuring that system components follow certain programming practices, comply with specified rules, and meet the assumptions." [98](p179). For example, in Java, it is highly

recommended that Swing components should not be used in EJBs because EJBs are intended to be business functionality specific server extensions, not clients with user interfaces [64]. If there is no policy enforcement; problematic code may not be detected during the development phase and the error may exist in the deployed system. The other example is the ubiquitous recommended rules crossing various programming languages. For example, although not being recommended for use, the “Goto” statement is supported by most popular programming languages, such as Java, C#, C, C++, and Visual Basic, etc. Software firms may want to prohibit the use of “Goto” statements in their software applications.

In summary, software firms may want to develop a set of reusable, general-purpose policies so that they can carry them to other projects. In addition, policies also help developers who are new to a technology – working as a mentor [98]. In the maintenance phase, policy enforcements ensure developers that the modifications to source code do not violate the existing policies.

8.4.2 Problem Statement

No programming language is perfect. Each language has its advantages and disadvantages. However, experienced developers summarise the “best programming practices”, which give the tips, tricks, do’s and don’ts of a specific programming language or universal rules to any programming languages. On the other hand, no software developer is perfect. Junior developers may lack relevant experience. Even senior developers may not be able to concentrate on coding for a long time and introduce bugs to the system.

SuperDev is a software firm providing enterprise management software and e-Business application solutions. SuperDev is engaged in developing and selling enterprise management software and e-Business application software, and middleware of e-Business and e-government platforms for

enterprises or government. Based on the code review, bugs reports, and development reports, SuperDev realizes that a set of common and reusable polices need to be developed and enforced to all software projects crossing various platforms and programming languages. As a result, the time of delivering software projects and the cost will be reduced dramatically.

8.4.3 Solution

To address the above problems, different groups of policies are defined in different platforms and programming languages as reusable assets, for example, as shown in Figure 8.10, the policies in C# projects, the policies in Java-based web projects, and the policies in C++ based desktop projects. The intersection of these policies is common policies crossing different platforms and programming languages.

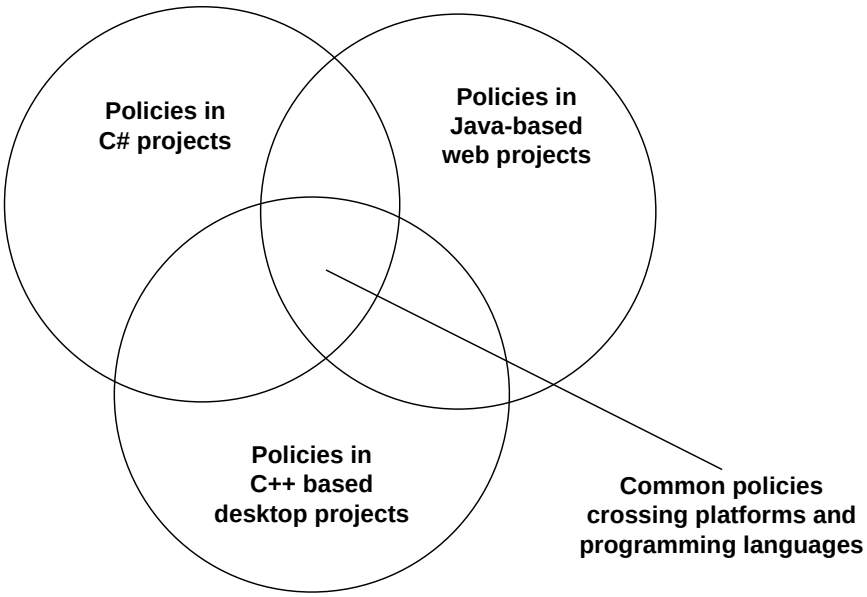


Figure 8.10 Polices for SuperDev

In the GAIN framework, policies are defined in AF format, and the appropriate Semantic Interpreters are used to transform these policies to concrete Aspects in different AOP platforms. If we define P1 as policies in C# projects, P2 as policies in Java-based web projects, and P3 as policies in

C++ based desktop projects, then the available Semantic Interpreters needed for P1, P2, P3 and their intersections are shown in Table 8.1below:

	SI for C#	SI for Java	SI for C++
P1	A		
P2		A	
P3			A
P1 ∩ P2	A	A	
P1 ∩ P3	A		A
P2 ∩ P3		A	A
P1 ∩ P2 ∩ P3	A	A	A

A: Available

Table 8.1 The availability of Semantic Interpreters for policies

In summary, the common policies in P1 ∩ P2, P1 ∩ P3, P2 ∩ P3, and P1 ∩ P2 ∩ P3 are defined repeatedly in P1, P2, and P3. In addition, the common polices enforcer are implemented repeatedly in C#, Java, and C++ with the same processing logic embedded, which cost extra money and time.

Therefore, the common policies need to be defined once in an appropriate format and can be implemented automatically when the programming language is confirmed. By using the Gain framework, all these common polices can be defined in an AF format and can be transformed to language specific policy enforcement code.

8.4.4 PCAS

In this section, as a demonstration, two company development policies are introduced:

- Object creation enforcement (P1). Considering further maintenance, all objects must be created by using factory pattern [32], rather than using “new” keyword.
- Using logging mechanisms rather than using standard output (P2). As using System.out or System.err in Java , or Console.WriteIn in C# is a poor way to perform logging, the company decides to use logging mechanisms and prohibit using standard output for logging purposes.

To implement the above policies, the following two Aspects are needed: namely PolicyEnforcement_NoStandardOutput, and PolicyEnforcement_NewObj. These policies are then described in a PCAS shown in Figure 8.11. The specification is created with the PCAS Editor by finding appropriate AFs and putting them into the AspectFramework. These Aspects are not woven into the original components because as a policy enforcer, rather than adapting the original component, they only need to give some suggestions or warning messages. Therefore, the value of most attributes (such as sourcefile, and path) of the AspectFramework tag and some attributes (class and method) of Apply-aspect tag are empty because these attributes are only required for the Aspect weaving process, which is not needed for policy enforcement.

```
<?xml version="1.0"?>
<AOP-Process   name="PolicyEnforcement"
                    xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
<AspectFramework name="PE_CodeConventionsOOP"
                        sourcefile=""
                        path=""
                        joinpointcomponent=""
                        joinpointmethod=""
                        when=""
                        returntype=""
                        parameters=""
                        awhen="">
<Apply-aspect    class=""
                    method=""
                    aspect_id="06"
                    aspect_level="Primitive"
                    aspect_type="PE_NoStandardOutput"
                    af_id="14"
```

```
                    af_name="PolicyEnforcement_NoStandardOutput"
                    synchronized="false"
                    comment="PE_No standard output"/>
<Apply-aspect    class=""
                    method=""
                    aspect_id="05"
                    aspect_level="Primitive"
                    aspect_type="PE_NewObj"
                    af_id="13"
                    af_name="PolicyEnforcement_NewObj"
                    synchronized="false"
                    comment="PE_NewObjectCreation"/>
</AspectFramework>
</AOP-Process>
```

Figure 8.11 The PCAS for policy enforcement

8.4.5 Aspects

The AF definition for P2 is shown in Figure 8.12. The basic information of an Aspect is provided within CommonStructure tag and the variations of PolicyEnforcement_NoStandardOutput Aspect is provided within the Variation tag. In this example, the AffectedClasses are the variation of PolicyEnforcement_NoStandardOutput Aspect, which means these classes will be the target classes that this policy enforces.

```
<?xml version="1.0" ?>
<Aspect name="PolicyEnforcement_NoStandardOutput">
  <!-- Common Structure -->
  <CommonStructure>
     <PointCut>
        <Name>pe_nso1</Name>
        <When>execution</When>
        <ReturnType>*</ReturnType>
        <ClassName>*</ClassName>
        <MethodName>*</MethodName>
        <Parameters>*</Parameters>
     </PointCut>
     <Advice>
       <When>before</When>
       <PointCutName ref="pe_nso1" />
     </Advice>
  </CommonStructure>
```

```
  <!-- Variations -->
  <Variation type="PE_NoStandardOutput">
```

```
    <AffectedClasses>
      <Class>ShoppingCart</Class>
      <Class>ShoppingCartOperator</Class>
    </AffectedClasses>
  </Variation>
</Aspect>
```

Figure 8.12 No standard output policy definition in AF

The no standard output policy can be applied to different components implemented by different programming languages. As shown in Figure 8.13, the AInst of this policy in the AspectJ platform is generated by a Semantic Interpreter automatically.

```
01 public aspect PolicyEnforcement_NoStandardOutput{
02 declare warning : get(* System.out) || get(* System.err)
03       && (within(ShoppingCart) || within(ShoppingCartOperator)):
04       "Consider using logging mechanism instead.";
05 }
```

Figure 8.13 No standard output policy definition in AInst in AspectJ

While compiling AInst of PolicyEnforcement_NoStandardOutput Aspect together with the target components, the policy will be enforced. For example, if the code in target components conflicts with the policy, the warning message will be provided as shown in Figure 8.14. Therefore, in this way, the policies can be forced to all classes in the target components.

```
D:\pe>ajc PolicyEnforcement_NoStandardOutput.java
          ShoppingCart.java ShoppingCartOperator.java
D:\pe\ShoppingCart.java:27 [warning]
 Consider using logging mechanism instead.
System.out.println(msg);
^^^^^^^^^^^^^^^^^^^^^^^^^^
      field-get(java.io.PrintStream java.lang.System.out)
      see also: D:\pe\PolicyEnforcement_NoStandardOutput.java:12::0
1 warning
```

Figure 8.14 Warning message while compiling PolicyEnforcement_NoStandardOutput Aspect with original component

If the developer wants to apply this policy to C# program, then the appropriate Semantic Interpreter can be used to generate related AInsts.

The corresponding AInst in C# to AF of no standard output policy in Figure 8.12 is shown in Figure 8.15.

```
01 using System;
02 using System.Text;
03 using System.Xml;
04 namespace gain
05 {
06    public class PolicyEnforcement_NoStandardOutput
07    {
08       public static void Main(string[] args)
09       {
10          PolicyEnforcerNSO pe = new PolicyEnforcerNSO();
11          pe.enforce(ShoppingCart);
12          outputMsg(pe.getEnforcementReport());
13          pe.enforce(ShoppingCartOperator);
14          outputMsg(pe.getEnforcementReport());
15       }
16       public void outputMsg(string msg) {
17          System.Console.WriteLine( msg );
18       }
19    }
20 }
```

Figure 8.15 No standard output policy definition in AInst in C#

While running AInst of PolicyEnforcement_NoStandardOutput Aspect, the policy will be enforced. For example, if the code in target components conflicts with the policy, the warning message will be provided as shown in Figure 8.16.

```
 PE_NSO -> D:\tmp\CSharpPEChecker.cs
Error in Line 18 : "System.Console.WriteLine("Usage: CSharpPEChecker PCASName");"
Use log mechnism instead!
PE_NSO -> D:\tmp\CSharpPEChecker.cs
Error in Line 26 : "System.Console.WriteLine( msg );"
Use log mechnism instead!
```

Figure 8.16 Warning message while running PolicyEnforcement_NoStandardOutput Aspect

8.4.6 Summary

In this case study, the approach framework defines a group of policies as reusable assets, and enforces them to various platforms and programming languages in corresponding to the requirements of the software firm. In

conclusion, this case studies shows that the approach framework and the related tool are capable of defining platform independent Aspects in AFs and transforming them to platform specific Aspects in AInsts by using Semantic Interpreters.

Chapter 9 Conclusions and Future Work

The main outcome of the research undertaken for this thesis was the development of a new technique using generative programming, software product line, component adaptation technology, and aspect oriented programming to support deep level component adaptation with high automation.

This chapter discusses three parts of the work that merit further examination and discussion. Firstly, the evaluation of the approach is carried out by employing the techniques in chapter 2 (section 2.1.6) and chapter 3 (section 3.3.5). Secondly, the conclusions are reached and the technique contributions are summarised. Thirdly, the future directions of the research are discussed.

9.1 Critical Analysis of the Approach

As the approach is the combination of / the improvement to AOP and component adaptation techniques, it is justified in applying the technique requirements of component adaptation approaches and AOP approaches respectively.

In table 9.1, the GAIN approach is compared with other component adaptation approaches. All these approaches are evaluated on how well each technique fulfils the specified component adaptation requirements in section 2.1.6.

Adaptation techniques	R1	R2	R3	R4	R5	R6	R7	R8
Copy-paste	-	+	-	-	-	-	+	-
Inheritance	-	+	-	-	-	-	+	-
Wrapping	+	-	+	-	-	-	-	+/-
SAGA	-	+	-	+	-	-	+	-
Superimposition	+	-	+	+	+	-	-	-
BCA	-	+	-	-	-	-	+	-
Customizable Components	-	-	+	+	+	+	-	-
Non-Invasive approach to WS	+	+	+	+	-	+/-	+	+
Wrapper for WS adaptation	+	-	+	-	-	-	-	+
GAIN	-	+	+	+	+	+?	+	+

R1: Black-box

R2: Transparent

R3: Composable

R4: Reusable

R5: Configurable

R6: Automatic

R7: Deep level adaptation

R8: Language independence

+: fulfilled

-: not fulfilled

+/-: fulfilled or not fulfilled depending on the application context

+?: Semi-automatic

Table 9.1 The comparison between GAIN and other component adaptation techniques

The comparison outcomes are justified as follows:

R1: Black-box

Since currently GAIN is a source code level adaptation technique, the implementation details of original component must be obtained prior to performing the adaptation. Therefore, the black-box requirement is not fulfilled.

R2: Transparent

Since the end-user of the adapted component is unaware of the adaptation between original component and the adapted component, the transparent requirement is fulfilled.

R3: Composable

Since the adaptation process can be composed with other adaptation methods, no matter how many adaptations are applied to the original component, the composable requirement is fulfilled. This process is repeatable.

R4: Reusable

With the support of the two-dimensional Aspect model, Aspect repository and related tool, GAIN is capable of providing platform-independent, highly reusable Aspects to deal with various adaptation circumstances if the required Aspects are available in the repository. Therefore, the reusable requirement is fulfilled.

R5: Configurable

With the support of product line based Aspect model, any abstraction level Aspect has two parts, namely, Common Structure (CS) and Variations (V). In the Variations part, flexible configurations for each type of Aspect are supported by providing various parameters to Aspects and combining different elements within Aspects. Therefore, the configurable requirement is fulfilled.

R6: Automatic

The approach is semi-automatic in performing component adaptation. The Aspect oriented component adaptation design still needs human

intervention. However, with the support of the CASE tool, the Aspect oriented component adaptation implementation is fully automatic. As long as the adaptation requirements are described in PCAS, the component adaptation can be performed by simply clicking a button. Benefits from AOP and the automation of the approach, the disadvantages of source code level adaptation such as maintenance and evolution concerns (refer to section 2.1.5.1) have been eliminated as the adaptation concerns are logically separated from the original component(s) and the adaptation process is semi-automatic.

R7: Deep level adaptation

As source code level adaptation is performed in GAIN, the deep level adaptation are supported in the approach by organizing various Aspects in PCAS and performing adaptation by parsing and executing the adaptation process defined in PCAS.

R8: Language independence

With the three abstraction level of Aspects and a set of Semantic Interpreters, language independence is achieved in GAIN. All Aspects can be represented as a language independent form as Aspect frames (AF).

On the other hand, considering GAIN as an improvement to Aspect-oriented programming technology, in table 9.2, GAIN approach is evaluated by comparing with other popular AOP technologies.

AOP approaches	T1	T2	T3	T4	T5	T6	T7
Aspectual Component	-	+	-	-	-	-	-
JAsCo	-	+	-	+	-	-	-
Shared Join Points Model	-	-	-	-	+	-	-
Framed Aspects	-	+	+	+	-	+	+
GAIN	+	+	+/-	+	+	+	+

T1: Short learning curve

T2: Reusable

T3: Light weight

T4: Configurable

T5: Advanced weaving process

T6: Language independence

T7: Generative Aspects

+: fulfilled
-: not fulfilled
+/-: whether fulfil the requirement or not depending on the target AOP platform

Table 9.2 The comparison between GAIN and other AOP techniques

The outcomes are justified as follows:

T1: Short learning curve

With support from the associated CASE tool, in the Aspect oriented design stage, end users only need to understand basic concepts in AOP such as pointcut and advice and fill relative parameters for selected AFs. The PCAS is generated automatically by the tool. In the Aspect oriented implementation phase, the job is even easier because the generation of individual Aspects and the adaptation process are fully automatic. Therefore, compared with existing AOP platforms, the GAIN framework is easy to use.

T2: Reusable

All Aspects in GAIN are designed following a software product line based Aspect model. While reflecting different variations in AAF, AF, and AInst, the

Aspects are highly reusable. For example, an AF can be reused in various AOP platforms by employing appropriate Semantic Interpreters. Therefore, the reusable requirement is fully fulfilled.

T3: Light weight

The GAIN framework could be heavy or light weight, depending on which target AOP platform is selected and which Semantic Interpreter is selected to generate AInsts. By default, Aspect Generator generates Aspect in the same programming language source code as the component being adapted. In this case, the framework is light weight. On the other hand, when the target AOP platform is an existing heavy weight AOP platform, the Aspect Generator generates heavy weight source code in the selected AOP platform. Therefore, in this case, the GAIN framework is heavy weight.

T4: Configurable

With the support of product line based Aspect models, any abstraction level Aspect has two parts, namely, Common Structure (CS) and Variations (V). In the Variations part, flexible configurations for each type of Aspect are supported by providing various parameters to Aspects and combining different elements within Aspects. Therefore, the configurable requirement is fulfilled.

T5: Advanced weaving process support

To meet the complex adaptation requirements, the advanced weaving process such as switch structure is supported to enhance the weaving in the join points in the GAIN approach.

T6: AOP programming language independence

With the three abstraction level of Aspects and a set of Semantic Interpreters, language independence is achieved in GAIN. All Aspects are

represented as a language independent form as Aspect frames (AF), and may be mapped to any AOP language if appropriate Semantic Interpreter is available. Currently, the supported languages in GAIN include AspectJ, Java, and C#.

T7: Generative Aspects

The automatic generation of individual Aspects is supported by the Aspect Generator and Semantic Interpreters in GAIN. Whether the required Aspects in AInst level can be generated by GAIN depends on the availability of the Semantic Interpreters.

9.2 Conclusions and Technique Contributions

9.2.1 Conclusions

Despite the success of component-based reuse, the mismatches (section 2.1.4.1) between available pre-qualified components and the specific reuse context in individual applications continue to be a major factor hindering component composition and therefore reusability. From a technical perspective, the reason is largely due to the difficulty of adapting these components to meet the specific needs of the user. The research during the study was based on the observation that existing reuse approaches and tools are weak in providing a mechanism to adapt components at an adequately deep level and meanwhile with sufficient automation.

The Aspect-oriented nature of the approach makes it particularly suitable for the improvement of non-functional features of the target component-based software, such as dependability and performance. However, existing AOP platforms do not support reusable Aspects and advanced weaving process effectively. Therefore, the reuse of Aspects and advanced weaving process support must be considered while applying AOP to component adaptation.

The approach applies Aspect-oriented generative adaptation to targeted components to correct the mismatch problem with eliminating the problems associated with current component adaptation and AOP approaches, so that the components can be integrated into the target application easily. The following work has been undertaken during the study:

A generative aspect-oriented component adaptation approach

Based on the successful points of existing technologies, such as generative programming, software product line, component adaptation, and AOP, a new generative aspect-oriented component adaptation approach focusing on non-functional issues to mismatch problems (section 2.1.4.1) has been developed. Meanwhile, the approach has provided the solution to the problems (section 3.1.7 and section 3.3.5) associated with current component adaptation approaches and AOP platforms.

In the approach, from the component view, there are two main parts for each Aspect: Common Structure and Variations. The Common Structure is the mechanism that is reused in several similar Aspects or target aspect oriented systems. On the other hand, Variations reflects the variations among different Aspects. Each Aspect shares the same structure in its Common Structure part and varies in its Variations part. In the abstract view, three levels are used to support the product family: Abstract Aspect Frame (AAF), Aspect Frame (AF), and Aspect Instance (AInst).

A framework to support the approach

As the embodiment of the approach, a generative aspect oriented component adaptation framework has been developed. The framework consists of two phases: Aspect oriented Component Adaptation Design and Aspect oriented Component Adaptation Implementation. The aim of Aspect oriented component adaptation design is to gather the specification for

component adaptation and the output of design stage is a PCAS. User intervention is required in the design stage. On the other hand, Aspect oriented component adaptation implementation is the automatic process of performing component adaptation according to the PCAS gathered in the design stage.

A prototype tool

As the implementation of the framework, a prototype tool (Chapter 7) was developed to support the component adaptation in the framework and to demonstrate its scalability. In the design phase, Component Analyzer and PCAS Editor were built to help developers building PCAS. On the other hand, Aspect Generator, Semantic Interpreter, and Aspect Weaver were developed to automate the Aspects generation and weaving process. Also, the Aspect Manager was developed to manage Aspects in three abstract levels.

Case studies

Three case studies (Chapter 8) have been undertaken to illustrate and evaluate the usability and correctness of the approach, in terms of its capability of building highly reusable and platform independent Aspects across various AOP platforms and providing an advanced flow control of weaving process.

In summary, the requirements defined in Section 3.4 have been fulfilled respectively:

- As a source code level adaptation technique, the approach performs deep level component adaptation focusing on non-functional issues. Although the aspect oriented design still needs user intervention, the aspect oriented implementation is a fully automatic process.

- With the two dimensional Aspect model, highly reusable Aspects in various AOP platforms have been supported in the approach.
- The advanced weaving process with the support of flexible flow control has been implemented in AspectJ, pure Java, and C#. Depending on different target platform, PCAS based weaving or pre-weaving process is supported.
- As the adaptation knowledge is hidden in the Semantic Interpreters, and in the Aspects, the system users do not need to know too much details of the complex syntax of AOP languages. Therefore, the approach has a short learning curve.

9.2.2 Contributions

The GAIN technology enables application developers to adapt the pre-qualified components to eliminate mismatches to the integration requirement of specific applications. From a component adaptation point of view, the original contribution of this thesis is the automation and deep level adaptation of components, focusing on non-functional issues by introducing extra process, operations and resources. As the feature inherited from AOP, all non-functional issues solved by AOP can also be addressed by GAIN, e.g., Monitoring, Policy enforcement, Persistence, Optimization, Authentication, Authorization, Transaction Management, and implementing business rules. From the AOP point of view, the original contribution is the improved reusability of Aspects and the support of advanced weaving process. The key technical contributions [41][103] are summarised below:

1) Product line based reusable Aspect model [44][103] (section 4.2). In the approach, a two dimensional Aspect model was developed, e.g. component view and abstraction view of Aspect. Within the two dimensional Aspect model, all Aspects are designed to be reusable in both dimensions. From the component point of view, each Aspect is split into common structures (CS) and variations (V), which support software

product line based reuse. On the other hand, from the abstraction point of view, each Aspect has three levels of abstraction: AAF, AF, and AInst. There are different types of variations in these abstractions, including functional variations, parameterisation, and platform variations. During the whole Aspect oriented adaptation process, from the designing of different Aspects in AAF, to the implementation of AOP platform independent Aspects in AFs, to the implementation of concrete AOP platform specific Aspects in AInsts, all Aspects are presented in two parts: CS and V, no matter which abstract level they are, such as AAF, AF, or AInst.

2) Highly reusable and AOP platform independent adaptation Aspects [101][103] (section 5.2). With the support from the product line based Aspect model, the adaptation knowledge is captured in Aspects and is reusable in various adaptation circumstances. Highly reusable Aspects, especially in AAF and AF level are achieved. Different types of Aspect are saved in XML schema as AAF. Platform independent Aspects are implemented in XML as AFs. With the support of AFs, the learning curve of AOP becomes shorter because AOP platform specific syntax is hidden in the related tool, namely the Aspect Generator and Semantic Interpreters. Platform specific Aspects are automatically generated from these AFs by selecting corresponding Semantic Interpreters as required.
3) Aspect Repository for Aspect reuse [42][43][44]. As an embodiment of the product line based reusable Aspect model, an expandable library of reusable adaptation Aspects at three abstraction levels is used as storage for various levels of Aspects. With the support of three abstract levels of Aspects in Aspect repository, the reusability of the framework is increased incrementally. In addition, the combination of Aspects and control flows are saved in the Aspect repository as Aspect Frameworks and can be reused in the similar adaptation situations.
4) Advanced Aspect weaving process [102][103] (section 6.5). In AOP, it is possible that several Aspects need to be woven at the same join point. In these cases, these Aspects may have problems such as determining the exact execution order and dependencies among the Aspects. The

enhanced Aspect weaver supports the advanced weaving processes, e.g. sequence and switch structure in a weaving process. Pre-defined advanced weaving processes may be also added into the Aspect repository as Aspect Framework for further reuse.

9.3 Future work

However, the GAIN approach still has its weakness. For example, currently, the approach does not support binary level component adaptation, which limits the wider use of the approach. Also, the Aspect oriented adaptation design still needs human intervention. Ideally, the Aspect oriented adaptation design and implementation should be fully automatic.

9.3.1 Aspect oriented binary code adaptation

As an Aspect oriented adaptation technique, currently, the work focuses on source code level adaptation in the weaving process. However, as a component adaptation technique, it is desirable to deal with both source code level component(s) and binary code level component(s). In the future, the research can be enhanced to deal with all types of component adaptations, and therefore, the approach can be applied to wider adaptation scenarios.

9.3.2 Classification of mismatch problems and adaptation types

Currently, the approach only works automatically in the Aspect oriented implementation stage including Aspect generation and Aspect weaving. In the Aspect oriented design stage, however, user intervention is still required, which limits the wider use of the approach.

The classification of mismatch problems and the adaptation types can be used to address this problem. Algorithms have to be developed to check for the identified component mismatch types during the adaptation design

stage. If the adaptation types can be classified corresponding to particular adaptation requirements, the adaptation patterns can be summarised and reused in the similar situations in the future. As a result, the Aspect oriented design can work in a semi-automatic or automatic manner. In addition, the prototype tool needs to be improved to support short listing potential suitable Aspects, and the selection of adaptation pattern(s) which can be applied to solve specific mismatch types.

9.3.3 Intelligent Aspect repository and automatic Aspect selection

To achieve a fully-fledged engineering approach to component adaptation, further effort will be required to develop an intelligent Aspect repository that requires much less human interaction than existing solutions and gives a larger return to application developers wishing to use them. Ideally, automatic Aspect selection should be supported corresponding to the specific adaptation requirements. The proposed intelligent repositories can only be achieved through the addition of semantics to existing Aspects. The Aspect ontology can be used to develop such an intelligent repository.

9.3.4 Aspect-oriented web service adaptation

When building service oriented systems, it is often the case that existing web services do not perfectly match user requirements in target systems. To achieve seamless integration and high reusability of web services, mechanisms to support automated evolution of web services are highly in demand. The GAIN approach can potentially solve the above problem associated with web services by applying the approach to the underlying components of web services. However, due to the unique characteristics of web services, the users of a web service may be distributed globally and are very diverse in their detailed requirements. Therefore, more research needs to be carried out to apply the approach to service oriented systems.

References

[1] Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R. and Wallnau, K (2000). Component Based Software Engineering Volume II: Technical Concepts of Component-based Software Engineering. *Technical Report CMU/SEI-2000-TR-008*, Carnegie Mellon Software Engineering Institute.

[2] Baker, T. (2002). Lessons learned integrating COTS into systems. *Proceeding of International Conference on COTS-based Software Systems*, Orlando, FL: Springer-Verlag.

[3] Balk, L. D., and Kedia, A. (2000). PPT: a COTS integration case study. *Proceedings of International Conference on Software Engineering*, Limerick, Ireland: ACM Press.

[4] Basili, V.R. & Boehm, B. (2001), COTS-based system top 10 list, *IEEE Computer*, Vol.34, pp.91-93.

[5] Bassett, P. (1997), Framing Software Reuse - Lessons from the Real World, Prentice Hall, ISBN: 978-0133278590.

[6] Batory, D. (October, 1998). Product-Line Architectures. *Invited Presentation, Smalltalk & Java in Industry and practical Training*, Erfurt, Germany.

[7] Batory, D., Chen, G., Robertson, E., & Wang, T. (May 2000) Design Wizards and Visual Programming Environments for GenVoca Generators, *IEEE Transactions on Software Engineering*, pp. 441-452.

[8] Batory, D., Johnson, C., MacDonald, B., & Heeder, D. V. (April 2002) Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 11(2), pp. 191-214.

[9] Becker, S. et al. (2006), Towards an Engineering Approach to Component Adaptation, *Architecting Systems*, LNCS 3938, pp. 193–215, Springer-Verlag Berlin Heidelberg.

[10] Becker, S., Overhage, S., and Reussner, R. (2004). Classifying Software Component Interoperability Errors to Support Component Adaption. Proceedings of 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Edinburgh, UK, May 24-25, 2004. Volume 3054 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer pp.68–83.

[11] Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., & Mecella, M. (2003). Automatic Composition of E-services That Export Their Behavior. *Proceedings of the International Conference on Service Oriented Computing*, 43-58.

[12] Biggerstaff, T. (1998). A perspective of generative reuse. *Annals of Software Engineering*, Vol 5, pp. 169-226.

[13] Birrer, I., Cechticky, V., Pasetti, A., & Rohlik, O. (2004), Implementing Adaptability in Embedded Software through Aspect Oriented Programming. *Proceedings of IEEE Mechatronics & Robotics*, Aachen, Germany, pp. 85-90.

[14] Birrer, I., Chevalley, P., Pasetti, A., and Rohlik O.(2004) An Aspect Weaver for Qualifiable Applications. *Proceedings of the DASIA 2004 – Data Systems in Aerospace Conference*, pp. 272-280.

[15] Blake, M. B. (September, 2004). A Specification Language and Service-Oriented Architecture to Support Distributed Data Management. *Software:* Practice and Experience , 34, 11, 1091-1117, John Wiley and Sons.

[16] Bondavalli, A., Chiaradonna, S., Cotroneo, D., & Romano, L. (2004), Effective fault treatment for improving the dependability of COTS and legacy-based applications, *IEEE Transactions on Dependable and Secure Computing*, 1(4), pp. 223-237.

[17] Brown, A.W., and Wallnau, H.C., The current state of CBSE, *IEEE Software*, Vol: 15, Issue: 5, pp. 37-46, Sep-Oct 1998.

[18] Bultan, T., Fu, X., Hull, R., & Su, J. (May, 2003). Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. *Proceedings of the 12th International World Wide Web Conference*.

[19] Burner, M. (March, 2003). The Deliberate Revolution: Transforming Integration With XML Web Services. *ACM Queue*, 1, 1, 28-37.

[20] Cable, S., Galbraith, B., Irani, R., Hendricks, M., Milbury, J., Modi, T., Tost, A., Toussaint, A., & Basha J. (2002). *Professional Java Web Service*. Peer Information Inc.

[21] Canal, C., Murillo, J.M., and Poizat, P.(2004). Coordination and Adaptation Techniques for Software Entities. *Proceedings of ECOOP 2004 Workshops*, Oslo, Norway, June 14-18, 2004, Final Reports. Volume 3344 of Lecture Notes in Computer Science., Springer pp.133–147.

[22] Cao, F., Bryant, B.R., Liu, S.H., & Zhao, W. (2005). A Non-Invasive Approach to Dynamic Web Services Provisioning. *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS'05)*.

[23] Carman, M., Serafini, L., & Traverso, P. (2003). Web Service Composition as Planning. *Proceedings of the 13th International Conference on Automated Planning & Scheduling.*

[24] Casati F., Shan, E., Dayal, U., & Shan, M.C. (October, 2003). Business-oriented management of Web services. *Communications of the ACM*, 46, 10, 55-60.

[25] Cervantes H., & Hall, R. S. (2004). Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Compoent Model. *Proceedings of the 26th International Conference on Software Engineering*, 614-623.

[26] Charfi, A., & Mezini, M. (2004). Aspect-Oriented Web Service Composition with AO4BEL. *Proceedings of the European Conference on Web Services*.

[27] Christian, K., and Störzer, M.(2004), PCDiff: Attacking the Fragile Pointcut Problem. *Proceedings of European Interactive Workshop on Aspects in Software*, Berlin, Germany.

[28] Cibran, M. A. (2004). Modularizing Web Services Management with AOP. *Proceedings of the European Conference on Web Services*.

[29] Cleaveland, J. C. (July 1998), Building application generators, *IEEE Software*, pp. 5(4):25-33.

[30] Colyer, A., and Clement, A. (2005). Aspect-oriented programming with AspectJ. *IBM Systems,* Vol 44(2), pp. 301-308.

[31] Constantinides, C., Skotiniotis, T., and Stoerzer, M.(2004), AOP considered harmful. *Proceedings of 1st European Interactive Workshop on Aspects in Software EIWAS'04*, September, 23-24, 2004, Berlin, Germany.

[32] Cooper J.W. (2000). Java Design Patterns. Addison Wesley. ISBN: 0-201-48539-7.

[33] Cottenier, T., & Elrad, T. (2004). Validation of Context-Dependent Aspect-Oriented Adaptations to Components. *Proceedings of the 9th International Workshop on Component-Oriented Programming*.

[34] Curbera, F., Khalaf, R., Mukhi, N., Tai, S., & Weerawarana S. (October, 2003). The next step in Web services. *Communications of the ACM*, 46, 10, 29-34.

[35] Diaz-Herrera, J.L., Knauber, P., & Succi, G. (2000), Issues and Models in Software Product Lines, *International Journal on Software Engineering and Knowledge Engineering*, 10(4):527-539.

[36] Doernhoefer, M. (2005), Surfing the Net for Software Engineering Notes, *ACM SIGSOFT Software Engineering Notes*, Vol 30(4), pp.10-18.

[37] Dustdar, S. (2004). Web Services Workflows - Composition, Co-ordination, and Transactions in Service-oriented Computing. *Concurrent Engineering*, 12, 3, 237-245.

[38] Egyed, A., and Gacek, C.(1999). Automatically Detecting Mismatches during Component-Based and Model-Based Development, *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, Cocoa Beach, Florida, October 1999, pp. 191-198.

[39] Elrad, T., Askit, M., et al. (2001). Discussing aspects of AOP. *Communications of ACM*, 44, 10, 33-8.

[40] Fayad, M. E. And Schmidt, D. C. (1997). Object-oriented application frameworks. *Communications of ACM*. Vol 40(10), pp. 32-38.

[41] Feng, Y., Liu, X., and Kerridge, J., Achieving Smooth Component Integration with Generative Aspect and Component Adaptation. *Springer-Verlag's LNCS 4039, (9th International Conference on Software Reuse)*, Turino, Italy, 2006.

[42] Feng, Y., Liu, X., and Kerridge, J. An Aspect-Oriented Component-Based Approach to Seamless Web Service Composition. *System and Information Sciences Notes*, to appear, ISSN 1753-2310, 2007.

[43] Feng, Y., Liu, X., and Kerridge, J., A product line based aspect-oriented generative unit testing approach to building quality components. *Proceedings of the 1st IEEE International Workshop on Quality-Oriented Reuse of Software*, Beijing, China, pp., 2007.

[44] Feng, Y., Liu, X., and Kerridge, J., Smooth Quality Oriented Component Integration through Product Line Based Aspect-Oriented Component Adaptation. Proceedings of International Conference on Software Engineering and Knowledge Engineering (SEKE'2007), Boston, USA, pp. 71-76, July 9-11, 2007.

[45] Ferris, C., & Farrell, J. (June, 2003). What are Web services?. *Communications of the ACM.* 46, 6, 31.

[46] Fiorano Software, Inc. (2004). Service Oriented Architecture Implementation Frameworks white paper.

[47] Frakes, W. B., and Kang, K. (2005), Software Reuse Research: Status and Future, *IEEE Transactions on Software Engineering*, Vol:31, No.7, pp. 529-536, July 2005.

[48] Fuchs, M. (2004), Adapting Web Services in a Heterogeneous Environment. *Proceedings of the IEEE International Conference on Web Services* (ICWS'04), pp.656-664, 6-9 July 2004.

[49] Gamma, E, Helm, R., Johnson, R., and Vlissides J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN: 0-201-63361-2.

[50] Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural Mismatch or Why it's hard to build systems out of existing parts. *IEEE Software*, November, pp. 17-26, 1995.

[51] Gray, J., Bapty, T., Neema, S., Gokhale, A., & Natarajan, B. (2002). Generating Aspect Code from Models. *Proceedings of the Workshop on Generative Techniques for Model Driven Architecture*. Settle, WA.

[52] Griss, M. L. & Wosser, M. (1995), Making reuse work at Hewlett-Packard. *IEEE Software*, 12(1), 105-7.

[53] Hanson, J. E., Nandi, P., & Levine, D.W. (2002). Conversation-enabled Web Services for Agents and e-Business. *Proceedings of the International Conference on Internet Computing*, 791-796.

[54] Hanson, J. E., Nandi, P., & Kumaran, S. (September, 2002). Conversation Support for Business Process Integration. *Proceedings of the 6th International Enterprise Distributed Object Computing*. Ecole Polytechnic, Switzerland.

[55] Heineman, G.T., (1998). A model for designing adaptable software components, *Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC)*, pp. 121-127, Vienna, Austria, August 1998.

[56] Heineman, G.T., and Ohlenbusch, H. (1998), Towards a theory of component adaptation, *Technical report WPI-CS-TR-98-20*, Worcester Polytechnic Institute.

[57] Herrejon, R. E. L., & Don, B (2002), Using AspectJ to Implement Product-Lines: A Case Study. *Technical Report*. Department of Computer Sciences, The University of Texas, Austin, Texas 78712. September 2002.
[58] Hölzle, U., Integrating Independently-Developed Components in Object-Oriented Languages, *Proceedings of ECOOP'93*, pp. 36-56, 1993.
[59] http://aspectwerkz.codehaus.org/
[60] http://eclipse.org/aspectj/
[61] http://en.wikipedia.org
[62] http://www.eclipse.org/projects/listofprojects.php
[63] http://java.sun.com
[64] http://java.sun.com/blueprints/qanda/ejb_tier/restrictions.html
[65] http://java.sun.com/docs/books/tutorial/java/concepts/index.html
[66] http://java.sun.com/docs/books/tutorial/reflect/index.html
[67] https://javacc.dev.java.net/
[68] http://labs.jboss.com/jbossaop/
[69] http://msdn.microsoft.com/en-us/library/ms173183(VS.80).aspx
[70] http://www.aophp.net/
[71] http://www.aspectc.org
[72] http://www.corba.org/
[73] http://www.ibm.com/developerworks/library/specification/ws-bpel/
[74] http://www.jboss.org/
[75] http://www.magicdraw.com/
[76] http://www.microsoft.com/com
[77] http://www.microsoft.com/net/
[78] http://www.oristand.com
[79] http://www.sei.cmu.edu/cbs/
[80] http://www.springframework.org/
[81] http://www.w3.org/2002/ws
[82] http://www.w3.org/TR/xslt20/
[83] http://www-306.ibm.com/software/awdtools/developer/rose/index.html
[84] Hull, R., Benedikt, M., Christophides, V., & Su, J. (June, 2003). E-Services: A Look behind the Curtain. *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.
[85] Ingham, David B., Shrivastava, Santosh K., & Panzieri, Fabio, "Constructing dependable Web services", IEEE Internet Computing, 2000, 4(1), pp. 25-33.
[86] Jacobsen, I., Griss, M., et al. (1997), *Software Reuse*, Reading, MA: Addison-Wesley.
[87] Jarzabek, S., Bassett, P., Zhang, H., and Zhang, W. (2003), XVCL: XML-based Variant Configuration Language, *Proceedings of 25th International Conference on Software Engineering*, May 2003, pp. 810-811.
[88] JAsCo cookbook
[89] Jhumka, A., Hiller, M., & Suri, N. (2002), Component-based synthesis of dependable embedded software, Formal Techniques in Real-Time

and Fault-Tolerant Systems. *Proceedings of 7th International Symposium*, FTRTFT 2002. Proceedings LNCS, Vol.2469, pp 111-28.

[90] Keller, R., & Hölzle, U. (1998). Binary Component Adaptation. *Proceedings of the 12th European Conference on Object-Oriented Programming*, July 1998.

[91] Kiczales, G. (December, 2001). Aspect-Oriented Programming: The Fun Has Just Begun. *Proceedings of the Workshop on New Visions for Software Design and Productivity: Research and Applications*.

[92] Kiczales, G., Hilsdale, E., et al. (2001). Getting started with AspectJ. *Communications of ACM*. Vol 44(10), pp. 59-65.

[93] Kim, S. M. (May 17-22, 2004). A Survey of Public Web Services. *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters,* 312-313. New York, USA.

[94] Kleijnen, S., & Raju, S. (March, 2003). An Open Web Services Architecture. *ACM Queue*, 1, 1, 38-46.

[95] Kreger, H. (June, 2003). Fulfilling the Web Services Promise. *Communications of the ACM*, 46, 6, 29-34.

[96] Kucuk, B., & Alpdemir, M.N. (1998), Customizable adapters for blackbox components, *Proceedings of the 3rd International Workshop on Component Oriented Programming*, pp.53-59.

[97] Kumaran, S., & Nandi, P. (2003). *Dynamic e-Business Using BPEL4WS, WS-Coordination, WS-Transaction, and Conversation Support for Web Services*

[98] Laddad, R. (2003). *AspectJ in Action*. Greenwich, CT, Manning Publications Co.

[99] Lieberherr, K., Lorenz, D., & Mezini, M. (March, 1999). Programming with Aspectual Components, *Technical Report*, NU-CCS-99-01.

[100] Lim, W.C., Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software*, Vol 11, Issue 5, pp.23-30, 1994.

[101] Liu, X., Feng, Y. and Kerridge, J. (2006), Achieving Dependable Component-Based Systems through Generative Aspect Oriented Component Adaptation. *Proceedings of the 30th IEEE International Conference on Computer Software and Applications (COMPSAC'06)*, Chicago, USA.

[102] Liu, X., Feng, Y. and Kerridge, J. (2006). Automated Responsive Web Services Evolution through Generative Aspect-Oriented Component Adaptation. *International Journals of Computer Applications in Technology*, to appear, ISSN (Print): 0952-8091.

[103] Liu, X., Feng, Y., & Kerridge, J. "Generative Aspect-Oriented Component Adaptation", *IET Software*, accepted.

[104] Liu X., Wang B., & Kerridge J. (2005). Achieving Seamless Component Composition Through Scenario-Based Deep Adaptation And Generation. *Journal of Science of Computer Programming (Elsevier), Special Issue on New Software Composition Concepts*, 56, 2.

[105] Lohmann, D., Blaschke, G., and Spinczyk, O. (2004), Generic advice: On the combination of AOP with generative programming in

AspectC++, *Proceedings of the 3rd Int. Conf. on Generative Programming and Component Engineering (GPCE '04)*, vol 3286 of LNCS, pp. 55–74. Springer, October 2004.

[106] Loughran, N., & Rashid, A. (2004), Framed Aspects: Supporting Variability and Configurability for AOP. *Proceedings of International Conference on Software Reuse*, 2004.

[107] Loughran, N., Rashid, A., Zhang, W., & Jarzabek, S. (2004), Supporting Product Line Evolution with Framed Aspects. *Proceedings of Workshop on Aspects, Components and Patterns for Infrastructure Software* (held with AOSD 2004).

[108] Majithia, S., Shields, M., Taylor, I., & Wang, I. (2004). Triana: A Graphical Web Service Composition and Execution Toolkit. *Proceedings of International Conference on Web Services*, 514.

[109] Majithia, S., Walker, D. W., & Gray, W. A. (2004). A Framework for Automated Service Composition in Service-Oriented Architectures. *Proceedings of the First European Semantic Web Symposium*.

[110] McIlroy, M. D. (1968), Mass-produced software components. *Proceedings of NATO Conference on Software Engineering*, Garmisch, Germany, Springer-Verlag.

[111] Mehner, K., & Rashid A. Towards a Generic Model for AOP (GEMA). *Technical Report,* No. CSEG/1/03. Computing Department, Lancaster University, UK.

[112] Mezini, M., Lorenz, D., & Lieberherr, K. Components and Aspect-Oriented Design/Programming. *Lecture Notes*.

[113] Mezini, M., Ostermann, K. (2005), A comparison of program generation with aspect-oriented programming, Lecture Notes in Computer Science, 3566, pp. 342-354.

[114] Mili, H., Mili, A., Yacoub, S., and Addy, E.(2002), *Reuse-Based Software Engineering, Techniques, Organization, and Controls*, Wiley Inter-Science, ISBN: 0-471-39819-5.

[115] Miller, G. (June, 2003). The Web services debate: .NET vs. J2EE. *Communications of the ACM*, 46, 6, 64-67.

[116] Moreira, A., and Araujo, J.(2004), Handling unanticipated requirements change with aspects, *Proceedings of the Software Engineering and Knowledge Engineering Conference*, Banff, Canada.

[117] Nagy, I., Bergmans, L., & Aksit, M(2005), Composing Aspects at Shared Join Points. *Proceedings of International Conference NetObjectDays*, Lecture Notes in Informatics, Vol 69, Erfurt, Germany.

[118] Pfarr, T. And Reis, J. E. (2002). The integration of COTS/GOTS within NASA's HST command and control system. Proceedings of 1st International Conference on COTS-based Software Systems, Orlando, FL: Springer-Verlag.

[119] Rada, R. (1995), Software Reuse, ISBN: 1-871516-53-6.

[120] Rao, J., & Su, X. (2004). A Survey of Automated Web Service Composition Methods. *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*.

[121] Rempel, M. & Lind, K. (September, 2002). *MCAD/MCSD C# (r) .NET (tm) Certification All-in-One Exam Guide*. McGraw-Hill Osborne Media.
[122] Samentinger, J. (1997). *Software Engineering with Reusable Components*. Springer Verlag.
[123] Sarkar, S. (August, 2002). Model Driven Programming Using XSLT: An Approach to Rapid Development of Domain-Specific Program Generators. *www.XML-JOURNAL.com*.
[124] Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F., *Pattern-Oriented Software Architecture – Volume 2 – Patterns for Concurrent and Networked Objects*. Wiley & Sons, New York, NY, USA (2000)
[125] Schult, W., & Polze A. (2002). Aspect-Oriented Programming with C# and .NET. *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*.
[126] Shaparau, D. Approaches to Web Service Composition. *Lecture Notes*, University of Trento.
[127] Shukla, D., Fell, S., & Sells, C. (2002). Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. *MSDN Magazine*.
[128] Singh, S., Grundy, J., & Hosking, J. (2004). Developing .NET Web Service-based Applications with Aspect-Oriented Component Engineering. *Proceedings of the Fifth Australian Workshop on Software and Systems Architectures*.
[129] Sirin, E., Hendler, J., & Parsia, B. (April, 2003). Semi-automatic composition of web services using semantic descriptions. *Proceedings of the Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*. Angers, France.
[130] Skogan, D., Gronmo, R., & Solheim, I. (2004). Modeling Web Service Composition in UML, *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference*.
[131] Skogan D., Gronmo, R., & Solheim I. (2004). Web Service Composition in UML. *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference*.
[132] Sommerville, I. (2007). *Software Engineering* (8th Ed.). Addison-Wesley, ISBN: 978-0-321-31379-9.
[133] Spinczyk, O., Gal, A., and Schröder-Preikschat, W. (2002), AspectC++: An aspect oriented extension to C++. *Proceedings of the 40th Int. Conf. on Technology of OO Languages and Systems (TOOLS Pacific '02),* pp. 53–60, Sydney, Australia, February 2002.
[134] Srivastava B., & Koehler, J. Web Service Composition – Current Solutions and Open Problems.
[135] Student Record System. Java course work for master students of Harbin Institute of Technology, China, 2007.
[136] Sullivan, G.T. (Oct 2001), Aspect-oriented programming using reflection and meta object protocols - Providing programmers with the capability to modify the default behaviour of a programming language.,*Communications of the ACM*, 44 (10), pp. 95-97.
[137] Suvee, D., Vanderperren, W., & Jonckers V. (2003). JAsCo: an Aspect-Oriented approach tailored for component Based Software

Development. *Proceedings of the 2nd international conference on Aspect-oriented software development*, 21-29. Boston, USA.

[138] Swe, S. M., Zhang, H., and Jarzabek, S. (2002), XVCL: A Tutorial, *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, Vol: 27, pp.341-349, Ischia, Italy.

[139] Szyperski, C. (2002), *Component Software Beyond Object-Oriented Programming* 2nd Edition. ACM Press, ISBN: 0-201-74572-0.

[140] The OWL Service Coalition. OWL-S: Semantic Markup for Web Services, 2003.

[141] Thone S., Depke, R., & Engels G. (2002). Process-Oriented, Flexible Composition of Web Services with UML. *Proceedings of the 21st International Conference on Conceptual Modelling*.

[142] Torchiano M. & Morisio M. (2004), Overlooked facts on COTS-based development, *IEEE Software*, Vol. 21, pp. 88-93.

[143] Tracz. W. (2001). COTS myths and other lessons learned in component-based software development. *Component-Based Software Engineering*. Boston: Addison-Wesley, pp. 99-112.

[144] Tsai, T.M., Yu, H.K., Shih, H.T., Liao, P.Y., Yang, R.D., & Chou, S. T. (October, 2003). Ontology-Mediated Integration of Intranet Web Services. *IEEE Computer*, 36, 10, 63-71.

[145] Turner, M., Budgen, D., & Brereton, P. (October, 2003). Turning Software into a Service. *IEEE Computer*, 36, 10, 38-44.

[146] Vallecillo, A., Hern´andez, J., and Troya, J. M. (2000). Component interoperability, *Technical Report. ITI-2000-37*, Departmento de Lenguajes y Ciencias de la Computaci´on, University of M´alaga, July 2000.

[147] Vanderperren, W. (September, 2001). Applying aspect-oriented programming ideas in a component based context: Composition Adapters. *Proceedings of NetObjectDays*, 201-206. Erfurt, Germany.

[148] Vanderperren, W., Suvée D., & Jonckers, V. (2003). Invasive Composition Adapters: an aspect-oriented approach for visual component-based development. *Proceedings of the ACP4IS workshop at AOSD 2003*.

[149] Vanderperren, W., Suvée, D., Verheecke, B., Cibrán, M.A., & Jonckers, V. (March, 2005). Adaptive programming in JAsCo. *Proceedings of the 4th international conference on Aspect-oriented software development*.

[150] Verheecke, B., Cibran, M. A., & Jonckers, V. (2003). AOP for Dynamic Configuration and Management of Web Services. *Proceedings of the* International Conference on Web Services – Europe.

[151] Viega, J., Voas, J. (Nov-Dec, 2000), Quality time - Can aspect-oriented programming lead to more reliable software?, *IEEE SOFTWARE*, 17(6), pp. 19-21.

[152] Wampler, D., Aspect Programming, Inc. Use Cases as Aspects – An Approach to Software Composition. *http://www.aspectprogramming.com/papers.html*.

[153] Wang, B., Liu, X. & Kerridge, J. (September, 2003). A Generative and Component based Approach to Reuse in Database Applications. *Proceedings of the 5th Generative Programming and Component Engineering Young Researchers Workshop*. Erfurt, Germany.

[154] Wang, B., Liu, X., & Kerridge, J. (November, 2004). Scenario-based Generative Component Adaptation in .NET Framework. *Proceedings of the IEEE International Conference on Information Reuse and Integration*. Las Vegas, USA.

[155] Wang, G., Hallberg, L.M., Saphier, E., Englander, E. W., and Bosch, J. (1999). Superimposition: a component adaptation technique. *Information and Software Technology*, Vol 41(5), pp. 257-273, 25 March 1999.

[156] Weerawarana, S. (2002). Business Process with BPEL4WS: Understanding BPEL4WS. *IBM developerWorks*.

[157] Williams, J. (June, 2003). The Web services debate: J2EE vs. .NET. *Communications of the ACM*, 46, 6, 58-63.

[158] Wirfs-Brock, R.J., and Johnson, R.E. (1990). Surveying current research in object-oriented design. Communications of ACM, Vol 33(9), pp.104-124.

[159] Yakimovich, D., Travassos, G., and Basili, V.(1999). A classification of software components incompatibilities for COTS integration. *Technical report, Software Engineering Laboratory Workshop*, NASA/Goddard Space Flight Centre, Greenbelt, Maryland.

[160] Yellin, D.M., and Strom, R.E., Protocol Specifications and Component Adaptors, *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 2, pp. 292-333, 1997

Appendix A Abbreviations and Acronyms

All the abbreviations and acronyms used in this thesis are defined below.

Abbreviation/Acronyms	Description
AAF	Abstract Aspect Frame.
AD	Active Directory.
AF	Aspect Frame
AI	Artificial Intelligence.
AOP	Aspect Oriented Programming.
AOSD	Aspect Oriented Software Development.
Aspect Framework	Aspect Framework is the combination of various aspects and control flows to support further reuse.
AInst	Aspect Instance.
BPEL4WS	Business Process Execution Language for Web Services.
CASE	Computer Aided Software Engineering.
CBSD	Component Based Software Development.
CIL	Common Intermediate Language.
CLR	Common Language Runtime.
COM	Common Object Model.
COM+	Common Object Model Plus.
CORBA	Common Object Request Broker Architecture.
COTS	Commercial Off-The-Shelf.
CS	Common Structure.
DCOM	Distributed Common Object Model.
DSL	Domain Specific Language
EJB	Enterprise Java Beans.
HTTP	Hyper Text Transfer Protocol.
IDL	Interface Description Language.
IL	Intermediate Language.
J2EE	Java 2 Enterprise Edition.
JavaCC	Java Compiler Compiler.
MSMQ	Microsoft Message Queuing.

Abbreviation/Acronyms	Description
MTS	Microsoft Transaction Server.
OLE	Object Linking and Embedding.
OMG	Object Management Group.
OOP	Object Oriented Programming.
ORB	Object Request Broker.
QoS	Quality of Service.
PCAS	Process-based Component Adaptation Specification.
SI	Semantic Interpreter.
SOAP	Simple Object Access Protocol.
SPL	Software Product Line
UDDI	Universal Description, Discovery and Integration.
UML	Unified Modelling Language.
V	Variations.
WSDL	Web Service Description Language.
XSLT	Extensible Stylesheet Language Transformations

Appendix B The screen dumps of the prototype tool

B.1 Main Interface

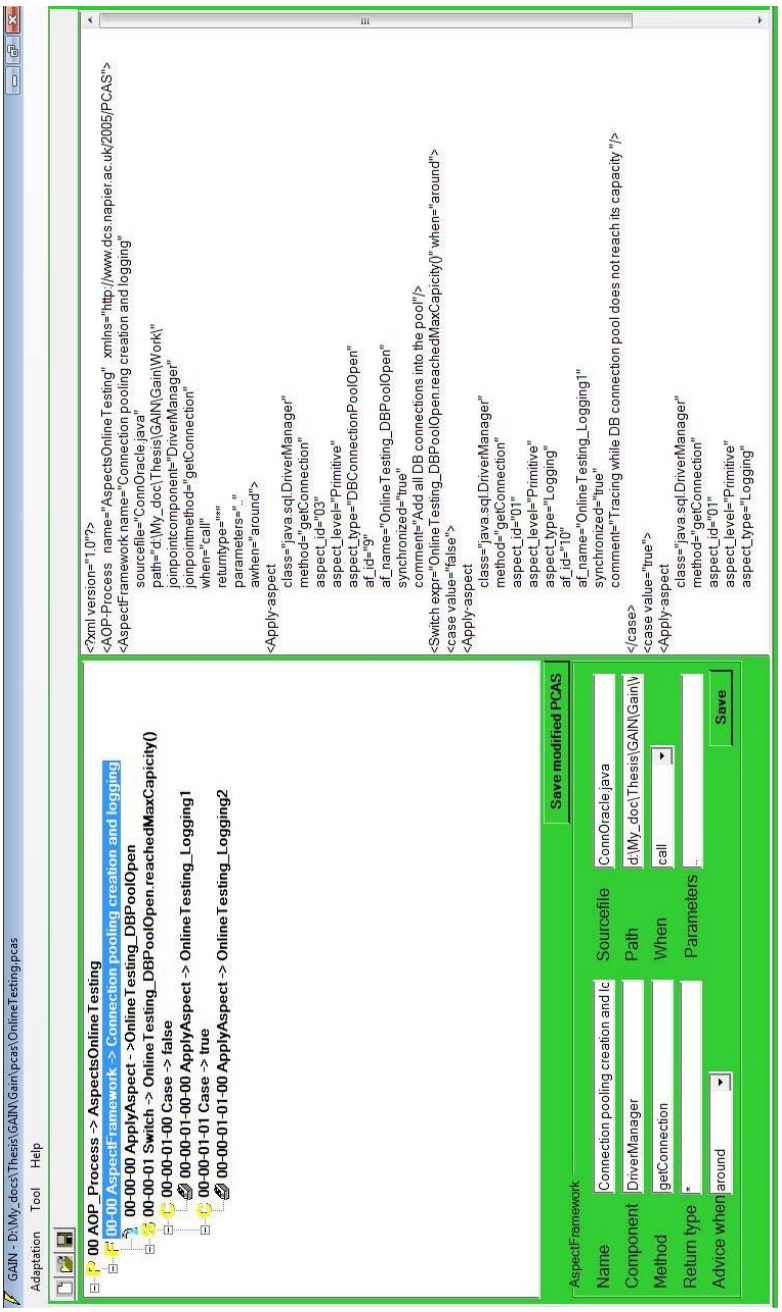


Figure B.1 Main interface

B.2 Aspect Manager

B.2.1 The manipulation of AFs

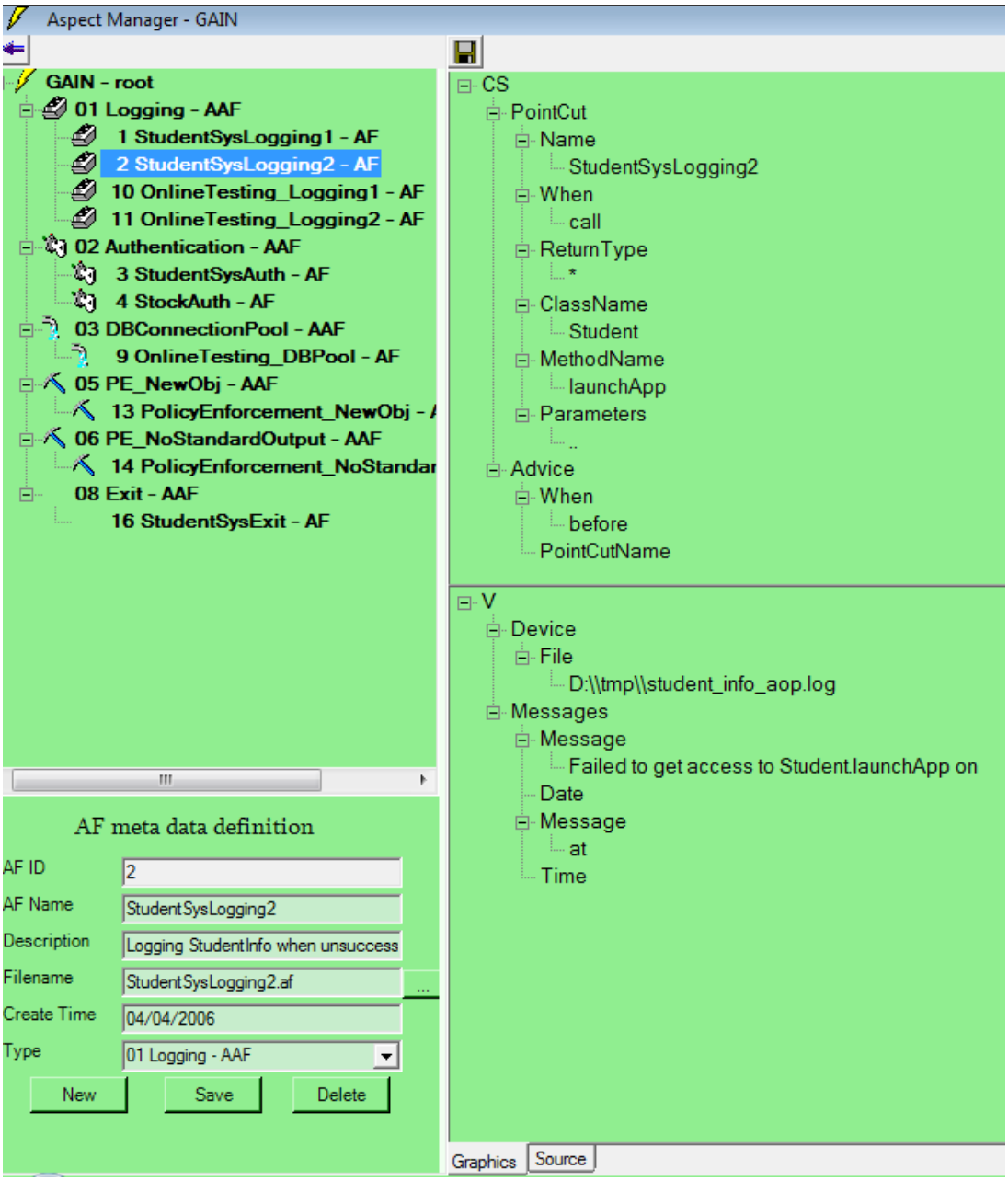


Figure B.2 Graphics view of AFs

Figure B.3 Source view of AFs

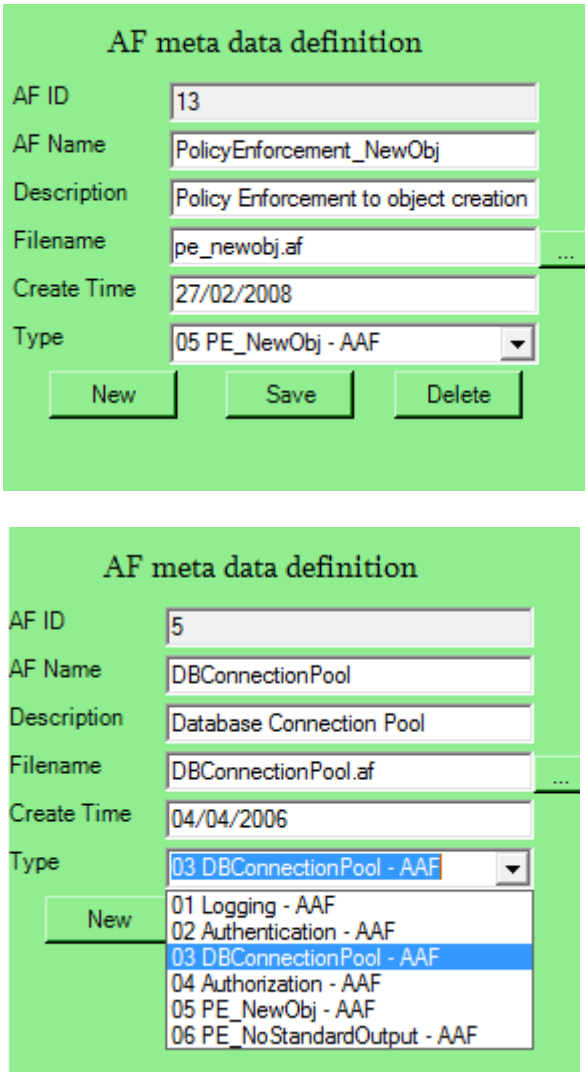


Figure B.4 AF meta data edit window

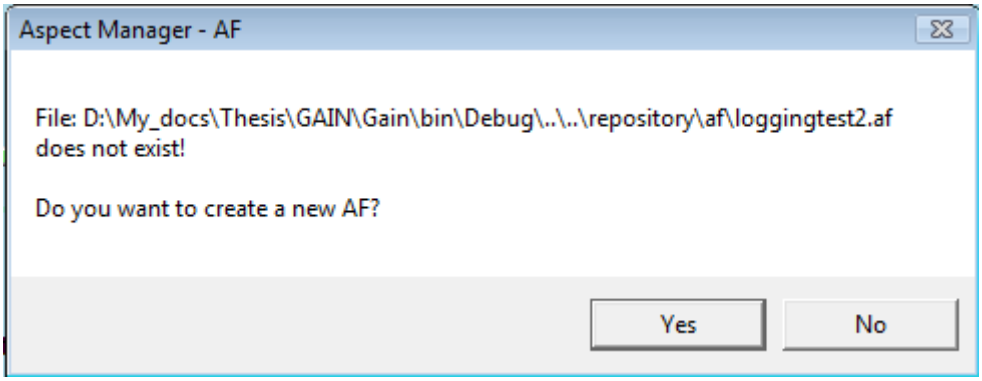


Figure B.5 Creating new AF file

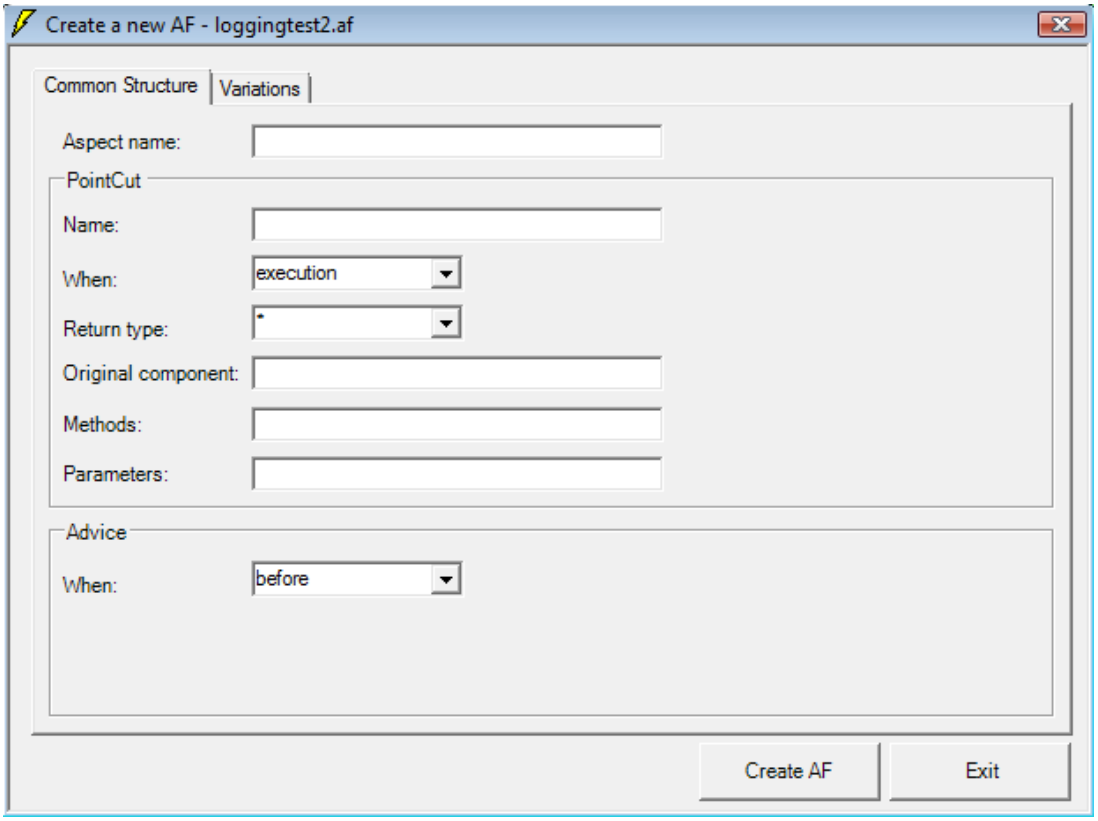


Figure B.6 Common structure part of AF

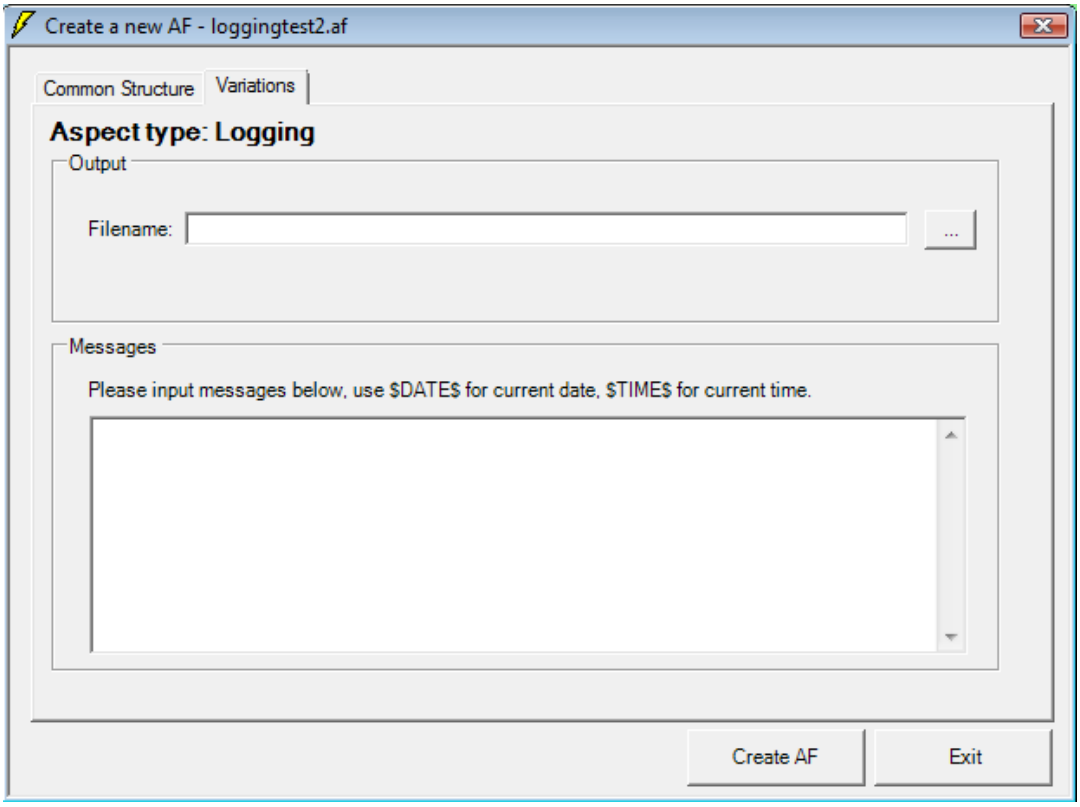


Figure B.7 Variation part of Logging Aspect

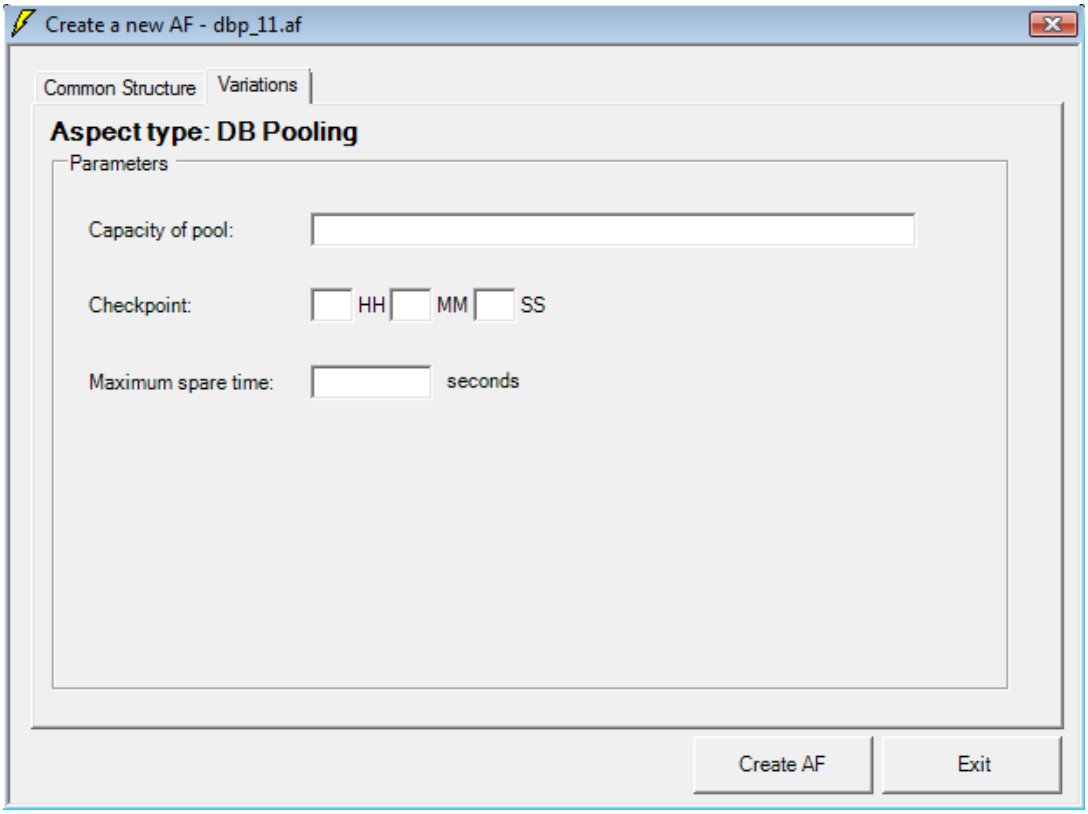


Figure B.8 Variation part of DBPooling Aspect

B.2.2 The manipulation of AAFs

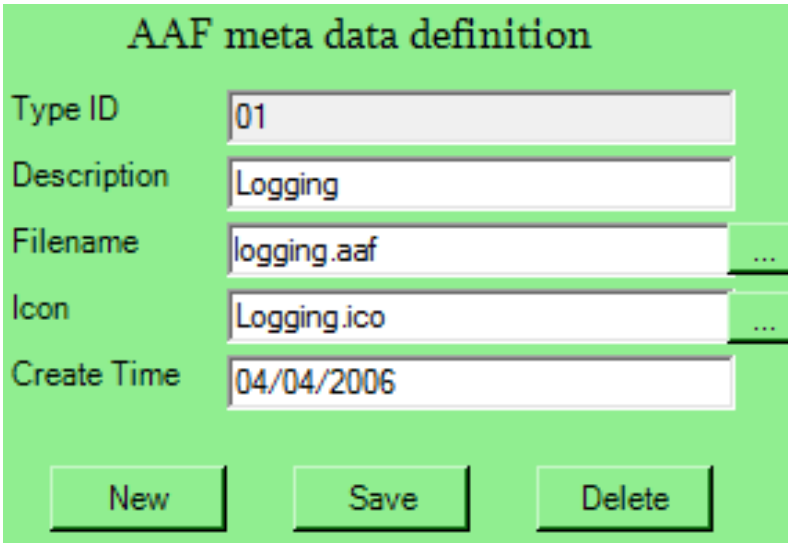


Figure B.9 AAF meta data edit window

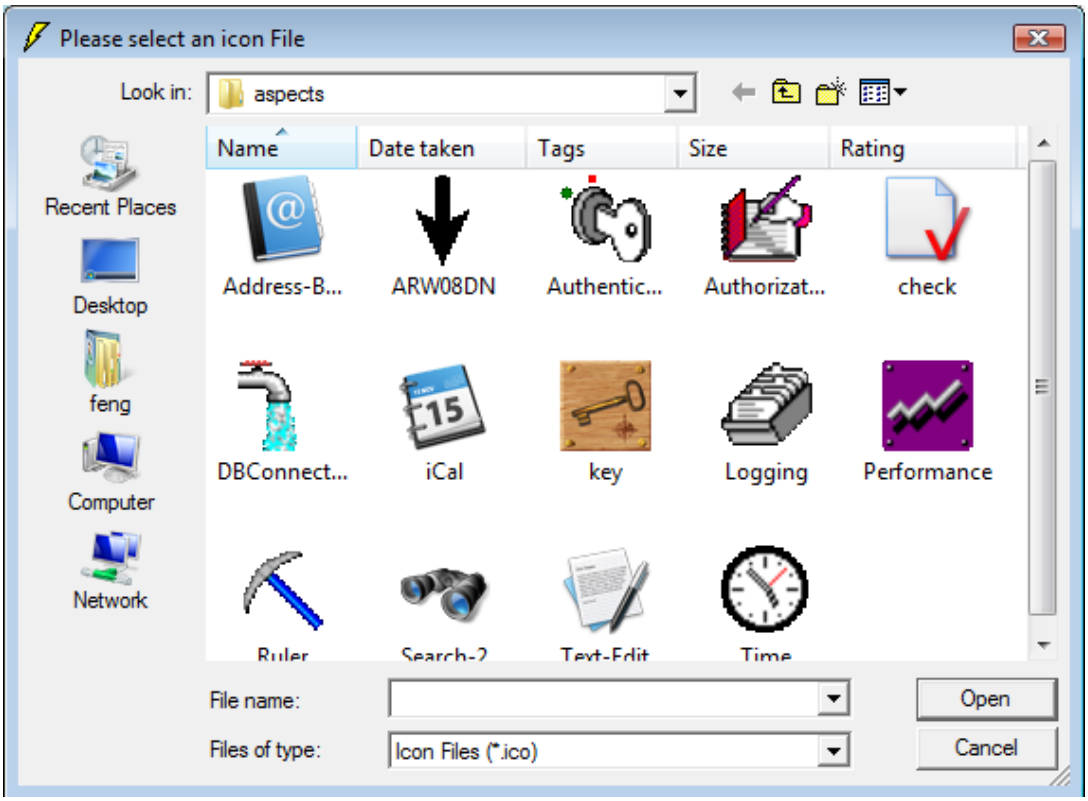


Figure B.10 Changing icon of AAF

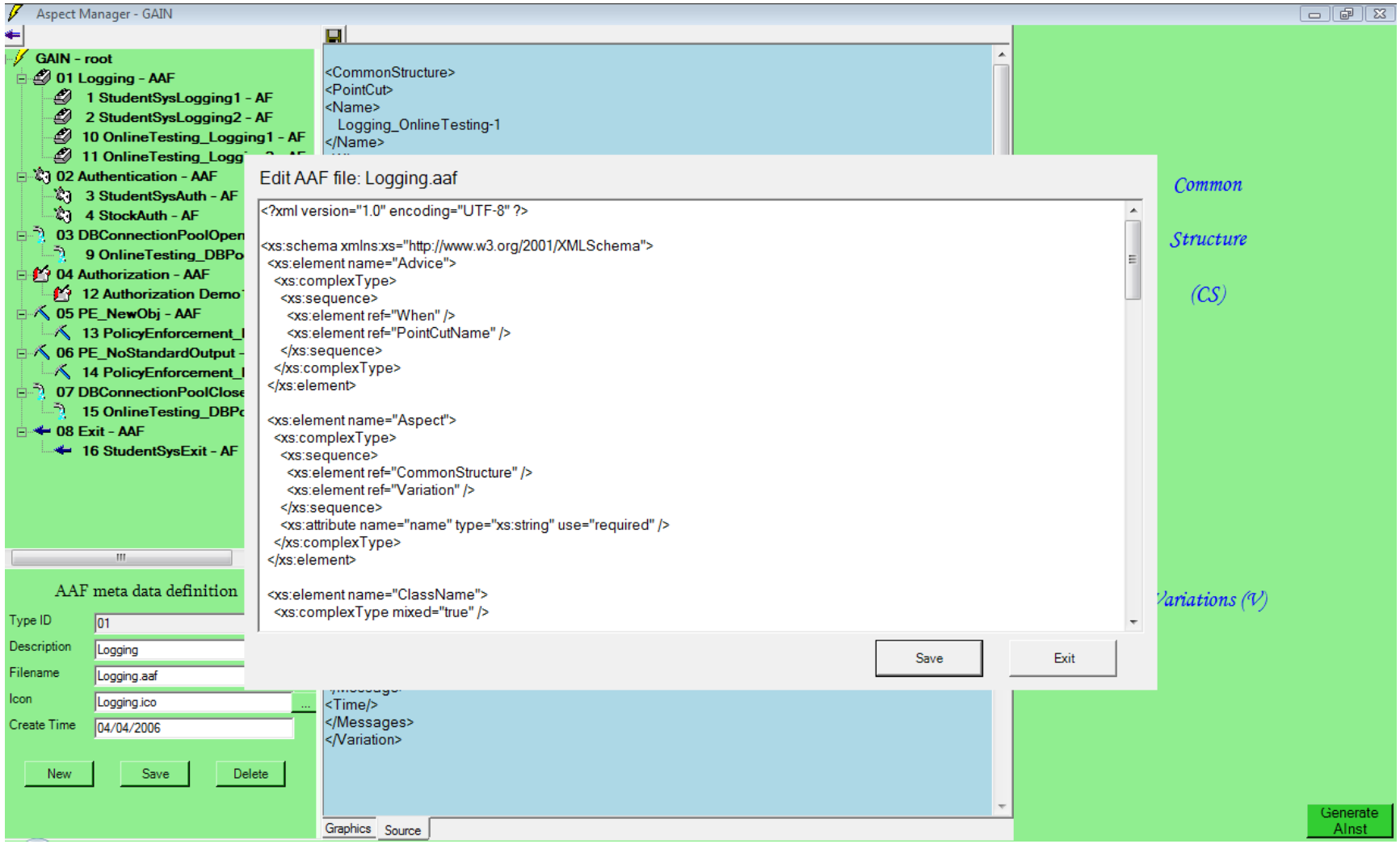


Figure B.11 AAF edit window

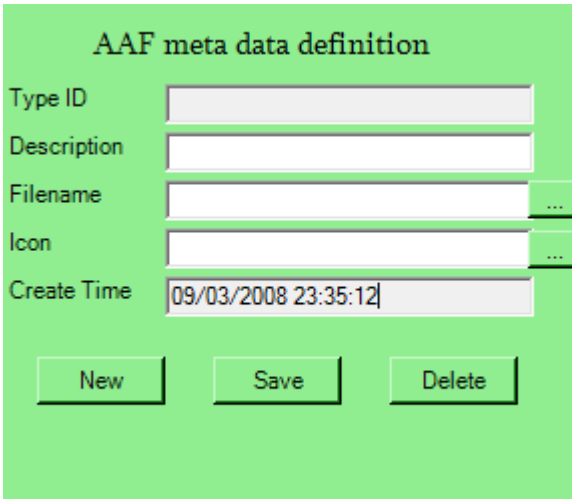


Figure B.12 Create AAF meta data

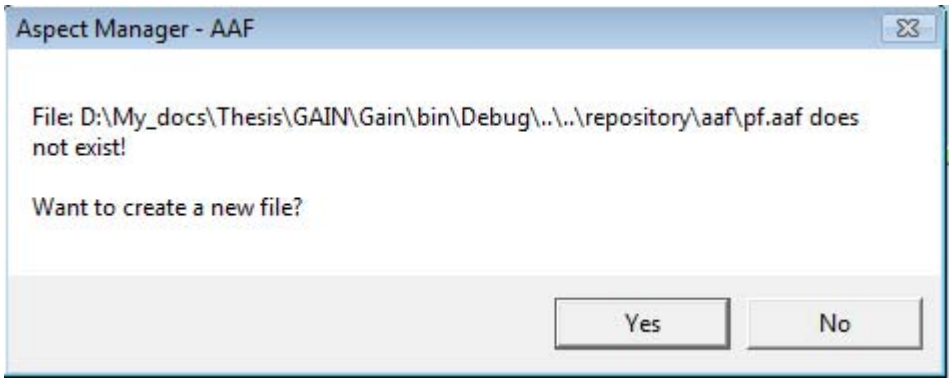


Figure B.13 Create AAF file

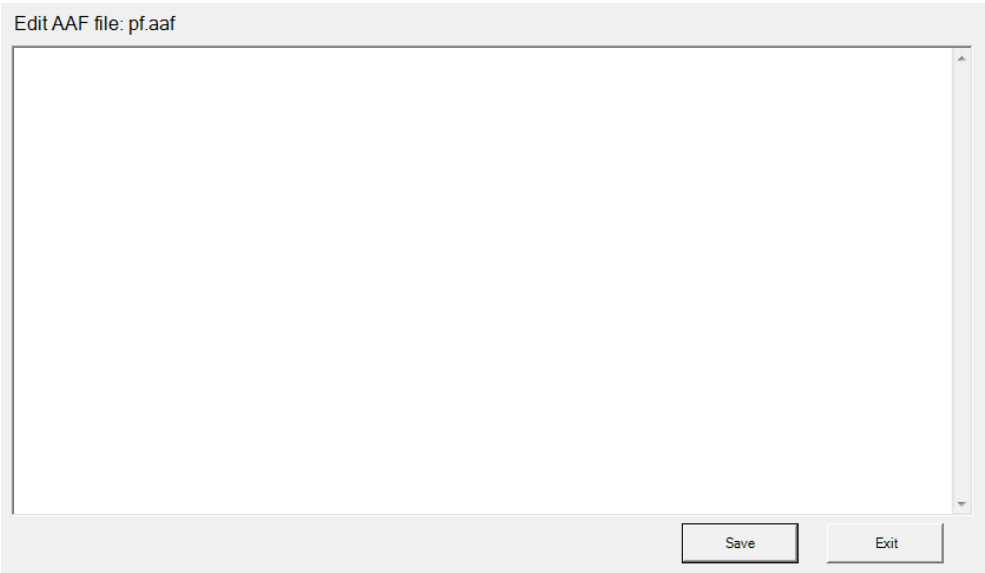


Figure B.14 AAF file edit window

B.2.3 AInsts

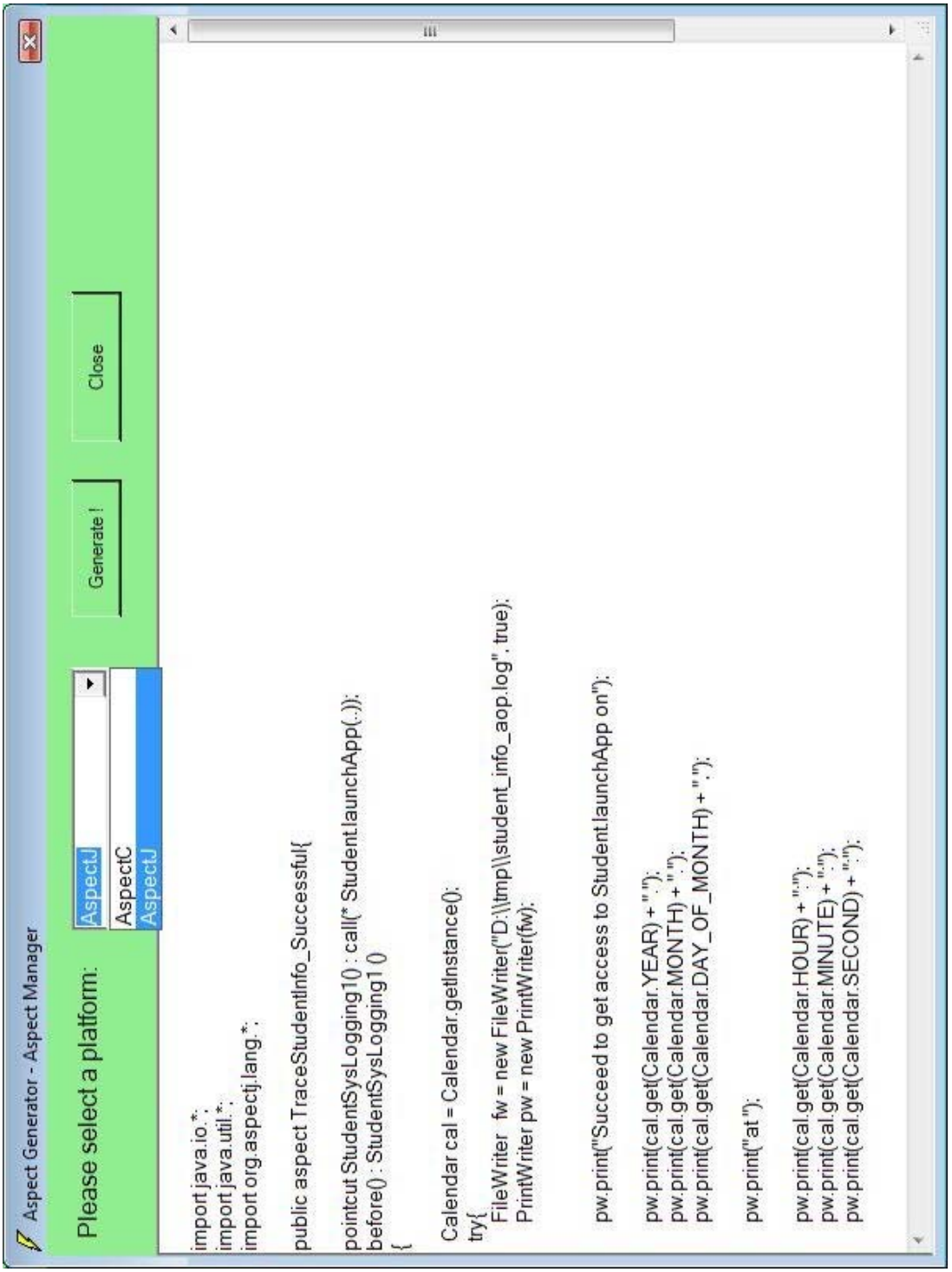


Figure B.15 Generated AInst

B.3 Component Analyzer

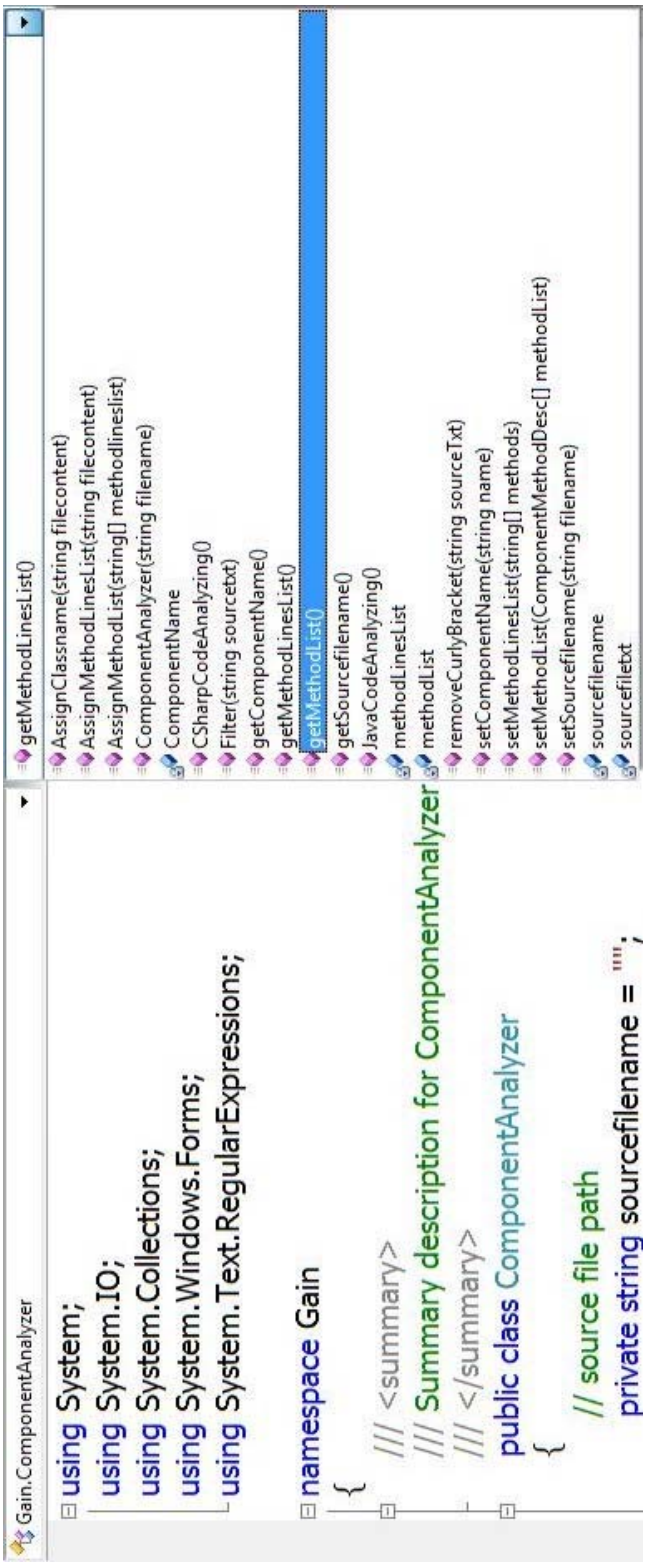


Figure B.16 Component Analyzer

B.4 Semantic Interpreter

Figure B.17 Semantic Interpreters

B.5 System Preferences

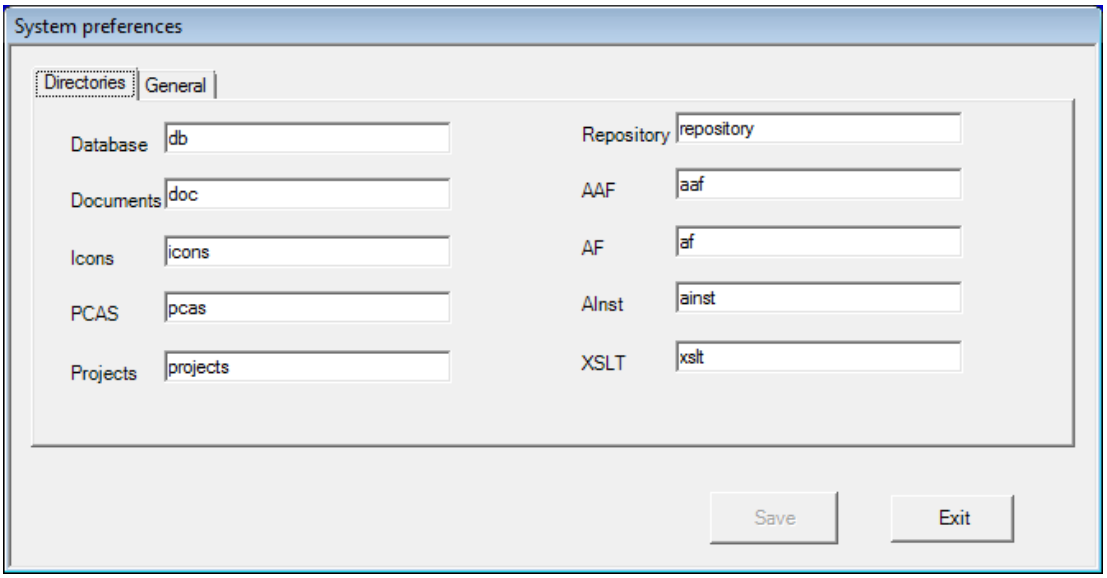


Figure B.18 System preferences

B.6 PCAS Editor

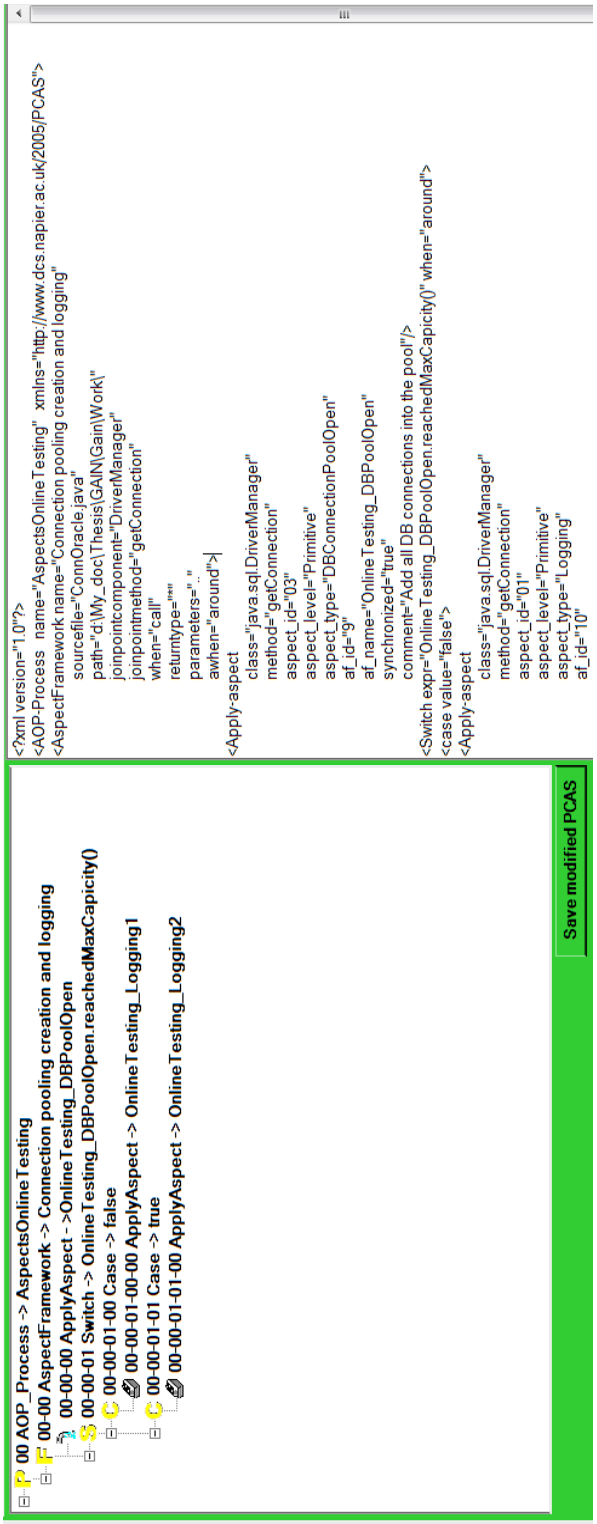


Figure B.19 PCAS Editor

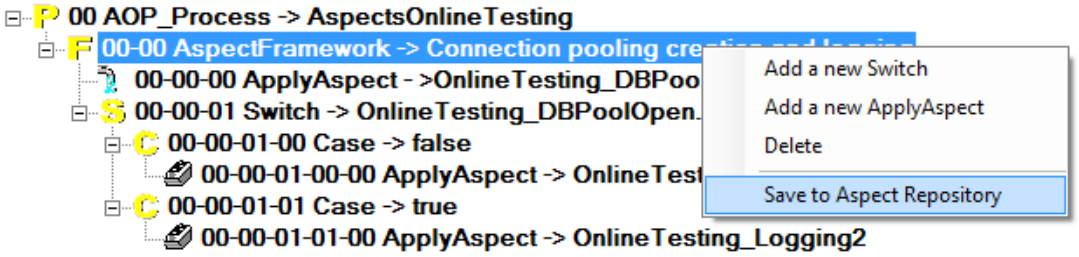


Figure B.20 Save Aspect Framework

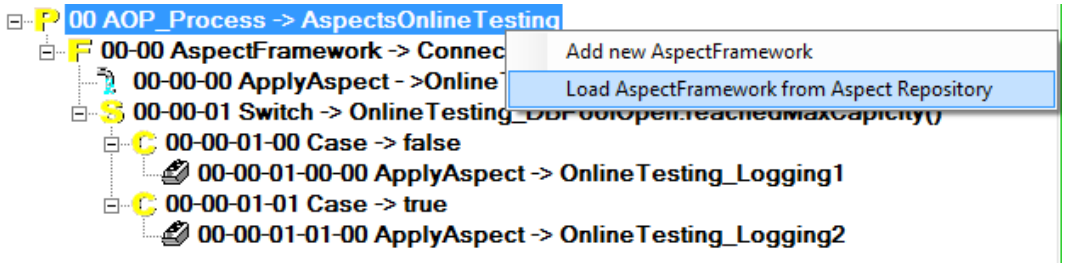


Figure B.21 Load Aspect Framework

B.7 Aspect Generation

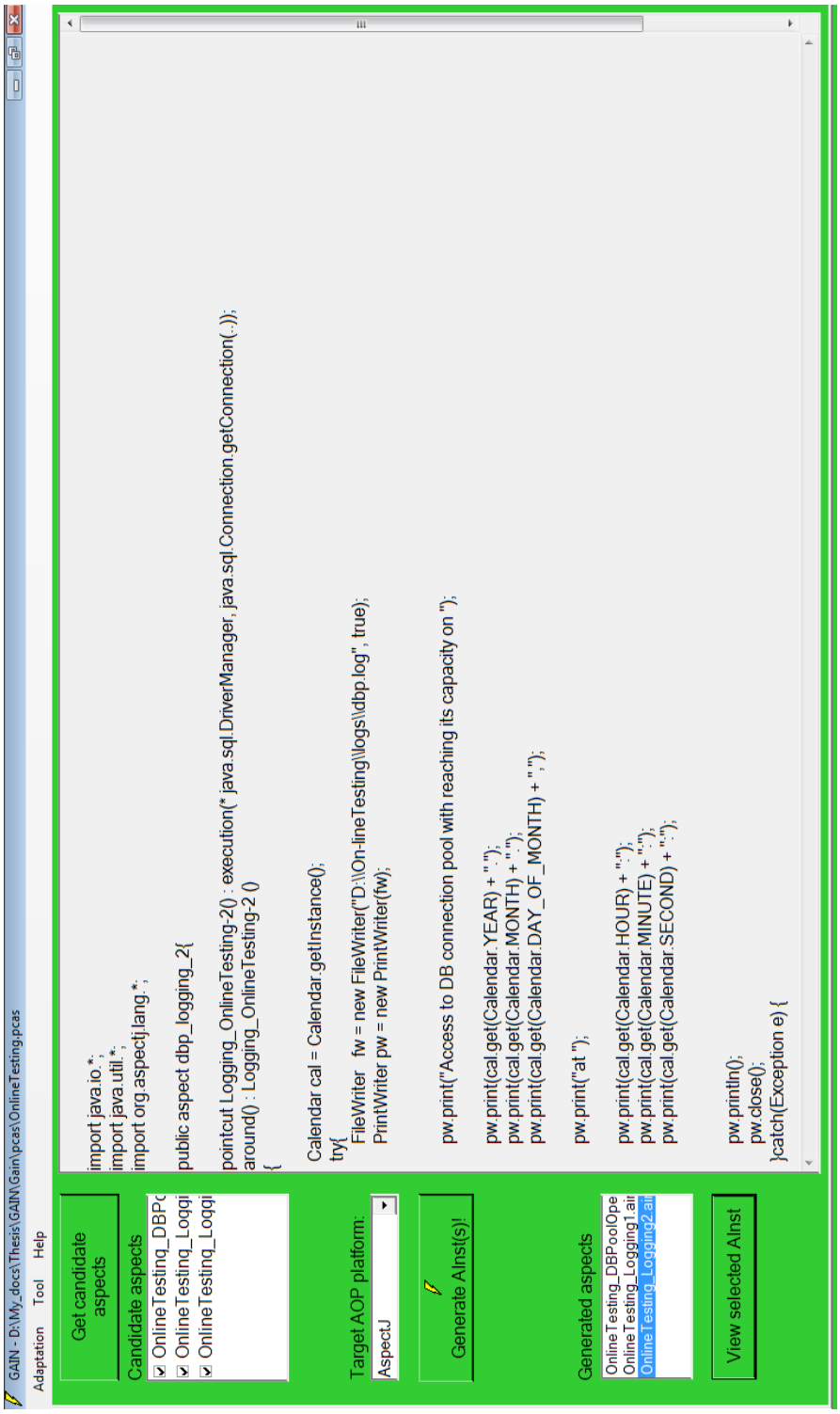


Figure B.22 Aspect Generation

B.8 PCAS Weaver

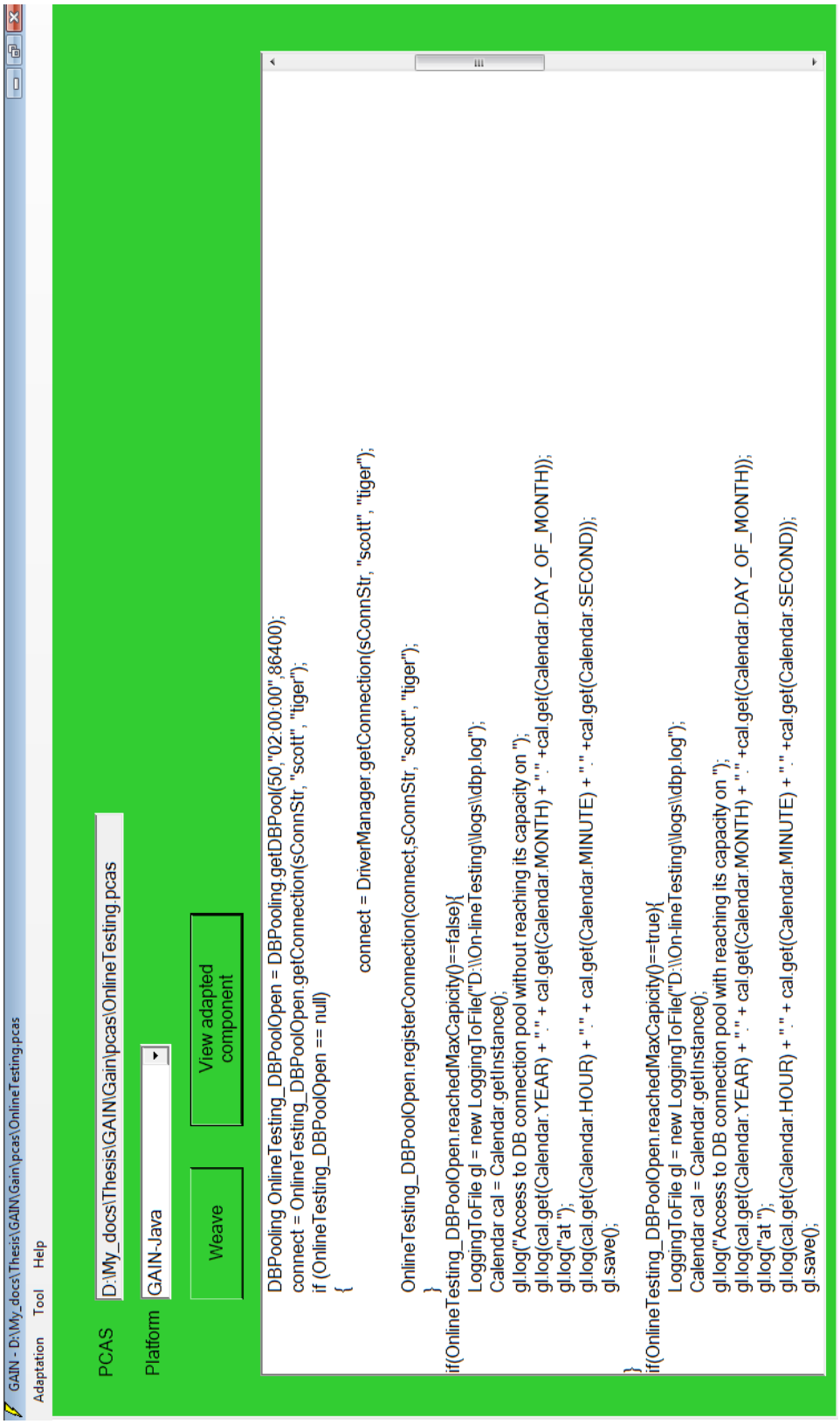


Figure B.23 PCAS weaving

Appendix C The case studies source code

C.1 Student record management system

C.1.1 PCAS

```
<?xml version="1.0"?>
<AOP-Process   name="Aspects_on_StudentSys"
                    xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
<AspectFramework name="Auth_loggingOnStudentinfo"
                        sourcefile="Student.java"
                        path="d:\My_doc\Thesis\GAIN\Gain\Work\"
                        joinpointcomponent="Student"
                        joinpointmethod="launchApp"
                        when="call"
                        returntype="*"
                        parameters=".."
                        awhen="before">
<Apply-aspect         class="Student"
                        method="launchApp"
                        aspect_id="02"
                        aspect_level="Primitive"
                        aspect_type="Authentication"
                        af_id="3"
                        af_name="StudentSysAuth"
                        synchronized="false"
                        comment="check user name and password"/>
<Switch expr="StudentSysAuth.getAuthenticationStatus()" when="before">
<case value="true">
<Apply-aspect         class="Student"
                        method="launchApp"
                        aspect_id="01"
                        aspect_level="primitive"
                        aspect_type="Logging"
                        af_id="1"
                        af_name="StudentSysLogging1"
                        synchronized="true"
                        comment="Log the access to DB"/>
</case>
<case value="false">
<Apply-aspect
                        class="Student"
                        method="launchApp"
```

```
                        aspect_id="01"
                        aspect_level="primitive"
                        aspect_type="Logging"
                        af_id="2"
                        af_name="StudentSysLogging2"
                        synchronized="true"
                        comment="log the rejection of access to DB"/>
<Apply-aspect          class="Student"
                        method="launchApp"
                        aspect_id="08"
                        aspect_level="Primitive"
                        aspect_type="Exit"
                        af_id="16"
                        af_name="StudentSysExit"
                        synchronized="false"
                        comment="Exit"/>
</case>
</Switch>
</AspectFramework>
</AOP-Process>
```

C.1.2 AFs

AF for authentication

```
<?xml version="1.0" ?>
<Aspect name="Auth_Student">
<!-- Common Structure -->

<CommonStructure>
  <PointCut>
    <Name>StudentSysAuth</Name>
    <When>call</When>
    <ReturnType>*</ReturnType>
    <ClassName>Student</ClassName>
    <MethodName>launchApp</MethodName>
    <Parameters>..</Parameters>
  </PointCut>
  <Advice>
    <When>before</When>
    <PointCutName ref = "StudentSysAuth"/>
  </Advice>
</CommonStructure>

<!-- Variations -->
<Variation type = "Authentication">
  <AuthenticationType>CommandLineBased</AuthenticationType>
  <AuthenticationDB>Account.mdb</AuthenticationDB>
</Variation>
</Aspect>
```

AF for logging

```
<?xml version="1.0" ?>
<Aspect name="TraceStudentInfo_Successful">
 <!-- Core asset -->
 <CommonStructure>
  <PointCut>
    <Name>StudentSysLogging1</Name>
    <When>call</When>
    <ReturnType>*</ReturnType>
    <ClassName>Student</ClassName>
    <MethodName>launchApp</MethodName>
    <Parameters>..</Parameters>
  </PointCut>
  <Advice>
    <When>before</When>
    <PointCutName ref="StudentSysLogging1" />
  </Advice>
 </CommonStructure>

 <!-- Variations -->
 <Variation type="Logging">
   <Device>
    <File>D:\\tmp\\student_info_aop.log</File>
   </Device>
   <Messages>
    <Message>Succeed to get access to Student.launchApp on</Message>
    <Date/>
    <Message>at </Message>
    <Time/>
   </Messages>
 </Variation>
</Aspect>
```

C.1.3 AInst

AInsts for authentication in AspectJ

```
import java.io.*;
import gain.authentication.Authentication;

public aspect Auth_Student{
   private boolean isAuthenticated = false;
   pointcut AuthStudentInfo():execution(* StudentInfo.getStudentInfo(..));
   before() : AuthStudentInfo () {
      String username="";
      String password="";
      try{
        BufferedReader in =
          new BufferedReader(new InputStreamReader(System.in));
```

```
          System.out.println("Username: ");
          username = in.readLine();
          System.out.println("Password: ");
          password = in.readLine();
        }catch(IOException ie){}
        if(username!=null && password!=null){
          Authentication auth = new Authentication();
          isAuthenticated = auth.authenticate(username, password);
        }
        if(!isAuthenticated) {
          System.out.println("Authentication failure!");
          System.exit(-1);
     }
   }
    public  boolean getAuthenticationStatus() {return isAuthenticated;}
}
```

AInsts for logging in AspectJ

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

public aspect TraceStudentInfo_Successful{

pointcut traceMethods() : call(* Student.launchApp(..));
before() : traceMethods ()
{
  Calendar cal = Calendar.getInstance();
  try{
     FileWriter   fw = new FileWriter("D:\\tmp\\student_info_aop.log", true);
     PrintWriter pw = new PrintWriter(fw);
     pw.print("Succeed to get access to Student.launchApp on");
     pw.print(cal.get(Calendar.YEAR) + ".");
     pw.print(cal.get(Calendar.MONTH) + ".");
     pw.print(cal.get(Calendar.DAY_OF_MONTH) + ",");
     pw.print("at ");
     pw.print(cal.get(Calendar.HOUR) + ":");
     pw.print(cal.get(Calendar.MINUTE) + ":");
     pw.print(cal.get(Calendar.SECOND) + ":");
     pw.println();
     pw.close();
  }catch(Exception e) {
     System.out.println("Error occured: " + e);
  }
}
}
```

C.1.4 Part of the adapted component source code in if selected target AOP language is Java

```
Student stu = new Student();
AuthenticationCmdDB StudentSysAuth=
new AuthenticationCmdDB("D:\\My_docs\\Thesis\\GAIN\\Gain\\\db\\Account.mdb");
StudentSysAuth.authenticate();
if(StudentSysAuth.getAuthenticationStatus()==true){
  LoggingToFile gl = new LoggingToFile("D:\\tmp\\student_info_aop.log");
  Calendar cal = Calendar.getInstance();
  gl.log("Succeed to get access to Student.launchApp on");
  gl.log(cal.get(Calendar.YEAR)     +     "."     +     cal.get(Calendar.MONTH)     +     "."
+cal.get(Calendar.DAY_OF_MONTH));
  gl.log("at ");
  gl.log(cal.get(Calendar.HOUR)     +     "."     +     cal.get(Calendar.MINUTE)     +     "."
+cal.get(Calendar.SECOND));
  gl.save();
}
if(StudentSysAuth.getAuthenticationStatus()==false){
  LoggingToFile gl = new LoggingToFile("D:\\tmp\\student_info_aop.log");
  Calendar cal = Calendar.getInstance();
  gl.log("Failed to get access to Student.launchApp on");
  gl.log(cal.get(Calendar.YEAR)     +     "."     +     cal.get(Calendar.MONTH)     +     "."
+cal.get(Calendar.DAY_OF_MONTH));
  gl.log("at ");
  gl.log(cal.get(Calendar.HOUR)     +     "."     +     cal.get(Calendar.MINUTE)     +     "."
+cal.get(Calendar.SECOND));
  gl.save();
  System.exit(-1);
}
stu.launchApp();
```

C.2 On-line testing system

C.2.1 PCAS

```
<?xml version="1.0"?>
<AOP-Process          name="Aspects_on_StudentSys"
                      xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
<AspectFramework name="Auth_loggingOnStudentinfo"
                      sourcefile="Student.java"
                      path="d:\My_doc\Thesis\GAIN\Gain\Work\"
                      joinpointcomponent="Student"
                      joinpointmethod="launchApp"
                      when="call"
                      returntype="*"
                      parameters=".."
                      awhen="before">
<Apply-aspect         class="Student"
                      method="launchApp"
```

```
                        aspect_id="02"
                        aspect_level="Primitive"
                        aspect_type="Authentication"
                        af_id="3"
                        af_name="StudentSysAuth"
                        synchronized="false"
                        comment="check user name and password"/>
<Switch expr="StudentSysAuth.getAuthenticationStatus()" when="before">
<case value="true">
<Apply-aspect         class="Student"
                        method="launchApp"
                        aspect_id="01"
                        aspect_level="primitive"
                        aspect_type="Logging"
                        af_id="1"
                        af_name="StudentSysLogging1"
                        synchronized="true"
                        comment="Log the access to DB"/>
</case>
<case value="false">
<Apply-aspect
                        class="Student"
                        method="launchApp"
                        aspect_id="01"
                        aspect_level="primitive"
                        aspect_type="Logging"
                        af_id="2"
                        af_name="StudentSysLogging2"
                        synchronized="true"
                        comment="log the rejection of access to DB"/>
<Apply-aspect
                        class="Student"
                        method="launchApp"
                        aspect_id="08"
                        aspect_level="Primitive"
                        aspect_type="Exit"
                        af_id="16"
                        af_name="StudentSysExit"
                        synchronized="false"
                        comment="Exit"/>
</case>
</Switch>
</AspectFramework>
</AOP-Process>
```

C.2.2 AFs

AF for DB connection pool

```
<?xml version="1.0" ?>
<Aspect name="OnlineTestingDBPoolOpenAspect">
```

```
  <!-- Common Structure -->
  <CommonStructure>
     <PointCut>
        <Name>connectionOpen</Name>
        <When>call</When>
        <ReturnType>java.sql.Connection</ReturnType>
        <ClassName>java.sql.DriverManager</ClassName>
        <MethodName>getConnection</MethodName>
        <Parameters>String url,String username,String password</Parameters>
     </PointCut>
     <Advice>
       <When>around</When>
       <PointCutName ref="connectionOpen" />
     </Advice>
  </CommonStructure>
  <!-- Variations -->
  <Variation type="DBConnectionPoolOpen">
       <Capacity>50</Capacity>
       <ExpireTime>
          <CheckPoint>02:00:00</CheckPoint>
          <MaxIdleTime>86400</MaxIdleTime>
        </ExpireTime>
  </Variation>
</Aspect>
```

AF for logging

```
<?xml version="1.0" ?>
<Aspect name="dbp_logging_1">
  <!-- Common Structure -->
  <CommonStructure>
    <PointCut>
       <Name>Logging_OnlineTesting-1</Name>
       <When>execution</When>
       <ReturnType>*</ReturnType>
       <ClassName>java.sql.DriverManager, java.sql.Connection</ClassName>
       <MethodName>getConnection</MethodName>
       <Parameters>..</Parameters>
    </PointCut>
    <Advice>
       <When>around</When>
       <PointCutName ref="Logging_OnlineTesting-1" />
    </Advice>
  </CommonStructure>
  <!-- Variations -->
  <Variation type="Logging">
     <Device>
       <File>D:\\On-lineTesting\\logs\\dbp.log</File>
     </Device>
     <Messages>
       <Message>Access to DB connection pool without reaching its capacity on
</Message>
       <Date/>
```

```
      <Message>at </Message>
      <Time/>
    </Messages>
  </Variation>
</Aspect>
```

C.2.3 Part of the adapted component source code in Java

```
DBPooling OnlineTesting_DBPoolOpen =
                                    DBPooling.getDBPool(50,"02:00:00",86400);
connect = OnlineTesting_DBPoolOpen.getConnection(
                                    sConnStr, "scott", "tiger");
if (OnlineTesting_DBPoolOpen == null) {
   connect = DriverManager.getConnection(sConnStr, "scott", "tiger");
   OnlineTesting_DBPoolOpen.registerConnection(
                                     connect,sConnStr, "scott", "tiger");
}
if(OnlineTesting_DBPoolOpen.reachedMaxCapicity()==false){
   LoggingToFile gl = new LoggingToFile("D:\\On-lineTesting\\logs\\dbp.log");
   Calendar cal = Calendar.getInstance();
   gl.log("Access to DB connection pool without reaching its capacity on ");
   gl.log(cal.get(Calendar.YEAR) + "." + cal.get(Calendar.MONTH) + "."
           +cal.get(Calendar.DAY_OF_MONTH));
   gl.log("at ");
   gl.log(cal.get(Calendar.HOUR) + "." + cal.get(Calendar.MINUTE) + "."
            +cal.get(Calendar.SECOND));
   gl.save();
}
if(OnlineTesting_DBPoolOpen.reachedMaxCapicity()==true){
   LoggingToFile gl = new LoggingToFile("D:\\On-lineTesting\\logs\\dbp.log");
   Calendar cal = Calendar.getInstance();
   gl.log("Access to DB connection pool with reaching its capacity on ");
   gl.log(cal.get(Calendar.YEAR) + "." + cal.get(Calendar.MONTH) + "."
          +cal.get(Calendar.DAY_OF_MONTH));
   gl.log("at ");
   gl.log(cal.get(Calendar.HOUR) + "." + cal.get(Calendar.MINUTE) + "."
         +cal.get(Calendar.SECOND));
   gl.save();
}
```

C.3 Policy enforcement

C.3.1 PCAS

```
<?xml version="1.0"?>
<AOP-Process   name="PolicyEnforcement"
                    xmlns="http://www.dcs.napier.ac.uk/2005/PCAS">
<AspectFramework name="PE_CodeConventionsOOP"
                        sourcefile=""
                        path=""
                        joinpointcomponent=""
```

```
                        joinpointmethod=""
                        when=""
                        returntype=""
                        parameters=""
                        awhen="">
<Apply-aspect    class=""
                    method=""
                    aspect_id="06"
                    aspect_level="Primitive"
                    aspect_type="PE_NoStandardOutput"
                    af_id="14"
                    af_name="PolicyEnforcement_NoStandardOutput"
                    synchronized="false"
                    comment="PE_No standard output"/>
<Apply-aspect    class=""
                    method=""
                    aspect_id="05"
                    aspect_level="Primitive"
                    aspect_type="PE_NewObj"
                    af_id="13"
                    af_name="PolicyEnforcement_NewObj"
                    synchronized="false"
                    comment="PE_NewObjectCreation"/>
</AspectFramework>
</AOP-Process>
```

C.3.2 AFs

AF for NoStandardOutput

```
<?xml version="1.0" ?>
<Aspect name="PolicyEnforcement_NoStandardOutput">
  <!-- Common Structure -->
  <CommonStructure>
     <PointCut>
        <Name>pe_nso1</Name>
        <When>execution</When>
        <ReturnType>*</ReturnType>
        <ClassName>*</ClassName>
        <MethodName>*</MethodName>
        <Parameters>*</Parameters>
     </PointCut>
     <Advice>
       <When>before</When>
       <PointCutName ref="pe_nso1" />
     </Advice>
  </CommonStructure>

  <!-- Variations -->
  <Variation type="PE_NoStandardOutput">
      <AffectedClasses>
```

```
        <Class>ShoppingCart</Class>
        <Class>ShoppingCartOperator</Class>
      </AffectedClasses>
      <ErrorMessage>Please use logging mechnism instead!</ErrorMessage>
  </Variation>
</Aspect>
```

AF for NewObj

```
<?xml version="1.0" ?>
<Aspect name="PolicyEnforcement_NewObj">
  <!-- Common Structure -->
  <CommonStructure>
     <PointCut>
        <Name>pe_newobj</Name>
        <When>execution</When>
        <ReturnType>*</ReturnType>
        <ClassName>*</ClassName>
        <MethodName>*</MethodName>
        <Parameters>*</Parameters>
     </PointCut>
     <Advice>
       <When>before</When>
       <PointCutName ref="pe_newobj" />
     </Advice>
  </CommonStructure>

  <!-- Variations -->
  <Variation type="PE_NewObj">
       <AffectedClasses>
         <Class>C1</Class>
         <Class>C2</Class>
       </AffectedClasses>
       <ErrorMessage>Please use factory method instead!</ErrorMessage>
  </Variation>
</Aspect>
AInsts
```

C.3.3 AInsts

AInst for NoStandardOutput aspect in AspectJ

```
public aspect PolicyEnforcement_NoStandardOutput{

declare warning : get(* System.out) || get(* System.err) :
       "Consider using logging mechanism instead.";

}
```

Appendix D The core implementation code of the system

D.1 XML schema for PCAS

Following code shows the XML schema for PCAS:

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="AOP-Process">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="AspectFramework"  maxOccurs="unbounded"/>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>

 <xs:element name="Apply-aspect">
  <xs:complexType>
   <xs:attribute name="method" type="xs:string" use="required" />
   <xs:attribute name="class" type="xs:string" use="required" />
   <xs:attribute name="synchronized" type="xs:string" use="required" />
   <xs:attribute name="comment" type="xs:string" use="required" />
   <xs:attribute name="af_name" type="xs:string" use="required" />
   <xs:attribute name="aspect_level" type="xs:string" use="required" />
   <xs:attribute name="af_id" type="xs:string" use="required" />
   <xs:attribute name="aspect_type" type="xs:string" use="required" />
   <xs:attribute name="aspect_id" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>

 <xs:element name="AspectFramework">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="Apply-aspect" maxOccurs="unbounded"/>
    <xs:element ref="Switch"  maxOccurs="unbounded"/>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required" />
   <xs:attribute name="awhen" type="xs:string" use="required" />
   <xs:attribute name="returntype" type="xs:string" use="required" />
   <xs:attribute name="when" type="xs:string" use="required" />
   <xs:attribute name="joinpointcomponent" type="xs:string" use="required" />
   <xs:attribute name="joinpointmethod" type="xs:string" use="required" />
```

```
    <xs:attribute name="parameters" type="xs:string" use="required" />
    <xs:attribute name="sourcefile" type="xs:string" use="required" />
    <xs:attribute name="path" type="xs:string" use="required" />
   </xs:complexType>
  </xs:element>
```

```
  <xs:element name="case">
   <xs:complexType>
    <xs:sequence>
     <xs:element ref="Apply-aspect"  maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="value" type="xs:string" use="required" />
   </xs:complexType>
  </xs:element>

  <xs:element name="Switch">
   <xs:complexType>
    <xs:sequence>
     <xs:element ref="case" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="when" type="xs:string" use="required" />
    <xs:attribute name="expr" type="xs:string" use="required" />
   </xs:complexType>
  </xs:element>

</xs:schema>
```

D.2 XML schema (AAF) for Logging aspect

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Advice">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="When" />
        <xs:element ref="PointCutName" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Aspect">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="CommonStructure" />
        <xs:element ref="Variation" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>
```

```
<xs:element name="ClassName">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="CommonStructure">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="PointCut" />
      <xs:element ref="Advice" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Date" type="xs:string" />

<xs:element name="Device">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="File" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="File">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="Message">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="Messages">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="Date" maxOccurs="unbounded"/>
      <xs:element ref="Message" maxOccurs="unbounded"/>
      <xs:element ref="Time" maxOccurs="unbounded"/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="MethodName">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="Name">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="Parameters">
  <xs:complexType mixed="true" />
</xs:element>
```

```
  <xs:element name="PointCut">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="When" />
        <xs:element ref="ReturnType" />
        <xs:element ref="ClassName" />
        <xs:element ref="MethodName" />
        <xs:element ref="Parameters" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="PointCutName">
    <xs:complexType>
      <xs:attribute name="ref" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="ReturnType">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Time" type="xs:string" />

  <xs:element name="Variation">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Device" />
        <xs:element ref="Messages" />
      </xs:sequence>
      <xs:attribute name="type" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="When">
    <xs:complexType mixed="true" />
  </xs:element>

</xs:schema>
```

D.3 XML schema (AAF) for DB connection pool aspect

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Advice">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="When" />
        <xs:element ref="PointCutName" />
```

```
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="Aspect">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="CommonStructure" />
      <xs:element ref="Variation" />
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="Capacity">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="CheckPoint">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="ClassName">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="CommonStructure">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="PointCut" />
      <xs:element ref="Advice" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="ExpireTime">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="CheckPoint" />
      <xs:element ref="MaxIdleTime" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="MaxIdleTime">
  <xs:complexType mixed="true" />
</xs:element>

<xs:element name="MethodName">
  <xs:complexType mixed="true" />
</xs:element>
```

```
  <xs:element name="Name">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Parameters">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="PointCut">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Name" />
        <xs:element ref="When" />
        <xs:element ref="ReturnType" />
        <xs:element ref="ClassName" />
        <xs:element ref="MethodName" />
        <xs:element ref="Parameters" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="PointCutName">
    <xs:complexType>
      <xs:attribute name="ref" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="ReturnType">
    <xs:complexType mixed="true" />
  </xs:element>

  <xs:element name="Variation">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Capacity" />
        <xs:element ref="ExpireTime" />
      </xs:sequence>
      <xs:attribute name="type" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="When">
    <xs:complexType mixed="true" />
  </xs:element>

</xs:schema>
```

D.4 Semantic Interpreter

As an example, the Semantic Interpreter of logging Aspect for AspectJ is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:aaf="http://dcs.napier.ac.uk/2005/AAF"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ev="http://www.w3.org/2001/xml-events">
<xsl:output method="text"/>
<xsl:template match="/">
import java.io.*;
import java.util.*;
import org.aspectj.lang.*;
<xsl:for-each select="//Aspect">
public aspect <xsl:value-of select="@name" />{
<xsl:apply-templates select="CommonStructure" mode="CAS"/>
}
</xsl:for-each>
</xsl:template>
<xsl:template match="CommonStructure" mode="CAS">
<xsl:apply-templates select="PointCut"      mode="PointCut"/>
<xsl:apply-templates select="Advice"        mode="Advice"/>{
<xsl:apply-templates select="../Variation"   mode="AA"/>
}
</xsl:template>
<xsl:template match="PointCut" mode="PointCut">
pointcut <xsl:value-of select="Name"/>() : <xsl:value-of select="When"/>(<xsl:value-of
select="ReturnType"/><xsl:value-of      select="substring('      ',1)"/>      <xsl:value-of
select="ClassName"/>.<xsl:value-of                  select="MethodName"/>(<xsl:value-of
select="Parameters"/>));
</xsl:template>

<xsl:template match="Advice"   mode="Advice">
<xsl:value-of select="When"/>() : <xsl:value-of select="PointCutName/@ref"/> ()
</xsl:template>

<xsl:template match="Variation" mode="AA">
 Calendar cal = Calendar.getInstance();
 try{
    FileWriter   fw = new FileWriter("<xsl:value-of select="Device/File/."/>", true);
    PrintWriter pw = new PrintWriter(fw);
    <xsl:for-each select="Messages/*">
   <xsl:choose>
       <xsl:when test="self::Message">
    pw.print("<xsl:value-of select="."/>");
  </xsl:when>
       <xsl:when test="self::Date">
    pw.print(cal.get(Calendar.YEAR) + ".");
    pw.print(cal.get(Calendar.MONTH) + ".");
    pw.print(cal.get(Calendar.DAY_OF_MONTH) + ",");
  </xsl:when>
<xsl:when test="self::Time">
```

```
      pw.print(cal.get(Calendar.HOUR) + ":");
      pw.print(cal.get(Calendar.MINUTE) + ":");
      pw.print(cal.get(Calendar.SECOND) + ":");
</xsl:when>

<xsl:otherwise>
      pw.print("<xsl:value-of select="."/>");
</xsl:otherwise>
</xsl:choose>
      </xsl:for-each>
      pw.println();
      pw.close();
  }catch(Exception e) {
     System.out.println("Error occured: " + e);
  }
</xsl:template>
</xsl:stylesheet>
```

Appendix E The sample test cases of the tool

E. 1 Sample test cases in unit testing

Test case 1: Testing Component Analyzer (getMethodList and getComponentname method)

Description: Test whether Component Analyzer provides basic information of a component.
Input: A source code level component file name (ConnOracle.java).
Steps:
(1) Create a new object of ComponentAnalyzer class by passing component file name (ConnOracle.java) as parameter to the constructor.
(2) Invoke getMethodList method and display the return value.
(3) Invoke getComponentName and display the return value.
Expected result:
Basic component information, e.g. component name(ConnOracle), method signatures (ConnOracle, executeQuery, executeUpdate).
Real result:
Classname: ConnOracle
MethodList: ConnOracle
executeQuery
executeUpdate
Status: passed.

Test case 2: Testing Semantic Interpreter for Logging in AspectJ

Description: Test whether Semantic Interpreter can be used to transform Aspect from AF to AInst
Input: An AF file (StudentSysLogging1.af)
Steps: using testing harness below:

```
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
public class TestSI {
  public static void main(String[] args) throws Exception {
    String docname = " StudentSysLogging1.af ";
    String xslname = "AspectJ_logging.xsl";
    File xmlFile = new File(docname);
    File xslFile = new File(xslname);
    Source xmlsource = new StreamSource(xmlFile);
    Source xslsource = new StreamSource(xslFile);
    File targetFile = new File( " StudentSystLogging1.ainst " );
    PrintWriter out = new PrintWriter(new FileWriter(targetFile));
```

```
    Result result = new StreamResult(out);
    TransformerFactory transFact = TransformerFactory.newInstance();
    Transformer trans = transFact.newTransformer(xslsource);
    trans.transform(xmlsource, result);
  }
}
```

Expected result: An AInst file (StudentSystLogging1.ainst)
Real result: StudentSysLogging1.ainst
Status: passed.

Test case 3: Testing Aspect Manager (LoadAF method)

Description: Test whether LoadAF method in Aspect Manager loads all AAFs/AFs into the tree structure.
Input: None
Steps:
(1) Create a new AspectManagerUI object
(2) Invoke show() method of that object
(3) Invoke loadAF() method of that object
Expected result: All AAFs and Afs in the Aspect Repository are loaded and displayed in the Aspect tree.
Real result: All AAFs and Afs are loaded and displayed in the Aspect tree.
Status: passed.

Test case 4: Testing PCASAnalyzer (getAF method)

Description: Test whether getAF method returns an AF object.
Input: " 1" as AF_ID
Expected result: The corresponding AF object to AF_ID
Steps:
(1) Create a new PCASAnalyzer object by passing "StudentInfo.pcas" to its constructor.
(2) Invoke getAF() method by passing "1" to this method
(3) Get the returned object, and invoke getters to test whether the returned object is correct.
Real result: The corresponding object to AF_ID ("1")
Status: passed.

Test case 5: Testing XMLOperator class (getCAS_from_AF method)

Description: Test whether getCAS_from_AF method returns the CS part of an AF
Input: An AF file name ("StudentSysLogging1.af")
Expected result: The string of CS part of provided AF
Steps:
(1) Create a new XMLOperator object by passing "StudentSysLogging1" to its constructor
(2) Invoke getCAS_from_AF method and output the return value of this method
Real result: The string of CS part of provided AF as shown below

```
<CommonStructure>
  <PointCut>
    <Name>StudentSysLogging1</Name>
```

```
      <When>call</When>
      <ReturnType>*</ReturnType>
      <ClassName>Student</ClassName>
      <MethodName>launchApp</MethodName>
      <Parameters>..</Parameters>
   </PointCut>

   <Advice>
      <When>before</When>
      <PointCutName ref="StudentSysLogging1" />
   </Advice>
  </CommonStructure>
```
Status: passed.

E. 2 Sample test cases in integration testing

Test case 1: Test the integration between Aspect Manager and Aspect Generator.

Desired Functionality: The selected AF in Aspect Manager can be passed to Aspect Generator and based on this, Aspect Generator can generate an AInst from the selected AF.

Steps:
(1) Launch Aspect Manager
(2) Select an AF ("StudentSysLogging1") by clicking it
(3) Click on "Generate AInst" to launch Aspect Generator
(4) Select "AspectJ" as target AOP platform from the listbox
(5) Click on "Generate !"
(6) The source code of generated AInst should be shown in the textbox

Status: passed.

Test case 2: Test the integration between PCAS Editor, Aspect Generator, and PCAS weaver (From a pre-defined PCAS file).

Desired Functionality: The tool should be able to deal with the whole process of component adaptation from PCAS loading to Aspect generation, to Aspect weaving.

Steps:
(1) Click on "Open" button to open an existing PCAS, select "OnlinTesting.pcas"
(2) Click on "Adaptation" menu
(3) Click on "Aspect Generation" menuitem to launch Aspect generation window
(4) Click on "Get candidate aspects" button
(5) In target platform listbox, select "AspectJ"
(6) Click on "Generate AInst(s)" button
(7) In "Generated aspects" listbox, Click on "OnlineTesting_Logging1.ainst"
(8) Click "View select AInst", the source code of selected AInst should be shown in the

textbox on the right side of the window.
(9) Click on “Adaptation” menu
(10) Click on “Aspect Weaving” to launch Aspect weaving window
(11) Select “AspectJ” in platform listbox
(12) Click on “Pre-weave” button
(13) Click on “View adapted Aspect in AspectJ” button, the adapted Aspect should be shown in the textbox.
(14) Compile adapted Aspect with original component.
(15) Run target system with adapted component.

Status: passed.

E. 3 Sample test cases in system testing

Test case 1: Test the main functionality of the tool based on student record system.

Desired Functionality: The tool should be able to deal with the whole process of component adaptation.

Steps:
(1) Click on “Open” button to open an existing PCAS, select “StudentSys.pcas”
(2) Click on “Adaptation” menu
(3) Click on “Aspect Generation” menuitem to launch Aspect generation window
(4) Click on “Get candidate aspects” button
(5) In target platform listbox, select “AspectJ”
(6) Click on “Generate AInst(s)” button
(7) In “Generated aspects” listbox, Click on “StudentSysAuth.ainst”
(8) Click “View select AInst”, the source code of selected AInst should be shown in the textbox on the right side of the window.
(9) Click on “Adaptation” menu
(10) Click on “Aspect Weaving” to launch Aspect weaving window
(11) Select “AspectJ” in platform listbox
(12) Click on “Pre-weave” button
(13) Click on “View adapted Aspect in AspectJ” button, the adapted Aspect should be shown in the textbox.
(14) Compile adapted Aspect with original component.
(15) Run target system with adapted component.
(16) Go back to Aspect weaving window, and select "GAIN-Java" in platform listbox.
(17) Click on "Weave" button
(18) Click on "View adapted component" button, the adapted component should be shown in the textbox.
(19) Compile adapted component.
(20) Run target system with adapted component.

Status: passed.