REAL WORLD EVALUATION OF ASPECT-ORIENTED SOFTWARE DEVELOPMENT

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Computer Science at Massey University, Palmerston North, New Zealand

CHRISTOPHER MARK ELGAR

2006

Abstract

Software development has improved over the past decade with the rise in the popularity of the Object-Oriented (OO) development approach. However, software projects continue to grow in complexity and continue to have alarmingly low rates of success.

Aspect-Oriented Programming (AOP) is touted to be one solution to this software development problem. It shows promise of reducing programming complexity, making software more flexible and more amenable to change. The central concept introduced by AOP is the aspect. An aspect is used to modularise crosscutting concerns in a similar fashion to the way classes modularise business concerns. A crosscutting concern cannot be modularised in approaches such as OO because the code to realise the concern must be spread throughout the module (e.g. a tracing concern is implemented by adding code to every method in a system). AOP also introduces join points, pointcuts, and advice which are used with aspects to capture crosscutting concerns so they can be localised in a modular unit.

OO took approximately 20 years to become a mainstream development approach. AOP was only invented in 1997. This project considers whether AOP is ready for commercial adoption. This requires analysis of the AOP implementations available, tool support, design processes, testing tools, standards, and support infrastructure. Only when AOP is evaluated across all these criteria can it be established whether it is ready to be used in commercial projects. Moreover, if companies are to invest time and money into adopting AOP, they must be aware of the benefits and risks associated with its adoption. This project attempts to quantify the potential benefits in adopting AOP, as well as identifying areas of risk.

SolNet Solutions Ltd, an Information Technology (IT) company in Wellington, New Zealand, is used in this study as a target environment for integration of aspects into a commercial development process. SolNet is in the business of delivering large scale enterprise Java applications. To assist in this process they have developed a Common Services Architecture (CSA) containing components that can be reused to reduce risk and cost to clients. However, the CSA is complicated and SolNet have

identified aspects as a potential solution to decrease the complexity.

Aspects were found to bring substantial improvement to the Service Layer of SolNet applications, including substantial reductions in complexity and size. This reduces the cost and time of development, as well as the risk associated with the projects. Moreover, the CSA was used in a more consistent fashion making the system easier to understand and maintain, and several crosscutting concerns were modularised as part of a reusable aspect library which could eventually form part of their CSA.

It was found that AOP is approaching commercial readiness. However, more work is needed on defining standards for aspect languages and modelling of design elements. The current solutions in this area are commercially viable, but would greatly benefit from a standardised approach. Aspect systems can be difficult to test and the effect of the weaving process on Java serialisation requires further investigation.

Acknowledgements

I wish to acknowledge my supervisors Dr. Jens Dietrich (Massey University) and Shane Griggs (SolNet Solutions Ltd) for their time, support, and ideas during this project. Without them, this project would not have been so successful.

Thank you to Technology New Zealand for their financial support of this project through the Technology in Industry Fellowship (TIF).

To all the staff I have been involved with at SolNet Solutions thank you for your time and feedback as I tried out ideas which will hopefully make their way into your everyday work! In particular, Peter Abbott and Antony Binns for their support when deploying and working with some complex systems, Simon Brierley for embracing AOP and taking it into a real project, and all the Senior Developers and Architects who were interviewed to find out more about how SolNet operates.

Thank you to my parents and relatives for their support during the preparation of this thesis. In particular, your proof reading and comments were greatly appreciated.

To anyone I have missed, apologies, and thank you for your contributions!

Table of contents

List of Figures

List of Tables

Listings

CHAPTER 1
INTRODUCTION

1.1 The Software Development Problem

Software development has long been prone to spectacular project failure rates that would be unacceptable in any other professional discipline. The 1994 Chaos Report from The Standish Group showed that just 16% of projects were successful. Of those unsuccessful projects 31% were never completed and 53% had problems such as cost or time overruns and missing functionality (The Standish Group 1994). An example of a high profile project failure in New Zealand was the Integrated National Crime Information System (INCIS). This ambitious project suffered numerous time delays and cost overruns before it was eventually abandoned with only a small portion of the system in operation (Small 2000).

We believe many project failures can be attributed to the sheer size and complexity of software system developments and the inability of traditional development methodologies to cope with this. Object-Oriented (OO) technology has become the major development methodology helping to reduce complexity with new concepts such as inheritance, abstraction, and polymorphism (Boner, Vasseur & Dahlstedt 2005*a*). The latest Chaos Report in 2003 shows a substantial improvement since 1995 with 34% of projects categorised as successful and only 15% of projects failing. However, 51% of projects still have some problems (The Standish Group 2003). Despite the advances made in recent years, professionals continue to strive to find ways to improve project success rates.

In this thesis, Aspect-Oriented Programming (AOP) is presented as a development approach which has the potential to reduce software complexity and increase software project success.

1.2 Aspect-Oriented Programming

Aspect-Oriented Programming is a relatively new programming paradigm invented at the Xerox Palo Alto Research Center (Xerox-PARC) in the mid nineties by Gregor

Kiczales and his research team. It attempts to reduce program complexity using the notion of separating crosscutting concerns from the core program concerns (Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier & Irwin 1997). This is considered to be one of the most promising approaches to reducing program complexity, and was ranked in the 10 emerging technologies that will change the world by Massachusetts Institute of Technology's (MIT) Technology Review (van der Werff 2001).

AOP adds the concept of an aspect for the purposes of designing and implementing crosscutting concerns. Aspects complement the more familiar concepts of procedures and objects found in the Structured and OO paradigms (Kiczales 2005). AOP is not a replacement for these other paradigms, but rather complements them with a new modularisation technique. Core program concerns can be implemented using traditional modularisation techniques and crosscutting concerns using aspects.

1.2.1 Object-Oriented Programming

Although aspects can be used with other programming paradigms such as Structured Programming, most implementations available are based around current OO languages. The reasons why the OO paradigm is not suitable for all problems faced in modularising code and how aspects complement this technology to solve these problems is discussed. OO was designed to model real-world domain entities and their behaviour as objects. However, there are many elements of a design that must be intermixed with these objects which are incongruent with the object's original intent. AOP addresses this problem by allowing behaviour to be added to objects in a non-intrusive, modularised fashion (Glover 2004).

A good example is a banking system with an 'Account' class containing a 'withdraw' method. Being a banking system there are many things that must happen before and after the 'withdraw' method modifies the account's balance such as security checks, auditing, transaction handling, and persistency. All these extra concerns are not directly part of the main concern of withdrawing funds from an account, but they must be coded with the logic for withdrawing money to ensure the system meets its non-functional requirements. Clearly these extra concerns will require more code than the actual withdrawal of money, and concerns such as transaction handling will be spread across multiple classes making it difficult to maintain and evolve. With AOP it is possible to remove these concerns from the core classes and modularise them as aspects. This will make the system easier to design, code, test, and maintain.

Although the concepts of AOP are not inherently linked to any particular OO

language, most of the current mainstream implementations are based around the Java language. This is probably attributable to the strong Java open source community rather than any inherent features of the Java language itself since other languages are having implementations developed such as Python, PHP, C#, Ruby, Perl, and Lisp (Wikipedia 2005*a*). Moreover, AOP implementations based on Java have received strong vendor support from groups such as IBM, BEA Systems, Xerox, and JBoss.

1.3 SolNet Solutions Ltd

1.3.1 Company Profile

SolNet Solutions is an Information Technology (IT) company based in Wellington and Auckland, New Zealand, with approximately 125 staff. Their core business is the delivery of J2EE solutions for large enterprise systems.

1.3.2 Current Development Environment

SolNet have invested substantial time and money into their existing development processes and tools to enable them to produce high quality, reliable systems, as cheaply and timely as possible. SolNet has developed a set of standard reusable components that can be used in typical J2EE projects. These components enable them to significantly reduce the cost and risk involved in conducting J2EE projects. This set of components is referred to as the Common Services Architecture (CSA).

1.3.3 Motivation for AOP Assessment

SolNet's current infrastructure (CSA) is complicated and relies on individual developers being familiar with the components available and how to correctly use them. SolNet are continually looking for ways to reduce the complexity and make their CSA easier for developers to use and more reusable across different types of projects. They would also like to increase flexibility such as having the ability to easily change the components used in a project. For example, changing from EJB Persistency to Hibernate by plugging in a different aspect.

Senior development staff at SolNet have recognised that AOP has potential to simplify their CSA and make it more accessible to different projects. SolNet Solutions entered into this Technology in Industry Fellowship (TIF) project to have an assessment undertaken of AOP technology and how it fitted into their development

lifecycle and to assess the potential benefits it could produce in their commercial environment.

1.4 Project Objective and Scope

There is a substantial amount of research being conducted on AOP, and tools are continually being developed. However, the availability of tools does not necessarily mean that AOP is ready to be used commercially (in the real world). To be used commercially there must also be availability of training resources, books, quality assurance tools, integrated development environments, patterns, diagramming techniques, and support infrastructure. Furthermore, the technologies must meet non-functional requirements such as scalability, fault tolerance, and openness.

In this thesis AOP is examined over several of these areas to try and establish its readiness for commercial adoption. Moreover, we try to quantify the commercial benefits of AOP by refactoring a real world project to measure the benefits as a result of using an AOP approach. In doing so we can identify areas where SolNet can benefit from AOP and assess the risks and affect on different areas of their development process.

The objective of this thesis is to show how Aspect-Oriented Software Development can be integrated into a real-world environment at SolNet Solutions with the ultimate goal of assessing the readiness of aspects for use in a commercial environment.

The areas investigated are:

- Approaches to Aspect-Oriented Programming.
- Tool support for Aspect-Oriented development.
- Fitting aspects into the design process.
- Aspect-Oriented standards.
- Testing aspects.
- Metrics for evaluating aspect software.
- Refactoring a real-world project to use AOP.
- Measuring the risks and benefits of using AOP with respect to the refactored project.

Prior to starting this project it was estimated that aspects could reduce the total cost of ownership for SolNet projects by 6%. In this thesis we try to quantify the benefits of AOP and evaluate them against this hypothesis. However, this may not be possible because of the limited historical data available from SolNet to provide a baseline for comparison.

The results obtained are reported in the context of the SolNet Solutions' environment. However, this environment is considered to be representative of many J2EE development companies. It is believed that other companies face similar problems and would have comparable benefits and risks in adopting an Aspect-Oriented approach. Therefore, it is inferred that the results obtained will be applicable to other commercial environments.

Due to the rapidly changing nature of the Aspect-Oriented community, certain limitations were realised when making some assessments. These were made from a practical perspective to enable the work to be completed despite changes happening concurrently with the technologies being evaluated. This was most critical when evaluating the different AOP approaches and tool support. During these two phases the major implementations continually released new versions and features as well as fixing bugs. To continually update and incorporate the new information would have been an endless task. For this reason the current version at the time of conducting the work was evaluated. Some upcoming features are mentioned, but they are not evaluated.

There are many different approaches similar to aspects for achieving separation of concerns such as Composition Filters (Aksit 2001), Hyperslices (Tarr & Ossher 2001), and Subjects (Wikipedia 2005*b*). However, these approaches are outside the scope of this thesis and are only briefly examined.

1.5 Overview of Thesis

This chapter has introduced the objectives and scope of this thesis. In the remaining chapters findings from applying aspects at different places in the software development lifecycle are discussed.

Chapter 2 reviews AOP concepts and terminology for unfamiliar readers.

The different techniques used for AOP are explored in Chapter 3. This compares and contrasts the most popular implementations available. One of the more experimental implementations is examined to see what motivates the development of the smaller frameworks and how their approach differs from the mainstream implementations.

The motivation for Aspect-Oriented tool support is explored in Chapter 4. This includes design, build, development, testing, documentation, and quality assurance tools. The quality of the current tools is assessed and the need for improved tools is identified.

In Chapter 5 the various notations developed to guide the design of Aspect-Oriented software are reviewed, in particular some extensions to the de facto standard Unified Modelling Language (UML) are considered. The use of a notation based on standard UML extensions is proposed. Finally, it is discussed how this notation can be integrated with SolNet Solution's design techniques.

Test driven development has become an important approach for developing quality software. In Chapter 6 the techniques available for testing Aspect-Oriented software are assessed.

To enable us to assess whether our Aspect-Oriented refactoring of a SolNet project has made any improvements, a set of objective, quantitative measurements is required for evaluating AO solutions. In Chapter 7 the use of traditional metrics is proposed. The potential to use new Aspect-Oriented metrics is discussed.

Chapter 8 explores the standards that have been developed for AOP. It then makes recommendations for future standardisation paths to make AOP easier to adopt.

The major goal of this project is to assess the benefits and risks to SolNet Solutions in adopting AOP. Chapter 9 shows the integration of AOP into two real projects which SolNet is undertaking with the New Zealand Qualifications Authority (NZQA). A qualitative and quantitative analysis of the benefits and risks in using AOP for these projects is presented. Finally, further possibilities for utilising AOP at SolNet Solutions are discussed.

Chapter 10 presents a summary of findings from this project and recommendations for future work. The applicability of the findings to environments outside of SolNet Solutions is discussed.

CHAPTER 2
AOP OVERVIEW

2.1 Introduction

This short chapter is intended to give a brief overview of AOP concepts to unfamiliar readers. Readers familiar with AOP concepts may choose to omit this chapter.

2.2 Important AOP Concepts and Terminology

AOP introduces several new concepts which allow modularisation of crosscutting concerns. The terminology is usually consistent between different AOP approaches for major concepts. However, each implementation may introduce new terms, or have alternative names for the same concepts. It is these differences and the unusual choices of terminology that can cause confusion for new users. The concepts described here are considered by the author to be the core concepts which are implementation independent and are based on definitions used by Filman, Elrad, Clarke & Akşit (2005).

2.2.1 Concerns

Engineering processes have many things about which they care. These things are called concerns and may range from high level requirements to low level implementation issues. There are different categories of concerns. However, AOP is usually directed at systematic behaviour such as 'all failed login attempts will be logged'.

2.2.2 Scattering and Tangling

When a concern is implemented it often needs to be realised in multiple places in the code base and is considered to be scattered. The concern may be implemented in a method or a class with another concern, these are considered to be tangled. A simple example is logging. Logging messages are required in numerous places so the concern has been scattered. Logging messages will also be tangled with other

concerns such as withdrawing money from bank accounts, even though this is not the key objective of the withdrawal method. These terms are quite distinct but are interconnected and are associated with crosscutting concerns.

2.2.3 Crosscutting Concerns

A crosscutting concern is any concern whose implementation must be scattered throughout the rest of the system's implementation. In OO systems these are concerns that can not be localised as a method or class but are instead implemented in methods or classes that involve other concerns. Prime examples are logging, security, persistency, concurrency, and error handling. AOP aims to modularise these concerns.

2.2.4 Aspect

An aspect is the unit of modularity for Aspect-Oriented (AO) software. Aspects are units designed to implement crosscutting concerns in a modular fashion. It is very similar to the concept of a class in OO, but instead of containing core concerns, an aspect contains crosscutting concerns.

2.2.5 Join Point

A join point is some identifiable place in the execution or structure of a program where additional behaviour can be attached. A particular AOP implementation will have a join point model defining what join points are available. Common examples are method calls, constructor calls, and field accesses.

2.2.6 Pointcut

A pointcut describes a set of join points. A programmer can specify that behaviour should be applied to all join points specified by the pointcut without explicit knowledge of each join point. This provides a layer of abstraction between the specification of the join points where behaviour must be applied and the use of these join points.

2.2.7 Advice

Advice is the behaviour to execute at a join point. Most AOP implementations provide the means to run advice before, after, or instead of a join point. Advice is

oblivious since there is no explicit reference at the join point to show that it will be run.

2.2.8 Intertype Declaration

It is possible to change the static structure of a program using intertype declarations (also commonly referred to as introductions). The modifications normally available are adding methods or fields to a class, and adding a parent class or interface to a class.

2.2.9 Weaving

Weaving is the process of combining the core functionality with the aspects to produce the fully functional system. Weaving can occur at various times (pre-compiler time, compile time, post-compile time, load time, and run time).

2.2.10 Obliviousness

Obliviousness means that by examining the base code, a programmer cannot tell that the aspect code will execute (i.e. it is transparent). Obliviousness is desirable as it allows for greater separation of concerns since it is possible to reason about a body of code free of the aspect code. This is in contrast to non-aspect approaches where explicit calls are made to subprograms to implement the functionality. Obliviousness was considered one of the most important properties of aspect programs when compared with other approaches of achieving crosscutting modularity. However, it also has some practical disadvantages which means less emphasis is placed on it in practice (Sullivan, Griswold, Song & Cai 2005).

2.2.11 Dynamic and Static Crosscutting

Dynamic crosscutting occurs when changes to a program's execution are made using AOP, where as static crosscutting refers to changes made to the static structure of the system using intertype declarations.

2.3 AOP System Overview

Figure 2.1 shows the general structure of an AO system. Of particular note is the separation of the core and crosscutting concerns into different units. The core units are implemented using traditional units such as classes and the crosscutting concerns

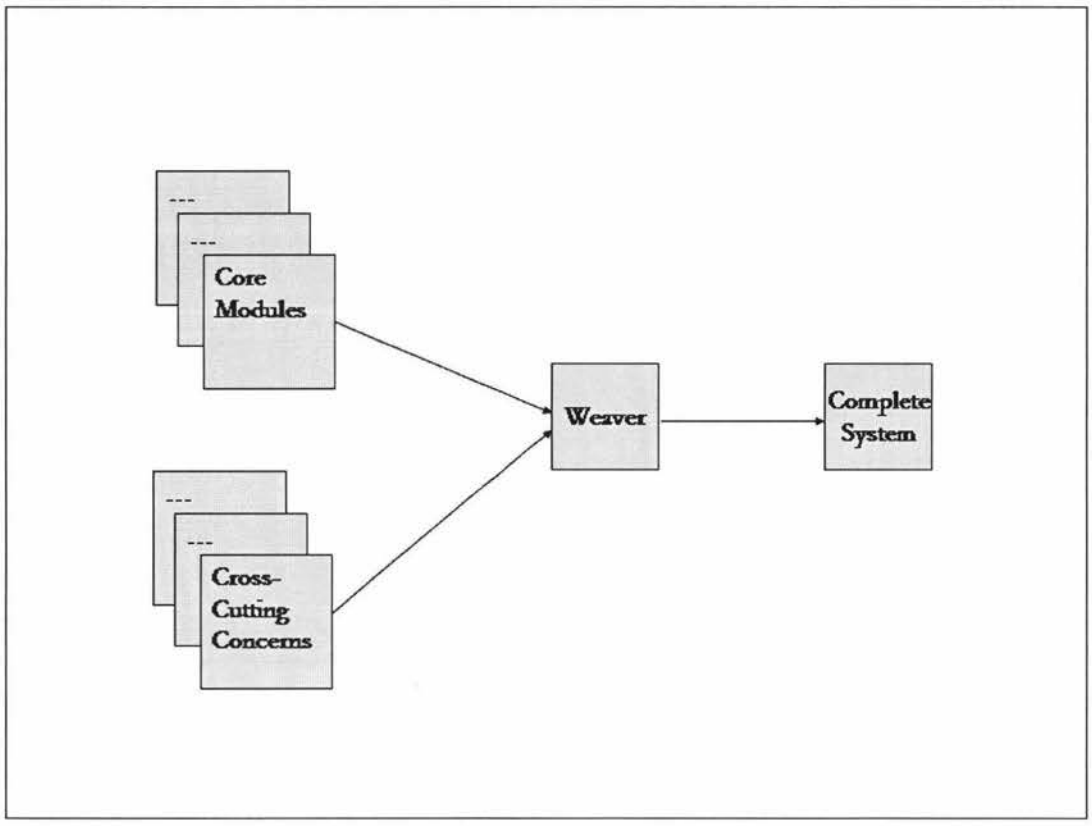

Figure 2.1: AOP System Diagram

are implemented using aspects. The core units now contain the fundamental logic of the system. The aspects contain both the logic to implement the crosscutting concerns and the rules of where to apply this logic. These units are passed to a weaver which takes the rules from the aspects and applies the crosscutting logic to the appropriate places in the core units. The output from the weaver is the fully functional system. Of important note is the resulting system functions are the same as a system that is coded using traditional approaches, but increased modularity has been achieved during the development process.

2.4 Summary

This chapter provides a brief introduction to AOP concepts and terminology for inexperienced readers. In the next chapter a criteria is proposed for choosing an AOP framework and an evaluation performed on several popular implementations.

CHAPTER 3

APPROACHES TO AOP

3.1 Introduction

There are many Aspect-Oriented programming languages emerging. Some of these implementations are well established with large user bases, while others are much smaller and specialised to a particular environment. AOP presents many challenges for adoption with many new concepts and techniques that must be mastered by developers. This is often equated to the challenges faced when programmers moved from Procedural Programming to Object-Oriented Programming. The problem is, developers often feel overwhelmed by the number of choices available and have little idea of the similarities and differences between the frameworks. Moreover, they do not know what criteria to judge frameworks on, and what features are strictly AO, and what have been added specifically for that framework. This situation is further escalated by the differences in terminology used by frameworks.

In this chapter a set of criteria to evaluate frameworks is established. This is used to evaluate four of the leading frameworks: AspectJ[1], AspectWerkz[2], JBoss AOP[3], and the Spring Framework[4]. In addition to this, the approach taken by the smaller Dynaop[5] framework is evaluated. The frameworks chosen are limited to those based around the Java programming language. This is because AOP is most mature in this area and because SolNet Solutions is a Java development house. There are frameworks available for languages such as PHP, Python, and C# (Wikipedia 2005*a*).

---

[1]http://www.aspectj.org

[2]http://aspectwerkz.codehaus.org

[3]http://www.jboss.com/products/aop

[4]http://www.springframework.org

[5]https://dynaop.dev.java.net

3.2 Establishing Criteria for Framework Evaluations

There are numerous criteria that frameworks may be evaluated against. In this section the key elements are identified which should be used when choosing a framework. This criteria is not only concerned with technical capabilities, but also those supplementary issues that are often more important to companies such as technical support and licensing arrangements. This list is very extensive and provides broad coverage that should satisfy most adopters when comparing the approaches. There are many other factors that are more general to AOP that are not considered. These items have been grouped into three general categories consisting of:

- Support Infrastructure
  - Vendor Backing - Who is driving the development of the framework? Are they reputable? This could be a vital factor in whether the framework is successful in the long term.
  - License - What requirements must be met to use this framework? Is the source code available? Can it be changed and distributed? What are the liability issues when distributing the framework to clients?
  - User Base - Provides an indication of the maturity of the framework. A framework with a larger user base is likely to identify bugs quicker.
  - Support - What support mediums are available? Is there paid commercial support available? How long does it take for bugs to be fixed?
  - Training Resources - How can developers quickly become familiar with the language? Are the resources high quality? Freely available?
  - Documentation - Is this kept up to date? Is it comprehensive?
  - Tool Support - What is available to support developers in deploying high quality aspect applications?
- Language Properties
  - Aspect Language - How are aspects written?
  - Composition Language - How is it specified where aspects should be applied?
  - Static Pointcut Checking - Are the pointcuts statically checked or are problems identified at runtime?

- Weave Times - What weave times are supported? Does the language support multiple weave times? Can aspects be deployed or undeployed at runtime? This could be important for flexibility and different uses of AOP.
- Standards Adherence - What AOP standards does this framework adhere to? Standards increase the ability to change frameworks and promote interoperability.
- Framework Integration - Can aspects developed for other frameworks be used?
- Join Point Model - How expressive is the framework? This influences the types of aspects that can be developed. A fine grained approach may be more complicated and produce aspects that are more tightly coupled to the core concern's implementation (fault tolerance when changes occur is decreased) than a coarse approach. However, it also increases flexibility and the range of aspects that can be developed.
- Types of Advice - Is there a single interceptor or advice for specific scenarios? Many people believe more specific advice types make development less prone to errors.
- Contextual Information - What information is available to advice?
- Intertype Declarations - Can the static structure of a class be changed? To what extent?
- Java Language Level Support - What versions are supported? In particular, is Java 5 supported? Java 5 introduces new possibilities for selecting join points based on annotations. These have the advantage over JavaDoc comments of being compiler checked. Annotations also increase the visibility of aspects.
- JVM Support - Does this approach have JVM weaving support?
- Advice Ordering - Can advice be ordered to ensure they are applied in the correct order for the application? This is important when multiple advice interact with the same join point.
- Aspect Lifecycle Models - When are aspects instantiated and how long do they exist?
- Pointcut Language - How are program elements identified? (e.g. Regular Expressions or Annotations)

- Adoption Issues
  - Ease of Adoption - Does this framework require many changes to existing tools and practices?
  - Environment Requirements - Does this framework require a particular application server or context to execute?
  - Build Overhead - Is the build and deployment time significantly affected?
  - Runtime Performance - How does this framework affect the performance of an application?
  - Debuggability - Are special tools required or are standard debuggers OK?
  - Testability - Are frameworks available that help verify program correctness?
  - Aspect Libraries - Are libraries available which support standard uses of AOP? Development time and risk can be greatly reduced by using well developed components.
  - Compatibility - Any compatibility issues? Does this framework maintain the SerialUID used by the Java serialisation process? This could be important when using EJBs or RMI.
- Other - Any other notable elements for the particular framework?

Note: This criteria contains points that are specific to Java based frameworks. There may be similar issues that could be considered when working with alternative languages.

3.3 Framework Evaluations

In this section the criteria discussed above are applied in the evaluation of five frameworks. The frameworks chosen are among the most well known and used frameworks for Java based AOP. The version of the framework evaluated is the most current at the time of evaluation. This has not been updated as new versions have been released and new features added. The frameworks and versions chosen are:

- AspectJ 1.5M1.
- AspectWerkz 2.0RC3.

- JBoss AOP 1.1.
- Spring Framework 1.2.
- Dynaop 1.0 beta.

Each point from the above criteria is presented as a separate section to allow comparison of the frameworks on each point. Finally, an approach will be recommended for use at SolNet Solutions and details of upcoming features in the chosen approach are discussed.

3.3.1 Vendor Backing

AspectJ was originally a Xerox-PARC project, but was later transferred to the Eclipse Foundation where it is backed by IBM. Many of the development staff contributing to AspectJ are employed by IBM. IBM has shown strong support for AspectJ and has used it in projects such as Web Sphere. For these reasons AspectJ is likely to be well supported in the foreseeable future.

BEA Systems has been the major backer of the AspectWerkz project. Similarly to AspectJ, key contributors to the framework have been employed by BEA. However, with the merger of AspectJ and AspectWerkz in early 2005 the AspectWerkz framework is no longer developed, although limited support is still available. BEA are now associated with the AspectJ 5 project.

JBoss AOP and Spring have no large vendor backing. Their AOP components are both elements of large development projects that have much community support. Both frameworks have developed commercial spin offs from the project offering enhanced support and training but these are not of the same scale as IBM and BEA.

Dynaop has no vendor support and is maintained by a small open source development team.

AspectJ 5 is obviously the framework that has the backing of key commercial entities and is likely to be the most stable in the future. However, Spring and JBoss AOP are both part of popular frameworks that are likely to enjoy long term success.

3.3.2 License

All the frameworks evaluated make it clear that using and distributing their frameworks is acceptable. However, if the framework is modified then the relevant license must be followed. AspectWerkz, JBoss AOP, and Dynaop all use the Lesser General Public License (LGPL), where as AspectJ uses the Common Public License and

Spring the Apache License. Some of these licenses require any changes be released to the community. Of most importance will be license requirements indemnifying developers of any liability when releasing the software with a commercial project. It is recommended that developers check their company's policies on the use of open source software and seek professional legal advice on the implications of the relevant license.

3.3.3 User Base

It is difficult to obtain an accurate measure of the user base of any of the frameworks. Download information is often not available and does not necessarily imply that the framework has been adopted. Furthermore, AOP is included in the Spring Framework but may not be a feature used. In this section the relative size of the user base is inferred from the traffic on the mailing lists and forums of the frameworks.

AspectJ experiences the highest traffic with approximately twenty five to thirty messages per week. This is similar to the level of AspectWerkz before the merger. Since this time AspectWerkz traffic has substantially declined. The AOP section of the Spring Framework has twenty to twenty five messages per week, and JBoss AOP ten messages per week. Dynaop has had no traffic in 2005.

It was expected that Spring would have lower traffic than JBoss AOP, however Spring has experienced much growth in recent times as developers look for alternative approaches to enterprise software development.

3.3.4 Support

AspectJ offers support through their user's mailing list. Questions are answered by members of the community as well as the key contributors to the development of AspectJ. This list is well frequented and questions are quickly answered. Issues with AspectJ are quickly identified and fixed. Alternatively, bug reports can be searched and new problems filed. Unfortunately, there is no commercial support option available. AspectWerkz operates in a similar fashion but its mailing list traffic has been significantly reduced since the AspectJ merger.

Both Spring and JBoss AOP offer forum, mailing list, and bug report options. However, they also offer paid commercial support. Spring does this through Interface21, a company formed by many of the experts behind Spring. JBoss offers extra support documentation and a priority help service.

Dynaop operates a mailing list, forum, and bug tracker. However, these all have very low traffic.

The commercial support options give JBoss and Spring the edge. However, AspectJ's support is satisfactory it just does not guarantee a response or give priority to any party.

3.3.5 Training Resources

Training resources are not only important for current staff but also enable recruitment of staff who are familiar with the technology. In this section books, training courses, web articles, and university courses are considered.

AspectJ has a huge volume of resources available in all the above areas. At least five books dedicated to AspectJ development have been published since 2003. These nicely complement the plethora of high quality online articles and examples available including publications by IBM. Training courses and workshops are available through several consulting groups including Aspect Mentor and New Aspects of Software. These involve intensive courses with substantial cost. Many university graduates are being introduced to AOP in postgraduate courses particularly using AspectJ.

While AspectWerkz is limited to online articles, many of these are excellent.

Several excellent books have been published on the Spring Framework. However, it should be noted that there is limited coverage of AOP in many of these books. However, use of Spring AOP is closely coupled to the Spring Framework so having a good book covering many topics would be essential. Several training courses are available, including an intensive four day course which covers Spring AOP as a core element. Furthermore, many freely available articles are available on the Internet.

One book has been released for JBoss that is known to cover AOP. It is the official guide to JBoss released by the JBoss Group. This is a very extensive publication that should be considered by anyone serious about JBoss AOP and the use of the JBoss Application Server. JBoss Group and several independent consulting groups offer courses on JBoss, with one particular course offering extensive AOP coverage as part of a five day course. Once again, extensive online articles are available.

Dynaop has very limited articles available on its use.

Of all these frameworks AspectJ has the advantage with extensive resources in all the areas examined. Of particular importance is its use in the teaching of AOP concepts by many of the universities offering AOP courses. Spring and JBoss AOP both offer plenty of resources for commercial users.

3.3.6 Documentation

The core documentation should be the first place that users check, but for this to occur it is essential that it is current and extensive in its coverage.

The AspectJ documentation is excellent. Its coverage includes AOP concepts, language features, and supporting tools. The material is extensive and includes plenty of examples. Furthermore, when errors are identified they are quickly fixed. Example projects are released which are also covered in the documentation. JavaDocs for AspectJ are available for users wanting more knowledge of the underlying AspectJ framework.

AspectWerkz is well documented with extensive coverage and examples. JavaDocs are provided. It is unlikely that this documentation will continue to be maintained as no there will be no further development.

The Spring Framework has extensive documentation. However, some information was missing such as AspectJ/Spring integration. There are lots of examples and coverage of most topics. Unfortunately, many advanced features are not documented and can only be found by examining the example projects and Spring test cases.

JBoss's freely available documentation is well maintained including a programmer's guide, JavaDocs, tutorials, wiki, tool support, and contributed aspects. The examples provided are good and sample projects are available.

The manual distributed with Dynaop contains many examples but lacks extensive coverage and is best described as minimal. Example projects are provided.

All the frameworks have adequate documentation, but AspectJ is the most extensive and well maintained. Dynaop is the only framework to lack the resources required for commercial development.

3.3.7 Tool Support

Tools which assist developers in producing high quality applications are necessary when working with aspect software. This is made necessary because of the invisible nature that AOP can bring to software development.

AspectJ has IDE support available for JDeveloper and Eclipse. The Eclipse plug-in is the best IDE plug-in available and offers extensive support for working with AspectJ. The JDeveloper plug-in is less mature and offers basic support. IDE support is also available for older versions of AspectJ with NetBeans and JBuilder, however these plug-ins are now obsolete. Build tool plug-ins are available for Maven and ANT to support the automated build process. The basic tools distributed with

AspectJ include a command line compiler and a visual aspect browser. Finally, a documentation tool is distributed which is similar to JavaDoc. Unfortunately, this tool is not available with the latest AspectJ versions.

An Eclipse plug-in is available for working with AspectWerkz. However, AspectWerkz uses plain Java so it, like the other frameworks evaluated, does not require as much IDE support as AspectJ. ANT and Maven build integration are also available.

Spring does not require any specialised tools other than letting the framework instantiate objects. However, it would still be useful to have some AOP support.

JBoss AOP has an excellent Eclipse plug-in available which offers support close to that of AspectJ. However, this plug-in is less mature and stable than the AspectJ plug-in. JBoss AOP distributes an ANT task to support building of software but does not include Maven support.

No tool support is available for dynaop.

Once again, AspectJ's maturity comes through with its extensive tool support.

3.3.8 Aspect Language

There are different approaches to writing aspects. AspectJ extends the Java language with a new compilation unit in an aj file called an aspect which is similar to a class. This approach clearly differentiates aspects from normal classes which are used in all the other approaches. The other approaches either implement an interface, use a standard method signature style, or use any Java method for advice within a class.

Once again it is felt that the AspectJ approach is cleaner. However, it is these language extensions which also make AspectJ more invasive when integrating it with tools and build processes. Furthermore, developers must learn new language elements and syntax.

3.3.9 Composition Language

The composition language used to apply aspects to a program can be specified in several ways. AspectJ uses new code elements called pointcuts. It is possible to configure the aspects used in a build with a list file. AspectWerkz allows configuration using JavaDoc annotations, Java 5 annotations, or XML. Configuration is done using an aop.xml file. This approach increases flexibility and allows developers to choose the most appropriate choice for the aspect being used (e.g. XML may be appropriate for tracing and annotations for transactions). Spring uses XML config-

uration. JBoss uses the same approach as AspectWerkz and Dynaop uses a Bean Script configuration file.

The AspectJ approach is simple and keeps aspects in code but this reduces the ability to externally configure aspects. XML configuration allows late binding of aspects just before an application is deployed. The approach of AspectWerkz and JBoss is preferred as it gives the most flexibility depending on the type of aspect being deployed.

3.3.10 Static Pointcut Checking

The static checking of pointcuts ensures that problems are detected at compile time rather than as runtime errors. Only AspectJ is statically checked. However, not all pointcuts can be statically determined and delaying the determination of join points can offer advantages.

3.3.11 Weave Times

The weaving of an aspect application can occur at different times depending on the framework. This can affect the performance of the application and the flexibility in the deployment of aspects.

AspectJ supports compile, post-compile, and load-time weaving. These options depend on the aspects being used. For example, an exception handling aspect may be an integral part of the application and not having it available at compile time can result in compilation errors. On the other hand, a tracing aspect may be best deployed at load time depending on the application configuration required.

AspectWerkz uses normal Java compilation combined with a post-compile build step called offline weaving. Alternatively, aspects can be woven at load or run time. AspectWerkz allows hot deployment and undeployment of aspects allowing configuration changes on running systems.

The Spring Framework uses runtime weaving using dynamic proxies. This also supports runtime configuration changes.

JBoss AOP offers the same options as AspectWerkz and Dynaop supports runtime weaving.

The ability to hot deploy aspects and use normal compilation steps gives the other frameworks an advantage over AspectJ. However, this flexibility does not come without a price as is shown by the runtime performance of these frameworks.

3.3.12 Standards Adherence

AOP standards increase the ability for interoperability of aspects between frameworks. The only current standard is the AOP Alliance. This has only been implemented partially by both Dynaop and Spring. None of the other frameworks implement this standard.

3.3.13 Framework Integration

The ability to make use of aspects developed for one framework with another is important when developers may not be certain which framework they will eventually adopt or a client may have specific requirements that force use of a framework that the aspects were not originally developed for.

AspectWerkz offers an Extensible Aspect Container which can run aspects from AspectJ, Spring, Dynaop, JAC, and any AOP Alliance aspects. This is the most extensive support for alternative aspect use available from any of the frameworks. However, there is no known support for AspectWerkz aspects with other frameworks.

Adrian Colyer from the AspectJ team has shown the use of AOP Alliance aspects with AspectJ using a special adapter aspect to manage the framework differences. It is thought that aspects developed by some of the other frameworks could also be handled in a similar fashion. AspectJ has opened up many of its APIs to encourage other frameworks to increase their support.

The Spring Framework can use AOP Alliance aspects and has focused on increasing integration with AspectJ. In particular it is possible to configure AspectJ aspects using Spring dependency injection. Adrian Colyer has joined the Spring team to increase AspectJ support including use of AspectJ aspects without the AspectJ compiler.

3.3.14 Join Point Model

The join points that can be advised by a framework influence the types of aspects that can be developed.

AspectJ, JBoss AOP, and AspectWerkz all offer fine grained approaches including method, constructor, and field accesses. They also offer dynamic pointcuts that depend on the control flow of the application although this is more limited in JBoss AOP. Spring and Dynaop both offer more limited join point access which consists only of method invocations. However, this is the most commonly used join point.

Of the frameworks AspectJ and JBoss AOP offer the most extensive join point

access.

3.3.15 Types of Advice

The most general form of advice is the around advice which allows behaviour to be added before, after, or instead of a join point. This is offered by all the frameworks often in the form of an interceptor. However, many experts argue that having more specific advice types reduces the risk of errors being introduced. AspectJ, Spring, and AspectWerkz therefore both offer before, and after advice types in addition to around advice.

3.3.16 Contextual Information

All the frameworks offer similar access to contextual information at a join point including arguments, the executing object, and the join point object (e.g. a Method object). The method with which the information is accessed is the major difference. AspectJ introduces a new keyword 'thisJoinPoint' which is similar to 'this'. 'thisJoinPoint' can be directly accessed to retrieve the contextual information. The other frameworks take an alternative approach where some sort of object is passed as a parameter to the advice method which contains the contextual information. Either of these approaches is equally acceptable and simple to use.

3.3.17 Intertype Declarations

All the frameworks offer the ability to change the structure of a class to a similar degree. However, where as AspectJ allows direct addition of methods, fields, interfaces, or parent classes to another class, the other approaches all use mixins. A mixin involves writing another class containing the items that should be introduced to a class and adding the mixin class to the class being modified. The AspectJ approach is recommended as simpler and more natural.

3.3.18 Java Language Level Support

With the recent addition of Java 5 it is interesting to know what support is offered for different language levels. Depending on the language features desired certain Java versions must be used. For example, Java 5 annotations can only be used with Java 5, but JBoss AOP and AspectWerkz both offer alternatives which can be used with previous Java versions using JavaDoc comments. AspectJ, AspectWerkz,

Spring, and JBoss AOP all support Java 1.3-1.5. Dynaop is only known to support Java 1.4, although 1.5 is most likely supported.

All the frameworks support similar Java versions, however it is yet to be seen what support will be provided for working with Java 5 language elements such as generics in pointcut expressions.

3.3.19 JVM Support

None of the frameworks evaluated offer any specific Java Virtual Machine (JVM) support for weaving. However, an experimental version of the BEA JRockit JVM has been developed with an aspect weaving API. A modified version of AspectJ 5 was developed using this API for weaving instead of byte code manipulation. This highlighted some problems that were faced in adding JVM support as well as the problems that could be solved. This will be an exciting feature to watch for in the future, but to be successful it must go through the Java standard's process.

3.3.20 Advice Ordering

It is important when multiple pieces of advice must be applied to a join point that they are executed in the correct order if they have any dependencies. All the frameworks have some rules which decide how advice should be applied. In AspectWerkz, Spring, JBoss AOP, and Dynaop this is determined by the order aspects are specified in the relevant configuration files. However, AspectJ requires that explicit precedence rules be specified if aspect ordering is required.

3.3.21 Aspect Lifecycle Models

The lifecycle of an aspect determines when it is created and when it is destroyed. Some aspects are shared by all objects where as other aspects are specifically created for a single object.

All the frameworks support the basic Singleton lifecycle where a single aspect is shared by all objects in the system. However, AspectJ can also create an aspect instance for each target object, each type of object an aspect advises, and for each control flow. AspectWerkz supports per type and per instance. Spring supports per class and per instance. JBoss AOP supports per type, per instance, and per join point. Finally, Dynaop supports per proxy.

AspectWerkz, AspectJ, and JBoss AOP offer the most powerful Aspect lifecycle models.

3.3.22 Pointcut Language

There are many ways to determine the join points that should be selected. This could be regular expressions, wild cards, XPath[6], Java 5 Annotations, JavaDoc Annotations, or objects. JBoss AOP, AspectJ, and AspectWerkz all use wild card type patterns. These type patterns can select annotations which allow aspects to easily pick the correct places to apply behaviour rather than the weaker join point signature methods traditionally used. However, having to annotate methods can also be time consuming and difficult to maintain if large numbers of join points are required. Spring and Dynaop both support the use of regular expressions which have similar properties to the use of wild cards. Spring does not make join points a language feature so it is possible to write custom join point classes.

The languages to specify join points are one of the major weaknesses of AOP as they often rely on naming conventions. Although annotations do help, they can require extensive addition of annotations resulting in minimal improvement from making method calls from the relevant join points. This is one area where AOP could benefit from a new approach, although it is not clear how this could be achieved.

3.3.23 Ease of Adoption

All the approaches are relatively easy to adopt. However, AspectJ is the most invasive as it requires new tool support such as IDE plug-ins and changes to the build process. However, this can be offset by the quality of the tools and documentation available to support it. On the other hand, the other approaches are generally less invasive and can operate with minimal impact. However, having quality tools and documentation available increases the ability to easily adopt an approach. JBoss and Spring both perform relatively well with documentation. Spring is more invasive as it requires the Spring Framework be used to manage advised objects. This approach is only recommended if the Spring Framework is also adopted. Dynaop is considered difficult to adopt due to poor documentation and lack of tool support and little in the way of support infrastructure. Its future is also uncertain with little development apparent which appears to plague smaller frameworks after their initial releases. AspectWerkz is simple to adopt and offers some tool support but it does not have a long term future with the merger with AspectJ.

---

[6]http://www.w3.org/TR/xpath

3.3.24 Environment Requirements

Some AOP environments are general purpose and can be applied with any application, where as others are integrated into some larger framework with which they must be applied. Spring and Dynaop are the prime candidates as objects that are advised must be managed by the frameworks. JBoss AOP has strong ties to the JBoss Application Server but can be run independently of it. All the other frameworks are free of environment concerns.

3.3.25 Build Overhead

The weaving process can add overhead to a build and deploy process through the addition of extra steps. Of all the approaches the only one to affect the build process is AspectJ. The AspectJ compiler is usually used instead of the javac compiler. Unfortunately, this compiler often requires complete rebuilds whenever pointcuts are updated which can slow the compilation process. The addition of an incremental compiler has helped to reduce the builds required. The other approaches use normal Java compilation followed by some post-compilation step such as an aspect compiler, load time weaver, or run time weaver. This allows aspects to be less intrusive on the build process but it can affect the load time and/or runtime performance or the application.

3.3.26 Runtime Performance

Many commercial applications have performance critical elements so it is vital that the use of AOP does not significantly affect the performance of the application when compared with the hand coded version. The results from the AspectWerkz benchmarks are used to compare the relative performance of the applications (Vasseur 2004). It should be noted that the overhead each framework produces depends on the type of advice being executed. In this comparison the simple before advice is used as it is representative of the relative performance over many of the advice types.

AspectJ and AspectWerkz display equal performance with a 15ns overhead. This is far better than the other frameworks which all apply proxy approaches which are well known to be slower. JBoss AOP comes in at 145ns, followed by Spring with 275ns, and Dynaop at 320ns. It should be noted that all these values are very small overheads; however there is a large relative performance differential between the approaches. When other advice types are considered AspectJ is likely to be faster than AspectWerkz by a small margin.

3.3.27 Debuggability

Debugging aspect software can be expected to pose new problems due to the weaving process. Fortunately, it has been found that most frameworks support normal debugging and with certain practices have minimal impact.

AspectJ is compatible with any JSR-45 compatible debugger which supports classes with multiple source files. This is certainly the case with the AspectJ plug-in for Eclipse and most of the latest IDE versions should support this JSR. The other frameworks all support normal Java debugging. However, it can be necessary to set debug points in the aspects otherwise stepping may unexpectedly pass over them.

3.3.28 Testability

Testing of aspect software is discussed more extensively in Chapter 6. However, in this section the properties of aspect languages which influence their ability to be tested using current testing techniques is considered.

AspectJ is difficult to test since new compilation units are produced by the weaving process which are difficult to unit test since the weaving makes them strongly dependent on the context to which they are woven. This is due to aspects lacking an independent identity. However, a new tool called aUnit is under development which aims to making testing of aspects as simple as current JUnit testing by hiding the framework details. Some traditional testing can be performed but this does not bring the benefits of separation of concerns to testing.

The other approaches all use normal Java classes. This allows unit testing to a certain extent. Unfortunately, the need for contextual objects which are difficult to create outside the frameworks can produce a hindrance (e.g. objects containing contextual information). This results in similar problems to AspectJ since the framework is needed to perform weaving into an application before testing can be easily performed.

Testing is one of the most difficult and critical areas that needs to be addressed with aspect software.

3.3.29 Aspect Libraries

Standard aspect libraries are useful to reduce development time and risk in a similar fashion to the libraries distributed with Java.

Both Spring and JBoss AOP both distribute aspect libraries. The Spring aspects

are not directly used but provide declarative services such as object pooling and transaction management. On the other hand, JBoss provides many aspects which can be configured for use in a particular environment. This is the most extensive aspect library available and is well documented.

3.3.30 Compatibility

It is important that AOP is compatible with other technologies being used in developing an application. In this section the issue of Java serialisation is identified as one possible compatibility issue.

The built in Java serialisation mechanism depends on a field attached to classes called the SerialUID. The serialisation process is used for Java Remote Method Invocation (RMI) and EJB passivation and activation. This makes it important that this field is maintained. However, since aspects can alter the structure of a class it is possible that there could be problems with this.

It is known that AspectJ can change the SerialUID of classes in the generated class files. This is very important and developers working with AspectJ should consider the impact of this. It is hoped that future AspectJ versions will avoid this problem. It is thought that the problems are more likely to influence long term persistency than short term persistency of objects. This requires further investigation as it could be critical to many uses of aspects.

Spring and Dynaop fully support serialisation of Java objects so must maintain this field correctly. It is not known if AspectWerkz and JBoss AOP maintain this field. It is likely that JBoss AOP does and AspectWerkz may not but this would require further investigation.

3.3.31 Other

This section briefly mentions some notable features of the various frameworks that may be important but do not fit into the above areas.

AspectJ and JBoss AOP both support the declaration of custom compiler warnings and errors. This allows aspects to be written which ensure that certain coding practices are followed.

AspectJ supports a process called exception softening which allows compiler warnings for checked exceptions to be suppressed when its known that an aspect will handle an exception. It also supports privileged aspects which allow the private members of classes to be advised.

AspectWerkz has been merged with AspectJ which means many features from AspectWerkz will be included in future AspectJ versions. Furthermore, AspectWerkz is unlikely to have any further development.

Appendix A shows an example of how aspects are written using each of the frameworks discussed.

3.4 Language Choice - SolNet Solutions

Choosing a language to use at SolNet Solutions is not an easy task as the different frameworks have many advantages and disadvantages. Furthermore, an increased understanding of SolNet's future direction and their architecture is required. One possibility is SolNet could adopt the Spring Framework which would make Spring AOP a likely choice. However, with the current environment it is felt that AspectJ offers the most comprehensive solution. This is driven by the tools to support developers, documentation, training resources, and language features such as ability to advise non-public members. These are required for some of the uses of AOP in later chapters. It is interesting to note that many of the reasons for adopting this language came down to infrastructural issues rather than functional requirements since many of the frameworks offer similar functionality.

Several issues that should be considered further by SolNet when using AspectJ are testing and serialisation of objects. In particular AspectJ may cause issues with the use of EJB in development projects. It is noted that no issues were discovered when working with EJBs in Chapter 9 but this does require urgent review.

It is also noted that AspectJ has many enhancements being developed for release in AspectJ 5 as a result of Java 5 being released and the merger with AspectWerkz. The features include selection of join points based on new Java features such as annotations and generics, enhanced load time weaving, aspect libraries, and the ability to write plain Java aspects using annotations. In particular, the last point gives increased flexibility to AspectJ and should reduce the need for new tools required for adoption.

3.5 Alternatives to AOP

There are various approaches that could be considered competitors to AOP. In this section four alterative approaches are discussed and their ability to be used instead of AOP is assessed.

3.5.1 EJB 3.0

JSR 220 is currently working on the next version of the EJB component model, more commonly known as EJB 3[7]. EJB 3 promises a simplified programming model and a move towards use of POJOs. To achieve this, JSR 175 annotations are used extensively for specifying container behaviour, service injection, object/relational mapping, and can replace XML deployment descriptors. Furthermore, interceptors can be used to intercept calls to business methods or lifecycle call back events in session and message driven beans. Any number of interceptor classes can be defined for a bean and their order specified. The interceptors are stateless, but state can be carried across multiple invocations using a context object.

This approach is a step towards the model achieved with AOP. However, the capability of aspects far exceeds the limited capabilities of interceptors in EJB 3. Furthermore, this approach is only available to session and message driven beans further reducing its capability. The model achieved is similar to that available in Spring. This approach will satisfy the need for many users but is not as flexible, portable, or reusable as a more general AOP approach.

3.5.2 Servlet Filters

Servlet Filters[8] provide a simple AOP like technology similar to that achieved with EJB 3. These filters allow transparent injection of services and pre-processing of servlet requests. However, this is very limited and only applies to web requests lacking the power and flexibility required for many users who deal with business objects.

3.5.3 Composition Filters and Hyperslices

Composition Filters and Hyperslices/Multidimensional Separation of Concerns (MDSOC) both promote alternative modularisation technologies to achieve separation of concerns. However, both these approaches have been static since 2001 and are not recommended.

---

[7] http://jcp.org/en/jsr/detail?id=220

[8] http://java.sun.com/products/servlet/Filters.html

3.5.4 Choosing an Approach

AOP may not be the correct approach in all circumstances. It seems to have won the battle of the alternative modularisation technologies with Composition Filters and Hyperslices not going any further. However, techniques such as EJB 3 and Servlet Filters both offer solutions in a limited context which may be useful. Using the correct tool for particular jobs is of importance. For this reason, it is recommended that servlet filters be used for many pre-processing tasks associated with web requests rather than using AOP interception. In similar fashion if only the limited features of EJB 3 interception are required then it makes sense to use this approach rather than introducing an AOP framework. However, it should be carefully considered whether AOP's flexibility is required when making these decisions.

3.6 Summary

In this chapter a criteria has been proposed for the evaluation of aspect frameworks. This considers three general categories covering support infrastructure, language features, and adoption issues. Each of these categories are subdivided into key points.

The criteria proposed are used in the evaluation of five frameworks and the AspectJ framework is proposed for use at SolNet Solutions. This framework choice is used for our work with SolNet projects and is used throughout this thesis. Upcoming features of AspectJ are also discussed. These are largely driven by the addition of Java 5 and the merger with AspectWerkz. This merger brings many of AspectWerkz advantages (and those of other frameworks) to AspectJ, making it by far the most flexible framework with a suburb support infrastructure in place which was not as robust with AspectWerkz.

Two areas that have been identified as requiring further work are the serialisation of Java objects, in particular with AspectJ, and the testing of aspect frameworks.

Finally, alternative technologies which offer some AOP like approaches are discussed and recommendations are made as to when these can offer better solutions than aspects.

CHAPTER 4
TOOL SUPPORT

4.1 Introduction

Aspects introduce a new dimension to software development which brings many benefits. However, with this added dimension there are new areas where developers need tool support. In this chapter the tools available for building, developing, testing, quality assurance, debugging, and visual design of aspect software are analysed. Recommendations of suitable tools are made and areas where new tools are required are identified.

4.2 Build Tools

Build tools such as Make[1], ANT[2], and Maven[3] are used by developers to automate the large number of steps that a typical project must go through when it is built and deployed. Some common steps are removing files from previous builds, copying libraries to the appropriate places, compiling code, running tests, and producing a deployable file. To fit into the build processes of many organisations, the ability to use aspects with these tools is essential. In this section the integration of two popular build tools used in Java development, ANT and Maven, are analysed with AspectJ, AspectWerkz, Spring, and JBoss AOP.

4.2.1 ANT Integration

ANT tasks are included with AspectJ releases, ensuring they contain the latest compiler features. An incremental compiler is available to allow developers to avoid complete rebuilds. Integration has been improved in recent times with the addition

[1]http://www.gnu.org/software/make

[2]http://ant.apache.org

[3]http://maven.apache.org

of an adapter task which replaces normal javac compilation with AspectJ compilation. The tasks available are fully featured, highly mature, well documented, and user friendly. Furthermore, ANT is used by SolNet to manage the build process making this support necessary for future aspect integration.

AspectWerkz can be used with ANT to perform offline weaving (i.e. post-compile weaving step). This can be achieved using the normal javac task to compile all the classes and then either a standard ANT task to execute the AspectWerkz weaver or the special task distributed with AspectWerkz. It is preferred to use the AspectWerkz task since it has all the necessary features to execute the weaver appropriately and is simpler to use. Furthermore, it is well documented.

JBoss distribute an ANT task for post compilation of aspects. Similarly to AspectWerkz, javac is used to compile the classes and then the JBoss task is executed to perform weaving. This task is well documented and easy to use.

No special support is required for working with Spring AOP. Normal compilation is performed using javac. This is due to Spring managing the creation and the weaving of objects using proxies at runtime.

It is clear from the discussion above that ANT provides excellent support for working with any of the four major frameworks. This fits well with SolNet's and many other companies use of ANT as their primary build tool.

4.2.2 Maven Integration

An increasingly popular alternative to ANT is the Maven build tool because of its improved support for project management. However, Maven has not been in use as long as ANT so therefore is not as mature and well documented. Maven is likely in the long term to supersede ANT as the de facto standard Java build tool. It has been investigated by SolNet as a replacement for ANT, but this is not likely to happen in the short term.

Maven provides support for AspectJ compilation. However, the documentation available is minimal making it more difficult to work with than the ANT tasks. Furthermore, since the tasks are maintained independently of the AspectJ project it is anticipated that new features will not be available immediately.

Support for AspectWerkz is available in Maven 1.0 but was dropped in version 2.0 because AspectWerkz was merged with AspectJ. The support in version 1.0 was extensive but poorly documented.

No specific Maven support is available for either JBoss AOP or the Spring Framework.

Additional work is required to improve the documentation of Maven's AOP support to better enable assessment of its AOP build capabilities. It is expected that frameworks may release their own Maven plug-ins or work more closely with those groups providing them as Maven's popularity increases. It should also be noted that it is possible to execute ANT tasks from Maven using the antrun plug-in. This provides developers with the opportunity to use Maven when tasks are only available for ANT. However, this solution retains a dependency on the use of ANT so it would be desirable to have pure Maven tasks developed.

4.3 Integrated Development Environments

Integrated Development Environments (IDEs) provide a range of development tools in one localised environment such as code editors, build tools, and debuggers. IDEs are one of the most important development support tools due to the invisible nature and development style changes aspects bring. This can be effectively bridged with appropriate tools to make aspects less transparent, particularly by showing where crosscutting occurs.

This section examines the integration of the leading AOP implementations with the commonly used IDE's Eclipse, IntelliJ IDEA, JDeveloper, JBuilder, and NetBeans. Of particular interest is the support the IDE's provide for working with aspects including help in writing pointcuts, checking where advice is applied, debugging, and code completion.

4.3.1 Eclipse

Eclipse appears to be the de facto standard IDE for working with Aspect-Oriented software. It provides some support for all the major frameworks through plug-ins. This is a result of open source collaboration to produce the necessary support instead of waiting for vendors to add support to their products. This is in contrast to most commercial products which are waiting to see how AOP progresses before adding tool support.

AspectJ

The AspectJ Development Tools Project (AJDT) is an Eclipse Foundation project which adds tool support to Eclipse for working with AspectJ software. This plug-in is well developed, mature, well documented, and stable. It continues to evolve with regular feature enhancements, bug fixes, and support for the latest AspectJ features.

In fact, the AJDT releases generally follow the AspectJ releases by only a day or two due to the project's collaborative nature. Some key features are:

- Syntax highlighting of aspects.
- AspectJ project creation wizard.
- New aspect wizard.
- Generation of ajdoc (Only with AspectJ 1.0) for documentation of aspects similar to JavaDoc.
- Full AspectJ compiler settings.
- Add or remove aspects from builds.
- Outline view shows advice types and pointcuts.
- Crosscutting markers and navigation - Allows viewing of where advice is being applied and to navigate between advice and advised members.
- Aspect Visualiser - High level graphical view of aspect crosscutting.
- Run configuration for aspect programs including addition of required libraries to the classpath.
- Debugging support ensures its just like debugging a regular Java application.

A screenshot of the AJDT plug-in support is shown in Figure 4.1. This shows the cross references view, gutter markers on methods and advice, and advised members in the context menu.

There are some currently known bugs that will be resolved in future releases of AJDT (e.g. non-functioning code completion and aspect refactoring behaviour is missing). Furthermore, when working on large projects it is necessary to disable the automatic builds to avoid lockups each time an action is performed.

This is an excellent tool that provides leading edge support for AspectJ development. This is the recommended tool for any AspectJ development project.

JBoss AOP

The JBossIDE brings JBoss AOP support to the Eclipse platform. However, it lacks the maturity and stability of AJDT. It offers numerous features including support for the writing of pointcuts which is missing in AJDT. The features include:

Figure 4.1: AJDT Screenshot

- Project wizard to create AOP projects. This automatically adds the jboss-aop.xml file and runtime libraries.
- No markers in the aspects, but classes that are advised have gutter markers. These can be navigated and viewed by using the quick fix feature (Ctrl + 1). This is a hidden feature that could be improved with increased visibility.
- An Aspect Manager provides a graphical view of the aspects, bindings, pointcuts, and interceptors in the jboss-aop.xml file.
- The Advised Member's view shows the aspects advising a class and its members.
- A run configuration automatically adds the configuration files and libraries required for JBoss AOP.
- Wizards for creating bindings, writing pointcuts, and adding advice or interceptors to a method.

A screenshot of the JBossIDE support is shown in Figure 4.2. This shows the aspect manager, advised members view, popup quick fix advice view, and gutter markers on advised members.

Overall this provides excellent support but has some hidden features and can be unstable. In particular it was found that restarts were required to register some changes to the XML file or other updates made. Furthermore, white space in the workspace path resulted in errors with unrelated messages. This is a good tool that will continue to develop and provide support for JBoss AOP development

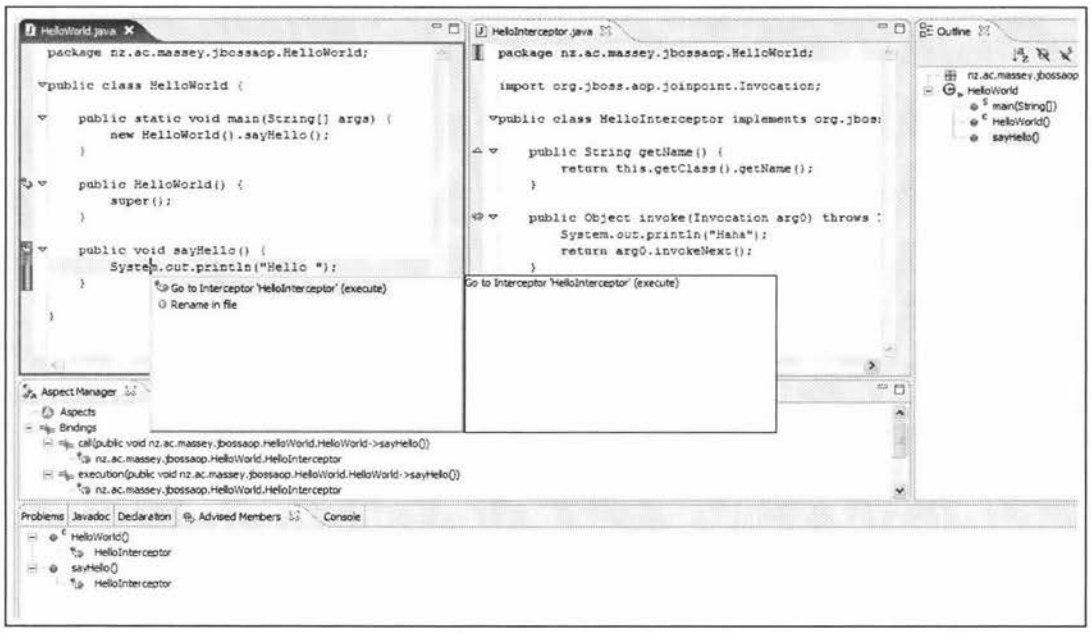

Figure 4.2: JBossIDE Screenshot

Spring Framework

A general Spring Framework plug-in exists which provides support for configuration of Spring Beans. However, no AOP support exists.

AspectWerkz

AspectWerkz can be used either with or without a plug-in. Without a plug-in, ANT is used to automate the build process. However, features such as crosscutting identification are missing. Debugging could be performed using the regular Java debugger provided that breakpoints are set in the aspect classes. If breakpoints are not set in the aspect classes then they cannot always be correctly 'stepped into' from the advised code.

It is preferable to use AspectWerkz with the plug-in despite it only being a beta release. However, with the AspectJ merger this is unlikely to be further developed. Nevertheless, this plug-in provides support for viewing and navigating crosscutting relationships and configuring weaving. This is only a rudimentary plug-in but it provides some of the most important support for developers.

4.3.2 NetBeans and JBuilder

An AspectJ plug-in was developed for NetBeans and JBuilder as an open source project. Unfortunately, this has become inactive as developer's interests have moved to other projects. This plug-in does not function with the latest NetBeans/JBuilder

and AspectJ versions. It provided support such as viewing and navigating cross-cutting relationships and setting compiler options. This is a rudimentary plug-in which needs a vendor sponsored replacement to bring AOP back to the NetBeans and JBuilder development environments.

It is possible to configure NetBeans without a plug-in to use AspectWerkz, but this provides no specialist AOP support. This is likely to be possible with most IDEs and aspect frameworks, but more development support is needed for developers to effectively work with aspects.

4.3.3 IntelliJ IDEA

Support for AspectJ was built into IDEA but later disabled. It is possible to re-enable this support but it did not function correctly for our tests. Forums postings indicate many users have experienced similar problems. Contact with the IntelliJ help desk indicates that support may be improved in upcoming releases, but this was yet to be confirmed.

4.3.4 JDeveloper

An open source plug-in is being developed for Oracle JDeveloper which intends to bring the same support as AJDT does for Eclipse. However, the releases for this are well behind the current AspectJ versions and the support provided is inferior to AJDT. It is a good rudimentary plug-in for viewing and navigating aspect crosscutting relationships. Furthermore, compiler options and configuration are available. It is hoped this plug-in will continue to develop and not meet the fate of others such as the NetBeans plug-in. With aspects increasingly popular it is vital that IDE support grows out of Eclipse and into other IDEs.

4.4 Testing

Support for unit testing of aspects depends on the framework being tested. AspectJ poses the most problems due to its extensions to Java to facilitate aspects. Unfortunately, testing theory for aspect software is still evolving so production ready tools are yet to be developed. Some tools have been developed to complement research in this area. However, none of these tools were available for download and evaluation. These testing techniques and tools are discussed further in Chapter 6.

Fortunately there is a tool being developed to allow testing of aspects created

using AspectJ called aUnit[4]. However, since this tool is currently a 0.1 release it lacks the maturity necessary to be used in most environments. It was found to only support a subset of the advice types available in AspectJ meaning it could not be used without modification for testing of SolNet projects. Moreover, attempts to modify the tool to support some of the missing features were unsuccessful.

Overall, this tool is not currently recommended for use until a newer version is released. This is scheduled for early 2006 when an extensive update is expected to bring much needed functionality and reliability. This tool shows considerable promise, but the release was too immature for an effective evaluation.

4.5 Debuggers

Debugger support can be complicated by the addition of aspects to a system. Once the code has been woven the classes being debugged no longer match back to corresponding source code of that particular class. This can make debugging more difficult and cause unexpected results. Debugging can often work with some pure Java frameworks when breakpoints are set in the aspect classes themselves. Trying to 'step into' an aspect that does not have a breakpoint set generally fails. For this reason it is necessary to have some debugger support to ensure a smooth debugging process.

The only tool to offer extensive debugger support is AspectJ with the Eclipse AJDT plug-in. This supports debugging through aspect code and hides the underlying AspectJ framework. This support makes the debugging of AspectJ programs akin to debugging normal Java. This support is made possible by JSR 45 for debugging which allows a class to have multiple source files. This should be supported in most of the latest debuggers and IDEs.

Unfortunately, no other specialist debugger support has been encountered for other languages. The other languages are not as significantly affected as AspectJ, but nevertheless do require some extra support. This is an area where further work is required. Moreover, it would be interesting to evaluate the AJDT debugger when remotely debugging EJB's in a container.

---

[4]http://www.aunit.org

4.6 Documentation

Java has an effective documentation tool in the form of JavaDoc. It would be ideal if a similar tool could be used to document Aspect-Oriented software.

It has been found that JavaDoc could continue to be used for AOP languages that use normal Java classes such as Spring, JBoss AOP, and AspectWerkz. However, it should be noted that the documentation appears like a class rather than as an aspect. Therefore, it is considered necessary to provide a new tool or possibly a doclet that could account for aspects. Furthermore, it may be desirable to allow aspects crosscutting nature to be navigated as hyperlinks in a similar way to class references in JavaDocs.

AspectJ introduced a tool called ajdoc in version 1.0. This tool provides similar documentation for aspects as that produced by JavaDoc. Unfortunately, this tool has not been updated for the latest versions of AspectJ. It appears that this is still on the agenda for future development; it has fallen in priority as AspectJ 5 was being developed.

Overall, there are possibilities for some documentation of aspects, but this area needs significant development.

4.7 Code Metrics

Code metric tools are important to allow evaluation of software in a quantitative fashion. Although many metrics exist for OO software that are still applicable to Aspect-Oriented software, there is also a need for metrics to measure the new types of coupling that result from using aspects. Several metrics have been proposed for Aspect-Oriented software and will be discussed further in Section 7.4.

At this stage the only tool available is an open source project being developed as part of an MSc project titled the AOP Metrics Suite[5]. It includes measures that have been altered to incorporate aspects from the OO metrics suite proposed by Chidamber & Kemerer (1994), as well as package dependency metrics such as those used in JDepend[6]. The documentation for metrics used in this suite is adequate, but some of the metrics are yet to be implemented. Furthermore, the tool is only executable as an ANT task. This is suitable for SolNet's purposes, but further support for standalone execution are needed to allow easy use. The tool supports output of the results to both XML and Excel allowing easy analysis.

[5]http://aopmetrics.tigris.org

[6]http://www.clarkware.com/software/JDepend.html

The tool is still very immature (draft release) and would not execute on any of the projects tried due to AspectJ or ANT version incompatibility that could not be resolved. It is difficult to assess the correctness of the metrics the tool produces, but the source code is available if users wish to verify it. The tool lacks support for any complexity measures which are used extensively in the assessment of SolNet projects in Chapter 9.

This is obviously an area where tool support will evolve as metrics become available and more stable.

4.8 Visual Design

There is currently limited tool support available for visual design of Aspect-Oriented systems. There have been systems developed for research projects but none of these were available. It is not surprising that support is limited when it is considered that a standard development notation has yet to be agreed upon. Because of this, most books published using AO notations have relied on general drawing tools such as Microsoft Visio. This is fine for drawing the diagrams, but lacks the ability to use automated design checks, reverse engineering, code generation, and round tripping tools that are bundled with CASE tools. However, it should be remembered that most CASE tools should contain support for specifying a UML profile and may also include support for custom scripts that can perform tasks such as code generation. If a company decides to settle on a standard approach that can make use of these facilities then their existing tools will still be supported.

Rational Rose[7] provides the necessary support to define a UML profile for using aspects. Moreover, it provides rose script which can be used to automate some code generation from design elements. This approach is shown by Zakaria, Hosny & Zeid (2002) with AspectJ as a target language.

Rhapsody[8] provides similar support to Rational Rose and has been used by Elrad, Aldawud & Bader (2005) to generate code from aspect UML models including state charts.

Finally, Enterprise Architect[9] provides similar support to Rational Rose and Rhapsody. Furthermore, it is possible to define a profile and export it as UML so it can be easily distributed to a team using aspects. Enterprise Architect has adopted

[7]http://www-306.ibm.com/software/rational/

[8]http://www.ilogix.com/sublevel.aspx?id=53

[9]http://www.sparxsystems.com.au/products/ea.html

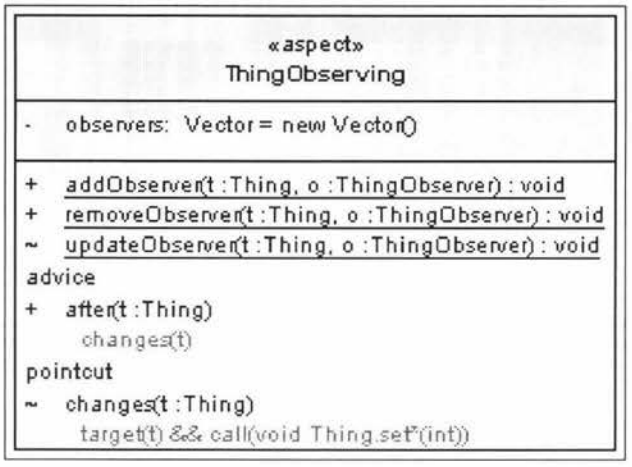

Figure 4.3: UML Notation for Enterprise Architect

a notation for AspectJ (Figure 4.3) allowing some support out of the box including reverse engineering. However, the notation used is not considered an ideal solution, but it is acceptable and when used consistently by a team will be constructive. This is the tool used by SolNet Solutions, making this support attractive.

4.9 Summary

This section has examined the tools available to support software engineers in the development of Aspect-Oriented software. It has been found that tool support is still developing in many areas. However, the areas that are better established in relation to the implementation of the software are reasonably well developed such as build tools and IDEs. It is vital that more IDE support is developed and that build tools such as Maven increase their support. Of most importance is the development of tools for design and testing of systems. Metrics tools are important and developing but are not as vital as these two other tools. Furthermore, debuggers and documentation tools also require some extensions to be more useful.

Overall, the basic tools exist to support AOP at a commercial level, but additional work is necessary to increase the breadth and quality of the tools.

In the next chapter the design of aspect systems is examined.

CHAPTER 5

ASPECT-ORIENTED DESIGN

5.1 Introduction

So far AOP has been discussed as an implementation technology. However, to integrate AOP into a software development methodology its impact on other areas such as requirements gathering, analysis, design, and testing must be considered. This holistic approach is referred to as Aspect-Oriented Software Development (AOSD) and allows the separation of crosscutting concerns to be realised throughout the development process (Jacobson & Ng 2004, Araujo, Baniassad, Clements, Moreira, Rashid & Tekinerdogan 2005).

In this chapter the focus is on the design stages of the process, in particular how aspects can be visually modelled using UML. There have been many proposals in this area (Han, Kniesel & Cremers 2005, Cottenier, Berg & Elrad 2005, Pawlak & Younessi 2004, Clement, Harley, Colyer & Webster 2004, Katara & Mikkonen 2002, Zakaria et al. 2002, Basch & Sanchez 2003, Cole, Piveta & Sampaio 2004, von Flach Chavez, Garcia, Kulesza, Anna & Lucena 2005, Kande, Kienzle & Strhmeier 2002, Stein, Hanenberg & Unland 2002*a*, Stein, Hanenberg & Unland 2002*c*, Stein, Hanenberg & Unland 2002*b*, Stein, Hanenberg & Unland 2003, Clemente, Hernandez, Herrero, Murillo & Sanchez 2005). A selection of the best approaches are considered in this chapter (Suzuki & Yamamoto 1999, Jacobson 2003, Jacobson & Ng 2004, Clarke & Baniassad 2005, Clarke & Walker 2005, Baniassad 2003, Baniassad & Clarke 2004, Clarke & Walker 2002, Aldawud, Elrad & Bader 2003, Aldawud, Elrad & Bader 2001, Elrad et al. 2005, Rausch, Rumpe & Cornel Klein 2003, Asteasuain, Contreras, Estvez & Fillottrani 2004). Recommendations are made on using one of these approaches within SolNet Solutions.

The notion of Aspect-Oriented design patterns and idioms is also considered. Design patterns and idioms are beginning to emerge as aspect best practices are formed. Both design patterns and Aspect-Oriented design are still immature elements of the Aspect-Oriented development lifecycle.

5.2 Aspect-Oriented Design Approaches

Without an aspect design approach programmers are given an OO specification and design which they implement. However, since the design does not allow specification of aspects they must follow a translation process and redesign to use aspects. In doing so, the implementation does not match the design and there could be new errors introduced through poorly informed design changes. In this section several design approaches are presented which tackle this problem by allowing specification of aspects in the design ensuring the programmer can implement directly from specification to code elements. Moreover, this is an important area since many professionals consider the system design to be the most significant element contributing to the success of a system (Clarke & Baniassad 2005).

5.2.1 Use Case Approach

Ivar Jacobson has suggested that use cases are a natural approach for taking aspect software from the initial requirements gathering through to testing (Jacobson 2003, Jacobson & Ng 2004). Jacobson is the inventor of the use case and one of the fathers of UML and the Rational Unified Process (Jacobson 2005). Use cases are one of the most commonly used methods to capture user requirements and are used to drive software development (referred to as Use Case Driven Development). Use cases are separate when specified, but when use cases are realised they get tangled as use cases become intermixed across implementation classes. This is mainly because there was no implementation technology available that could keep use cases separate at the implementation phase, so it made sense to tangle the use cases at the realisation stage to ensure that the design could be easily implemented. Moreover, this meant that UML was not built with explicit support for keeping use cases separate during this phase. For example, the use case extend relationship has no mapping for use case realisation.

Jacobson proposes that use case slices are added to UML to allow support for the use case extension mechanism. This new feature would allow use cases to be kept separate right through to implementation, testing, and maintenance.

A use case slice collates parts of classes and operations that are specific to a particular use case and groups them in a single model. Figure 5.1[1] shows an example of a use case slice. Notice that each use case is represented by a use case slice which contains a few elements from the various implementation classes. The composition

[1]Diagram reprinted from Jacobson & Ng (2004)

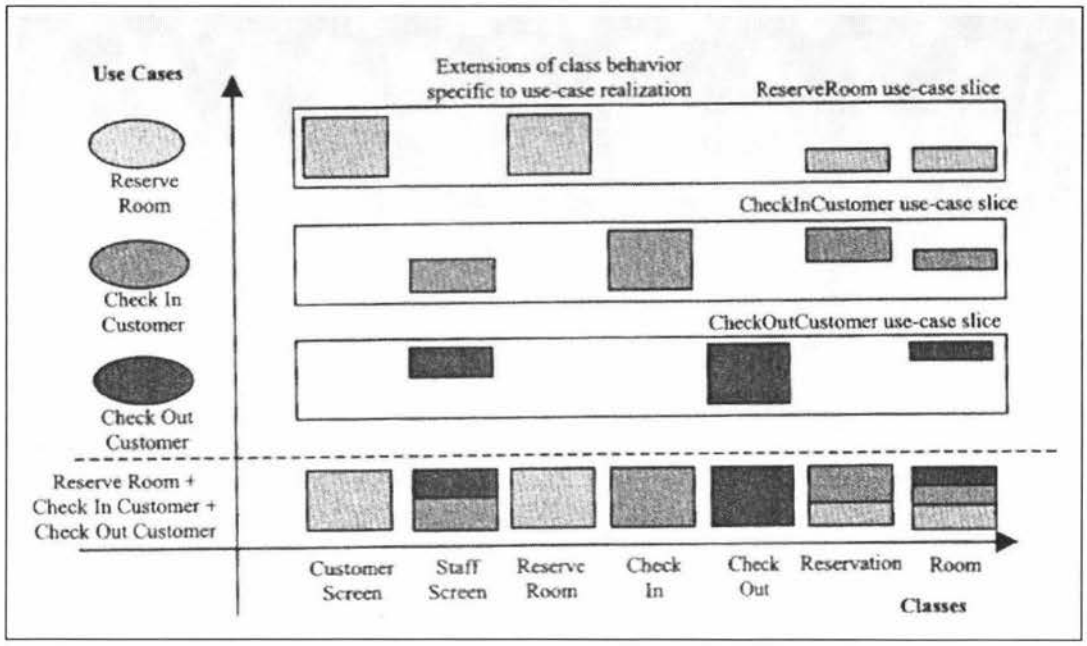

Figure 5.1: Use Case Slice

of the use case can then be performed by the particular implementation technology (i.e. it is not dependent on a particular AOP approach). The superposition of all the use case slices is the entire design model.

There are two types of use cases which benefit in different ways from this approach. The first is peer use cases which have no relationship between each other but their realisations are tangled. Peer use cases benefit since attributes and methods from the realisation classes are identified that are specific to each particular use case, allowing them to be maintained separately. The portions of each class contained in a use case slice are called a class extension and represent those parts of the class required to implement the use case. The merging of all class extensions using intertype declarations produces the complete system.

The UML extend relationship allows an extension use case to add behaviour to another use case at an extension point. However, this cannot be implemented using OO so have only been used as named places in a control flow where another use case flow should be added. AOP techniques such as pointcuts and join points combined with advice allow these to finally be realised to their potential in the implementation phase.

Finally, Jacobson proposes a use case module which contains all artifacts specific to a use case over the entire lifecycle. This allows use case modules to be developed separately and concurrently (with some coordination work) and gives one place to look for information on any use case and its path through the lifecycle (traceability from requirements to implementation). Furthermore, the implementation of use

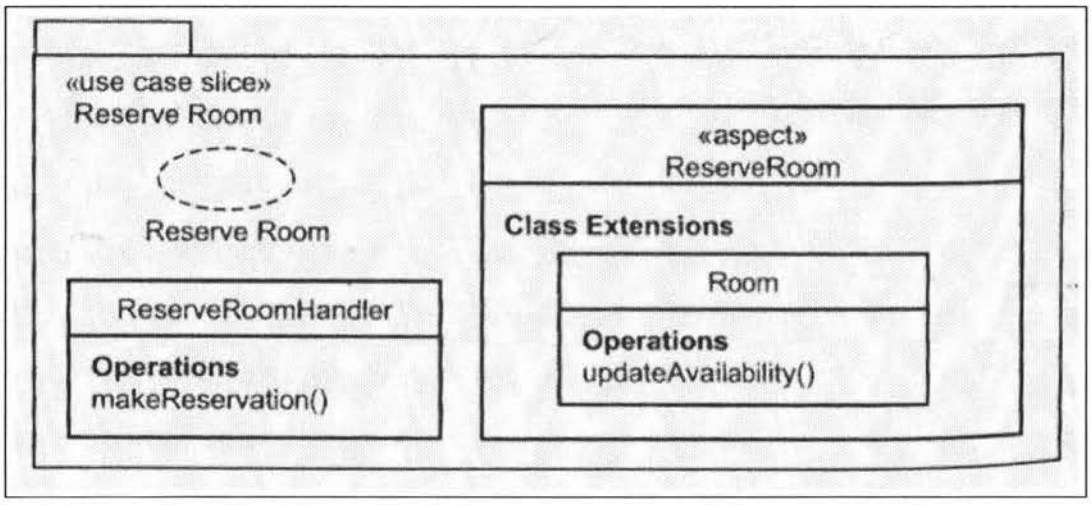

Figure 5.2: Use Case Aspect Representation

cases across iterations can be prioritised to allow the most important functionality to be designed and implemented first.

However, this approach does require some changes to UML to support new artifacts such as aspect and use case module. Jacobson has produced a comprehensive book (Jacobson & Ng 2004) which guides developers through this approach and details the required elements and how they can be mapped to AspectJ (they can be mapped to other languages too). Furthermore, who better than one of the fathers of UML to drive a change to UML to incorporate features needed to support aspects? Figure 5.2[2] shows an example of an aspect in Jacobson's notation containing a single class extension and a pointcut.

The use case approach has the major advantage of being widely used making many developers and designers familiar with its use. It therefore makes sense to build on this approach rather than replace it. However, we should be sure that a replacement would not produce a more appropriate approach for a new technology. Furthermore, this approach seems to assume that aspects will be incorporated well beyond infrastructural concerns. Instead aspects are used to produce malleable software that can be composed using aspect technology. For many this is beyond what they intend to use aspects for, particularly in the early stages of adoption. It is not clear how little of this approach could be applied and still be effective for adopters wishing to start at a higher level. If it is not suitable for these less invasive uses of AOP, then it makes it more likely that adopters will use a different approach and never realise the potential this approach brings to the building of very versatile and easily changeable software.

[2]Diagram reprinted from Jacobson & Ng (2004)

5.2.2 Theme/UML

A completely new approach to software development with aspects is Theme/UML proposed by Siobhan Clarke (Clarke & Baniassad 2005, Clarke & Walker 2005, Baniassad 2003, Baniassad & Clarke 2004, Clarke & Walker 2002). The Theme/UML approach is used to identify and model aspects from a set of requirements using a symmetrical approach. A symmetrical approach modularises both the core and aspects which represent some piece of separate functionality which when combined form the functionality of the whole system. In contrast an asymmetric approach considers aspects as being separate from the core program. Aspects are treated like events which are triggered when appropriate events are dispatched from the core program. This works well when aspects are executed at many places in the system (such as infrastructure aspects).

The most central concept is the theme, which is an encapsulation of a concern. This makes a theme more general than an aspect, since themes represent some piece of functionality or aspect from the system. The two key components to the theme approach are Theme/Doc and Theme/UML.

Theme/Doc is a set of heuristics which help in the analysis of software requirements to identify themes and determine whether they should be modelled as an aspect. This is referred to as theme and aspect identification. A Theme/Doc tool provides graphical views of the relationships between requirements and themes.

Theme/UML provides a means to write themes as UML. Each theme has a separate design model which allows it to be independently designed regardless of whether it crosscuts or overlaps another theme. All the classes and methods that are pertinent to a concern are designed within the theme. Most of the UML used is standard's compliant. However, some new elements have been introduced for modelling the parameterisation of the behaviour that is triggered by a base theme. A composition relationship is introduced to identify the parts of a theme that are related and that should be composed. Figure 5.3[3] shows a crosscutting theme (aspect) using the Theme/UML notation.

The Theme approach can be used with different lifecycle models such as waterfall and iterative. Using it with an agile approach is not as easy, but with correct selection of diagrams and appropriate updating of diagrams it can be applied effectively.

This approach gives excellent traceability between requirements and implementation, making the mapping from design to code easy for developers. This avoids the

[3]Diagram reprinted from Clarke & Baniassad (2005)

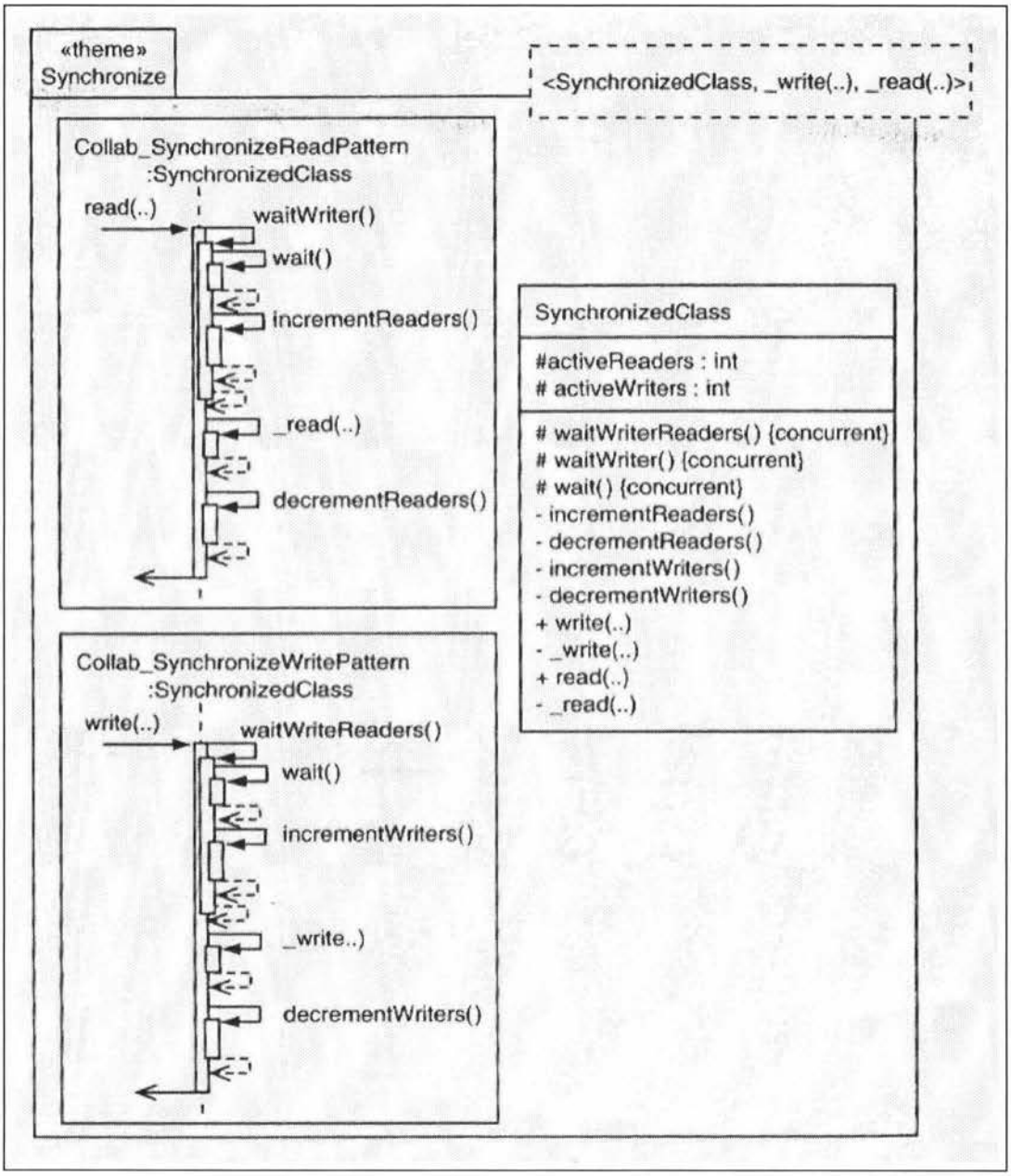

Figure 5.3: Theme/UML Crosscutting Theme

problems of ad hoc redesigns to incorporate aspects by developers and gives more opportunity to identify domain specific aspects, rather than the smaller number of general purpose aspects that are typically applied using ad hoc approaches. Clarke & Walker (2005) show how to map from themes to Hyper/J and AspectJ showing the language independence of the approach. Furthermore, they show abstract aspects in AspectJ allow the creation of highly reusable aspects from reusable themes by separating the crosscutting specification (advice) from the composition specification (pointcuts) using abstract pointcuts. However, dynamic pointcuts such as cflow in AspectJ are missing from Theme.

Overall, this approach should be easily applied by UML designers, but it requires more changes in the process than the use case driven approach. In particular a new means of specifying system functionality/requirements with Theme/Doc instead of the de facto standard use case would discourage many potential adopters. Moreover, there is a need for further tool support to allow automatic code generation and design round tripping.

5.2.3 General UML Extension

Suzuki & Yamamoto (1999) propose an extension to the UML meta-model to incorporate aspects. Additionally, they propose an XML aspect description language to allow CASE tools to share aspect model information.

Aspects and woven classes are added to the UML meta-model and the existing realize relationship is reused for modelling aspect/class relationships. The $\langle\langle aspect \rangle\rangle$ stereotype is used on a class box to represent an aspect, with attributes representing weave definitions (pointcuts) and operations weave declarations (advice). A $\langle\langle weave \rangle\rangle$ stereotype is added to operations which model advice. A model of the woven structure of the application can be developed. Classes in this model that have been crosscut by an aspect have the $\langle\langle wovenclass \rangle\rangle$ stereotype.

Asteasuain et al. (2004) identifies some key points from this approach including:

- It hinders separate development since aspects have explicit references to objects.
- Learnability and user friendliness is reduced since addition of new elements requires considerable attention to behaviour.
- There are no clear composition rules which reduces understandability.
- Rich and expressive models can be built resulting in a high degree of reusability of aspect designs.

- Good traceability.

It is felt this approach has the advantage of being compatible with existing CASE tools and requires only a minimal level of effort to be introduced. In particular it can be helpful when aspects are being used for infrastructural purposes. However, it is also felt that as aspects become more prevalent a technique which has more support for separate development will be more suitable.

5.2.4 Model-Based Approach

Many applications follow a model-based development approach where models are produced at different levels of abstraction and mappings for transformations between the different levels are defined. Rausch et al. (2003) proposes a model-based approach which provides a merger of the implementation of a requirements model with a predefined aspect implementation. This is a reusable framework. Currently many of the infrastructural use cases for using AOP are associated with the desire to connect an application to a framework in a transparent manner. This approach helps with the design of this by making aspects visible in the model allowing reasoning before implementation.

New stereotypes $\langle\langle callJoinPoint \rangle\rangle$, $\langle\langle aspect \rangle\rangle$, and $\langle\langle advice \rangle\rangle$ are introduced along with $\langle\langle introduction \rangle\rangle$ on classes for modelling intertype declarations. Both structural and behavioural diagrams are used. The connections between the framework and the application are made using some constraint language such as OCL and the $\langle\langle aspectBindings \rangle\rangle$ stereotype.

Rausch et al. (2003) has left the development of a full UML profile for this approach as future work. This approach lacks tool support and the use of OCL makes it complex for specifying constraints. However, this approach could be useful when aspects are being used for infrastructural purposes.

5.2.5 UML Structural and Behavioural Diagrams

An approach which makes use of standard UML extension mechanisms for the specification of aspects is proposed using both structural and behavioural diagrams (Aldawud et al. 2003, Aldawud et al. 2001, Elrad et al. 2005). They produce a full UML profile which tailors UML to the AOP domain using stereotypes, tagged values, and constraints. The aim is to produce reusable design components and to automate code generation and round tripping to keep design and implementation consistent.

Aspects are modelled using the $\langle\langle aspect \rangle\rangle$ stereotype to ensure reusability. Synchronous aspects control the behaviour of another class, where as asynchronous

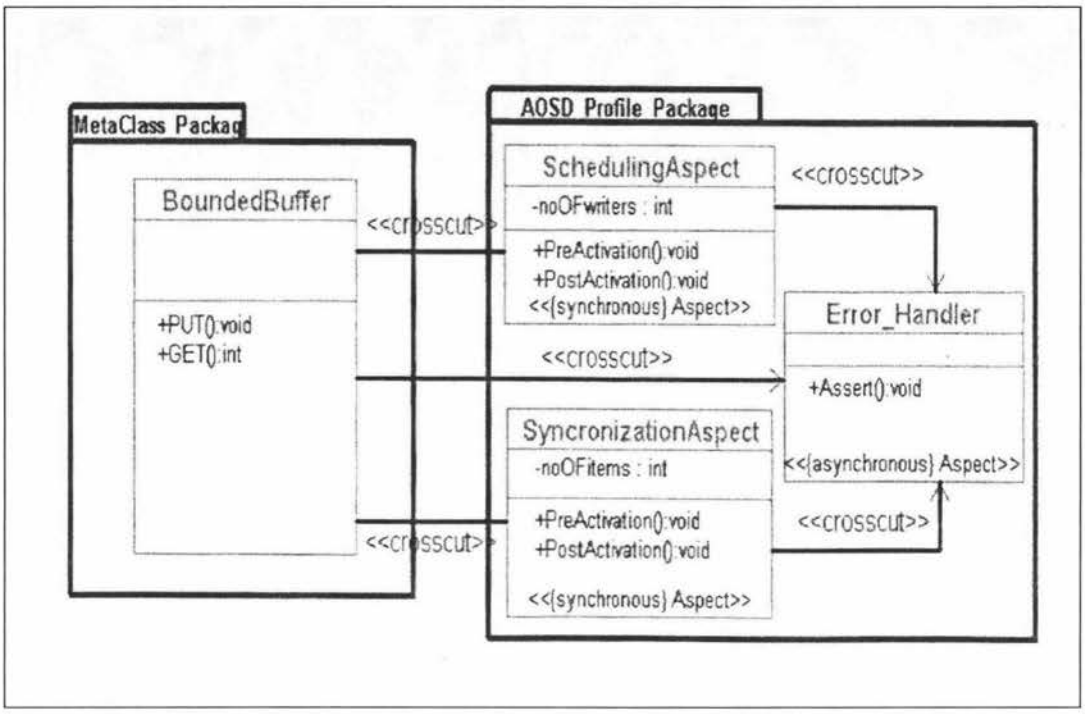

Figure 5.4: UML Class Diagram

aspects have no impact on a base class. This is controlled using a tagged value. A $\langle\langle crosscut \rangle\rangle$ stereotype is used to represent relationships between classes and pre/post activation operation stereotypes are used for advice. In Figure 5.4[4] a basic scenario is presented showing the representation of aspects, pointcuts, advice, and a basic relationship with a class.

So far this approach is similar to that shown in Section 5.2.3, however this approach goes further by making extensive use of state charts for behavioural modelling. State charts are designed for use in modelling intra-object behaviour. However, this approach shows that some of the advanced mechanisms available can be used to give a full behavioural specification for crosscutting behaviour. AOSD is supported by the extracting of the hardwiring of transition conditions making designs reusable. Furthermore, the approach is semantic preserving between design and implementation. Rose script has been used to generate AspectJ skeletons from the specification.

The event notification mechanism and use of concurrent states allows implicit weaving to be performed using broadcasting techniques. This results in loose coupling and extensibility. Furthermore, since concerns are modelled separately, requirements can be effortlessly traced to design elements and implementation. Round tripping is also performed to allow the model to be kept consistent with the code.

[4]Diagram reprinted from Elrad et al. (2005)

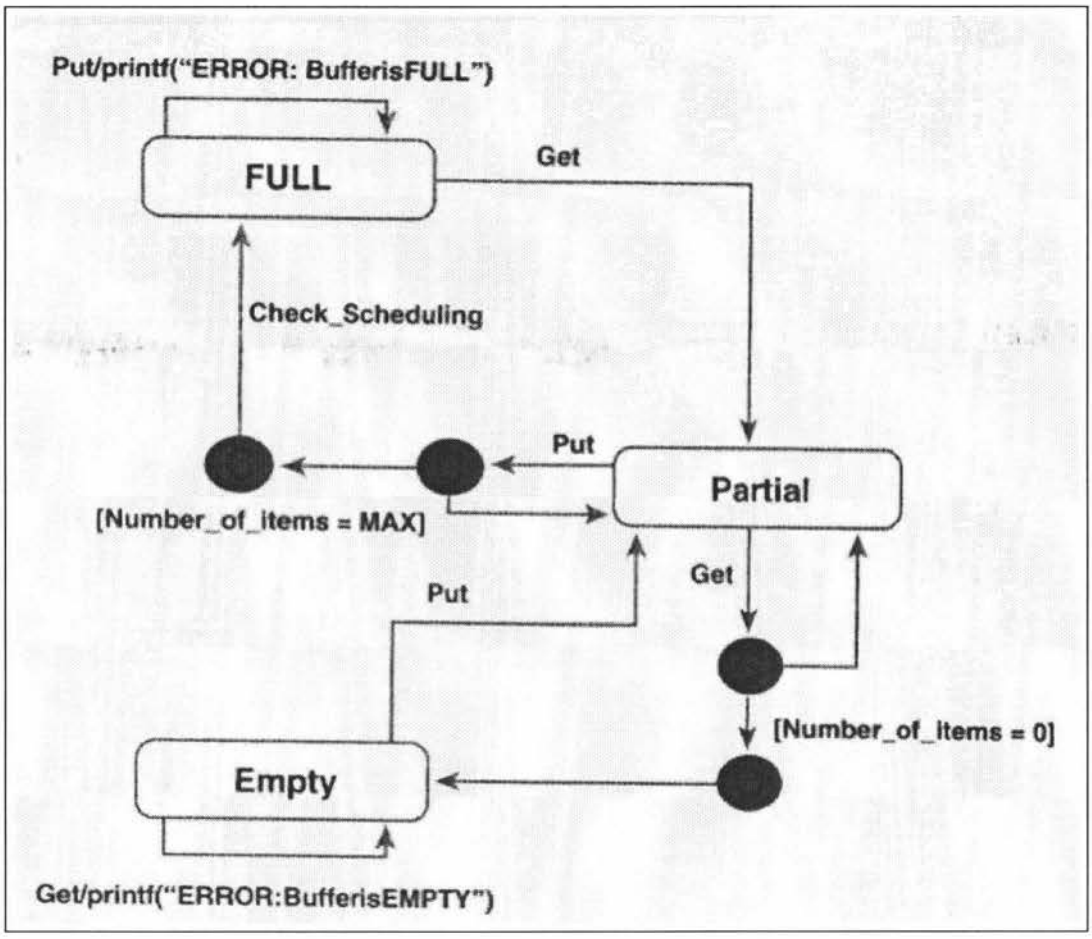

Figure 5.5: State Diagram - Tangled Model

Figures 5.5[5] and 5.6[6] show the use of state charts with a tangled model and with separation of concerns using this approach respectively.

We believe this is the most desirable of the simpler, less invasive approaches. It has many points similar with Suzuki's model, but with the addition of state charts to allow extensive behavioural modelling. This allows for fuller specifications when needed. Moreover, the ability to use standard CASE tools and code generation facilities make it ideal for early adopters.

5.3 Fitting with SolNet Solutions

SolNet Solutions uses a use case driven development approach. This involves full specification of use cases which are then realised as class diagrams. Depending on the criticality and risk associated with the component and the client's requirements there can be further state and behavioural modelling. Some projects take an agile

[5]Diagram reprinted from Elrad et al. (2005)

[6]Diagram reprinted from Elrad et al. (2005)

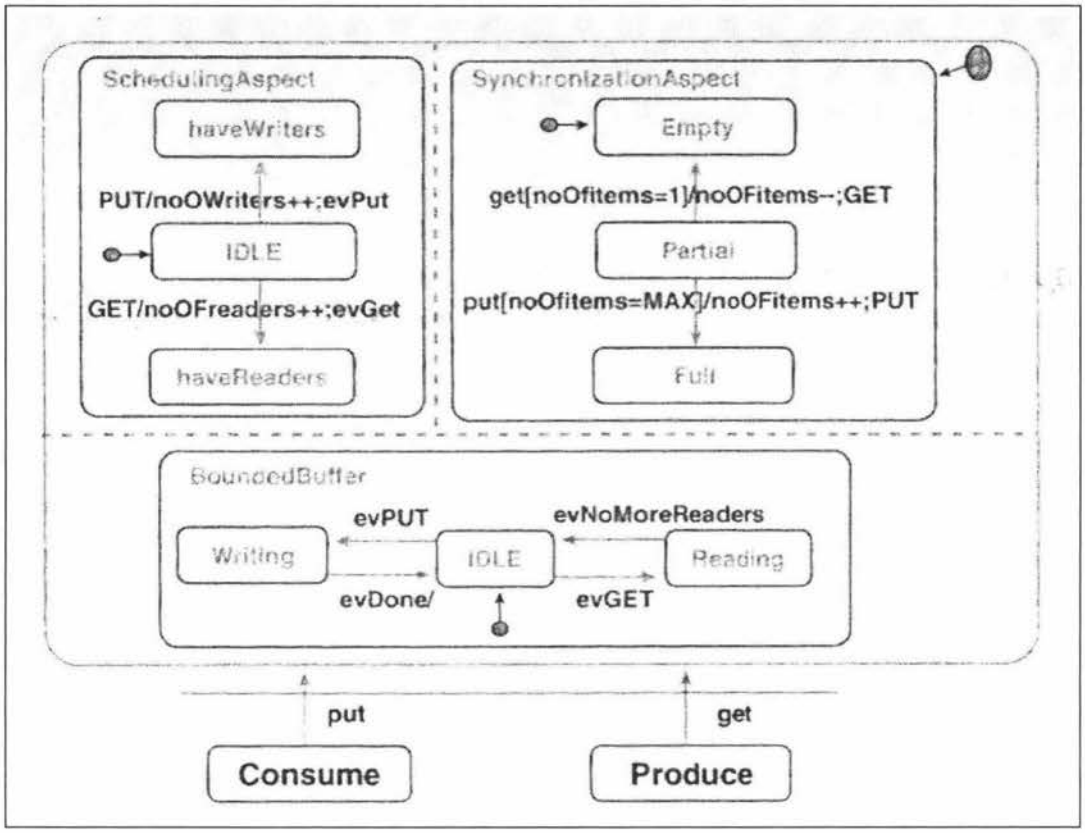

Figure 5.6: State Diagram - Separate Concerns

approach where development can be virtually performed from the use case specification with minimal modelling.

It would seem likely that the most advantageous approach for SolNet to use would be Jacobson's use case techniques. However, it should also be considered that aspects will most likely only be used for infrastructural purposes in the near future making this approach too demanding. Furthermore, it requires changes to tools to support the new UML features proposed. The Theme approach also suffers from similar problems, but is further complicated by the change to Theme/Doc instead of use cases. In many ways these two approaches share many similarities in the types of elements they capture and artifacts produced. If aspects are to be used more extensively both of these approaches are well developed and have useful book resources to guide their usage. We consider them to be the two ultimate solutions, but feel the technology is in advance of what most companies will be willing to adopt at this early stage. Moreover, tool support is yet to be developed for these approaches and there is unlikely to be a formal UML Profile or change to the specification adopted by OMG in the short term. This is essential for a UML notation to be adopted.

Alternatively, either of Suzuki's or Elrad's approaches to the use of UML extensions could be readily applied since they provide the ability to give some representa-

tion of aspects in the design, but do not require the same level of specification and intrusiveness as found in the use case and Theme approaches. It is likely that many early users would be satisfied with modelling aspects on this level. If aspect use becomes more extensive then the design may have to start using something which allows greater separation of concerns and traceability such as Theme or use cases. Elrad's approach is preferred since it offers more resources such as a book chapter and several papers, as well as behavioural specification with state charts. Moreover, it has been successfully used in the generation of code and round tripping of the design.

We do not view the model-based approach as a likely solution due to its lack of extensibility to other areas outside of pure framework aspect specification.

At this stage it seems likely that more techniques will continue to be developed and developers will need to choose and consistently apply a technique that suits their requirements. It is a big improvement to have some modelling of aspects at the design stage, and like the introduction of aspects performing this as a staged approach may give further time for staff development and familiarity with the technology to increase. Ultimately, we hope to see AOP used to its full potential using the more sophisticated techniques of Jacobson or Clarke. These techniques show great promise in producing more flexible and extensible software than has been possible in an OO world. However, in the meantime Elrad's approach is recommended as a short to mid-term solution for SolNet Solutions in specifying aspects.

5.4 Aspect-Oriented Design Patterns and Idioms

With new design methodologies comes the need for design best practices. Design patterns can be used to capture best practices for Aspect-Oriented software in a similar manner to their use in OO. Two classes of design patterns are presented in this section. Firstly, the refactoring of existing design patterns using aspects to make them less invasive and more reusable is discussed. Then a new class of patterns that have been proposed for writing aspect software is presented. Finally, idioms for AspectJ development are considered. An idiom is more language specific and has a smaller scope than a design pattern.

5.4.1 Refactoring OO patterns using Aspects

Design patterns present a solution to a recurring problem in some context. The work of Gamma, Helm, Johnson & Vlissides (1994) in producing the Gang of Four

(GOF) pattern catalogue is regarded as the key movement towards developers (deliberately) using patterns in software. However, many OO patterns exhibit crosscutting behaviour such as affecting multiple classes and being difficult and invasive to reuse (Lesiecki 2005*a*). Furthermore, it can be difficult for developers to identify the use of patterns in a system making tools for design recovery necessary (Dietrich & Elgar 2005). Hannemann & Kiczales (2002) present the original work in refactoring of GOF patterns with AspectJ. In comparing the Java and AspectJ solutions of the twenty three GOF patterns they found they could remove code level dependencies from participants in seventeen patterns. Twelve patterns were refactored to the point where they were reusable and could be included in a library. Most patterns were improved either by reducing the number of participants or having different participants. In some cases the code was simply moved from the participant to the aspect. This work is used as the basis for work by Miles (2004) to show how AspectJ can improve the Creational Patterns Singleton, Prototype, Abstract Factory, Factory Method, and Builder. Miles finds that pattern mechanics are modularised and less intrusive on business logic making the code cleaner and easier to understand. Moreover, the freedom to use inheritance relationships in Java is improved since there is less need to inherit from abstract base classes (classes can only inherit from one class in Java).

A simplified example of the Singleton pattern implemented using aspects shows how a pattern can be refactored. The first step is to define a tag[7] interface Singleton. An aspect can be used to capture all calls to constructors of any class implementing the Singleton interface and either return the existing object or a new object if this is the first constructor call. This is highlighted in Listing 5.1 which uses a Hashtable to store Singleton objects and an around advice to return the correct object instead of calling the constructor. The Singleton interface is defined as part of the same aspect for modularity. Finally, an aspect is used to make classes that need to be Singletons implement the Singleton interface as in Listing 5.2. This is now a pattern that can be easily reused by implementing an aspect to make the necessary classes implement Singleton. Furthermore, it is possible to use a class as a Singleton in one project and not in another project simply by including or excluding it from the aspect. This allows more reuse prospects for business classes which is also noted by Lesiecki when refactoring the Decorator, Observer, and Adapter patterns (Lesiecki 2005*a*, Lesiecki 2005*b*). Developers can simply identify use of patterns and more easily maintain classes without the aid of design recovery tools reducing the risk of

[7] A tag interface doesn't contain any members - it is used to signal a property of its implementers

Listing 5.1: Singleton Aspect

```
// Library aspect implementing the Singleton Pattern
public aspect SingletonAspect {

    // Interface for classes that are Singletons
    public interface Singleton{}

    // Store the single instance for each class
    // that is a Singleton
    private Hashtable singletons = new Hashtable();

    // Pick out constructor calls to classes
    // implementing Singleton
    pointcut singles() : call(Singleton+.new(..));

    // Intercept calls to Singleton class constructors
    // and return the single instance of the class
    Object around(Object obj) : singles() && this(obj) {
        Class clazz = obj.getClass();
        if(singletons.get(clazz) == null) {
            singletons.put(clazz,proceed(obj));
        }
        return singletons.get(clazz);
    }
}
```

Listing 5.2: Make class implement Singleton

```
// Application specific aspect to make any classes that must
// be Singletons implement the Singleton interface
public aspect MySingletonAspect {
    declare parents: MyClass implements
       SingletonAspect.Singleton;
}
```

damaging software flexibility through ill informed changes.

5.4.2 Aspect-Oriented Patterns

AOP opens the opportunity for a new class of patterns associated with the use of aspect software. Laddad (2003) has proposed several new patterns for aspect software. There are few patterns yet to emerge, but as aspects become more prominent more patterns will no doubt be discovered and documented much like the progression of OO patterns in the decade since the publishing of the GOF catalogue (Gamma et al. 1994). A brief summary of the patterns presented by Laddad and how they could be applied at SolNet is presented.

Worker Object Creation Pattern

A worker object is used to encapsulate a method in an object so it can be passed around, stored, and invoked. It is commonly implemented in Java by implementing Runnable and having the run() method delegate the method call to the worker method. It is heavily used in implementing Swing applications since all methods that change or access the state of Swing components need to be wrapped in worker objects and passed to the event queue to be executed[8]. As a result of having to wrap methods in worker objects the code becomes polluted and difficult to read and maintain. The worker object pattern presents a solution which involves making normal method calls which are intercepted by an aspect and wrapped in worker objects. Laddad presents this solution using a Swing example that greatly simplifies the code required to correctly write a Swing GUI. These Swing aspects could be useful for SolNet applications that require the use of desktop or applet solutions instead of web interfaces.

Exception Introduction Pattern

Aspects can introduce new checked exceptions when implementing crosscutting behaviour. These exceptions may not be part of the original set of checked exceptions that can be handled by a method. The exception introduction pattern presents a method of dealing with this situation in a way that ensures the application can still be compiled and behaves as expected when the new exception is encountered. This pattern was initially used when implementing exception handling in the SPER project in Chapter 9. However, this was later replaced by a more suitable solution for the problem context. Nevertheless this is a very useful pattern which could be applied to SolNet projects to enable handling of framework exceptions.

Wormhole Pattern

Often there is contextual information which must be passed from a caller to a callee through a set of methods in a control flow resulting in API pollution. The wormhole pattern solves this by allowing contextual information to be passed directly from the caller to the callee without polluting intermediate APIs with extra parameters.

---

[8]This is required since Swing components are not thread safe

Participant Pattern

The participant pattern allows classes to opt into using an aspect to implement behaviour based on some characteristic it possesses such as 'method executes slowly' which cannot be otherwise easily identified based on naming patterns and is best associated with a class. However, this pattern is most likely to have been rendered obsolete with the addition of JSR 175 annotations in Java 5.

5.4.3 AspectJ Idioms

This is a brief introduction to some idioms that have been presented for AspectJ. These are 'programming tips' which can help avoid common errors when writing AspectJ aspects and solve common programming problems.

Laddad (2003) presents solutions to infinite recursion (caused by aspects advising themselves) by excluding an aspect from its pointcut definitions. Additionally he presents methods to nullify advice and the use of empty pointcut definitions which are useful when extending base aspects where no join points match in the specific system. All these idioms have been used in implementing aspects for SolNet projects in Chapter 9.

Hanenberg & Schmidmeier (2003) present the most comprehensive set of idioms encountered which cover those presented by Laddad (with different names) as well as several additional idioms. Some of the idioms we have used are Template Advice, Abstract Pointcut, and Composite Pointcut. They also specify the Pointcut Method, Container Introduction, Marker Interface, Chained Advice, and Advised Creation Method.

The Abstract Pointcut is extensively used when writing base aspects. In this idiom advice is written to act on the join points of an abstract pointcut. This abstract pointcut is overridden in the sub aspects to provide the real set of join points in the application.

The Composite Pointcut is used to split up a complicated pointcut into several easily understood pieces which are combined to provide the full definition. This has been used when specifying pointcuts in the SPER application to make the pointcuts more reusable.

Template Advice is used to allow some behaviour to be changed depending on the join point that is being executed. It was used in the EOS application (See Chapter 9) to account for variability in exception handling policies.

These idioms have been useful in solving AspectJ specific problems when implementing aspects for SolNet projects. These can help to form best practices for

future SolNet work. Furthermore, additional idioms may be formed for company coding standards and no doubt new idioms will be created as aspects become more prevalent.

5.5 Summary

This chapter has investigated the effect of aspects on the design phase of software development. In particular it was found that separating requirements at the design phase can bring the benefits of aspects in the implementation phase into the design phase. Applying aspects earlier in the design phase helps to ensure consistency between the design and implementation by avoiding ad hoc redesigns by developers. Moreover, it promotes traceability from requirements to implementation.

There are many design notations that can be applied when using aspects, of which five have been presented. Of these we believe Jacobson's approach has the longest term potential for SolNet, but it requires a bigger investment in the use of aspect technology than they are likely to make in the short term. Therefore, Elrad's approach is recommended for development in the short term. This should be easily grasped by the Business Analysts and Developers alike. Tools should be unaffected and potential for code generation and round tripping can be further explored.

One of the first questions faced when presenting AOP to SolNet developers was how AOP affects patterns. Patterns are an important part of the development best practice at SolNet where they make use of classic patterns such as the Abstract Factory (Gamma et al. 1994), and J2EE patterns such as Business Delegate and Service Locator (Alur, Crupi & Malks 2001). We have shown that aspects can help to simplify and make design patterns more reusable, as well as opening opportunities for a new range of patterns associated with the use of aspects. In particular, the ability to refactor existing patterns to make them reusable and the base code more reusable presents another opportunity to increase flexibility of SolNet frameworks and reduce development effort.

Finally idioms used with AspectJ and how these have been applied in SolNet projects were discussed. These idioms solve very specific design problems but allow more effective and productive use of AspectJ. Furthermore, they assist in the development of more reusable and flexible aspects.

CHAPTER 6
TESTING

6.1 Introduction

Aspects introduce new challenges when verifying program correctness. However, it is only recently that testing has received attention from the aspect development community. The early focus on testing looked at how aspects could be used to aid testing of software rather than testing of aspect software. Aspects were used to write test cases by allowing specification of system invariants and substitution of mock objects (Isberg 2002, Monk & Hall 2002). This is a distinct act from the testing of the correctness of software that realises functionality using aspects. Moreover, it may be desirable to test aspects independently so they can be included in reusable aspect libraries or sold as off the shelf components (COTS).

The other element that should be considered is validation. Recall that verification is checking the program built works correctly, where as validation ensures the correct program has been built (Pressman 2001). In fact, aspects should make this task easier than before since requirements can be traced directly to implementation elements and verified with test cases (Jacobson & Ng 2004). This was difficult with traditional approaches because requirements tended to be spread across multiple implementation units making direct traceability and testing of a requirement difficult. Validation testing is not explored any further as the focus of this chapter is verification techniques.

In this chapter various approaches and tools that have been proposed for performing testing (Lopes & Ngo 2005, Xie, Zhao, Marinov & Notkin 2005, Souter, Shepherd & Pollock 2003, Mortensen & Alexander 2005, Xu, Xu & Nygard 2005, Zhao 2003, Zhao 2002, Alexander, Bieman & Andrews 2004, Ceccato, Tonella & Ricca 2005, Lesiecki 2005*c*) are discussed. Testing challenges introduced by aspects are identified and how these can be resolved is discussed. Similar problems are encountered to those faced when testing OO software as well as new challenges specific to AO software. Introducing tests on separate units can bring the advantages of AOP to the testing phase. Tests are decomposed to specific requirements improving

traceability and making detection of causes of errors easier to determine. Furthermore maintenance time and costs can be reduced as tests only need to be updated and rerun on the concerns being modified rather than the whole application (Souter et al. 2003).

Finally, performance testing of aspects has been identified as an important quality assurance requirement. The techniques examined in this chapter do not consider performance testing as we believe that existing techniques can still be used to ensure that aspects do not introduce unacceptable overheads into an application. The separation achieved using aspects can make bottlenecks in applications easier to identify since performance can be tested on individual components including aspects.

6.2 Testing Elements

The first question faced when testing Aspect-Oriented software is what should be tested? We believe there are several distinct elements that need to be tested depending on the phase of the software development. These elements are:

- Pointcut matching - Verifying the strength of the pointcut (too weak matches too many places, too strong matches too few places) (integration testing).
- Advice - Does what it is suppose to do (unit testing).
- Advice interaction - Particularly the effect of different advice orderings (integration testing).
- Advice/System interaction - Effect of the advice on the base system (integration testing).

These are the new elements that we believe must be tested. However, it must be remembered that Aspect-Oriented languages such as AspectJ are a superset of their respective base languages such as Java. This means every Java program is a valid AspectJ program. Therefore, Aspect-Oriented programs have the same faults as Object-Oriented programs, but they can also have additional sources of program faults. We think the elements above are the key new elements that need to be verified to ensure aspect software behaves correctly. This is similar to the fault model for AOP proposed by Alexander et al. (2004). These faults are:

- Incorrect pointcut strength (Too many, or too few join points are selected by a pointcut).

- Incorrect aspect precedence (Aspects are not applied to a join point in the correct order).
- Failure to establish expected post conditions (Aspects make changes to the execution of a join point which does not meet the post conditions specified by the original developer).
- Failure to preserve state invariants (Changes are made to an objects state by an aspect that violate established invariants).
- Incorrect focus of control flow (Some join points should only be selected during certain execution sequences).
- Incorrect changes in control dependencies (Around advice can change the control flow of a join point).
- Incorrect changes in exceptional flows (Advice throwing exceptions or handling exceptions cause implicit control flow changes).
- Failures due to intertype declarations (The control sequences could be changed for code that depends on the structure of a class such as what parent classes or interfaces it has).
- Incorrect changes in polymorphic calls (Method introductions which override a super class method can modify expected system behaviour).

This is a more comprehensive fault model for AOP, however we believe many of these errors can be picked up through detection of traditional OO faults (e.g. state invariants). From a minimalistic approach we believe that aspects could be verified using a two step approach. The first is the unit testing of the aspect logic, and the 2nd is the normal integration testing of the aspects with the system. However, this is unlikely to bring about as many of the benefits of using AO throughout the lifecycle and may make faults harder to detect. Therefore, in this chapter several approaches that have been proposed to support the testing of Aspect-Oriented software are presented.

6.3 Aspect Testing Challenges

Aspects present additional challenges to testing, in fact some authors have said current aspect languages are impossible to test (Lopes & Ngo 2005). However, we believe that although aspects can be harder to test than classes, there are techniques

that can effectively deal with aspects. The major challenges in testing aspects come from:

- Aspects do not have a separate identity (they are bound to some context) (Alexander et al. 2004, Lopes & Ngo 2005).
- Aspects are tightly bound to the woven context (changes in classes propagate into aspects) (Alexander et al. 2004, Lopes & Ngo 2005).
- The control and data dependencies are not obvious (Alexander et al. 2004).
- Emergent Behaviour - Fault may be from class, aspect, or a side effect of the weave order (Alexander et al. 2004).
- Cannot easily verify pointcuts (Colyer 2004).

In our experience, the major challenge faced when unit testing aspects is the tight binding of an aspect to its context. The ability to easily substitute objects to represent contextual objects would make the testing of aspects simpler. At this time the simplest way to achieve this is to weave the aspects into a dummy application. The ability to isolate aspects and advice at a lower level, and the avoidance of weaving when unit testing would be advantageous.

Secondly, it has been found that verifying pointcut strength is difficult and involves a tedious manual examination process. We question if this can be made easier by considering different specification languages. For example naming patterns can easily capture unexpected points, where as a structured language such as annotations may improve the ability to write correct pointcuts and verify them.

6.4 AOP Testing Approaches

This section looks at the various approaches that have been proposed for testing AO software. This is still an area where research is evolving and little in the way of best practice has been developed. The practicality of these approaches, tool support available, fit with existing practices, and challenges faced are discussed.

6.4.1 Data Flow Testing

An approach has been suggested to use data flow based testing of aspects and classes affected by aspects in (Zhao 2003, Zhao 2002). Data flow testing tests how values associated with variables can effect program execution. Three levels of testing are

proposed: inter-module, intra-module, and intra-aspect/class. A control flow graph is used to calculate the def-use pairs used for data flow testing. It is claimed that aspects acting on a class must be considered when unit testing a class. However, we believe that since this is a unit test, it should be possible to test the class independently of the aspects in the same way we would test a class free of other services such as a database (normally using mock objects). This allows verification of the base logic rather than the extra concerns that must also be applied. There may be situations where this is not possible because the aspects are required to implement the basic functionality of the class and removing them will cause problems (e.g. using AspectJ to implement checked exception handling). Intra-module testing only looks at individual units such as advice, an introduction, or a method. Inter-module testing tests the public modules and some of the modules they call directly or indirectly. Intra-Aspect/Class testing tests the interactions of multiple modules in an aspect when called in a random sequence. There are plans to implement tool support. The tool will gather control and dataflow information to generate the test cases.

6.4.2 Test Adequacy

Mortensen & Alexander (2005) have proposed an approach using mutation and coverage testing to ensure that AspectJ programs have been suitably tested. Fault based testing is used in combination with coverage and mutation testing to ensure the adequate coverage of faults. Mutation testing involves injecting faults to see if the tests can detect the error. Coverage testing of the woven program uses statement and branch coverage, and def-use pairs. There are different coverage criteria that are necessary depending on the environment being tested (factors such as control changes and data dependencies of the aspect). Mutations applied include pointcut strengthening, pointcut weakening, and precedence changes. The mutations are currently made manually, but there are plans to develop an automated approach.

We believe this approach is useful to assess whether appropriate tests have been produced. However, there are concerns about the need to make changes to the pointcuts used in the application without tool support to automate these steps, including the necessary reversals. Moreover, it is uncertain whether incorrect pointcut strength would be easily detected since errors made in specifying the pointcut may also be replicated in tests.

6.4.3 Test Generation

Xie et al. (2005) proposes Wrasp, a framework to automatically generate tests for AspectJ programs. Wrasp generates wrapper classes which enable AspectJ programs to have tests generated using standard Java tools such as JTest[1]. JTest generates tests automatically from Java byte code. Wrasp provides wrapper classes to allow this tool to work taking into account weaving issues. However, some situations currently cannot be dealt with such as context classes like JoinPoint from the AspectJ framework and the AroundClosure class used for around advice. There are plans to produce mock objects to allow these to be handled in the future. Xie, Zhao, Marinov & Notkin (2004) have also proposed Aspectra for the detection of redundant tests for AspectJ programs which could be used in conjunction with Wrasp.

It is certainly useful to automate the generation of tests and to be able to use existing tools. However, we question whether this tool can effectively deal with the new faults created by the use of AspectJ programs such as pointcut strength. It would seem reasonable to assume this framework could be changed to work with different aspect languages in a similar way so the general techniques may be portable.

6.4.4 Unit Testing Aspects

The Java Aspect Mark-up Language (JAML) provides a means to write aspects using plain Java for logic and XML for aspect bindings (Lopes & Ngo 2005). JAML aspects are easy to unit test using the proposed JamlUnit, an extension of the Java testing tool JUnit[2]. Testing of aspects with languages such as AspectJ is problematic because of the tight coupling between the woven context and the aspect behaviour. This makes traditional testing methods inappropriate and unit testing impossible (aspects do not have independent identity). This challenge is a result of the language design not the crosscutting concepts. However, JamlUnit only tests the logic of aspects, not the bindings. Helper classes are provided to enable regular JUnit tests to be written for aspect code. This research has also found that most aspects implement orthogonal concerns which become library aspects, yet there has been no way to ensure that these are correct. There are still challenges with this approach such as selecting appropriate mock objects and creating execution context.

Issue is taken with several aspects of this approach. Firstly, there are many aspect languages that allow separation of the logic and bindings using XML or

[1]http://www.parasoft.com/jsp/products/home.jsp?product=Jtest

[2]http://www.junit.org

annotations, so many of the claims about current aspect languages only really apply to AspectJ. Secondly, their claim that AspectJ aspects cannot be tested is quite false as will be shown in Section 6.4.7. It is quite possible to use delegation with AspectJ to achieve the same logical structure as JAML. This paper quite clearly identifies that aspect testing faces the same problems as normal Java programs such as creation of context and mock objects. We do not believe it is necessary to adopt a new aspect language solely for the purpose of allowing testing of aspects. All the aspect languages we have applied traditional unit tests to had problems related to the need to provide appropriate mock objects and context. New testing frameworks and techniques can solve these problems and better tackle the faults introduced by aspect programs rather than focusing solely on allowing use of existing techniques and ignoring the new sources of program faults.

6.4.5 State Based Testing

An alternative to the data flow testing approach is to use state based testing. Xu et al. (2005) uses Aspectual State Models (ASM) which are an extension of the testable FREE (Flattened Regular Expression) state model (Binder 2000). The ASM allows the capture of impact of aspects on the state model of classes. FREE is used to model the base elements of the program and ASM introduces elements that model crosscutting elements and their relationships. This approach only considers join points, advice, and pointcuts rather than the more complex elements such as aspect composition, aspect inheritance, and introductions. Since the base classes are unaware of the extra states introduced by aspects the testing must exercise these states as well. However, this approach can use Transition Tree-Based testing to generate tests directly from the ASM. This is a huge advantage for automation. However, these tests should be complemented by tests for unintended behaviour of aspects since this only generates tests for expected behaviour. This testing approach can reveal OO faults as well as aspect faults like incorrect pointcut strength and failure to preserve state invariants. Like all state based testing this suffers from 'state explosion'.

Although this approach has the ability to generate tests it does require extensive state modelling to be performed prior to this. This would only fit in well to an environment that performs state modelling as part of the design process. This could fit well with the aspect design process described by Elrad et al. (2005).

6.4.6 Fault Based Testing

Ceccato et al. (2005) discusses how a combination of new and old testing techniques can be used to ensure that both traditional OO faults and those new AOP faults proposed by Alexander et al. (2004) can be tested. Coverage testing is extended to expose weak pointcuts (too many join points) by detecting traversal of incorrect branches by aspects. Traditional testing techniques should expose strong pointcuts (too few join points) by detecting missing behaviour. Post conditions and invariants should still be detectable with unit tests. Branch coverage testing of the base code should detect static crosscutting problems and changes in control flow. However, two faults that require new techniques are composition order and dynamic pointcuts. If weaving order matters then dominance constraints should be specified otherwise errors could occur. Different composition orders can be tested, but the best approach is to test just those that differ by at least one data dependency from any other test. It is not possible to test all dynamic pointcuts (could be infinite) so the k-limiting approach used in path coverage criteria is applied. However, these approaches do not consider how to test aspects in isolation or how to decide which code needs to be tested on changes. For this purpose they suggest an incremental AOP testing approach.

This approach covers the problems that needed to be tested, but as was mentioned does not cover unit testing of aspects which is particularly desirable when developing reusable aspect libraries.

6.4.7 Traditional Testing Techniques

The approaches above mainly use theoretical approaches for testing of aspect software. Many of these techniques (or slight variations) underpin or influence the way testing is conducted using many of today's tools. However, the techniques described so far offer limited maturity and tool support making them unsuitable for commercial development. This section discusses patterns that have been proposed to allow testing of AspectJ programs using traditional tools. These techniques are not perfect, but they use many of today's commonly used tools to allow effective testing of aspects. As testing matures these techniques can be replaced by more adequate tools, but in the meantime these could be considered best practice when undertaking commercial development. The techniques described here are proposed by Lesiecki (2005*c*) to take advantage of aspects making testing of crosscutting behaviour easier since the behaviour is now modularised (cannot unit test without a unit). The tests cover both the functionality (advice) and specification (pointcuts).

Integration Tests

Integration tests can be written as normal JUnit tests. These tests check both the specification and functionality are correct. However, it requires experience to write tests that will detect aspect misbehaviour.

Visualisation Tools

Visualisation tools such as AJDT can be used to visualise and verify that aspects are being applied in the appropriate places. Unfortunately, this process cannot be automated and aspects with large numbers of matches throughout a system can be hard to manually verify. It is thought to be more difficult to find unintended matches than missing matches using this technique.

Crosscutting Comparison Tool

AJDT also offers a tool to capture the set of currently advised join points. These can be used to periodically check where changes have occurred. This can help to detect when new join points have been inadvertently advised or removed. If a large number of changes occur then this method can be difficult to manually work with.

Delegation

One of the major problems faced in unit testing aspects is they are not easily instantiated without being woven into some context. One approach to solve this problem is to delegate the aspect logic to a class. This doesn't work well when per object instantiation or contextual information is required, but it does provide a way to unit test many aspects.

Mock Objects

Mock objects can be used to check whether advice is being triggered in appropriate places. The mock objects can detect whether they have had advice triggered and this can be used to verify pointcut strength. JMock[3] is commonly used for mock objects and can be applied for this purpose.

---

[3]http://www.jmock.org

Mock Targets

Mock targets can be used to substitute those objects that would be used in a production system. This allows library aspects to be tested with a context, but independent of an actual system. There are a few approaches that can be used to seamlessly perform this:

- Extend an abstract aspect and provide pointcuts to test the mock targets.
- Write mock targets to match pointcuts in the aspect.

This technique can also be used to verify more complex pointcuts such as AspectJ's cflow.

6.5 AOP Testing Frameworks

Techniques that can be used to test Aspect-Oriented software have been presented along with patterns which allow use of current tools such as JUnit and JMock. When using these tools for testing Aspect-Oriented software we are working around some of the testing problems by performing extra work. We would like to have a similar framework to those used in testing Object-Oriented software (e.g. JUnit) for testing Aspect-Oriented software. One such tool is aUnit[4] which is being developed by Russell Miles as part of his MSc. Adrian Colyer proposed aUnit in November 2004 for unit testing aspects in isolation on the AspectJ mailing list. The tool was to address problems faced such as (Colyer 2004):

- Cannot easily unit test aspects in isolation from a program.
- Cannot easily check pointcuts match all and only the join points wanted.
- Cannot easily test an advice body in isolation.

The objective is to write tests without needing to create packages and having to weave and run external classes just to perform testing. aUnit is an extension of JUnit based on the xUnit[5] architecture. It works by allowing a programmer to specify a sequence of join points that can be played back to the aspect. aUnit can check how the aspect responds to these join points to verify correctness. Contextual information can be supplied as either real or mock objects just like normal test cases.

Some early problems identified with the aUnit vision are:

[4]http://www.aunit.org

[5]http://sourceforge.net/projects/xunit

Listing 6.1: Proposed example aUnit code

```
public void testCallMatching() {

    // define the join points to test
    String [] jps = new String[] (
                        {"call(void Account.doFoo() &&
                            within(org.xyz.abc)"}
                    );
    // get an instance of the aspect being tested
    X x = X.aspectOf();
    // run the aspect against the join points
    playBack(jps, x);
    // check the results
    assertInvoked(x, "before", "1");
}
```

Listing 6.2: JMock style aUnit code

```
// specify the join point and conditions using
// API calls with the required values as parameters
joinPoint = new Call().to(Account.class, "doFoo")
                      .with(int(ss)).from(Facade.class)
                      .under(SomePreviousJoinPoint)
                      .will(returnValue("Foo done!");
```

- Testing around advice that uses proceed is difficult.
- Representing cflow.
- Context passing.
- Aspects with perXXX instantiation models.
- Making code paths as close as possible to the real deployment situations.

Listing 6.1 shows an example of a proposed aUnit test case. In this listing join points are specified using strings (as in aspects). A reference is obtained to the aspect and this is passed along with the join points to the framework to playback the sequence. It finishes by checking that the correct invocations occurred. One problem with this approach is the ability to make consistent mistakes when specifying pointcuts since they are specified in the same way as in an aspect. Another approach could be using Java code as in Listing 6.2 or a scripting language (Lesiecki 2004). This approach is similar to that used in JMock.

So far the proposed nature of aUnit has been addressed. The 0.1 release of April 2005 is now assessed. When developing this release of the framework several

Listing 6.3: Aspect Annotations

```
// Aspects being tested using aUnit must have the
// TestableAspect annotation
@TestableAspect
public aspect TestAspect {

    // each advice being tested must be named using the
    // TestableAdvice annotation so it can be identified
    // by the aUnit framework (advice don't have names)
    @TestableAdvice("uniqueid")
    before() : somePointcut() {
        ...
    }
}
```

practical issues when working with the AspectJ framework were encountered that affect the ability to conduct tests.

The first issue is advice cannot be named in AspectJ. Tests shouldn't depend on the ordering in the source code so it is necessary to find a way to identify advice. This was solved by introducing annotations (making it Java 5 dependent). AspectJ 5 will provide another way to solve this problem with the introduction of a new reflection API (thus bypassing the need to use standard Java reflection). An Aspect class will be available with methods such as getAdvice() and getDeclaredAdvice() (as in java.lang.Class).

The annotation @TestableAspect must be added to the aspect definition, and the @TestableAdvice to each piece of advice that will be tested. Listing 6.3 shows an example. Notice the use of annotations to name an advice as "uniqueid".

Secondly, instantiating join points are being investigated further as this is a challenging task due to the need to create different contexts. One possible solution is the use of XML configuration for specifying join points.

The 0.1 release was primarily about providing a tool supporting the concept exploration for unit testing aspects since this area lacks theoretical foundations and is still developing. The eventual aim is to merge the project with the AspectJ or AJDT project trees.

This release was found to be too immature for commercial work, however its use with a SolNet project is shown as part of Chapter 9.

6.6 Summary

This chapter has examined the verification and validation of aspect systems. It is asserted that validation is made easier using aspects since there is direct traceability between requirements and implementation units allowing tests to be decomposed for each unit.

The challenges in testing aspects are discussed. A simplistic approach to testing aspects is proposed using current techniques, but it is acknowledged that it is not satisfactory because it does not realise the benefits of using aspects in the earlier lifecycle phases. This motivates the exploration of new and more sophisticated aspect testing approaches. These approaches show much promise but lack examples of practical application and tool support necessary to adopt them in a commercial environment. Moreover, some of them make the testing process too complex. However, we have presented an approach that uses current frameworks and a set of best practices to effectively test aspects. This is the approach we currently recommend. A specialised aspect testing framework is also presented. This tool is currently too immature but promises to provide an effective testing framework for the future.

Finally, the need for performance testing of aspects and the potential benefits in identifying bottlenecks in applications is briefly discussed.

CHAPTER 7
METRICS

7.1 Introduction

It would be useful to have a quantitative comparison of Aspect-Oriented and Object-Oriented software. This allows comparative statements regarding the productivity and quality of the solutions. However, a suitable metrics suite is required to ensure that fair and accurate measurements can be made. In this chapter a selection of existing Object-Oriented and Aspect-Oriented metrics are examined and a suitable set is proposed to be used in the evaluation of SolNet projects in Chapter 9. Our selection aims to provide a small and balanced set rather than a comprehensive analysis of the software as recommended by Wiegers (1999). New metrics are not proposed, but it is argued why certain metrics are relevant and others are not to our study of SolNet software.

7.2 Motivation for Metrics

Measurement is fundamental to any engineering discipline as it allows us to gain insight by providing a mechanism for objective evaluation. Software metrics are measurements made related to computer software in areas such as software process, estimation, quality control, and productivity assessment. Quantitative measurements can be taken and compared with past averages (in-house or external) to determine if improvements have been made, or to pinpoint problems. They are important to ensure that judgement is not based solely on subjective evaluations (Pressman 2001).

According to the IEEE Standard Glossary of Software Engineering Terms (*IEEE Software Engineering Standards* 1990) a metric is defined as:

> a quantitative measure of the degree to which a system, component, or process possesses a given attribute.

Some measurements may be taken directly, where as some measurements can

only be made indirectly. Indirect measurements are harder to make and include properties such as maintainability and reliability.

This study is mainly concerned with the use of quality and productivity measures as opposed to those used for other tasks such as project estimation. Many of these quality measures will need to be made indirectly.

7.3 Traditional Metrics

Traditional metrics is considered to cover those metrics that have been used with Procedural and Object-Oriented software. Many of the metrics shown have been applied to both classes of software and we argue that some of these measurements will also have relevance when assessing Aspect-Oriented software.

Two classes of commonly applied metrics are size-oriented and function-oriented metrics. Size-oriented metrics use the lines of code (LOC) to normalise quality or productivity metrics (e.g. errors/KLOC). Function-oriented metrics use a measure of functionality for normalisation (Pressman 2001).

Size-oriented metrics are a simple measure that can be applied to any project but they have received much opposition due to their programming language specific nature, penalising short well designed programs, and they don't easily accommodate non-procedural languages (Pressman 2001). However, if the metric is being used in-house with clear methods to make LOC measurements and are used to make comparisons between the same languages then they could be considered a viable approach (Jones 1994).

Function Points are used to provide a technology independent measure of the systems functionality based on five classes of general system characteristics (external inputs, external outputs, external inquiries, internal logical files, and external interface files). Being a technology independent measure the number of function points stays constant and the only variable is the amount of effort required to deliver a set of function points. Typical values for different languages such as Java are available or in-house values could be applied. However, no data is available for AspectJ. Function points require trained, experienced personnel to be performed accurately (Longstreet 1992, Software Composition Technologies 1997).

Halstead metrics were developed to measure a module's complexity directly from source code placing an emphasis on computational complexity as determined by the operators and operands in the module (VanDoren 1997). The JHawk Metric tool (Virtual Machinery 2005) provides several flavours of the Halstead metrics including:

- Halstead Length - Calculated from the operator and operand occurrences.
- Halstead Effort - Estimated mental effort to develop the code.
- Halstead Bugs - Estimated number of errors in the code.

Another measurement of complexity that is generally better accepted than the Halstead metrics is Cyclomatic Complexity (McCabe 1976). Cyclomatic Complexity provides a quantitative measurement of the logical complexity of a program. This is based on the number of independent paths through the program. Cyclomatic Complexity is independent of the physical program size and complexity and depends only on the decision structure of the program. A low Cyclomatic Complexity indicates a program is more understandable and more amenable to modifications at lower risk than more complex programs (VanDoren 2000). It is the most widely used of the static software metrics (VanDoren 2000).

According to Jones (1994) LOC and Halstead metrics are invalid under certain conditions (such as comparisons between languages) and should not be used. In fact, LOC is one of the most widely used metrics despite being one of the most imprecise metrics ever used in Science/Engineering disciplines. However, under our conditions of use it does not suffer from these limitations as we are not really considering different programming languages. Jones (1994) also says that complexity and function point metrics are useful since excessive complexity raises defect potential and reduces productivity. Function points are also free of the economic distortions of LOC measures. No function point data was available for AspectJ (used in our refactoring project) and expertise in applying function points was not available. For these reasons, the application of function points was not considered further. It would be interesting to see productivity measures for AspectJ such as LOC/FP compared with those of Java and other OO languages.

How metrics can be applied to projects must be investigated. For instance, an AspectJ program consists of a primary decomposition containing Java code and crosscutting behaviour provided by aspects. The aspects will be woven into the primary decomposition to produce an equivalent Java program. So, from the primary programmer's perspective the complexity may decrease because of the code injection, yet the fully woven program is of equal or greater complexity to the original. We believe it is appropriate to consider the base program independent of the aspects when using standard metrics as this is what they are designed to measure. By considering this increased level of abstraction we are making a valid assumption like we do when working with different level languages such as assembler and

Java (which both end up as equivalent machine level instructions). It is clear that aspects create new associations and new elements of complexity that may need to be considered. However, how can we compare these new associations and make comparative measures against a program that doesn't use aspects? It is shown by Ceccato & Tonella (2004) that decreases in one area of the program were often compensated for with increases in new relations as a result of aspects being added. The question is which measure should be increased at the expense of the other? The answer to this question is likely to depend on the context to which aspects are being applied. In our study aspects implement infrastructural concerns that crosscut a number of components in a consistent way. The nature of infrastructure aspects allows them to be abstracted from a particular project and be reused at no cost. This 'free' nature allows us to disregard aspects when analysing the base program. In this situation we believe aspect relationships are better than the relationships being removed. However, this may not be true in all situations.

When considering just the base program, it would be expected that complexity would decrease from the programmer's point of view. If the assumption is made that obliviousness applies then it would seem valid to measure the complexity solely on the base program. If this principle doesn't apply then it may not be a valid comparison. This would be determined by the type of aspect (e.g. an aspect that provides tracing to all public methods would not require any expertise from the base programmer. Other aspects may require the programmer to consider their existence in the way they program. The extent to which this occurs would likely be the deciding factor). This is backed up by Zhang & Jacobsen (2003) who states changes can be measured independent of the primary program since aspects are maintained separately. The complexity of the program is the sum of the aspect and primary program. However, this is managed by the compiler not the architect. Zhang & Jacobsen (2003) also predict that AOP should reduce Cyclomatic Complexity as crosscutting elements are removed, and size should be reduced.

Based on the above arguments the traditional metrics of LOC, Halstead Effort, and Cyclomatic Complexity will be used to evaluate SolNet projects. These metrics should give a reasonable coverage of the effort required to develop aspect applications and the complexity (indication of time to understand, number of bugs, and maintainability) when compared with the Object-Oriented application.

The next section discusses metrics that have been developed for use with Aspect-Oriented software.

7.4 Aspect-Oriented Metrics

There has been some research into new metrics for aspect software. Some of these metrics are not entirely new, but extensions of old metrics to incorporate Aspect-Oriented features (e.g. including intertype declarations when evaluating the depth of the inheritance tree or number of methods in a class). It is well known that aspects introduce new couplings between aspects and the principal decomposition so there are also new metrics to account for these relationships.

Ceccato & Tonella (2004) propose a metrics suite that extends the commonly used OO metrics suite of Chidamber & Kemerer (1994). This suite was designed to investigate the advantages and disadvantages of using AOP. In particular they consider the implicit coupling between aspects and the principal decomposition. Many of the existing metrics are easily adapted by unifying classes and aspects, and methods and advice. Additionally minor adaptions are made for static crosscutting. However, they also recognise the need for specific measures for aspects. The metrics suite includes: Weighted Operations in Module, Depth of Inheritance Tree, Lack of Cohesion in Operations, Coupling on Advice Execution, and Coupling of Method Call. They have developed a tool to support collection of these metrics, and these metrics are also used by the AOP Metrics tool (Stochmialek 2005). In their small example they find that improvements in one metric as a result of applying AOP were at the detriment of another metric. A decision must be reached as to which one of these properties is more desirable.

Another metrics suite to measure aspect coupling is proposed by Zhao (2004). Since coupling is often used to indicate better maintainability, reliability, and reusability they have focused on metrics measuring the couplings between aspects and classes.

We don't believe that the metrics available for Aspect-Oriented software are mature enough, or provide compelling reasons to be applied in our study. We use similar metrics to those applied by Zhang & Jacobsen (2003) and have used similar reasoning in motivating their relevance.

7.5 Summary

This chapter shows that there are existing metrics that can be used to make valid comparisons between different aspect programs and between aspect and OO programs. There are certain assumptions required when making these comparisons. However, for companies making in-house comparisons these assumptions are likely

to be fair. It would be an interesting area for future work to evaluate the use of function point metrics with aspect software. Aspect software creates new associations and couplings that should be measured. However, we question the usefulness of these comparisons against traditional implementations where these relationships do not exist and decisions must be made as to what type of relationship is more desirable. When evaluations are being made between aspect systems these new measurements will be important.

CHAPTER 8
AOP STANDARDS

8.1 Introduction

In this chapter standards related to AOP frameworks used for implementing Aspect-Oriented software are investigated. This includes the motivation for standards, investigation of current standards, and recommendations for potential standards. This chapter does not investigate standards across the rest of the software development lifecycle. Chapter 5 examines the design phase of the lifecycle including possible extensions to the UML standard for aspect modelling.

8.2 Motivation for Standards

There are many AOP frameworks available for implementing Aspect-Oriented software in a Java environment such as AspectJ, JBoss AOP, Spring, AspectWerkz, Nanning[1], and Dynaop. It is easy for new users of aspect technology to be overwhelmed by the large number of frameworks available and the differences and similarities between them (See Chapter 3 for a comparison of the major frameworks). Unfortunately, each of these frameworks uses its own aspect representations and syntax. Although there are some common elements, there are enough differences to be frustrating to developers. This reduces the ability of users to easily switch between frameworks, and also the ability to take aspects developed for use in one framework to another (very useful for reusable aspect libraries). Moreover, the frameworks offer many of the same features, although in slightly different ways and often with different terminology or semantics. Therefore, it would make sense to develop a standard to reduce the needless differences between frameworks (Kiczales 2003).

In computing, standards are often considered to be synonymous with interoperability. However, standards are a formal protocol adopted by a group, where as

[1]http://nanning.codehaus.org

interoperability is less formal and only requires the ability of one system to be able to access features or resources of another (Shirky 2001). Interoperability can be achieved without a standard by groups developing their own compatibility layers. Standards can also help achieve other design goals such as avoiding vendor lock-in, promoting reuse, and sharing knowledge. These goals are all positive motivators for forming standards for AOP. However, it must also be established as to whether standards are needed to achieve these goals, or if they can be achieved via other means such as frameworks providing their own compatibility layers.

We believe a standard can help achieve the following for AOP:

- Writing aspects once that can be used with other AOP implementations (interoperability and reuse).
- Developers can concentrate on learning one technology regardless of the final deployment technology (avoid vendor lock-in).
- Good aspects written on one platform become available to others (sharing knowledge and reuse).
- Better tool support - no need for custom plug-ins for each tool (reuse).

In early 2005 the two largest frameworks, AspectJ and AspectWerkz, merged. This merger has the potential to produce a de facto standard AOP implementation for Java (Almaer 2005). However, other major implementations such as JBoss AOP and the Spring Framework have strong ties to their respective environments ensuring they will not be rendered obsolete in the near future. Moreover, there are also smaller frameworks competing in niche areas such as Dynaop and Nanning. Some of these smaller frameworks take different approaches to that of the mainstream frameworks such as the use of semantic pointcuts in the JAC framework (Pawlak 2005*b*). These frameworks could be the popular implementations of tomorrow and provide new ideas to continue to innovate AOP. This leads us to the question of whether it is too early to standardise. Standardising early could mean innovative ideas are missed that could make AOP a more popular and proficient technology, but failure to standardise or doing so too late creates more challenges for developing a user base (Shirky 2001).

From a commercial perspective standardisation opens the path for an aspect component market. Vendors can sell standard aspects which implement a particular concern. Companies can purchase these off the shelf (COTS) and reuse them in their projects. This requires components that can be deployed in different AOP

environments and the ability to specify how the component should be integrated with the particular system. Moreover, standards should mitigate the risk in purchasing these components since they can still be used if a particular vendor or AOP implementation disappears (e.g. bankrupt). This reduction in risk spreads to other areas of adopting the technology as standards imply increased maturity and stability providing greater confidence for businesses to invest in the technology.

Now that the need for standards has been motivated, the remainder of this chapter will look at current standards efforts, potential standardisation paths, and how some frameworks are currently trying to achieve some of these goals without implementing standards.

8.3 Candidates for Standardisation

Since there are few standards available for AOP a list is formed of potential candidates for standardisation. These are not intended to be standards, but a road map to where standards could be formed and what they could potentially contain. We are certain that AOP will undergo a major standardisation in the future but at this point in time competing frameworks provide the necessary innovation to continue to advance AOP into the mainstream of software development.

The most obvious place to start standardisation is the join point model. This is crucial if frameworks are to fully inter-operate and make use of each other's aspects. Most frameworks expose similar join points with only a small number of variations so the core elements should be easy to standardise. However, alternative approaches like the semantic pointcuts of the JAC framework should also be considered as a potential future model. Also, frameworks such as Spring expose a reduced number of join points to prevent breaking OO principles such as encapsulation and data hiding (e.g. Spring does not allow advising of field accesses). We believe it would be better to have a full join point model as part of the standard, but some frameworks may only partially implement the standard (similar to what happens in SQL).

It would be ideal if advice could be standardised. This is another necessary step in moving aspects from one framework to another. However, there are challenges here when approaches such as AspectJ have extended the Java language. A standard here would need to make a choice as to whether standard Java will be used or language extensions. Java 5 annotations add new possibilities to avoid language extensions (e.g. AspectJ 5 will include support for pure Java aspects using annotations). Advice is so similar to a method that if a standard Java approach was to be taken then this would enable the most cross platform support and would be the

least intrusive option.

Configuration of aspects is often left to each framework. Some approaches are XML, Annotations, and AspectJ pointcuts. This is acceptable because when aspects are moved between applications they need to be reconfigured for the specific environment. However, it could still be useful to standardise the languages used, particularly the patterns used to match code elements.

```
e.g.  execution(* *(..)) in AspectJ and
execution(* *->*(..))in JBoss AOP are equivalent.
```

These minor differences are totally unnecessary and should be eliminated. Most frameworks are using AspectJ's style so this would be a sensible choice for the basis of any future work.

A general area that could be standardised is the terminology used by AOP frameworks. The core terminology is currently quite stable, but there are still some small differences between frameworks. An example would be Introductions and Intertype Declarations which both refer to the static modification of a class. Different choices of terminology are one of the biggest hindrances to new users of aspects.

Standardising these areas would still allow different frameworks the ability to do different things 'under the hood' in terms of weave times, tool support, and runtime efficiency improvements. Widely adopted standards in terms of join points, pointcut expression languages, and advice have yet to evolve. This seems to be a major hindrance to AOP because there are many implementations out there, each doing their own thing. Of the four implementations with a significant user base (AOSD.NET 2005), Spring, AspectJ, and AspectWerkz have large numbers of similarities in comparison to JBoss AOP which has its own pointcut language and semantics.

Weaving tools could be standardised by incorporating AOP into the Java Virtual Machine. This would allow very efficient AOP implementations to be created as many of the overheads currently introduced by AOP tools will be removed. Many of the current weaknesses in AOP can be solved with this support. This is discussed further in Section 8.8.

8.4 Current Standards Efforts

There have only been two attempts to standardise AOP thus far. One of these was the AOPI (AOP Interfaces) project started by Renauld Pawlak in April 2003, and the AOP Alliance started in March 2003. The AOPI was merged into the AOP

Alliance in June 2003 leaving the AOP Alliance as the only major standards effort. In 2004 Bill Burke (JBoss) revealed a major effort that was going to be launched involving the majority of the big names in AOP (Burke 2005). Unfortunately, for reasons unknown, this never eventuated. Therefore, in the next section the focus will be solely on the AOP Alliance's standard.

8.5 AOP Alliance

The AOP Alliance[2] is the only major standards effort thus far for AOP. The AOP Alliance is self-described as a group of people interested in Java and AOP who collaborated to try and form some standards (AOP Alliance 2003). This group consists of many high profile people from the AOP community who have worked together to produce a set of interfaces to allow interoperability between implementations. Unfortunately, this group seems to have focused largely on dynamic proxy issues resulting in a standard that is only really applicable to certain types of implementation (i.e. proxy based interception). Of the four major frameworks only Spring has implemented the interfaces, and even then, only partially. However, smaller frameworks such as JAC and Dynaop have adopted the interfaces (Pawlak 2005*b*, Lee 2005). Version 1.0 of the standard was released in March 2004. This is the only release and the project has been inactive since.

8.5.1 Goals of the AOP Alliance

The current problem is AOP tools are designed for use in a particular environment. This is often because of the need to use a weaver to modify classes. Weavers are well fitted to an environment, but when used elsewhere they are liable to cause problems. The AOP Alliance believes it is useful to have an implementation specific to the particular problem being solved, however they want to have a core language that is shared by all these implementations (Pawlak 2003). This would allow:

- Reuse of existing AOP components.
- Simplify reuse by having a common API.
- Simplify adaption of existing AOP components for a given target environment.
- Simplify IDE/tool integration.

[2]http://aopalliance.sourceforge.net

The AOP Alliance decided to focus on issues outside of weaving logic and configuration logic since they are too tightly coupled to the particular AOP implementation. The goals of the AOP Alliance are similar to what we have proposed and they have tackled similar areas. However, we believe that weaving should be a goal for standardisation as this would remove many of the current problems faced by AOP implementations and tools.

8.5.2 AOP Alliance Components API

This section looks at the structure of the AOP Alliance interfaces including an analysis of its strengths and weaknesses. Recommendations for improvements are made. The AOP Alliance interfaces are separated into three distinct APIs.

Reflection and Program Instrumentation APIs

The reflection API allows location and identification of pertinent code fragments (e.g. Fields or Methods). It also provides access to information associated with the code fragments such as meta data (the AOP Alliance has a basic key/value meta data API which predates annotations). This API is used to give flexibility beyond that provided by the standard reflection packages which might not always be available depending on when weaving is performed (e.g. preload time). The API allows access to classes, members (fields or methods), code, and program units. It also has corresponding locators for these units such as ClassLocator. This API provides comprehensive access to a program to allow modification using the instrumentation API's.

The instrumentation interfaces provide useful methods for making program modifications at certain join points. Modifications available are add after code, add before code, add interface, add meta data, set super class, add around code, add field, add method, create class, and undo (remove instrumentation). This is a comprehensive set of program modifications and does not appear to lack any features available in the major frameworks. It is not clear whether this API would be useful for more advance functionality (e.g. dynamic pointcuts like cflow and cflowbelow used in AspectJ). Dynamic features would need to be supported in any future standard that is to be successful.

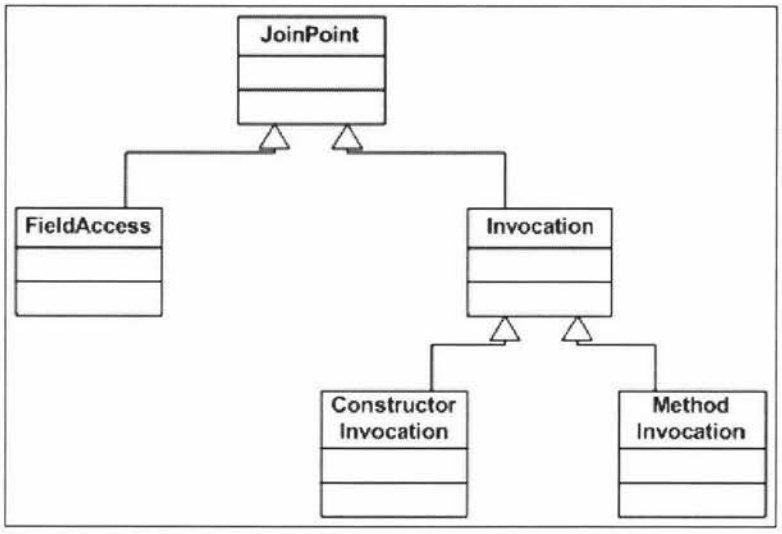

Figure 8.1: AOP Alliance Join Point Hierarchy

Interception API

The interception framework is based around an invocation join point model. The join point model consists of field accesses, constructor invocations, and method invocations. This hierarchy is described in Figure 8.1. The key problem here is the small number of join points exposed. This may be testament to the proxy focus taken by the AOP Alliance. Instantly this cuts down the ability for AOP Alliance aspects to be fully compatible with AspectJ, AspectWerkz, and JBoss AOP which all have far richer join point models.

The advantage of this approach is that most frameworks can support this as a minimum join point set as these join points are also the most commonly used. Examples of join points not available are exception handlers and dynamic control flow which are both supported by AspectJ and JBoss AOP. The Spring Framework does not support FieldAccess so only partially implements the AOP Alliance interfaces.

The join point model also contains access to some contextual information. Available to all join points is access to the object containing the join point, and the join point object (e.g. the method object). The method and constructor invocations provide further access to the arguments of the method or constructor being called. Finally, a proceed method is used to invoke the next interceptor in the chain (or the actual join point if all interceptors have been executed).

The second part of the interceptor API is the advice model shown in Figure 8.2. There is an interceptor for each of the runtime events described by the join point model. At this stage there are:

- MethodInterceptor - has an invoke(MethodInvocation) method.

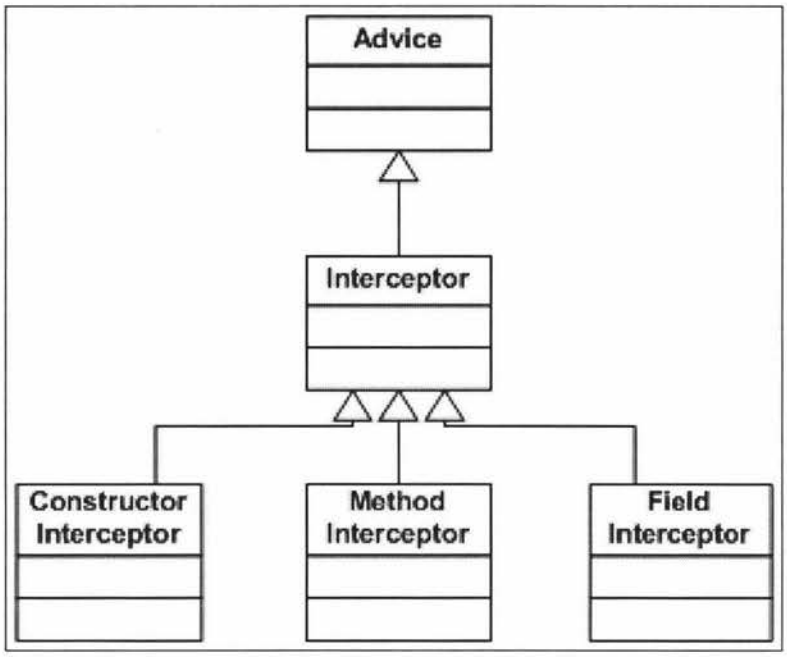

Figure 8.2: AOP Alliance Advice Hierarchy

- ConstructorInterceptor - has a construct(ConstructorInvocation) method.

- FieldInterceptor - has a method for set(FieldAccess) and get(FieldAccess).

There are several advantages and disadvantages to this type of model. Firstly, a separate interceptor class must be created for each piece of advice, unless runtime checks are made to determine which method/field/constructor has been invoked. This does not make sense when a concern needs to be realised as multiple pieces of advice that are best grouped in a single aspect. Typing information is lost using interceptors, adding further risk of incorrect advices being written and problems not being detected until runtime. Furthermore, there is only an around advice when using interceptors. This is also risky as it requires the programmer to remember to make calls to the proceed method to ensure execution of the next interceptor in the chain. Having a before and after semantic is important for this reason. The major advantages are the simplicity and use of pure Java to write advice.

General AOP API

Finally they define a general AOP API. This consists of a tag interface Advice. One possible implementation is the Interception API. There is currently discussion on adding an Annotation API as an alternative implementation (Pawlak 2005*a*).

8.5.3 AOP Alliance Interoperability

Aspects written using the AOP Alliance interfaces can be plugged into any other implementation that also implement these interfaces. This is a key goal of introducing AOP standards. The AOP Alliance interfaces focus on interception of method and constructor invocations, as well as field access. All their types of advice are interceptors. There is no notion of before and after advice, instead a call to proceed() is required to execute the next advice or the actual join point. However, this is the approach taken by several frameworks, many of whom have implemented the AOP Alliance interfaces.

It would seem that AOP Alliance aspects and AspectJ are incompatible. This is actually only true in one direction. The project lead from AspectJ (Adrian Colyer) has shown how to write an AspectJ adapter aspect which allows AOP Alliance aspects to be used with AspectJ. Listing 8.1 shows this adapter aspect. By extending the adapter aspect, specifying which AOP Alliance aspects to use and the join points to apply it to, AOP Alliance aspects can be used in AspectJ. Unfortunately, if aspects have been written using AspectJ, configuration for use with other frameworks is impossible unless specific support has been added by the framework such as that available in the Spring Framework (Johnson, Hoeller, Arendsen, Sampaleanu, Harrop, Risberg, Davison, Kopylenko, Pollack, Templier & Vervaet 2005).

Listing 8.2 shows an AOP Alliance aspect MyMethodInterceptor (implements MethodInterceptor). This interceptor contains no references to show where it should be applied. This is normally defined somewhere by the implementation language the aspect is being applied to (often XML configuration files). In AspectJ this is carried out using pointcuts, but since they know nothing about how to use this aspect directly, it is necessary to extend the base aspect AOPAllianceAdapter to provide the pointcuts and a reference to the AOPAlliance aspect to use. This is shown in Listing 8.3.

This example has served multiple purposes. Firstly, it highlights how to write an AOP Alliance aspect. Secondly, it shows how these aspects are free of connections to the platform which specifies how the aspect is used in its own proprietary way. Thirdly, it shows how a framework can provide an interoperability layer between itself and another framework to allow use of another framework's aspects.

Currently AOP Alliance aspects can be plugged into Spring, Nanning, Dynaop, and AspectWerkz. It is also shown that they can be easily used with AspectJ. JBoss AOP has chosen not to implement the AOP Alliance APIs because they do not fit into their development model. However, they believe their architecture could allow

Listing 8.1: AOPAllianceAdapter Aspect

```
// Library aspect which allows AOP Alliance Method Interceptor
// aspects to be used within AspectJ (Adapted from Adrian
// Colyer's version)
public abstract aspect AOPAllianceAdapter {

    // sub-aspects use this to specify the AOP Alliance Method
    // Interceptor to use
    protected abstract MethodInterceptor getMethodInterceptor();

    // sub-aspects use this to specify where to apply the AOP
    // Alliance aspect in their specific application
    protected abstract pointcut targetJoinPoint;

    // method interceptors should only be triggered at method
    // execution join points
    pointcut methodExecution() : execution(* *(..));

    //invoke method interceptor at target execution join points
    Object around() : targetJoinPoint() && methodExecution() {
       MethodInvocationClosure mic = new
          MethodInvocationClosure(thisJoinPoint) {
         public Object execute() { return proceed(); }
       }
       MethodInterceptor mInt = getMethodInterceptor();
       if (mInt != null) {
         try { return mInt.invoke(mic); }
         catch(Throwable t) { throw new SoftException(t); }
       } else {
         return proceed();
       }
    }
}
```

Listing 8.2: MyMethodInterceptor Aspect

```
// A very simple AOP Alliance Method Interceptor which logs
// a methods entry and exit
public class MyMethodInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation jp) throws Throwable {
        System.out.println("About to invoke: " +
           jp.getMethod().getName());
        Object ret = jp.proceed();
        System.out.println("Completed invocation of: " +
           jp.getMethod().getName());
        return ret;
    }

}
```

Listing 8.3: MyAOPAllianceAdapter Aspect

```
// Application specific aspect to configure and apply an AOP
// Alliance Method Interceptor using the AspectJ
// AOPAllianceAdapter
public aspect MyAOPAllianceAdapter extends AOPAllianceAdapter {

    private MethodInterceptor myMethodInterceptor = null;

    // Provides the AOP Alliance Method Interceptor to be used
    // to the super aspect
    protected MethodInterceptor getMethodInterceptor() {
        if (myMethodInterceptor == null) myMethodInterceptor =
            new MyMethodInterceptor();
        return myMethodInterceptor;
    }

    // specifies the join points where this interceptor should
    // be applied in this application
    pointcut targetJoinPoint() : within(org.xyz..*);
}
```

AOP Alliance interceptors to be 'plugged in' using their interceptor factories (Burke 2004). The major problem is the AOP Alliance aspects are not fully supported by the major frameworks because it was not developed in a way that supported their development models. This is a barrier that will need to be overcome if any standard is to be widely adopted.

8.5.4 Future of the AOP Alliance

The AOP Alliance is inactive. In August 2005 a major discussion began on their mailing list as to whether the project could continue to develop. Renauld Pawlak discussed about how he, like many others had lost interest in the AOP Alliance and become occupied with alternative projects. This is always a problem with groups formed in the open source community. This initiated a discussion on the possibility of adding a new area to the standard using the new Java 5 annotations. The disadvantage is that JBoss AOP, AspectWerkz, and AspectJ 5 have already carried out their own work using annotations. Developing a standard without agreement from these groups would be pointless. The AOP Alliance doesn't have enough power with the major frameworks to achieve its goals. This leads to further discussion in Section 8.6 as to where standardisation could go.

8.6 Potential Standardisation Paths

There are many potential paths that could be followed to produce standards for AOP. This section proposes the four options that we believe currently exist for forming a standard:

- JCP/JSR - add AOP to the Java language/Virtual Machine Support.
- Standard's groups, such as the Object Management Group (OMG) - standardise AOP at a higher level than Java. This would probably also result in standards for UML and Model Driven Development for AOP. At this stage the OMG is not doing any work in this area.
- Community Standard - this is similar to what the AOP Alliance set out to achieve. However, there is some conflict between some groups that could possibly make this a hard task to complete. This approach could result in an applicable and widely adopted standard if a consensus can be reached that is acceptable to the major players and smaller frameworks.
- De facto Standard - Users select a platform which becomes the standard through share popularity. The new AspectJ 5 is heading confidently in this direction. However, JBoss AOP and Spring have close ties to their environments which will help them remain popular.

We believe that a JSR is the best option for an AOP standard for the Java language to succeed. Currently there is no JSR in progress, which is surprising since IBM, BEA, and JBoss have the power to achieve this. Bob Bickel from JBoss has said they wanted AOP to become a standard feature of Java, and IBM have also backed the need for a standard (LaMonica 2003). There is also a prototype JRockit JVM with native AOP support. This results in more efficient weaving and memory usage. It would be possible to extend the Java language with a JCP for AOP, but this could also be achieved using other approaches such as annotations.

AOP has been developing over the past few years and many new ideas have been experimented with. It is getting to the stage where the frameworks are getting closer and closer in ability. At this point, we believe things appear stable enough for efforts to be best concentrated on a single framework. This would allow more work on making aspects accessible to the general population and building reusable aspect libraries. There is also a move towards other areas outside of AOP such as early aspects and testing which could also benefit from a standard development approach.

8.7 Framework Interoperability

While no standards have been adopted by all the frameworks, there have still been some efforts to promote interoperability. In this section compatibility layers that exist between the frameworks are discussed. For small numbers of frameworks the approaches described here are acceptable. However, there is a need to be aware of the n^2 problem. This occurs when each framework is required to write a compatibility layer for each of the others. For small n this is acceptable, but as n increases this becomes unwieldy. Until a standard is developed this is the most appropriate approach to ensure that aspects can be reused within different frameworks.

The biggest effort has come from AspectWerkz in the form of an extensible aspect container. The aspect container was designed to allow aspects from different platforms to be plugged in and used with AspectWerkz. This container supports Spring, AOP Alliance, and AspectJ aspects. The idea is to share the common elements from each of the frameworks and have a custom extension that is plugged in to handle the framework specific details (Boner & Vasseur 2005). The container could easily support other frameworks such as JBoss AOP if the necessary plug-in was developed. The container was slightly slower at running aspects from other languages than native support but the ability to plug-in different aspects should outweigh this concern in many environments. It should be noted that AspectWerkz has now merged with AspectJ so this will not be developed further.

Adrian Colyer has shown how AspectJ can use AOP Alliance using an adaptor aspect written in AspectJ to configure where the aspect should be applied and make the necessary translations. This is discussed in Section 8.5.3. We believe this could be applied to some of the other pure Java frameworks such as Spring and JBoss AOP in a similar way.

JBoss AOP does not have any compatibility layers for using other frameworks, and the other frameworks do not support JBoss aspects. It is possible to use the AspectWerkz container if a plug-in is written to allow it, but this has not been pursued. JBoss does not appear to have good relationships with the other AOP frameworks as they continue to work in the opposite direction producing proprietary solutions.

Spring has support for AOP Alliance aspects that do not have field accesses. Additionally it has made steps towards achieving some interoperability with AspectJ. AspectJ aspects can be configured using Spring's dependency injection and configuration of Javabean properties. Aspects can be instantiated using a Spring factory method instantiation model which was added to allow this integration. In Septem-

ber 2005, Adrian Colyer announced he was joining Spring to bring closer integration between the two projects. In particular Spring will adopt the AspectJ pointcut language and be able to parse AspectJ annotations. The AspectJ weaver and compiler will not be required; instead Spring's proxy based model can be used. It is this type of step between AspectJ and another major framework that appears to suggest AspectJ will continue to become the de facto standard AOP implementation.

8.8 JVM Support

We believe that ultimately AOP needs to be incorporated into the JVM. This would produce clear APIs for adding aspectual behaviour at runtime. Furthermore it would remove any overheads currently associated with using AOP since the framework will remove unnecessary code. This standard could be produced in one or two phases. The first phase would be the actual virtual machine API used by aspect frameworks to modify the code. The second phase could be a change to the Java language to incorporate aspects as first class citizens. This would ultimately turn Java into an Aspect-Oriented language allowing compilers, debuggers, etc to work as they do for current native Java code. Of these two changes the addition of support to the JVM is seen as the most crucial since this will define what it is possible to do with aspects and what cannot be done. Frameworks can stop developing competing APIs for manipulating code and concentrate on taking their aspect representation and applying it to the code using the API.

There is currently an experimental version of the BEA JRockit JVM with native support for AOP (Boner et al. 2005*a*, Boner, Vasseur & Dahlstedt 2005*b*). This was developed to address issues with current byte code manipulation techniques such as:

- Inefficient instrumentation - can be very CPU intensive and consume significant memory.
- Double bookkeeping - need to build up a database of information to allow the weaver to make necessary join point matches. This information is already gathered by the JVM to support the reflection API but is not available to the weaver.
- Runtime byte code changing using the Hotswap API adds complexity.
- Multiple agents a problem - precedence issues, changes conflicting, and undoing changes made by another agent all cause significant problems.

- Cannot intercept reflective calls.

It is believed that these problems can be addressed using JVM weaving. The JRockit team have implemented a prototype subscription based API for AOP support. This solves the above problems since the JVM already has the information available for weaving from forming the reflective database. Moreover, the JVM has bookkeeping information on which methods call others to support the Hotswap API. This would allow advice to be easily dispatched before a matching join point without byte code modification. Since the JVM can just dispatch advice as it is matched it becomes easy to add more advice to join points transparently and at linear cost. However, JVM support is not without its problems. There are some current semantics that will be difficult to address at the JVM level such as the initialization pointcut in AspectJ (all constructor invocations leading to the initialisation of an instance) and the need to support JVMs without support would be costly. A JSR would be an ideal solution to the final point and we believe this is the path AOP standards for Java should take.

8.9 Summary

At this stage a clear move towards AOP standards has not occurred. This may be positive since AOP is still evolving as different frameworks and ideas are being tried and innovation continues to be strong. However, the longer it takes to cement these ideas and move them into the mainstream, the slower adoption will be. We believe that AOP will be standardised through the Java Community Process to add native virtual machine support and a standard way of writing aspects. With AspectJ and AspectWerkz merging IBM and BEA may be seen combining to lead a JSR for AOP. A community standard can be capable of producing an excellent standard. However, this is unlikely as too many different players need to be satisfied and an agreement is unlikely to be reached.

Ultimately standards could help drive the formation of an off the shelf aspect component market and reduce the risk to businesses adopting aspects.

This chapter has focused on standards at the implementation stage of the software development lifecycle. However, having standard processes to help govern software development through the entire lifecycle will also be necessary. This has been explored in more detail in Chapter 5 which examines how aspects are identified and designed before implementation.

CHAPTER 9

INTEGRATING ASPECTS INTO A SOLNET SOLUTIONS PROJECT

9.1 Introduction

The major objective of this project was to apply aspects to a real world project. This allows assessment of the benefits AOP can bring to a complex environment in conjunction with enterprise technologies. This chapter explores the process used to aspectise a SolNet Solution's project and the results obtained by comparing an aspect and non-aspect version of the systems under examination.

9.2 SolNet Development Frameworks

This section gives a brief overview of some of the technologies used in SolNet development projects. The aim is to provide enough information to enable the reader to increase their understanding of the material referred to throughout the remainder of this chapter.

9.2.1 Application Structure

Many applications developed by SolNet are web applications built using the Model-View-Controller (MVC) pattern, implemented using the Struts framework (The Apache Software Foundation 2000). In the model there is a service and domain layer built using Enterprise JavaBeans (EJBs) (Sun Microsystems 2005). The Service layer consists of EJB Session Beans and the Domain layer contains EJB Entity beans. The Entity beans are made persistent using EJB Container Persistence to a Sybase relational database.

9.2.2 Common Services Architecture

The Common Services Architecture (CSA) provides development support with the following (Griggs 2005):

- Support components for development.
- Service components/modules.
- Build tools.
- Deployment and development support.
- Process information.
- Documentation - design, developer's guides.
- Application and environment setup.

When CSA is referred to, unless otherwise stated, the reference is to the component libraries it provides.

9.2.3 Incident Reporting Framework

The SolNet Incident Reporting Framework (part of the CSA) provides base classes for exceptions to inherit from and a reporting framework for instances of those subclasses. Figure 9.1 provides a simplified diagram of the exception classes. The idea is each application creates specific exceptions through the extension of BusinessException. In addition to creating these exceptions, developers also provide a XML file with information used in the production of error messages. Some exceptions that will be encountered come from frameworks such as the EJB components. An example of this is javax.ejb.FinderException. The Incident Reporting framework provides a wrapper class (ReportableFinderException) used to convert FinderException from a checked exception to an unchecked exception. This also allows FinderException to be reported through the framework. This is performed similarly for other common framework specific exceptions.

9.2.4 Business Object Framework

The Business Object Framework (BOF) provides base classes for inter-application interfaces. The base class of most importance is ServiceBaseBean used in the service layer of the application. The ServiceBaseBean is a standard EJB stateless Session

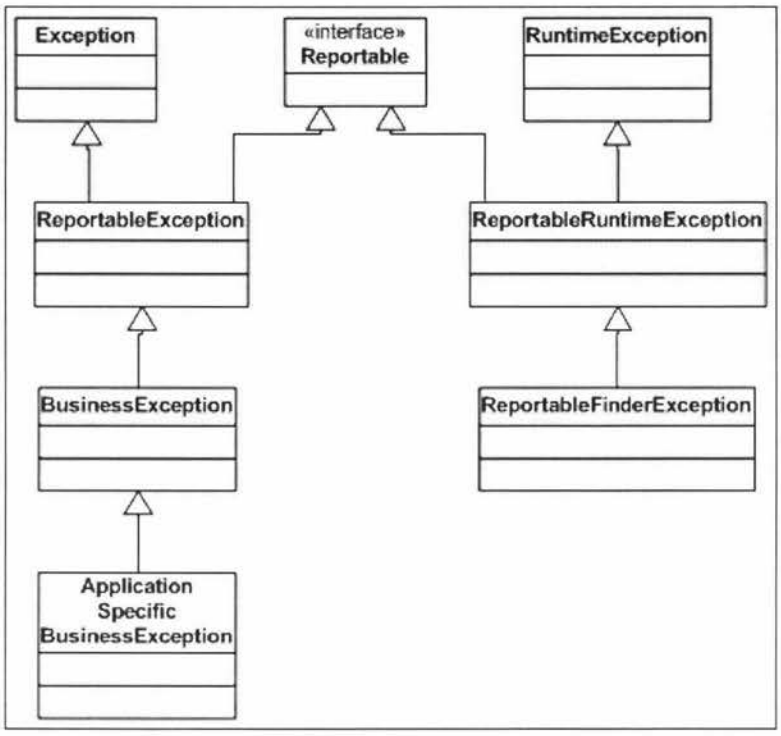

Figure 9.1: SolNet Incident Reporting Framework

Bean. However, this class eases the development tasks required to construct a Session Bean. As part of the base class there are methods called begin() and end(). These 'hook methods' allow services such as security and auditing to be easily performed. Developers need to make calls to these methods from each service method (i.e. public methods in Service Beans) they create.

9.2.5 Transaction Handling

EJB container transaction handling is extensively used by SolNet applications. Transactions are automatically rolled back by the EJB container if an exception is encountered that is a subclass of RuntimeException. However, checked exceptions (such as Business Exceptions) require the developer to notify the EJB container if the transaction needs to be rolled back. This is executed by getting a reference to the context and calling the setRollbackOnly() method.

9.3 NZQA - SPER Project

The New Zealand Qualifications Authority (NZQA) is a Crown Entity that was established to ensure the quality of and to coordinate national qualifications in New Zealand. NZQA is currently undertaking the eQA (Electronic Qualifications Authority) project in association with its development partner SolNet Solutions.

This is a project to replace legacy Information Technology (IT) applications with web centric solutions.

The SPER (Students, Processing, Entries, and Results) application allows NZQA to collect learner information (e.g. student enrolment and results) from providers. It also provides services such as record of learnings, awarding qualifications, and invoicing for services provided. SPER is the target system of this study. However, because of its dependencies on several other systems they will only be briefly mentioned. These other systems include eQA applications, NZQA legacy systems, and external systems. These systems have dependencies with SPER:

- Exams - The Exam's system facilitates NZQA's management of exams including which exams are available, where and when they will be sat, who is sitting them, the marker, and the materials used and their distribution. SPER makes information available to Exams for processing through direct database access.

- QUAL - The Qualification's system provides a single repository of information for all national qualifications and standards, as well as rules for qualification verification. It supports quality assurance and the registration of new/revised qualifications. SPER and QUAL communicate through the QUAL-SPER interface using RMI and Data Transfer Objects (DTOs).

- Contacts - The source of all provider and contact information. SPER accesses information from the Contact's database.

- Other NZQA systems used include finance and website systems. External systems such as school management systems and the National Student Index (NSI) maintained by the Ministry of Education (MoE) must also be interfaced with by SPER.

The relationships between the eQA applications as described above are shown as Figure 9.2[1]. There are many different forms of communication between the systems such as direct database access, manual batch files, and Java Remote Method Invocation (RMI). We are most concerned with the SPER External Interface (SXI) which presents access to SPER's functionality via RMI to other systems using the Facade pattern (Gamma et al. 1994).

SPER was selected because of its relative maturity and size when compared to some of the other SolNet projects. Initially the EOS (Educational Organisation System) was selected. However, this was a proof of concept (POC) project that was

[1]Figure reprinted SolNet Solutions/NZQA Design Documentation

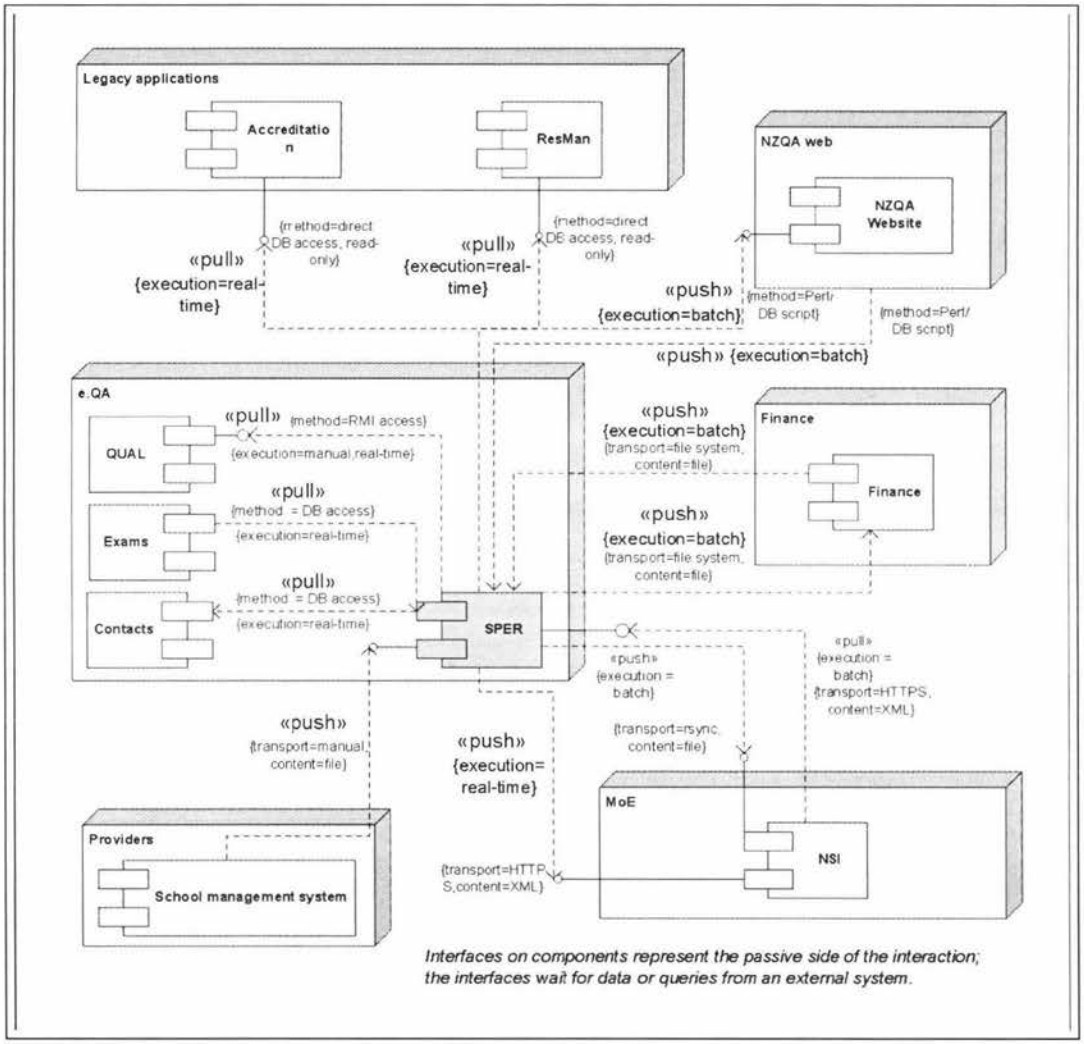

Figure 9.2: NZQA Applications

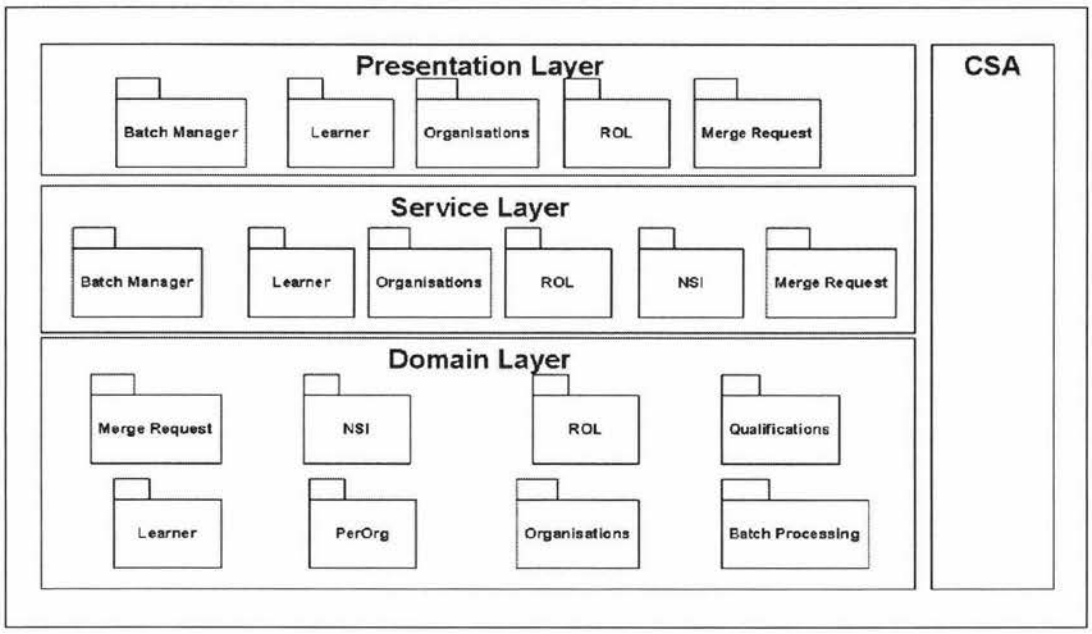

Figure 9.3: SPER Layered Architecture

not at a level of maturity where it could be used to reliably assess the impact of using aspects. In fact, it was initially used but because of the difficulty in reliably deploying the system, it wasn't feasible to continue. Moreover, it was still too small to give an indication as to the usefulness of aspects. EOS could be the first eQA system to be developed with aspects, pending approval. This will be discussed further in Section 9.4.

9.3.1 General Architecture

The SPER application is far from trivial. It is made up of approximately 115,000 lines of code spread across 1,022 classes. It is a J2EE application built using technologies such as EJBs and Struts in a layered fashion. The layered architecture is shown as Figure 9.3[2].

The SPER application is a Model View Controller (MVC) based web application implemented using the Struts framework. The presentation layer consists of Java Server Pages (JSPs), backed by Struts actions. The Service Layer consists of ServiceBeans (Session Beans) which allow use of common EJB services such as transactions. Most of the logic is delegated to Plain Old Java Objects (POJOs). The domain layer contains the Entity Beans used to manage persistent objects. The CSA provides core services such as auditing and security components which crosscut the application throughout the presentation, service, and domain layers. Factories are used to make naming service lookups between the various layers transparent.

[2]Figure reprinted SolNet Solutions/NZQA Design Documentation

Listing 9.1: General Service Bean method structure

```
// typical service method in a SolNet Service Bean
public void doService() {
    try {
        // start service wrapping
        begin();
        // perform business logic
    }
    catch(BusinessException e) {
        // transaction rollback
        getSessionContext().setRollbackOnly();
        throw e;
    }
    // exception handling
    catch(FinderException e) {
        throw new ReportableFinderException(e);
    }
    catch(...){}
    finally {
        // end service wrapping
        end();
    }
}
```

9.3.2 Identifying Potential Aspects

Due to the scale of the eQA applications there was a reliance on information from senior development staff at SolNet to identify an area of the system where attention could be focused on identifying aspects. It was decided to focus on the service layer as this was known to include code that could be potentially aspectised. By focusing on this layer the study was restricted to a much smaller code base from the full 115,000 lines of code available. Additionally it was considered to be an area that could bring benefits at lower risk, providing a natural path to have aspects integrated into future projects. Furthermore, it is generally recommended that aspects should be phased into use so developers can become familiar with the technology before applying it to critical and higher risk sections of code (Laddad 2003).

A code inspection process was used to identify similarities and differences in the code used for the classes in the service layer in order to identify aspects. It soon became very clear that there was a very common structure to the methods in the Service Beans. This is shown in Listing 9.1.

Three distinct services have been identified from this listing that are being performed in addition to the core logic of the bean. These are:

- Service Wrapping - This is the name used to describe the calls made to begin()

and end(). These methods perform common operations such as security checks and auditing. Potentially these services could be added into separate aspects, but at this stage the calls to these methods are made.

- Exception Handling - This involves capturing exceptions of certain types and converting them to another type.

- Transaction Handling - Checked exceptions must signal to the container that a rollback must occur. Runtime exceptions are automatically rolled back by the container. This normally involves capturing the checked exception, signalling the rollback, and re-throwing the exception.

In the next section the design of these services is explored and an argument formed as to whether they should be implemented as a single aspect, or separate aspects for each distinct service.

Having identified these potential aspects it was necessary to evaluate the degree to which the SPER project follows this structure. Service wrapping was the easiest of the three to evaluate as it was necessary to perform this for all public non-static methods. There was one exception to this in the project and this was when a public method delegated its work to another public method with different parameters. Furthermore, there is an alternative form of service wrapping that uses the begin(String comment) method which allows a custom audit comment to be made. This is rarely used except for external interfaces with other systems. However, exception handling and transaction handling were not as straight forward to evaluate. Exception handling presented numerous scenarios, however further analysis showed many of these could be refactored, allowing these options to be collapsed into fewer states. One common scenario highlighting the refactoring of code to collapse the number of options when dealing with a particular exception is:

- Catching a FinderException (checked exception), rolling back, and re throwing the exception. The caller then catches and re throws this exception as a ReportableFinderException (unchecked exception).

- Catching a FinderException, rolling back, and throwing a ReportableFinderException.

- Catching a FinderException and throwing a ReportableFinderException

An astute reader may notice that these can be made equivalent. The 2nd and 3rd choices are identical since the container automatically invokes a rollback for runtime

exceptions (ReportableFinderException). The first option can be refactored to one of the other two options by moving the responsibility for handling the conversion of the FinderException to a ReportableFinderException to the service method. This leads us to collapse all of these scenarios into the third option. This can be seen similarly with the CreateException/ReportableCreateException.

Most other cases of exception handling involve rolling back ReportableExceptions or their subclasses. However, there are a few remaining exceptions to these rules. There are two scenarios to be considered. The first is that developers have incorrectly used the incident reporting framework (which all business exceptions should subclass) or the method has some special exception handling logic that must be applied to it. It would seem that these two scenarios can explain the remaining differences. The worst of these scenarios are catch all exception handlers which rethrow a new runtime exception. This tends to be a definite sign of bad design (bad smell). In the following design these scenarios are taken into consideration, but we believe that aspects will force better design to avoid these issues on other projects.

The final question that must be resolved is whether there could be scenarios where checked exceptions are thrown and a transaction rollback should not occur. This has not identified, but a design may have to take this scenario into account.

A further area where a design should take account of variation in the use of these services is the service wrapping calls to begin(). There is a version of begin() that takes a String parameter used as a comment. This comment is used for custom audit comments and is used extensively in the SPER External Interface. This form can be used from any of the service bean methods, but is rarely used. However, we must allow it to be handled gracefully in the design.

9.3.3 Aspect Design and Implementation

In the previous section three key services have been identified that could be turned into aspects. In this section it is shown how an iterative approach has been applied to the design and implementation of the relevant aspects. Throughout this process adaptions have been made and problems identified that have needed to be overcome, many as a result of the complex environment that the aspects must be integrated with. These are practical issues that were not considered before this project commenced, but they needed to be resolved to ensure a reliable and low risk approach for future use of aspects in this commercial environment.

In the first iteration the variations of Service Methods in the base project have been ignored and a single aspect is used to implement all three services. This

approach highlighted some key problems. Firstly although the aspect is simple, the crosscutting concerns cannot evolve and change individually. The idea behind using aspects is to separate these concerns from the main code into localised unit. This is achieved to a certain degree, but it seems far better and more reusable if these concerns are separated. Moreover, the assumption is that these services are applied to the entire project in the same way. This is clearly incorrect as our analysis shows and this solution limits our ability to work around the differences in the various beans. What this approach has given us is the ability to simply test the use of aspects and see how they integrate with the build process. This is discussed further in Section 9.3.4. The potential benefit of using aspects is also seen since some sections of code have been refactored. In Listing 9.2 a method from a Service Bean is shown before the use of aspects, and then in Listing 9.3 the refactored version with aspects is shown. The refactored version is simpler. It is easy to identify what the method is doing without tediously going through auxiliary concerns that are framework related. The aspect to provide the extra services is similar to that shown in Listing 9.4. However, note this has been simplified for the purposes of illustrating this approach. There is a deliberate error with the use of checked exceptions which can be fixed, but for the purposes of this illustration it is ignored.

The second iteration saw the separation of the various services into an individual aspect for each service. Now that services are becoming more reusable base aspect approach is adopted. This means the logic is captured in high level aspects that can be reused across projects. The sub aspects are used to identify points in the current application where the services need to be applied. The situation is reflected in Figure 9.4. An example aspect for service wrapping is shown as Listing 9.5. Note the use of an abstract pointcut to allow easy specification of the methods where the calls need to be made. This can now be adapted for each application to take into account its requirements. Exception handling still suffers from the same problem as the last example when it is implemented using an around advice similar to Listing 9.4. However, because of the checked exceptions being thrown it requires that all methods the advice is applied to can handle these. This is not the case, and we know quite clearly that aspects will be dealing with these appropriately, unfortunately the compiler does not! To solve this the Exception Introduction Pattern is applied (Laddad 2003). The idea behind this pattern is to use the exception softening process to bypass the compiler errors, and then convert the exceptions back to their original checked exception before they reach the base code. This solution was less than desirable since it required substantial application specific code to perform the exception conversion process. This extra code depleted the benefits of using aspects

Listing 9.2: Enrolment Fees Method - Before Refactoring

```
public void recalculateEnrolmentFees(OID oid) throws
    LearnerException, OptimisticConcurrencyException {
     try {
         begin();
         // Start of business logic
         Enrolment enrolment = EjbUtil.getEnrolmentHome()
                                        .findByPrimaryKey(oid);
         enrolment.recalculateFees();
         // End of business logic
     }
     catch (FinderException e) {
         getSessionContext().setRollbackOnly();
         throw new ReportableFinderException(oid,
             EnrolmentServiceBean.class, e) ;
     }
     catch (LearnerException e) {
         getSessionContext().setRollbackOnly();
         throw e ;
     }
     catch (OptimisticConcurrencyException le) {
         getSessionContext().setRollbackOnly();
         throw le;
     }
     finally {
         end();
     }
}
```

Listing 9.3: Enrolment Fees Method - After Refactoring

```
public void recalculateEnrolmentFees(OID oid) throws
    LearnerException, OptimisticConcurrencyException {
     // Only contains business logic!
     Enrolment enrolment =
        EjbUtil.getEnrolmentHome().findByPrimaryKey(oid);
     enrolment.recalculateFees();
}
```

Listing 9.4: Single Aspect Approach

```
// Intercepts service method calls
Object around(ServiceBaseBean bean) : servicemethods() &&
    this(obj) {
     try {
          // service wrapping
          bean.begin();
          // execute service method
          proceed(bean);
     }
     // handle exceptions
     catch (OptimisticConcurrencyException e) {
          // rollback transaction
          bean.getSessionContext().setRollbackOnly();
          throw e;
     }
     catch(...) {}
     finally {
          // service wrapping
          bean.end();
     }
}
```

for exception handling so as a result a more appropriate solution was sought. It became clear that a very simple solution was the use of the after throwing advice instead of an around advice combined with the Exception Introduction Pattern.

Iteration three saw a restructure of the application to reduce the number of pointcuts being rewritten in each of the sub aspects. To solve this problem the approach used by Griswold, Sullivan, Song, Shonle, Tewari, Cai & Rajan (2006) to add a layer between the base system and the aspects was applied. This approach is called Crosscutting Programming Interfaces (XPI) and is designed to reduce the

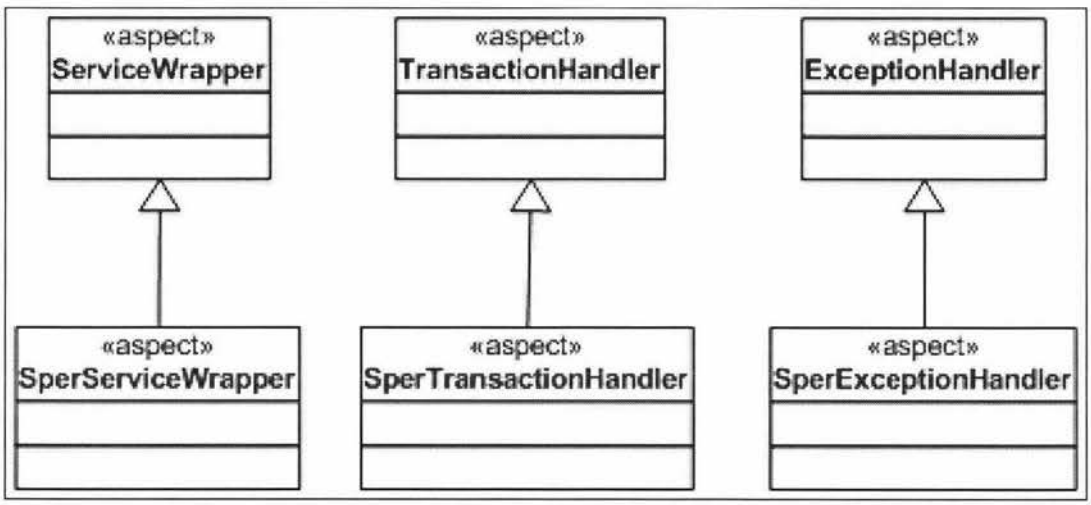

Figure 9.4: Basic Aspect Diagram

Listing 9.5: Base Service Wrapper Aspect

```
// Base aspect to make calls to begin() and
// end() for Service Bean methods
public privileged aspect ServiceWrapper {
    // sub aspects define service methods
    public abstract pointcut serviceMethods();

    // make calls to begin
    before(ServiceBaseBean bean) : serviceMethods()
                    && this(bean) {
        bean.begin();
    }

    // make calls to end
    after(ServiceBaseBean bean) : serviceMethods()
                        && this(bean) {
            bean.end();
    }
}
```

coupling between the aspects and base code. By taking this approach it is possible to have changes between the aspects and the base code localised to one aspect where all the necessary pointcuts are specified. It was also noticed that the aspects were not being applied in the correct order (through byte code decompilation) so an aspect was added to control the weaving order. This ensures that service wrapping is applied first, followed by exception handling, and finally transaction rollbacks.

At this stage there are well modularised concerns and a reasonable amount of flexibility. It is time to consider how variations in the Service Methods can be dealt with. The first of these is the use of begin(String comment) instead of begin() for service wrapping. This is only used a small number of times in the SPER application, but it must be allowed for when using aspects. Analysis showed that the comments can be unstructured and may use contextual information. No occurrences were found that referred to local variables. It was also found that all of these instances must still make the call to the end() method. The solution to this problem was to introduce a pointcut to identify the methods that must use the begin(String comment) method. This allows us to exclude these methods from the call to begin(), but still make the calls to end() with no modifications. The excluded methods themselves have a piece of advice added to the sub aspect to implement the call to begin() with the necessary comment. An example aspect is shown as Listing 9.6. Note that thisJoinPoint is used to access parameter information needed as part of the comment.

So far the aspects used have been privileged aspects. This means the aspects can access the private and protected methods and fields of a class despite the Java

Listing 9.6: begin(String comment) Aspect Example

```
before(ServiceBaseBean bean) : excludedmethod1() && this(bean) {
    // get a parameter from the method which is
    // included as part of the audit comment
    String nameParam = thisJoinPoint.getArguments(1);
    bean.begin("Random Comment: " + nameParam);
}
```

access protocols. This is achieved by the automatic weaving of the necessary public accessor and mutator methods to allow the aspect access. Advantage has been taken of this so far in our approach because certain methods of the ServiceBaseBean are declared protected (e.g. begin()) that need to be called from the aspects. This approach is acceptable, but faces challenges in the SolNet build process. The SolNet build process involves making multiple copies of the libraries used by an application and placing them into different folders depending on their deployment location (e.g. WAR). Unfortunately, only one copy of the JAR file gets woven by the AspectJ compiler. This could result in the system not functioning correctly because the necessary public accessors and mutators are not available in the particular deployment environment (e.g. Tomcat and EAServer have their own copy of the JAR file). There are options to solve this problem. The first is to change the build process so that only one copy of the JAR file is used, which is woven and then deployed to all the correct places. This is not a trivial exercise and is outside the project scope and has little benefit to SolNet as they are currently making changes to their build process. The second option is to make the necessary methods public so they can be freely accessed. This is not desirable since it breaks encapsulation. However, it is a simple approach and with the use of an enforcement aspects[3] to ensure that only authorised accesses to the object are made, this could be a satisfactory solution. This is likely to be the approach used by SolNet if aspects are used in future projects. It was not used in this project since it required a new release of the CSA components which was complicated by versioning issues. The final solution and one used for this project was to move the aspects into the same package as the CSA components they access. This works since access is only needed to protected, not private methods, and each aspect only accesses classes from one package. If this was not the case then this could not be successful. This may not be desirable since it more closely couples the classes and aspects.

[3]AspectJ allows the use of aspects to specify compiler errors and warnings using pointcuts. One possible use is to say it is an error to call the method begin() from outside the ServiceBaseBean hierarchy unless the call is from the ServiceWrapper aspect.

Listing 9.7: Service Wrapper Base Aspect

```
public abstract aspect ServiceWrapper {
  // method which allows subaspects to make calls to
  // begin(String) from a different package
  public void begin(ServiceBaseBean bean, String comment) {
    bean.begin(comment);
  }
  ...
}
```

Listing 9.8: Service Wrapper Sub Aspect

```
// Advice for a method that makes a comment to begin()
before(ServiceBaseBean bean) : excludedmethod1() && this(bean) {
    String nameParam = thisJoinPoint.getArguments(1);
    // delegate the call to the super aspect
    begin(bean, "Random Comment: " + nameParam);
}
```

The solution to the privileged aspect problem does require a slight redesign to allow the sub aspects to make the calls to begin(String) without being moved into the CSA packages (i.e. we would like SPER specific aspects to be located in SPER packages). For this problem a method is introduced in the base aspect which makes the call to the bean as in Listing 9.7 and Listing 9.8.

The final issue is the SPER External Interface (SXI). This allows other systems to access the SPER system through RMI. Because information is being passed between different virtual machines some special handling is required for implementing the concerns. The differences are:

- Exceptions are from a different hierarchy than those for the rest of the SPER application. This is because nested exceptions cannot be reliably passed from one JVM to another. Instead a special exception hierarchy is used that converts nested exceptions so they can be passed as a string within the exception. This implies that a special aspect will be required to handle exceptions so they are converted to the correct types instead of using the standard exception conversion process.

- Auditing is normally performed through the use of a call to begin() in service methods. However, for the external interface information about the user accessing the service is not available, so this must be passed as a string to the external methods. This is then used to make an audit comment with the begin(String comment) method. This is also best performed using a separate

aspect as it affects a large number of methods and is performed in a consistent manner meaning it can be simply turned into an aspect. This could potentially be moved into the SPER ServiceWrapping aspect, but it is felt separating it is better for modularity.

- Transaction rollbacks are normally triggered by ReportableExceptions from the standard exception hierarchy. Since the exceptions are from another hierarchy, custom code must be written for this purpose. Once again this is designed as a separate aspect.

We believe that the requirements of the external interface can be appropriately dealt with using separate aspects for each of the concerns. This highlights how classes that vary from the standard behaviour can use custom aspects in place of the standard behaviour.

In this section the iterative approach used to produce a set of reusable library aspects that meet the requirements of the SPER application and other SolNet applications of a similar nature has been discussed. See Appendix C for full listings of the final aspects developed.

9.3.4 Integrating Aspects into the Build Process

SolNet uses ANT to build and deploy its applications. Each project follows a similar organisational structure and has a configuration file that allows various build parameters to be set. These parameters allow the set of build scripts to change their behaviour according to differences in projects. The build scripts are complicated and took significant time to understand where the changes required to include aspects in the build scripts should be made.

Due to the complexity of the scripts it was decided to replace the standard javac task with the corresponding AspectJ compiler task iajc. This required few changes to the rest of the scripts. A better solution would be to allow a flag to be set depending on whether the AspectJ or Java compiler should be used. An alternative could have been to use AspectJ for a post compilation phase or load time weaving. This is not a viable option since AspectJ's exception softening features are used when incorporating exception handling into aspects. Exception softening results in checked exceptions being wrapped with a runtime exception (org.aspectj.SoftException) so the standard Java exception checking does not produce errors when aspects are handling an exception unknown to the base program. It would be good if this problem could be solved in the future, as load time weaving is an ideal solution that

Listing 9.9: SPER ANT aspect properties

```
<!-- aspectj properties -->
<property name="noweave" value="false" />
<property name="showweaveinfo" value="false" />

<!-- List jar files that need to be woven -->
<path id="inpath.path"> ... </path>

<!-- List the jar files that contain binary aspects -->
<path id="aspect.path">
  <pathelement
     path="lib/ejbjar/SolnetAspects-20051012-032151.jar" />
</path>
```

would give SolNet far more flexibility when deploying applications (e.g. adding and removing services such as logging).

The AspectJ task added to the build scripts can be configured through the use of several ANT variables that have been included for this purpose. This allows specification of which JAR files contain library aspects, which JAR files need to be woven, and where the source files are located. An example of the options is shown in Listing 9.9 and the iajc task in Listing 9.10.

It should also be noted that SolNet is developing a new set of build scripts that contain many enhancements on the current scripts. These were not examined, however it is known that the compilation process has been considerably changed from the version we have applied aspects to. The new version uses a custom compilation task rather than the standard javac task. It is currently unknown how aspects will be incorporated into these scripts, but there may be a simple solution since this custom task is based on the Eclipse Java Development Tools (JDT) compiler as is AspectJ.

9.3.5 Project Testing

SolNet Solutions invest considerable time and money in developing CSA components which may be reused across many projects in order to reduce development time, cost, and risk for their clients. Before a component can be accepted into the CSA it must meet some stringent requirements. The process for developing a new component for the CSA is:

- Planning Stage - Requirements capture, impact assessment, prioritisation, and road mapping.

Listing 9.10: AspectJ compilation task

```
<!-- Import the AspectJ ANT tasks -->
<taskdef resource=
  "org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties"/>

<!--    Setup the incremental Aspect compiler with the
  options specified in the application specific
  variables -->
<presetdef name="core_compile" description="Compile Source">
  <iajc    debug="${javacdebug}"
    deprecation="false"
    showWeaveInfo="${showweaveinfo}"
    source="${javacsrcversion}"
    Xnoweave="${noweave}"
    inpathref="inpath.path"
    aspectpathref="aspect.path" />

</presetdef>

<!-- Macro to run the compilation task
<macrodef name="core_compile_macro">
  <attribute name="sourceroots"/>
  <attribute name="destdir"/>
  <element name="classpath2use"/>
  <sequential>
    <mkdir dir="@{destdir}"/>
    <core_compile sourceroots="@{sourceroots}"
       destdir="@{destdir}">
      <classpath2use/>
    </core_compile>
  </sequential>
</macrodef>
```

- Design - Designed by Common Services Team (CST) in conjunction with other project teams and clients (if applicable). The design is reviewed by CST architects, possibly in conjunction with client architects.

- Implementation - Developed by CST developers. Goes through a code review for new/high risk components.

- Testing - Full JUnit testing (or similar) in conjunction with test coverage checks. Metrics and QA tools (e.g. Agitator) should also be applied. Performance testing may also be necessary. Note: code coverage and metrics are new processes that are still being introduced.

- Release - Documentation, developer education, publish to interested parties (e.g. clients).

- Changes - Must go through a change request procedure for approval.

Several aspects have been developed using the SPER project, with the ultimate aim of reusing them across SolNet projects as part of the CSA. For these components to be accepted they will need to go through the process above. Of utmost importance is the need to have a test suite associated with the aspect component. Unfortunately, aspects present several challenges when it comes to unit testing as explained in Chapter 6.

Four techniques have been used to verify the aspects' behaviour: AJDT, byte code decompilation, unit testing, and scenario testing (limited integration).

AJDT

Chapter 6 described how the AJDT could be used to verify that aspects are being applied in the correct places through manual checking using the cross references view. This approach is used to verify that all the aspects are being applied to the correct places. Although there were a large number of matches to individual methods across the sixteen classes, this was found to be a manageable task. In fact, several errors were found using this technique (e.g. one of the classes was in a different package structure and was inadvertently missed from the pointcut expressions).

Byte Code Decompilation

AJDT verification did not make it immediately obvious as to whether the order in which the aspects were been woven was correct. Furthermore, just because the

weaving was occurring in the development environment, we wanted to be sure that the classes that were actually deployed were correctly woven. For this purpose the DJ Java Decompiler was applied. The class file is input and DJ produces the corresponding source code. From this it is easy to verify that the weaving was correct, although it is not the clearest Java code. An example from the SPER project is shown as Listing B.1 (Appendix B due to size constraints). Notice that the exception handling code is difficult to follow, but it is clear the concerns have all been woven in and are correctly ordered. Note: This listing is simplified due to its complexity and length.

Unit Testing

JUnit was considered for testing of the base aspects. This would have shown how library aspects could be verified as correct, before using them. However, the easiest technique when using AspectJ aspects is to test these in classes by delegating the logic. This was not possible with some of the aspects since they were trivial (hence testing wasn't critical) and there was little to delegate. Furthermore, as explained earlier, most of these aspects were developed as being privileged. If logic was delegated then some of the methods and fields necessary to implement the concern could not be accessed (such as calling begin()). Delegation could be used if the aspects were refactored into the same package as the classes they were operating on, but there were still several framework issues that made testing more difficult than usual. This was due to the need to instantiate the SolNet framework in the appropriate manner. Furthermore, testing must be performed with dummy beans in an EJB container. These problems are not limited to aspect testing, but did make the process more difficult when there was limited verification value (i.e. the aspects could be inspected to find errors that most unit tests would detect).

The aUnit framework was also trialled. There were several problems with the base aspects being abstract, therefore it was necessary to provide a dummy sub aspect for testing purposes. It was also found that some of the AspectJ features used were not supported by the aUnit framework (e.g. after throwing advice). An attempt was made to modify the framework to support some of the features required but this resulted in more problems. An example of a potential aUnit aspect is shown in Listing B.2 (Appendix B due to size constraints). Of further note was aUnit did not have support for passing and failing tests, rather it could only print out results of executing join points. This framework cannot easily be used in an automated fashion. aUnit is currently too complex and immature for practical use. aUnit is

expected to be much improved by a scheduled release for early 2006.

Scenario Testing

This was originally intended to be an integration test using the various test cases available in the SPER packages. However, the test cases were out of date for the SPER version being refactored. Because it was not feasible to update all the test cases it was decided to update one test case to allow testing of a bean's service method. This was used to ensure that the correct results were returned, including an exception scenario. This testing is limited, but it did verify that for a service method the aspects had been applied correctly. This could not verify if the refactoring was correct across all the service methods. Unfortunately, errors could have been introduced that would not be discovered by these tests. This does not reflect the use of aspects, but rather the refactoring of a large number of methods not designed to use these aspects. Some inconsistent sections of code required modification that could have unintentionally been performed incorrectly to account for the new policies. Despite this, it can be argued that had the original code been written with aspects in mind, these scenarios would not have been an issue. It is these problems that aspects should help resolve.

A further stage of testing involved a scenario through the web interface. The same steps were repeated using both the original and refactored versions to ensure they both produced the same results, including error messages resulting from exceptions and rollbacks. For the services used, it was found that the aspect version behaved correctly. Logging was used for verifying elements of these scenarios.

Although the testing regime was not as thorough as originally planned, it did provide every indication that the aspects were functioning as expected.

9.3.6 Metrics

The refactored version of SPER and the original version were compared using a selection of metrics. In particular McCabes Cyclomatic Complexity (MCC), Number of Statements (NOS), and Halstead Effort (HE) are used. There are some assumptions made in using these measures as detailed in Chapter 7, but we believe them to be fair. Table 9.1 show the results before and after applying aspects to the service beans in the SPER application. These will be referred to throughout this section.

Class	CC - Original	CC - Aspects	NOS - Original	NOS - Aspects	HE - Original	HE - Aspects
ProviderServiceBean	14	14	76	34	1495.52	267.64
TopScholarServiceBean	2	1	14	9	315.48	8
BatchManagerBean	31	20	137	51	11556.3	1560.27
CompassionateServiceBean	3	1	14	7	1087.36	59.4
EnrolmentServiceBean	15	9	64	39	12508.8	7691.49
EntryServiceBean	36	32	206	171	44995.9	36890.3
FinanceServiceBean	51	26	185	68	15560.7	2606.24
LearnerServiceBean	89	61	405	165	39519.3	12778.9
MergeRequestServiceBean	8	4	37	16	2517.28	273.71
NsiServiceBean	5	2	25	13	880.68	20
PfuServiceBean	2	1	11	6	386.46	12
ProcessorBean	49	39	210	124	25488	15091.8
QualificationServiceBean	18	10	76	30	3298.82	183.81
ReconsiderationServiceBean	13	4	42	14	4525.09	52.24
ReportServiceBean	10	5	43	18	3050.05	442.38
SXIServiceBean	177	113	750	522	253550	169694
Totals	**523**	**342**	**2295**	**1287**	**420736**	**247632**

Table 9.1: SPER Metrics

Project Complexity

It is assumed that since the aspects have been developed as CSA style components their value is gained for free. This allows us to analyse the base program free of aspects. This results in large savings in complexity from the point of view of the developer since this complexity is hidden. Measures such as McCabes Cyclomatic Complexity do not take into account aspects. Cyclomatic Complexity looks at the number of paths through a program's execution. For SPER this is greatly reduced from the programmer's perspective since tasks such as exception handling have been removed and placed into aspects. This means the path still exists but is now hidden. It is acknowledged that aspects will introduce new couplings between aspects and different control flows that don't exist in the base program, but since the analysis is from a developer's point of view this does not need to be considered. This complexity has been moved away from the developer and into the compiler. The Cyclomatic Complexity has been reduced by 35%. This indicates the program should be easier to understand, maintain, and less prone to bugs.

The Halstead Effort provides a measure of the time it takes to understand a system. Like Cyclomatic Complexity it is also a complexity measure, but it is based on the operators and operands in the module. The Halstead Effort was reduced by approximately 41%. The approximate time to understand a piece of code can be found by dividing the Halstead Effort by 18 (Virtual Machinery 2005). This is a time saving of 2.7 hours for the SPER Service Layer. This should make adding new developers an easier task, as well as reducing the time required by existing developers for developing and maintaining the code.

These readings provide quantitative evidence to back our observation that the final system has been greatly reduced in complexity by using aspects. In fact, most of the service methods became trivial using aspects. This is testament to the methods being simple before adding aspects, but the simplicity was hidden in the need for developers to make calls to allow security, transactions, exception handling, and logging. This code was not complex, but it did have a bad smell to it since so much code was copied and pasted, yet using standards OO techniques this is necessary to make the system function correctly.

Effort Reduction

The NOS is similar to a LOC measure, but it has been normalised so that artifacts such as white space and comments do not influence the result obtained. NOS is used to give a measure of the amount of effort that is required to develop a piece of

code. A significant reduction in the NOS required to implement the service beans using aspects instead of traditional OO technology is shown. The sixteen classes that were refactored consisted of 2295 statements before aspects were used. After refactoring they were reduced to 1287 statements. This represents a 44% saving in boiler plate code. However, there was additional code that was transferred into aspects. Some of this code is application specific and some is designed for reuse. This aspect code is incompatible with the JHawk metrics tool so the NOS manually is estimated manually. It is estimated there are 140 statements in aspects, of which approximately 50% are reusable. This leaves a total code saving of 38%. This level of code saving is thought to be achievable for classes used in this context (e.g. Session Beans from other eQA projects have similar structure). This is not representative of the code saving possible across the entire project, which would be considerably less. A result more in line with our expectations was reported by Zhang & Jacobsen (2003) who experienced a 9% code reduction when refactoring middleware systems.

Project Maintainability

Aspects make software more maintainable because concerns are better modularised. They also make the code more understandable and reduce its complexity and the effort required to comprehend a section of code. This is highlighted by a reduction in the total Cyclomatic Complexity of the refactored classes by 35%. Most of the methods were trivial once aspects were applied making them easier to understand and less prone to bugs. This is further backed by a 41% reduction in the Halstead Effort. These reductions would also indicate that it would be easier and quicker for new developers to join a project as the resulting systems are far easier to comprehend. However, these measurements do not consider any added complexity in understanding the aspects, but we believe this is still valid since aspects can be written by specialists in an area and do not need to be maintained by all developers. Moreover, the resulting system may have the same complexity as the original system, but the reduction in complexity is realised by moving tasks into the compiler/weaver instead of being performed by the developer.

Consistency

Aspects give the opportunity to ensure certain practices are consistently applied throughout a system. This not only increases maintainability since it is easy to make system wide changes to these policies, but it also reduces the time to code, provides less opportunity for bugs, and makes the system easier to understand. Although

there is not a metric for showing this, an example earlier shows how Exception Handling was performed in an ad hoc manner depending on the developer. Using aspects this is made consistent, making the application easier to understand. When the EOS project is discussed another example of this type of practice with logging will be provided.

Performance Assessment

One question that must be addressed when using AspectJ is what is the performance cost of using aspects instead of coding the logic directly. Ideally the performance of an aspect application will be identical or better than its equivalent. In this section the impact on build time and memory consumption is considered, as well as the runtime performance impact. The runtime performance is the element we are most concerned about, but it also the more difficult of the two to assess as it is more prone to external factors influencing results (e.g. database slowdown, caching, other machine users/processes, and network load). Further isolation would be required to verify the results obtained. Aspects could have been used to make these measurements, but we have doubts about the use of aspects to benchmark aspect performance.

Build Impact

The incorporation of aspects into the build process has previously been discussed. However, no assessments of the impact on the build process in terms of time and memory required for an aspect build versus a normal Java application build have been made. For the purposes of this assessment both clean compiles (when previous compiled classes are removed) and compiles where the previous compile is retained are examined. AspectJ has an option to display information about the weaving taking place. This option was found to delay the build, probably due to screen I/O, therefore we have performed builds with this option turned off.

The first test was conducted using the SolNet ANT tool to execute a clean compile (sant clean compile). The total build time was recorded and this process repeated five times. This test allowed the ANT JVM 512MB of memory using the ANT_OPTS=-Xmx512m environment variable. The AspectJ version produced a median time of thirty three seconds and the Java version a median of twenty nine seconds. This shows a negligible affect on build time using ANT. However, it should also be considered that aspects are only being applied to a small portion of the entire application. Further testing would be required to see how substantially this

changes with aspects that crosscut larger numbers of classes and with more aspects.

The second test was operated under the same conditions, but instead of conducting a clean, the existing classes were simply recompiled. This highlights a well known deficiency with AspectJ that it must rebuild the entire project every time rather than only updating modified classes. The median for AspectJ was thirty two seconds and Java eight seconds. The AspectJ compiler does have an incremental option that allows compiling to only be performed when it is needed. It is not clear how this would fit into SolNet's build process since it requires the task to stay active and the user to push a key each time they want to recompile. Perhaps this could be a separate task to allow developers quick builds when deploys to the application server are not also required.

The third test evaluates the memory consumption of the AspectJ compiler. It was clear it was using more memory as the default ANT settings caused Out of Memory errors when using AspectJ. In this test standard memory options were used to find the minimum memory that was required when building this project for both the Java and AspectJ versions. The windows process monitor was used to check the actual memory usage. The AspectJ version required that the -Xmx option allowed 128MB of memory. The process was found to consume 161MB. On the other hand the Java version required the -Xmx option to be 64MB and the process consumed 87MB. This highlights the fact that AspectJ uses almost twice as much memory for builds than the normal javac compiler. For most modern machines this should be manageable and the standard SolNet environment typically uses at least 1GB of RAM for development machines. Other areas of the build process often use more RAM than what this compilation process was consuming, such as the generation of skeletons for the EJBs. Therefore, this should not be a significant issue.

Runtime Impact

The AspectJ Development Manual states that performance is targeted to be at least as good as a normal Java application. Anything less should be considered a bug (Eclipse Foundation 2005). We were interested to see what overhead, if any, the aspects applied to a service method under two scenarios. The first was a sunny day scenario and the second is when a transaction requires a rollback due to an exception being thrown. When considering benchmarks such as (Vasseur 2004) we believe there should be a small overhead encountered in the first scenario, and a larger overhead when dealing with exceptions in the second scenario. The impact of aspects on memory consumption was not considered.

Results for the first test were based around a simple query for an item 'Smith' from the Learner Service using one hundred queries with the total time averaged. This was repeated ten times and the best and worst times were removed as being outliers. The final result produced an average time of 79ms for the non-aspect version and 96ms for the aspect version. However, we also repeated this experiment with 10000 queries once for each system. This produced results of 85ms and 95ms. We believe both of these results show that the aspects have introduced minimal overhead, which is negligible. These results are influenced by many factors such as machine load, database server, and probably caching. We believe the repetition and relative comparisons between the systems reduces the risk of this affecting the results.

Another example was used that results in the system signalling an exception (InvalidProviderCodeException). This test fails if the exception is not detected. Otherwise, it is repeated one hundred times. It is expected that the aspect version will be slower. However, our results for each repetition were identical or within 1ms of each other. This difference cannot be explained and further testing is required as this is in conflict with previous benchmarks. Moreover, all advice from the previous example should execute in addition to the extra exception handling code meaning at least the same overhead is expected. Byte code inspection was used to ensure the correct code was deployed and logging used to verify the paths taken.

9.4 NZQA Project - EOS

The EOS (Education Organisation Systems) project aims to improve the way business processes are managed. This project was in a proof of concept (POC) stage when it was examined. It was the original system that we were to target refactoring because it was a smaller and a more manageable project. However, it was also in a state of flux and too immature to effectively evaluate aspects on. Despite this, in this section the work completed on this project before moving to SPER is presented. In particular the differences in the project approaches are highlighted and how they affect the aspects developed is explained. Metric data showing the improvement in the EOS project by applying aspects in terms of effort rather than complexity measures is also presented. Because the code shows many similarities to the SPER project, similar results are achieved when evaluating the effort reduction. It is expected that complexity would follow a similar pattern because the service layer is also being refactored in EOS.

The EOS project takes a different approach to many of the existing eQA projects

Listing 9.11: EOS Service Method

```
// typical EOS service method
public void sendEmail(String fromEmailAddress, String
    toEmailAddress,String status) throws ServiceFailedException {
     // logging start
     if(logger.isDebugEnabled()) logger.debug("sendEmail() -
        start");

     try {
       // service wrapping
       begin();
       // business logic
       NotificationApplicationService.getNewInstance().sendEmail(
           fromEmailAddress, toEmailAddress, status);
       // logging end
       if(logger.isDebugEnabled()) logger.debug("sendEmail() -
          end");
     }
     // exception handling
     catch(Exception e) {
         // log error - notice developer mistake!
         if(logger.isDebugEnabled()) logger.debug("sendEmail() -
            start");
         // convert to ServiceFailedException
         throw convertException(e);
     }
     finally {
       // finish service wrapping
       end();
     }
}
```

in order to evaluate new technologies and approaches. For example it is built around a Service Oriented Architecture (SOA) using web services. The web services are still backed by beans similar to those used in the other projects. However, the session beans do not extend directly from the SolNet framework base classes. Instead they introduce a Session Facade class which provides services such as an exception converter. The exception converter is used by the beans to change the checked exceptions thrown in the service methods to a runtime exception called ServiceFailedException. This is an important difference from the SPER project where checked business exceptions are used. Moreover, it also means that transaction handling is no longer required since this will be handled by the container whenever the runtime exceptions are encountered. One final difference is the extensive use of logging within the service methods.

In Listing 9.11 a typical service method in the EOS beans is shown. Note that

although it looks similar to the SPER project, there are enough differences to require a slight redesign in some aspects, as well as some new aspects. The services tangled with the business logic are logging, service wrapping, and exception handling. Service wrapping can be performed using the same base aspects as that applied in the SPER project. Exception handling can make use of the base aspect applied to the SXI as this allows custom exception handling to be performed. Finally, we require a new aspect for logging. We make use of an aspect with some modifications presented on the AspectJ mailing list. The logging aspect is reusable across many projects. All these aspects use base aspects and some small amounts of code to connect them with the EOS project. For full code listings refer to Appendix C.

One important benefit highlighted by the EOS project was how copy and paste coding can result in errors being propagated throughout the system. Although a relatively minor error, one example was that an incorrect logging message was propagated through the methods of a couple of the beans. This error could have been crucial had that information been required for diagnosing an error in a deployed system. Furthermore, being a non-functional error it would have been more difficult to detect and easily overlooked. Aspects make this type of error less likely to occur, and easier to correct should it happen as only one update is needed in one place in the code base.

We have found that aspects saved approximately eighteen lines of code in every service method. When aspects are excluded from the equation a 43% saving in LOC was achieved. When aspect code specific to EOS is included that drops to 27%. This is still a significant saving and both these results are similar to those achieved in the SPER project showing these results are achievable across SolNet projects, even when there are project specific differences to account for.

9.5 Benefits and Tradeoffs

Aspects can produce many benefits to the Service layer of SolNet projects. In this section the major benefits are presented, as well as considering some of the drawbacks of using aspects.

9.5.1 Benefits

Aspects have substantial benefits in terms of reducing the complexity and effort required to develop the code. This was a result of the code being simpler due to performing only one task, not several infrastructure tasks in addition to their

core logic. This reduction in developer burden is a result of framework code being removed from developer responsibility and moved to the aspect compiler. It was shown that it is possible to enforce a consistent policy for tasks such as exception handling, logging, and transaction rollbacks. Not only are these consistent, they can be easily maintained in one body of code. Overall aspects can reduce the development time, reduce the bugs in the code, make the code more maintainable, and more amenable to changes in requirements.

9.5.2 Tradeoffs

Like all technologies aspects have their risks and associated problems. This project has highlighted testing as the most substantial area where work is required to produce a practical solution. Testing theory is still developing and it is not helpful if it's too difficult and cumbersome to be applied in a commercial environment. Outside of this, we see limited IDE support and transparency to developers as minor issues. Changes to the build process may require some work, but this is transparent to most developers who still will deploy applications as they currently do. Overall these drawbacks are well compensated for by the benefits obtained when using aspects.

9.6 Aspects Future at SolNet

Following this evaluation of aspects, many of the senior development staff at SolNet are eager to start using aspects in the development of their projects. At the time of writing, SolNet was in the process of presenting a proposal to NZQA to start using aspects in the EOS project. It would appear that aspects have a positive future at SolNet and really do promise to reduce complexity and improve the development process. Obviously we have targeted a very specific problem in the SolNet code base, so finding other areas where SolNet can benefit from using aspects is paramount to their success. Two potential areas that have been proposed (not officially) are:

- Persistency - The idea would be to develop POJOs for domain objects and then use an aspect framework to control the persistency of the objects. Currently it is difficult to change between persistency frameworks, so once one is selected it must be persevered with unless numerous changes are to be made. A persistency framework using aspects should allow a less complex way to switch between frameworks such as EJB Persistency and Hibernate.

- POJOs - Currently SolNet makes extensive use of Session and Entity beans as part of their CSA development framework. Aspects could be applied to hide the underlying framework such as EJBs. Developers can write POJOs and then the aspects are used to make the necessary connections with the framework, such as making classes extend the required session base beans. The advantage is that developers can write basic objects without concerning themselves with the underlying implementation details of the framework. This is closely related to the first option as this also involves a move to POJOs and using aspects to hide the underlying framework (but more specifically persistency).

Both of these areas will present challenges that are currently unknown without a full assessment. These are certainly two areas that could greatly reduce the complexity of the SolNet framework and give more flexibility in meeting client requirements.

9.7 Summary

This chapter has presented the work accomplished to refactor a large real world system at SolNet Solutions. It has shown how aspects have been designed to remove an area of problematic code from the system so that it is well modularised, easy to maintain, and less complex than the original system.

Several practical problems have been identified when testing these aspects which cannot be easily resolved using the techniques presented in Chapter 6. These problems were a combination of the deployment environment, SolNet Infrastructure classes, and AspectJ language. These aspects are difficult to unit test since they do not have logic that can be delegated to other classes, but in most cases they were trivial making unit tests less important. Instead, a manual verification process consisting of use of the AJDT cross references view, byte code decompilation, and finally some integration tests using JUnit were used. This is an area where further work is required. Performance testing has shown that aspects add negligible overhead. Further work is needed to verify the exception handling results as these conflict with published benchmarks (Vasseur 2004). Build time is significantly affected by the need to always perform full builds rather than only building classes that have changed since the last build. The full build time is similar to that achieved for the normal system.

Aspects have been shown to be applicable to other SolNet projects with similar

problems. In particular the EOS project is likely to start using aspects in the near future. Two areas have been identified where aspects could be extensively applied to make the SolNet CSA framework more flexible and free developers from many infrastructure issues.

Ultimately, this chapter has shown that aspects can be applied in a complex real world environment and produce compelling benefits and cost savings. They have decreased the complexity of the development framework for developers reducing the risk of development errors and making it more consistent. This was the major hypothesis of this thesis.

In the remaining chapter the conclusions and areas for future work are presented.

CHAPTER 10
CONCLUSION

10.1 Introduction

This chapter provides a summary of the findings of this thesis in relation to the project goals outlined in Chapter 1. It concludes with recommendations for future work.

10.2 Summary of Findings

It has been shown using the refactoring of two SolNet Solution's projects that aspects can reduce the complexity of software. This is important as the high complexity of software is often cited as one of the major contributors to software project failures. The refactored SPER project had complexity reduction in the Service Layer of 35% and 41% as measured using the McCabes Cyclomatic Complexity and Halstead Effort respectively. Furthermore, both the refactoring of the SolNet SPER and EOS project's Service Layers showed significant size reductions in excess of 30% by applying aspects. This reduces the effort required to both develop the software and for future maintenance.

While refactoring the two SolNet projects a small reusable aspect library was developed. By using this library to implement infrastructural concerns it was shown that developers can be sheltered from the complexity of the CSA. Furthermore, removing these infrastructural concerns from the developer's responsibility helps to ensure that the CSA is applied consistently throughout projects. This increases understandability, reduces risk of introducing errors, and makes the software more maintainable. Finally, it has been shown that there are other areas where SolNet could benefit from aspects.

It was not possible to measure and quantify the true commercial benefits to SolNet Solutions in adopting aspects from this project. There are several reasons for this. Firstly, SolNet did not have historical data available that could be used in providing a baseline for comparisons. Secondly, the size of the projects under-

taken by SolNet was prohibitive to the refactoring of an entire project with the resources available, hence only a small portion of the projects considered were refactored. Thirdly, the code refactored has been significantly improved through the use of aspects, however, it is not expected that the results obtained are achievable throughout the system. Therefore, to quantify the benefits solely off this code would be misleading. The benefits to this section of code have been quantified in terms of effort reduction. With appropriate data this could have been used to determine the cost savings in implementing this code using aspects.

The integration of aspects into the SDLC has been considered over many phases. It was found that outside of the implementation phase aspects are still maturing and further research is being conducted. Despite this, there is enough maturity in most areas for aspects to be considered commercially ready. In the early phases there are many specification languages that can be used. Until one is standardised it will be unlikely that appropriate tools will be developed to support them. The design patterns and idioms being documented have proved useful in designing aspects for SolNet. For implementation there are many options for choosing a development framework. Any of the four major frameworks could be applied depending on the business requirements. The criteria provided in Chapter 3 will be useful in making this choice. Testing provides challenges, but there are methods that can be used until more appropriate tools and frameworks are developed. Tool support is established in most areas, but further work is required to increase the breadth and quality of the tools. A standard for AOP implementations would be useful to reduce needless differences between the frameworks. Ultimately, this would be best realised as a JSR providing Java Virtual Machine support.

Overall, it was decided that aspects can and should be adopted in commercial projects.

10.3 Applicability of Results to other Environments

The results obtained in this study have been predominantly considered in the SolNet Solutions' context. This is because it is difficult to verify their applicability in other environments due to most J2EE software development being closed source commercial systems. However, two particular J2EE development frameworks have emerged in recent times that show the problems with development complexity faced by SolNet are common and that AOP is an appropriate solution. The two frameworks

are:

- EJB 3 (JSR 220) - This specification specifically aims to simplify J2EE development. This immediately confirms that other developers face similar problems to SolNet Solutions with the complexity and cumbersome nature of EJB development. As was discussed in Chapter 3, EJB 3 places greater emphasis on the use of POJOs and introduces interceptors which allow some basic aspect like activities to be performed. The addition of aspect like behaviour suggests that aspects provide a means to solve many of the problems currently faced by developers when using EJBs.

- Spring Framework - The Spring Framework has gained a lot of traction with its promise to simplify J2EE development. POJOs with dependency injection help to make development less complex and Spring provides an AOP framework to allow services to be transparently injected.

Both of these frameworks attempt to reduce programming complexity in the J2EE field and do so by moving towards POJOs with transparent injection of additional services. These frameworks were created to try and solve problems such as those faced by SolNet Solutions. This enables us to infer that the results obtained are applicable to other environments. It does not infer that all companies will get the same complexity and code reductions achieved by SolNet. However, it is thought that most companies will benefit from improved modularity, reduced complexity, and better productivity as a result of applying aspects.

10.4 Future Work

There are several key areas where future work is required:

- Testing - More work is required to ensure that aspects can be easily tested. In particular there is a need to help test pointcut correctness and unit testing of aspects. More investment in frameworks such as aUnit may provide the necessary solution.

- Design Languages - It is important that a design language is adopted, preferably an extension of UML. Until a profile is adopted by a group such as the OMG it will be difficult for CASE tools to be developed to aid the process, particularly diagramming and code generation.

- Benefit Evaluation - A study that uses similar project teams to design, develop, and maintain a commercial level application using AOP and a traditional approach should be undertaken to allow further evaluation of the benefits and risks of AOP than was possible in this study.

- Serialisability - More investigation is needed to establish when AspectJ changes the SerialUI of a class. This could be critical to enterprise applications which use technologies such as RMI.

- Standards - A JSR would be useful to establish JVM support for weaving of aspect applications. This would help move the focus from development of competing frameworks to the establishment of a commercial aspect component market.

10.5 Summary

This chapter has presented the findings of this project and made recommendations for future work. The remainder of this thesis contains a glossary of terms used, references, and appendices.

References

Aksit, M. (2001), Composition Filters, Available from: http://trese.cs.utwente.nl/oldhtml/composition_filters/ [15 December 2005].

Aldawud, O., Elrad, T. & Bader, A. (2001), A UML Profile for Aspect Oriented Modeling, *in* 'Proceedings of the OOPSLA 2001 Workshop on Aspect Oriented Programming'.

Aldawud, O., Elrad, T. & Bader, A. (2003), UML Profile for Aspect-Oriented Software Development, *in* 'Proceedings of the Third International Workshop on Aspect-Oriented Modeling held in conjunction with the International Conference on Aspect-Oriented Software Development'.

Alexander, R., Bieman, J. & Andrews, A. (2004), Towards the Systematic Testing of Aspect-Oriented Programs, Technical Report CS-4-105, Department of Computer Science, Colorado State University, Fort Collins, Colorado, USA.

Almaer, D. (2005), AOP: No more need for a standard in Java space, http://www.almaer.com/blog/archives/000660.html [16 May 2005].

Alur, D., Crupi, J. & Malks, D. (2001), *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall/Sun Microsystems Press.

AOP Alliance (2003), AOP Alliance (Java/J2EE AOP Standards), Available from: http://aopalliance.sourceforge.net [18 May 2005].

AOSD.NET (2005), Tools for Developers, Available from: http://aosd.net/wiki/index.php?title=Tools_for_Developers [28 April 2005].

Araujo, J., Baniassad, E., Clements, P., Moreira, A., Rashid, A. & Tekinerdogan, B. (2005), Early Aspects: The Current Landscape, Technical Report COMP-001-2005, Lancaster University.

Asteasuain, F., Contreras, B., Estvez, E. & Fillottrani, P. (2004), Evaluation of UML Extensions for Aspect Oriented Design, *in* 'Proceedings of JIISIC'04'.

Baniassad, E. (2003), Theme: Intro, Available from: http://www.dsg.cs.tcd.ie/index.php?category_id=361 [2 June 2005].

Baniassad, E. & Clarke, S. (2004), Theme: An Approach for Aspect-Oriented Analysis and Design, *in* 'Proceedings of the 26th International Conference on Software Engineering (ICSE'04)'.

Basch, M. & Sanchez, A. (2003), Incorporating Aspects into the UML, Available from: http://whitepapers.zdnet.co.uk/0,39025945,60092638p-39000629q,00.htm [20 December 2005].

Binder, R. (2000), *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley.

Boner, J. & Vasseur, A. (2005), AspectWerkz Extensible Aspect Container, Available from: http://aspectwerkz.codehaus.org/extensions.html [15 December 2005].

Boner, J., Vasseur, A. & Dahlstedt, J. (2005*a*), JRockit JVM Support For AOP, Part 1, Available from: http://dev2dev.bea.com/pub/a/2005/08/jvm_aop_1.html [15 August 2005].

Boner, J., Vasseur, A. & Dahlstedt, J. (2005*b*), JRockit JVM Support For AOP, Part 2, Available from: http://dev2dev.bea.com/pub/a/2005/08/jvm_aop_2.html [15 August 2005].

Burke, B. (2004), The Open Source Pro Circuit, Available from: http://www.pyrasun.com/mike/mt/archives/2004/12/02/15.21.30/ [18 May 2005].

Burke, B. (2005), The Server Side: Interview with Bill Burke, Available from: http://www.theserverside.com/talks/videos/BillBurke2/interview.tss?bandwidth=dsl [5 May 2005].

Ceccato, M. & Tonella, P. (2004), Measuring the Effects of Software Aspectization, *in* 'Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004). November, 2004. Delft, The Netherlands.'.

Ceccato, M., Tonella, P. & Ricca, F. (2005), Is AOP code easier or harder to test than OOP code?, *in* 'Proceedings of the First Workshop on Testing Aspect-Oriented Programs (WTAOP 2005)', Chicago, Illinois.

Chidamber, S. R. & Kemerer, C. F. (1994), 'A metrics suite for object-oriented design', *IEEE Transactions on Software Engineering* **20**(6), 476–493.

Clarke, S. & Baniassad, E. (2005), *Aspect-Oriented Analysis and Design: The Theme Approach*, Pearson Education Inc., NY.

Clarke, S. & Walker, R. (2002), Towards a Standard Design Language for AOSD, *in* 'Proceedings of the 1st International Conference on Aspect-Oriented Software Development'.

Clarke, S. & Walker, R. (2005), *Aspect-Oriented Software Development*, Pearson Education Inc., NJ, chapter 19, pp. 425–458.

Clement, A., Harley, G., Colyer, A. & Webster, M. (2004), *Eclipse AspectJ*, Addison-Wesley.

Clemente, P., Hernandez, J., Herrero, J., Murillo, J. & Sanchez, F. (2005), *Aspect-Oriented Software Development*, Pearson Education Inc., NJ, chapter 18, pp. 407–423.

Cole, L., Piveta, E. & Sampaio, A. (2004), RUP Based Analysis and Design with Aspects, *in* 'Proceedings of the XVIII Brazilian Symposium on Software Engineering - SBES'04'.

Colyer, A. (2004), Announcing aUnit, AspectJ Users Mailing List, 8 November 2004.

Cottenier, T., Berg, A. V. D. & Elrad, T. (2005), Modeling Aspect-Oriented Compositions, *in* 'Proceedings of the 7th International Workshop on Aspect-Oriented Modeling held in conjunction with the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)'.

Dietrich, J. & Elgar, C. (2005), A formal Description of Design Patterns using OWL, *in* 'Proceedings of the Australian Software Engineering Conference (ASWEC'05), IEEE Computer Society'.

Eclipse Foundation (2005), AspectJ Frequently Asked Questions, Available from http://www.eclipse.org/aspectj/doc/released/faq.html [23 November 2005].

Elrad, T., Aldawud, O. & Bader, A. (2005), *Aspect-Oriented Software Development*, Pearson Education Inc., NJ, chapter 20, pp. 459–478.

Filman, R. E., Elrad, T., Clarke, S. & Akşit, M. (2005), *Aspect-Oriented Software Development*, Pearson Education Inc., NJ.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.

Glover, A. (2004), AOP banishes the tight-coupling blues, Available from: http://www-128.ibm.com/developerworks/java/library/j-aopsc/ [20 February 2005].

Griggs, S. (2005), CSA, SolNet Solutions Internal Document.

Griswold, W., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y. & Rajan, H. (2006), 'Modular Software Design with Crosscutting Interfaces', *IEEE Software, Special Issue on Aspect-Oriented Programming.*

Han, Y., Kniesel, G. & Cremers, A. (2005), Towards Visual AspectJ by a Meta Model and Modeling Notation, *in* 'Proceedings of the 6th International Workshop on Aspect-Oriented Modeling held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)'.

Hanenberg, S. & Schmidmeier, A. (2003), AspectJ Idioms for Aspect-Oriented Software Construction, *in* 'Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany'.

Hannemann, J. & Kiczales, G. (2002), Design Pattern Implementation in Java and AspectJ, *in* 'Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)', pp. 161–173.

IEEE Software Engineering Standards (1990). Standard 610.12-1990.

Isberg, W. (2002), 'Get Test-Inoculated!', *Software Development Magazine.* Available from: http://www.sdmagazine.com/documents/s=7134/sdm0205b/0205b.htm [7 March 2005].

Jacobson, I. (2003), 'Use Cases and Aspects - Working Seamlessly Together', *Journal of Object Technology* **2**(4), 7–28.

Jacobson, I. (2005), About Ivar Jacobson, Available from: http://www.ivarjacobson.com/html/content/about.html [12 January 2006].

Jacobson, I. & Ng, P.-W. (2004), *Aspect-Oriented Software Development with Use Cases*, Pearson Education Inc., NJ.

Johnson, R., Hoeller, J., Arendsen, A., Sampaleanu, C., Harrop, R., Risberg, T., Davison, D., Kopylenko, D., Pollack, M., Templier, T. & Vervaet, E. (2005), Spring - Java/J2EE Application Framework Reference Documentation, Available from: http://static.springframework.org/spring/docs/1.2.x/reference/index.html [15 December 2005].

Jones, C. (1994), 'Software Metrics: Good, bad, and missing', *IEEE Computer* **27**(9), 98–100.

Kande, M., Kienzle, J. & Strhmeier, A. (2002), From AOP to UML - A Bottom-Up Approach, *in* 'Proceedings of the Aspect-Oriented Modelling with UML as part of 1st International Conference on Aspect-Oriented Software Development'.

Katara, M. & Mikkonen, T. (2002), Refinements and Aspects in UML, *in* 'Proceedings of the Workshop on Aspect-Oriented Modeling with UML'.

Kiczales, G. (2003), Interview with Gregor Kiczales, Available from: http://www.theserverside.com/talks/videos/GregorKiczalesText/interview.tss [8 April 2005].

Kiczales, G. (2005), 'Once More, From the Top', *Software Development Magazine.* Available from: http://www.sdmagazine.com/documents/s=9512/sdm0502g/sdm0502g.html [7 March 2005].

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. & Irwin, J. (1997), Aspect-Oriented Programming, *in* 'Proceedings of the European Conference on Object-Oriented Programming (ECOOP)', Finland.

Laddad, R. (2003), *AspectJ In Action*, Manning Publications Co., Greenwich, CT.

LaMonica, M. (2003), IBM, JBoss eye new Java plan, Available from: http://news.com.com/2100-1007_3-5081831.html [16 May 2005].

Lee, B. (2005), Dynaop, Available from https://dynaop.dev.java.net/ [23 November 2005].

Lesiecki, N. (2004), Announcing aUnit, AspectJ Users Mailing List, 8 November 2004.

Lesiecki, N. (2005*a*), AOP@Work: Enhance Design Patterns with AspectJ, Part1, Available from: http://www-128.ibm.com/developerworks/java/library/j-aopwork5/index.html [17 May 2005].

Lesiecki, N. (2005*b*), AOP@Work: Enhance Design Patterns with AspectJ, Part2, Available from: http://www-128.ibm.com/developerworks/java/library/j-aopwork6/index.html [17 May 2005].

Lesiecki, N. (2005*c*), AOP@Work: Unit test your aspects, Available from: http://www-128.ibm.com/developerworks/java/library/j-aopwork11/?ca=dgr-lnxw01AOPtesting [11 November 2005].

Longstreet, D. (1992), Fundamentals of Function Point Analysis, Available from: http://www.ifpug.com/fpafund.htm [15 August 2005].

Lopes, C. & Ngo, T. (2005), Unit-Testing Aspectual Behavior, *in* 'Proceedings of the Workshop on Testing Aspect-Oriented Programs (WTAOP), held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD05)'.

McCabe, T. (1976), 'A Complexity Measure', *IEEE Transactions on Software Engineering SE-2.*

Miles, R. (2004), *AspectJ Cookbook*, O'Reilly.

Monk, S. & Hall, S. (2002), Virtual Mock Objects using AspectJ with JUnit, Available from: http://www.xprogramming.com/xpmag/virtualMockObjects.htm [18 February 2005].

Mortensen, M. & Alexander, R. (2005), An Approach for Adequate Testing of AspectJ Programs, *in* 'Proceedings of the 2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD 2005)'.

Pawlak, R. (2003), The AOP Alliance: Why Did We Get In?, Available from: http://aopalliance.sourceforge.net/white_paper/white_paper.pdf [18 May 2005].

Pawlak, R. (2005*a*), AOP Alliance Proposal, Available from: http://sourceforge.net/mailarchive/message.php?msg_id=12755402 [25 August 2005].

Pawlak, R. (2005*b*), JAC, http://jac.objectweb.org/ [24 November 2005].

Pawlak, R. & Younessi, H. (2004), 'On Getting Use Cases and Aspects to Work Together', *Journal of Object Technology*.

Pressman, R. (2001), *Software Engineering: A Practitioner's Approach 5th Ed*, McGraw-Hill, New York.

Rausch, A., Rumpe, B. & Cornel Klein, L. H. (2003), Aspect-Oriented Framework Modeling, *in* 'Proceedings of the 4th AOSD Modeling with UML Workshop (UML Conference 2003)'.

Shirky, C. (2001), Interoperability, Not Standards, Available from: http://www.openp2p.com/pub/a/p2p/2001/03/15/clay_interop.html [8 September 2005].

Small, F. (2000), Ministerial Inquiry into INCIS, Available from: http://www.justice.govt.nz/pubs/reports/2000/incis_rpt/ [12 January 2006].

Software Composition Technologies (1997), Function Point FAQ, http://ourworld.compuserve.com/homepages/softcomp/fpfaq.htm [1 December 2005].

Souter, A., Shepherd, D. & Pollock, L. (2003), Testing with Respect to Concerns, *in* 'Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM'03)'.

Stein, D., Hanenberg, S. & Unland, R. (2002*a*), A UML-based aspect-oriented design notation for AspectJ, *in* 'Proceedings of the 1st International Conference on Aspect-Oriented Software Development'.

Stein, D., Hanenberg, S. & Unland, R. (2002*b*), Designing Aspect-Oriented Crosscutting in UML, *in* 'Proceedings of the AOSD-UML Workshop at AOSD'02'.

Stein, D., Hanenberg, S. & Unland, R. (2002*c*), On Representing Join Points in the UML, *in* 'Proceedings of the Workshop on Aspect-Oriented Modeling with UML'.

Stein, D., Hanenberg, S. & Unland, R. (2003), Position Paper on Aspect-Oriented Modeling: Issues on Representing Crosscutting Features, Available from: http://lglwww.epfl.ch/workshops/aosd2003/papers/Stein-AOMissues.pdf [20 December 2005].

Stochmialek, M. (2005), AOP Metrics, Available from: http://aopmetrics.tigris.org [8 August 2005].

Sullivan, K., Griswold, W., Song, Y. & Cai, Y. (2005), On the criteria to be used in decomposing systems into aspects, *in* 'Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005, Lisbon, Portugal, September 5–9 2005'.

Sun Microsystems (2005), Enterprise JavaBeans Technology, Available from: http://java.sun.com/products/ejb/ [11 December 2005].

Suzuki, J. & Yamamoto, Y. (1999), Extending UML with Aspects: Aspect Support in the Design Phase, *in* 'Proceedings of the 3rd Aspect-Oriented Programming (AOP) Workshop at ECOOP'99'.

Tarr, P. & Ossher, H. (2001), HyperJ, Available from: http://www.alphaworks.ibm.com/tech/hyperj [15 December 2005].

The Apache Software Foundation (2000), Apache Struts Project, Available from: http://struts.apache.org/ [11 December 2005].

The Standish Group (1994), The CHAOS Report, Available from: http://www.standishgroup.com/sample_research/chaos_1994_1.php [15 December 2005].

The Standish Group (2003), Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%, Available from: http://www.standishgroup.com/press/article.php?id=2 [15 December 2005].

van der Werff, T. J. (2001), 10 Emerging Technologies That Will Change the World, Available from: http://www.globalfuture.com/mit-trends2001.htm [4 August 2005].

VanDoren, E. (1997), Halstead Complexity Measures, Available from: http://www.sei.cmu.edu/str/descriptions/halstead_body.html [22 August 2005].

VanDoren, E. (2000), Cyclomatic Complexity, Available from: http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html [22 August 2005].

Vasseur, A. (2004), AOP Benchmark, Available from: http://docs.codehaus.org/display/AW/AOP+Benchmark [15 December 2005].

Virtual Machinery (2005), JHawk Metrics Tool, Available from: http://www.virtualmachinery.com/jhawkprod.htm [12 August 2005].

von Flach Chavez, C., Garcia, A., Kulesza, U., Anna, C. S. & Lucena, C. (2005), Taming Heterogeneous Aspects with Crosscutting Interfaces, *in* 'Proceedings of the Brazilian Symposium on Software Engineering 2005'.

Wiegers, K. (1999), 'A Software Metrics Primer', *Software Development Magazine.*

Wikipedia (2005*a*), Aspect-Oriented Programming, Available from: http://en.wikipedia.org/wiki/Aspect-oriented_programming [22 November 2005].

Wikipedia (2005*b*), Subject-Oriented Programming, Available from: http://en.wikipedia.org/wiki/Subject-oriented_programming [15 December 2005].

Xie, T., Zhao, J., Marinov, D. & Notkin, D. (2004), Detecting Redundant Unit Tests for AspectJ Programs, Technical Report UW-CSE-04-10-03, Department of Computer Science & Engineering, University of Washington, USA.

Xie, T., Zhao, J., Marinov, D. & Notkin, D. (2005), Automated Test Generation for AspectJ Programs, *in* 'Proceedings of the AOSD 2005 Workshop on Testing Aspect-Oriented Programs (WTAOP'05)', Chicago, USA.

Xu, D., Xu, W. & Nygard, K. (2005), A State-Based Approach to Testing Aspect-Oriented Programs, *in* 'Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)', Taiwan.

Zakaria, A., Hosny, H. & Zeid, A. (2002), A UML Extension for Modeling Aspect-Oriented Systems, *in* 'Proceedings of the Aspect Modeling with UML workshop at the Fifth International Conference on the Unified Modeling Language - the Language and its Applications'.

Zhang, C. & Jacobsen, H.-A. (2003), Quantifying Aspects in Middleware Platforms, *in* 'Proceedings of the Aspect-Oriented Software Development Conference (AOSD 2003), Boston, MA USA'.

Zhao, J. (2002), Tool Support for Unit Testing of Aspect-Oriented Software, *in* 'Proceedings of the OOPSLA'2002 Workshop on Tools for Aspect-Oriented Software Development', Seattle, WA, USA.

Zhao, J. (2003), Data-Flow-Based Unit Testing of Aspect-Oriented Programs, *in* 'Proceedings of the 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003), pp.188-197. Dallas, Texas, USA, November 2003.'.

Zhao, J. (2004), Measuring Coupling in Aspect-Oriented Systems, *in* 'Proceedings of the 10th International Software Metrics Symposium (METRICS'2004), Chicago, USA, September 14-16, 2004'.

Glossary

Agitator	A testing tool being trialled by SolNet
AJDT	AspectJ Development Tools - Tool support for developing AspectJ applications in the Eclipse IDE
ANT	Another Neat Tool - Used to automate the Java build process
AO	Aspect-Oriented - An approach to software development that uses aspects
AOP	Aspect-Oriented Programming - A development approach which extends traditional modularity technologies with the aspect. An aspect is used to modularise crosscutting concerns whose implementation is traditionally scattered throughout objects rather than being localised
AOP Alliance	A group that has released a standard for Aspect-Oriented programming
AOPI	Aspect-Oriented Programming Interfaces - A standards effort for AOP that merged with the AOP Alliance
AOSD	Aspect-Oriented Software Development - A full development methodology for aspects which incorporates the benefits of aspects into all phases of the software development lifecycle
Apache Tomcat	A servlet container released by The Apache Software Foundation
API	Application Program Interface - An interface which allows a program to access the functionality of another
ASM	Aspectual State Models - An extension of FREE which allows state based testing of aspect software

AspectJ	An Aspect-Oriented implementation for Java which extends the Java language with new concepts for aspects
AspectWerkz	An Aspect-Oriented implementation for Java which uses plain Java classes
aUnit	A testing framework for AspectJ
Bad Smell	Signs which indicate bad design
Bean Script	Bean Script is a scripting language for writing dynamically interpreted Java
BOF	Business Object Framework - A SolNet Solutions CSA framework to simplify business object development
Boiler Plate Code	Code that is consistently added to implement infrastructural concerns
Checked Exception	A Java exception which must be explicitly dealt with by the developer, otherwise a compiler error will be signalled
Container Persistence	The EJB container manages storage, updates, and retrievals to Entity Beans
COTS	Commercial Off The Shelf - The term used to describe software components that can be purchased from component vendors
CSA	Common Services Architecture - A set of components and processes that can be reused on projects
CST	Common Services Team - A SolNet team that works on standardising their development approach and support infrastructure
DTO	Data Transfer Object - A simple object used to pass data between application layers
EJB	Enterprise Java Beans - A component framework for developing J2EE applications
Entity Bean	An EJB used to model persistent objects

EOS	Education Organisation System - An eQA application
FREE	Flattened Regular Expression - A state model used to model applications which allows state based testing to be performed
GOF	Gang of Four - The name given to group that released the first OO pattern book for software development. This group consists of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
GUI	Graphical User Interface - The visual interface of the application which the user interacts with
Halstead Metrics	A set of software complexity measures based on operator and operand complexity
IDE	Integrated Development Environment - A tool for software development which combines tools such as code editors, debuggers, and testing frameworks
J2EE	Java Enterprise Edition - The Java platform used to develop enterprise applications
JAC	Java Aspect Components - A framework for developing Aspect-Oriented applications in Java
JAR	Java Archive - A zip file used to distribute a related set of Java classes
javac	The standard Java compiler distributed with the Sun Microsystems's reference Java implementation
JBoss AOP	An Aspect-Oriented implementation for Java from the JBoss Group
JCP	Java Community Process - An process used to create new Java APIs or make modifications to existing APIs
JDT	Java Development Tools - The compiler and related tools developed for use in the Eclipse IDE

JMock	A tool for writing Java tests using mock objects
JSP	Java Server Page - A page served over the web which contains dynamic elements resolved on the server
JSR	Java Specification Request - A formal request for a new Java API. Proceeds through the JCP
JTest	A tool for generating tests from Java byte code
JUnit	A testing framework for Java
JVM	Java Virtual Machine - Executes Java applications
LOC	Lines of Code - A simple measure of program size
Make	A tool for building software
MCC	McCabes Cyclomatic Complexity - A measure of software complexity based on the number of paths through the program
MoE	Ministry of Education - A Government department that runs the NZ school system
Nanning	An Aspect-Oriented implementation for Java
NOS	Number of Statements - A normalised measure of program size that accounts for physical layout, comments, and white space
NSI	National Student Index - A database of all students in NZ schools
OMG	Object Management Group - An industry consortium that promotes standards for object technology
OO	Object-Oriented - A development approach where systems are built from objects
POC	Proof of Concept - An experimental development project used to try new development techniques
POJO	Plain Old Java Object - A simple Java object with no framework dependencies

QA	Quality Assurance - The process of ensuring that a product meets certain quality criteria
RMI	Remote Method Invocation - A Java protocol for invoking methods on objects that are remotely located
Rose Script	A scripting language for extending Rational Rose
Semantic Pointcuts	Pointcuts which describe a property of join points such as methods which modify the state of an object
Session Bean	An EJB used to implement services
SOA	Service Oriented Architecture - A way of building loosely coupled applications using web services
SPER	Students, Processing, Entries, and Results - An eQA application
Spring Framework	A framework for developing J2EE applications without EJB. Spring Framework includes an Aspect-Oriented Programming implementation for Java
Struts	An Apache Software Foundation framework for building Model-View-Controller Web Applications
SXI	SPER eXternal Interface - An EJB Facade which presents functionality from the SPER application to other applications
Sybase EAServer	An EJB container released by Sybase Inc.
TIF	Technology in Industry Fellowship - A fellowship offered by Technology New Zealand to support university and business research relationships which can result in commercial advantage
UML	Unified Modelling Language - A formal language for modelling software artifacts standardised by the OMG

Unchecked Exception	A Java exception which optionally may be handled by the developer
WAR	Web Archive - A zip file used to deploy Java Web Applications
Web Services	An open set of protocols and standards used to exchange data between applications over networks
XML	eXtensible Markup Language - A way of structuring documents and data to allow open exchange of information
xPath	A W3C recommendation for addressing parts of an XML document

APPENDIX A
FRAMEWORK EXAMPLES

A.1 Introduction

This appendix provides a simple 'Hello World' example using each of the AOP Frameworks discussed in Chapter 3.

A.2 Example Application Class

The Java class shown in Listing A.1 is a simple application that prints the message 'Hello There!' to the console. This class requires tracing of its method executions. This will be demonstrated using AspectJ, AspectWerkz, JBoss AOP, Spring, and Dynaop.

Listing A.1: HelloWorld Base Class

```
package nz.ac.massey.aop;

public class HelloWorld {

  public HelloWorld() {
    super();
  }

  public void sayHello() {
    System.out.println("Hello There!");
  }

  public static void main(String[] args) {
    new HelloWorld().sayHello();
  }
}
```

A.3 AspectJ

AspectJ allows both the aspect logic and configuration to be combined in one unit as in Listing A.2. The output produced when this application is run is shown in Listing A.3.

Listing A.2: AspectJ Tracing

```
package nz.ac.massey.aop.aspectj;

// Aspect to trace method entries
public aspect AspectJTracing {

  // trace execution of methods in the system
  public pointcut methodsToTrace() : execution(*
      nz.ac.massey.aop.HelloWorld.*(..));

  // advice that executes before the join points
  // captured by methodsToTrace
  before() : methodsToTrace() {
    System.out.println("Entering: " + thisJoinPoint);
  }

  // advice that executes before the join points
  // captured by methodsToTrace
  after() : methodsToTrace() {
    System.out.println("Exiting: " + thisJoinPoint);
  }
}
```

Listing A.3: AspectJ Output

```
Entering: execution(void
    nz.ac.massey.aop.HelloWorld.main(String[]))
Entering: execution(void nz.ac.massey.aop.HelloWorld.sayHello())
Hello There!
Exiting: execution(void nz.ac.massey.aop.HelloWorld.sayHello())
Exiting: execution(void
    nz.ac.massey.aop.HelloWorld.main(String[]))
```

A.4 AspectWerkz

AspectWerkz uses a normal Java class to implement the aspect logic as in Listing A.4. The advice is implemented as normal Java methods which take the join point being advised as a parameter.

Listing A.4: AspectWerkz Tracing

```
package nz.ac.massey.aop.aspectwerkz;

import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

// Simple tracing aspect using AspectWerkz
public class AspectWerkzTracing {

    // method to log method entries
    public void beforeTrace(JoinPoint joinpoint) {
        System.out.println("Entering: " +
            joinpoint.getSignature());
    }

    // method to log method exits
    public void afterTrace(JoinPoint joinpoint) {
        System.out.println("Exiting: " +
            joinpoint.getSignature());
    }
}
```

The aspect needs to be connected to the application. This is done using a XML configuration file such as Listing A.5. Finally, the output from the application is shown in Listing A.6.

Listing A.5: AspectWerkz Configuration File

```
<aspectwerkz>
  <system id="AspectWerkzTracing">
    <package name="nz.ac.massey.aop.aspectwerkz">
      <!-- Identify aspect-->
      <aspect class="AspectWerkzTracing">
        <!-- Pointcut containing methods -->
        <pointcut name="methodsToTrace"
            expression="execution(public void
            nz.ac.massey.aop.HelloWorld.*(..))"/>
  <!-- Link advice to pointcut -->
  <advice name="beforeTrace" type="before"
     bind-to="methodsToTrace"/>
        <advice name="afterTrace" type="after"
            bind-to="methodsToTrace"/> -->
      </aspect>
    </package>
  </system>
</aspectwerkz>
```

Listing A.6: AspectWerkz Output

```
AspectWerkz - INFO - Pre-processor
   org.codehaus.aspectwerkz.transform.AspectWerk
zPreProcessor loaded and initialized
Entering: public static void
   nz.ac.massey.aop.HelloWorld.main(java.lang.String[]
)
Entering: public void nz.ac.massey.aop.HelloWorld.sayHello()
Hello There!
Exiting: public void nz.ac.massey.aop.HelloWorld.sayHello()
Exiting: public static void
   nz.ac.massey.aop.HelloWorld.main(java.lang.String[])
```

A.5 JBoss AOP

JBoss AOP requires separate aspect logic and configuration similar to that of AspectWerkz. However, notice that JBoss AOP uses a single interceptor instead of separate methods for before and after advice. This requires the use of a proceed() method to invoke the method being advised. Listing A.7 shows the aspect class and Listing A.8 the XML configuration. Finally, Listing A.9 shows the resulting output from the application.

Listing A.7: JBoss Tracing

```
package nz.ac.massey.aop.jboss;

import org.jboss.aop.joinpoint.MethodInvocation;

//JBoss AOP aspect for tracing
public class JBossTracing {
   // advice method
   public Object trace(MethodInvocation invocation) throws
      Throwable {
       try {
         System.out.println("Entering: " +
            invocation.getMethod());
         // proceed to next advice or actual call
         return invocation.invokeNext();
      } finally {
         System.out.println("Exiting: " +
            invocation.getMethod());
      }
   }
}
```

Listing A.8: JBoss Configuration File

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<aop>
    <!-- Identify aspect classes -->
    <aspect class="nz.ac.massey.aop.jboss.JBossTracing"
        scope="PER_VM" />

    <!-- Declare Pointcut to capture methods to trace -->
    <pointcut expr="execution(public *
        nz.ac.massey.aop.HelloWorld-&gt;*(..))"
        name="methodsToTrace" />

    <!-- Link pointcut to advice method -->
    <bind pointcut="methodsToTrace">
        <advice aspect="nz.ac.massey.aop.jboss.JBossTracing"
            name="trace" />
    </bind>
</aop>
```

Listing A.9: JBoss Output

```
Entering: public static void
   nz.ac.massey.aop.HelloWorld.main(java.lang.String[])
Entering: public void nz.ac.massey.aop.HelloWorld.sayHello()
Hello There!
Exiting: public void nz.ac.massey.aop.HelloWorld.sayHello()
Exiting: public static void
   nz.ac.massey.aop.HelloWorld.main(java.lang.String[])
```

A.6 Spring Framework

The Spring Framework promotes the use of programming to an interface. For this reason, an interface is created for HelloWorld to implement as in Listing A.10. It is possible to advise classes that don't implement an interface but it requires extra configuration. Further changes are made to the base class to account for the new interface and changes to the application main method to account for the requirement that objects are instantiated using the Spring Framework rather than directly. This class is shown as Listing A.11.

Listing A.10: Spring HelloWorld Interface

```
package nz.ac.massey.aop;

// Interface class for HelloWorld

public interface IHelloWorld {
    public void sayHello();
}
```

Listing A.11: Spring HelloWorld Class

```
package nz.ac.massey.aop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.*;

// Simple class that creates an object
// and prints out Hello There! on the console
public class HelloWorld implements IHelloWorld {

  public HelloWorld() {
    super();
  }

  public void sayHello() {
    System.out.println("Hello There!");
  }

  public static void main(String[] args) {
    // Read the configuration file
    ApplicationContext ctx = new
        FileSystemXmlApplicationContext("springconfig.xml");

    //Instantiate an object
    IHelloWorld hw = (IHelloWorld)
        ctx.getBean("helloworldbean");

    // Execute method
    hw.sayHello();
  }
}
```

The Spring aspect is implemented using an AOP Alliance interceptor (Listing A.12 and the configuration is implemented using XML (Listing A.13). Finally, the output is shown in Listing A.14.

Listing A.12: Spring Tracing

```
package nz.ac.massey.aop.spring;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class SpringTracing implements MethodInterceptor {

  // Advice method
  public Object invoke(MethodInvocation invocation) throws
     Throwable {

        System.out.println("Entering: " + invocation);
        // execute advised method
        Object result = invocation.proceed();
        System.out.println("Exiting: " + invocation);
        return result;
  }
}
```

Listing A.13: Spring Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

   <!-- Bean configuration -->
   <bean id="helloworldbean"
   class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="proxyInterfaces">
         <value>nz.ac.massey.aop.IHelloWorld</value>
      </property>
      <property name="target">
         <ref local="beanTarget"/>
      </property>
      <property name="interceptorNames">
         <list>
            <value>SpringTracing</value>
         </list>
         </property>
   </bean>
   <!-- Bean Classes -->
   <bean id="beanTarget"
   class="nz.ac.massey.aop.HelloWorld"/>

   <!-- Advisor pointcut definition for around advice -->
   <bean id="SpringTracing" class=
   "org.springframework.aop.support.RegexpMethodPointcutAdvisor">
      <property name="advice">
         <ref local="aroundTracingAdvice"/>
      </property>
      <property name="pattern">
         <value>.*</value>
      </property>
   </bean>

   <!-- Advice classes -->
   <bean id="aroundTracingAdvice"
```

```
        class="nz.ac.massey.aop.spring.SpringTracing" />

</beans>
```

Listing A.14: Spring Output

```
Entering: invocation: method 'sayHello', arguments []; target is
    of class [nz.ac.massey.aop.HelloWorld]
Hello There!
Exiting: invocation: method 'sayHello', arguments []; target is
   of class [nz.ac.massey.aop.HelloWorld]
```

A.7 Dynaop

Dynaop, similarly to Spring, requires that objects are instantiated using the framework. For this reason, a class is added to launch the application in Listing A.15.

Listing A.15: Dynaop Application Launcher

```
package nz.ac.massey.aop.dynaop;

import dynaop.ProxyFactory;
import nz.ac.massey.aop.HelloWorld;

public class ApplicationLaunch {

   // main method to launch HelloWorld with dynaop aspects
  public static void main(String[] args) {
    // instantiate HelloWorld using dynaop
    HelloWorld hw = (HelloWorld)
        ProxyFactory.getInstance().extend(HelloWorld.class);
    // invoke method that has been advised
    hw.sayHello();
  }
}
```

The tracing aspect is implemented as an interceptor in Listing A.16. Configuration is performed using Bean Script as in Listing A.17. Note this is the shortest and simplest of the external configuration files. However, the requirement that objects are instantiated using the framework makes it more invasive. Finally, the output is shown in Listing A.18.

Listing A.16: Dynaop Tracing

```
package nz.ac.massey.aop.dynaop;

import dynaop.Interceptor;
import dynaop.Invocation;

public class DynaopTracing implements Interceptor {

  public Object intercept(Invocation invocation) throws
      Throwable {

    System.out.println("Entering: " + invocation.getMethod());
    // Call intercepted method
    Object result = invocation.proceed();
    System.out.println("Exiting: " + invocation.getMethod());
    return result;
  }
}
```

Listing A.17: Dynaop Configuration File

```
// Apply interceptor to all methods.
interceptor
(
  nz.ac.massey.aop.HelloWorld.class,
  ALL_METHODS,
  new nz.ac.massey.aop.dynaop.DynaopTracing()
);
```

Listing A.18: Dynaop Output

```
Entering: public void nz.ac.massey.aop.HelloWorld.sayHello()
Hello There!
Exiting: public void nz.ac.massey.aop.HelloWorld.sayHello()
```

A.8 Summary

The tracing example given in these appendices provide a concrete example of the use of the frameworks.

APPENDIX B
EXTENDED CODE LISTINGS

Listing B.1: Decompiled Service Bean

```
// Partial source from decompilation of the
// TopScholarServiceBean by DJ Decompiler
// Simplified for readability

package nz.govt.nzqa.sper.award.service.impl;

// imports removed

public class TopScholarServiceBean extends ServiceBaseBean
{
    // service method
    public void generateMsdTopScholarFile(String year)
        throws MsdTopScholarFileException
    {
  // large number of nested try/catch blocks for
  // exception handling (most catch blocks removed)
        try { try { try { try { try { try { try { try { try
            { try { try { try { try {
      // Call to Service wrapper begin()
      SperServiceWrapper.aspectOf().
        ajc$before$ServiceBeanWrapper$1$133f7d78(this);
      // Business Logic
      TopScholarFactory.generateMsdTopScholarFile(year);
  }
  catch(Throwable throwable)
  {
      // Call to Service wrapper end() when
      // exception is thrown
      SperServiceWrapper.aspectOf().
```

```
        ajc$after$ServiceBeanWrapper$2$d46692c4(this);
      throw throwable;
  }
  // Call to Service wrapper end() when
  // no exception is thrown
  SperServiceWrapper.aspectOf().
    ajc$after$ServiceBeanWrapper$2$d46692c4(this);
  return;
  }
      // Exception handling for Create Exception
  catch(CreateException createexception {
    // will convert exception to a ReportableCreateException
    SperExceptionHandler.aspectOf().
      ajc$afterThrowing$ExceptionHandler$1$77f415ae(
        createexception);
    throw createexception;
  }

  // lots of other exception handlers removed

  // Code used to soften exceptions to avoid compiler
  // warnings when aspects deal with the exception
  catch(CreateException createexception1) {
    if(createexception1 instanceof RuntimeException)
        throw createexception1;
    else
        throw new SoftException(createexception1);
  }
    }
}
```

Listing B.2: Example aUnit Test Aspect

```
package nz.co.solnet.infrastructure.transactionhandling;

// imports removed

//Tests the advice that rollsback transactions on exceptions
public class TestReportableExceptionRollback extends TestCase {

    // Setup environment for tests
    protected void setUp() throws Exception {
        super.setUp();
        // Setup and initialise Solnet CSA
        // Most of this is removed
        System.setProperty("log4j","log4j.properties");
        Initialisation.checkInitialisation(
            Initialisation.getCoreInitialisers()
        );
    }

    // specify the steps to be run in the test
    public TestStep[] constructSimpleTestSteps() throws Exception
        {
        TestStep[] steps = new TestStep[1];

        // Create the 'this' reference
        MockServiceBean thisReference = new MockServiceBean();

        // Get method1
        Method method1 = thisReference.getClass()
                .getMethod("doSomething1", new Class[0]);

        // Create the join points static part using method1
        JoinPoint.StaticPart staticPart =
            Factory.makeEncSJP(method1);

        // Specify that there are no arguments to be passed to
            method1
        Object[] args = new Object[0];
```

```
        // Create a join point, specifying the static information,
            the target,
        // and the object pointed to by the this reference.
        // Target in this case is the same as the thisReference.
        JoinPoint joinPoint = Factory.makeJP(staticPart,
            thisReference,
                thisReference, args, new
                    ReportableCreateException(new
                    CreateException()));

        JoinPointContext context = new
            JoinPointContext(joinPoint);

        // Regular expression used to control the selection of
            appropriate
        // advice blocks to be invoked by the supplied context.
        String adviceSelectorExpression = "id1";

        // create the aspect
        ReportableExceptionRollback rollbackHandler =
            (ReportableExceptionRollback)
            ConcreteMockReportableExceptionRollback.aspectOf();

        // create the test step using the above context and the
            aspect
        TestStep testStep =
            TestStepFactory.createControlledTestStep(
            rollbackHandler, context, adviceSelectorExpression);

        steps[0] = testStep;

        return steps;
    }

    // run the test
    public void testAspectWithSimpleTestSpecification()
    {
        // Create a test specification
        // Get the test steps
```

```
        // Add the steps to the test specification
        // Execute the test specification
        // Examine the results of the test steps
    }
}
```

APPENDIX C
ASPECT CODE LISTINGS

C.1 Base Aspects

Listing C.1: Service Wrapper Base Aspect

```
package nz.co.solnet.infrastructure.bof;

/**
 * Provides a wrapper for Service Bean methods that are wrapped
 * in begin/end statements. This should not be used for
 * external interfaces.
 * This aspect must be extended with a concrete definition
 * of the serviceBeanMethods() and excludedServiceBeanMethods()
 * pointcut.
 * serviceBeanMethods() should provide access to method
 * execution of all methods that call begin and end.
 * excludedServiceBeanMethods() should provide access to the
 * execution of all methods that use the begin(String) form. All
 * of these excluded methods should have advice in the subaspect
 * to call begin(String) with an appropriate comment. Note: this
 * aspect will call end() for excludedMethods that are included
 * in the serviceBeanMethods().
 * @author Chris Elgar
 */
public abstract aspect ServiceBeanWrapper {

    public abstract pointcut serviceBeanMethods();
    public abstract pointcut excludedServiceBeanMethods();

    // Allow sub aspects to make calls from different packages
    protected void begin(ServiceBaseBean bean, String message) {
        bean.begin(message);
```

```
    }

    before(ServiceBaseBean bean): serviceBeanMethods() &&
        this(bean) && !excludedServiceBeanMethods() {
        bean.begin();
    }

    after(ServiceBaseBean bean): serviceBeanMethods() &&
        this(bean) {
        bean.end();
    }
}
```

Listing C.2: Exception Handler Base Aspect

```
package nz.co.solnet.infrastructure.exceptionhandling;

import nz.co.solnet.infrastructure.incidentreporting.*;
import javax.ejb.*;
import javax.naming.NamingException;
import nz.co.solnet.infrastructure.bof.oids.OID;
import org.apache.log4j.Logger;

/**
 * Enforce policy to deal with exceptions in service methods of
 * session beans.
 */
public abstract aspect ExceptionHandler {

    // methods to apply handling to
    public abstract pointcut serviceBeanMethods();

    // soften exceptions that aspect captures
    declare soft: FinderException: serviceBeanMethods();
    declare soft: CreateException: serviceBeanMethods();
    declare soft: NamingException: serviceBeanMethods();
    declare soft: RemoveException: serviceBeanMethods();

    // Wrap exceptions in a Solnet exception
    after() throwing (CreateException e) : serviceBeanMethods()
       {
        throw new ReportableCreateException(e);
    }

    after() throwing (RemoveException e) : serviceBeanMethods()
       {
        throw new ReportableRemoveException(e);
    }

    after(OID oid) throwing (FinderException e) :
       serviceBeanMethods() && args(oid) {
        throw new ReportableFinderException(oid,
                thisJoinPointStaticPart.getSignature()
```

```
                                                .getDeclaringType(),
                                                    e);
    }

    after() throwing (FinderException e) : serviceBeanMethods()
       && !args(OID) {
        throw new EJBException(e);
    }

    // Deal with remaining exceptions
    after() throwing (Throwable t) : serviceBeanMethods() {
        logger.debug("Checked exception for: " + t);
        if(!(t instanceof Reportable)) {
                try {
                    throw new Exception(t);
                } catch(Exception e) {
                    throw new EJBException(e);
                }
        }
    }
}
```

Listing C.3: Transaction Rollback Base Aspect

```
package nz.co.solnet.infrastructure.transactionhandling;

import nz.co.solnet.infrastructure.bof.ServiceBaseBean;
import nz.co.solnet.infrastructure.incidentreporting.*;
import org.apache.log4j.Logger;

// Aspect to rollback all reportable exceptions
// in the solnet framework
public abstract aspect ReportableExceptionRollback {

    public abstract pointcut serviceBeanMethods();

    after(ServiceBaseBean bean) throwing(ReportableException e)
        : serviceBeanMethods() && this(bean) {
        bean.getSessionContext().setRollbackOnly();
    }

}
```

Listing C.4: Service Wrapper Base Aspect - External Interface

```
package nz.co.solnet.infrastructure.bof;

/**
 * Wrapper for ServiceBaseBeans which are external interfaces
 * between systems.
 * All mutators must call begin with the username as the
 * argument
 * We follow the rule that this MUST be the last argument of the
 * method. All accessors simply call begin().
 * The accessors and mutators pointcuts should be method
 * executions.
 * @author Chris Elgar
 */
public abstract aspect ExternalInterfaceServiceBeanWrapper {

    // identify methods
    public abstract pointcut accessors();
    public abstract pointcut mutators();

    // allow custom handling in sub aspects
    protected void begin(ServiceBaseBean bean, String message) {
        bean.begin(message);
    }

    // mutators version of begin
    before(ServiceBaseBean bean) : mutators() && this(bean) {
        Object [] args = thisJoinPoint.getArgs();
        String user = (String) args[args.length-1];
        bean.begin(user);
    }

    // accessors version of begin
    before(ServiceBaseBean bean) : accessors() && this(bean) {
        bean.begin();
    }

    // all methods call end()
```

```
    after(ServiceBaseBean bean) : (accessors() || mutators()) &&
        this(bean) {
        bean.end();
    }

}
```

Listing C.5: Custom Exception Handler (External Interface) Base Aspect

```
package nz.co.solnet.infrastructure.exceptionhandling;

import nz.co.solnet.infrastructure.bof.ServiceBaseBean;

// Base class for exception handling that has
// application specific properies
public abstract aspect CustomExceptionHandler {

    // methods to apply handling to
    public abstract pointcut serviceBeanMethods();

    // method with exception handling logic
    public abstract void exceptionHandler( ServiceBaseBean bean,
        Exception e);

    // advice to handle exception handling
    Object around(ServiceBaseBean bean): serviceBeanMethods() &&
        this(bean) {
        Object result = null;
        try {
            result = proceed(bean);
        }
        catch(Exception e) {
            exceptionHandler(bean, e);
        }
        return result;
    }

}
```

Listing C.6: Transaction Rollback (External Interface) Base Aspect

```
package nz.co.solnet.infrastructure.transactionhandling;

import nz.co.solnet.infrastructure.bof.ServiceBaseBean;

// Aspect to roll back exceptions on external interfaces
public abstract aspect ExternalInterfaceTransactionRollback {

    public abstract pointcut serviceBeanMethods();

    // after all exceptions call rollback
    after(ServiceBaseBean bean) throwing(Exception e) :
        serviceBeanMethods() && this(bean) {
         bean.getSessionContext().setRollbackOnly();
    }
}
```

Listing C.7: Base Tracing Aspect

```
package nz.co.solnet.infrastructure.tracing;


import org.aspectj.lang.*;

// Base tracing Aspect from AspectJ Mailing List
// with minor modifications
public abstract aspect Tracing {

      private pointcut staticContext () : !this(Object);
      private pointcut nonStaticContext (Object obj) :
         this(obj);
      private pointcut toStringMethod () : execution(String
         toString());

      private pointcut excluded () :
            within(Tracing+)
            || toStringMethod();

      pointcut tracedMethod () :
            execution(public * *(..))
            && !excluded();

      protected abstract pointcut shouldTrace ();

      before (Object obj) : tracedMethod() &&
         nonStaticContext(obj) && shouldTrace() {
            enter(thisJoinPoint,obj);
      }

      before () : tracedMethod() && staticContext() &&
         shouldTrace() {
            enter(thisJoinPoint);
      }

      after() returning(Object ret) : tracedMethod() &&
         shouldTrace() {
            exit(thisJoinPointStaticPart,ret);
```

```
    }

    after() throwing(Exception ex) : tracedMethod() &&
        shouldTrace() {
            exception(thisJoinPointStaticPart,ex);
    }

    protected abstract void enter (JoinPoint jp, Object obj);

    protected abstract void enter (JoinPoint jp);

    protected abstract void exit (JoinPoint.StaticPart sjp,
        Object ret);

    protected abstract void exception (JoinPoint.StaticPart
        sjp, Exception ex);

}
```

Listing C.8: Base JDK Tracing

```
package nz.co.solnet.infrastructure.tracing;

import java.util.logging.*;
import org.aspectj.lang.*;

// Logging using JDK1.4 from the AspectJ mailing List
// with minor modifications
public abstract aspect JDK14Tracing extends Tracing {

    protected abstract pointcut tracingScope ();

    protected pointcut shouldTrace () : tracingScope();
        // if(tracingEnabled) && tracingScope();

    public final static Level ENABLED = Level.FINER;
    public final static Level DISABLED = Level.OFF;

    private static boolean tracingEnabled = false;
    private Logger logger;

    protected void initLogger (String name) {
        logger = Logger.getLogger(name);
        if (!tracingEnabled) {
            tracingEnabled = isTracingEnabled(logger);
        }
    }

    public static boolean isTracingEnabled () {
        return tracingEnabled;
    }


    private static boolean isTracingEnabled (Logger logger) {
        return logger.isLoggable(ENABLED);
    }

    protected void enter (JoinPoint jp) {
```

```
            if (isTracingEnabled(logger)) {
                Signature signature = jp.getSignature();
                logger.entering(
                    signature.getDeclaringTypeName()
                    ,signature.getName(),jp.getArgs());
            }
        }

        protected void enter (JoinPoint jp, Object obj) {
            if (isTracingEnabled(logger)) {
                Signature signature = jp.getSignature();
                logger.entering(
                    signature.getDeclaringTypeName()
                    ,signature.getName(),jp.getArgs());
            }
        }

        protected void exit (JoinPoint.StaticPart sjp, Object
            ret) {
            if (isTracingEnabled(logger)) {
                Signature signature = sjp.getSignature();
                logger.exiting(signature.getDeclaringTypeName()
                    ,signature.getName(),ret);
            }
        }

        protected void exception (JoinPoint.StaticPart sjp,
            Exception ex) {
            if (isTracingEnabled(logger)) {
                Signature signature = sjp.getSignature();
                logger.exiting(signature.getDeclaringTypeName()
                    ,signature.getName(),ex);
            }
        }
}
```

Listing C.9: Base Log4j Tracing

```
package nz.co.solnet.infrastructure.tracing;

import org.apache.log4j.*;
import org.aspectj.lang.*;

// Logging using Log4J from the AspectJ mailing List
// with minor modifications
public abstract aspect Log4jTracing extends Tracing {

      protected abstract pointcut tracingScope ();

      protected pointcut shouldTrace () :
            if(tracingEnabled) && tracingScope();

      public final static Level ENABLED = Level.DEBUG;
      public final static Level DISABLED = Level.OFF;

      private static boolean tracingEnabled = false;
      private Logger logger;

      protected void initLogger (String name) {
            logger = Logger.getLogger(name);
            if (!tracingEnabled) {
                  tracingEnabled = isTracingEnabled(logger);
            }
      }

      protected void initLogger (Class clazz) {
          logger = Logger.getLogger(clazz);
          if (!tracingEnabled) {
                tracingEnabled = isTracingEnabled(logger);
          }
      }

      public static boolean isTracingEnabled () {
            return tracingEnabled;
      }
```

```
private static boolean isTracingEnabled (Logger logger) {
      return logger.isEnabledFor(ENABLED);
}

protected void enter (JoinPoint jp) {
      if (isTracingEnabled(logger)) {
            Signature signature = jp.getSignature();
            logger.debug(signature.getDeclaringTypeName()
              + "." + signature.getName() + " - start");

      }
}

protected void enter (JoinPoint jp, Object obj) {
      if (isTracingEnabled(logger)) {
            Signature signature = jp.getSignature();
            logger.debug(signature.getDeclaringTypeName()
              + "." + signature.getName() + " - start");
      }
}

protected void exit (JoinPoint.StaticPart sjp, Object
   ret) {
      if (isTracingEnabled(logger)) {
            Signature signature = sjp.getSignature();
            logger.debug(signature.getDeclaringTypeName()
              + "." + signature.getName() + " - end");
      }
}

protected void exception (JoinPoint.StaticPart sjp,
   Exception ex) {
      if (isTracingEnabled(logger)) {
            Signature signature = sjp.getSignature();
            logger.debug(signature.getDeclaringTypeName()
              + "." + signature.getName() + " - end");
      }
}
```

```
}
```

Listing C.10: Pertype JDK Tracing

```
package nz.co.solnet.infrastructure.tracing;

// Logging using JDK1.4 from the AspectJ mailing List
// with minor modifications
// Creates a separate logger for each class
public abstract aspect PTWJDK14Tracing extends JDK14Tracing
    pertypewithin(*) {

        before(): staticinitialization(*) && tracingScope() {
                String name = thisJoinPointStaticPart.getSignature()
                                                .getDeclaringTypeName();
                initLogger(name);
        }

}
```

Listing C.11: Pertype Log4j Tracing

```
package nz.co.solnet.infrastructure.tracing;

// Logging using Log4J from the AspectJ mailing List
// with minor modifications
// Creates a separate logger for each class
public abstract aspect PTWLog4jTracing extends Log4jTracing
    pertypewithin(*) {

    before(): staticinitialization(*) && tracingScope() {
            Class clazz = thisJoinPointStaticPart.getSignature()
                                                .getDeclaringType();
            initLogger(clazz);
    }

}
```

C.2 SPER Aspects

Listing C.12: Sper Service Wrapper

```
package
   nz.govt.nzqa.sper.infrastructure.servicewrapper.servicebeans;

import nz.co.solnet.infrastructure.bof.ServiceBeanWrapper;
import nz.govt.nzqa.sper.infrastructure.common.servicebeans.*;
import nz.co.solnet.infrastructure.bof.ServiceBaseBean;
import nz.co.solnet.infrastructure.bof.oids.OID;
import nz.govt.nzqa.sper.entry.common.dto.EntryDto;
import nz.govt.nzqa.sper.learner.common.dto.LearnerDto;
import java.util.List;

/**
 * Provide access to sper service beans that are not external
 * interfaces.
 * Provide advice for service methods that make a comment.
 * @author Chris Elgar
 */
public aspect SperServiceWrapper extends ServiceBeanWrapper {

    public pointcut serviceBeanMethods():
       ServiceBeanPointcuts.serviceBeanMethods();
    public pointcut excludedServiceBeanMethods():
       ServiceBeanPointcuts.excludedServiceBeanMethods();

    //Use of begin(String comment)
    before(ServiceBaseBean bean) :
       ServiceBeanPointcuts.excludedMethod1() && this(bean) {
        EntryDto entryDto = (EntryDto)
           thisJoinPoint.getArgs()[0];
        begin(bean, entryDto.getReasonForChange());
    }

    //Use of begin(String comment)
    before(ServiceBaseBean bean) :
       ServiceBeanPointcuts.excludedMethod2() && this(bean) {
        List inputOids = (List) thisJoinPoint.getArgs()[0];
```

```
        Long batchNumber = (Long) thisJoinPoint.getArgs()[3];
        begin(bean, "Processing input record OID " +
            inputOids.get(0) + " of batch " + batchNumber);
    }

    //Use of begin(String comment)
    before(ServiceBaseBean bean) :
        ServiceBeanPointcuts.excludedMethod3() && this(bean) {
        OID batchOid = (OID) thisJoinPoint.getArgs()[1];
        begin(bean, "Preprocessing learner of batch " +
            batchOid);
    }

    //Use of begin(String comment)
    before(ServiceBaseBean bean) :
        ServiceBeanPointcuts.excludedMethod3() && this(bean) {
        LearnerDto learnerDto = (LearnerDto)
            thisJoinPoint.getArgs()[0];
        begin(bean, learnerDto.getUserComment());
    }
}
```

Listing C.13: Sper Exception Handler

```
package nz.govt.nzqa.sper.infrastructure.exceptionhandling
                                                .servicebeans;

import nz.co.solnet.infrastructure.exceptionhandling.*;
import nz.govt.nzqa.sper.infrastructure.common.servicebeans.*;
import nz.govt.nzqa.qual.qsi.common.exception.*;
import java.io.IOException;
import java.rmi.RemoteException;

// Link SPER to the Exception Handling aspect
public aspect SperExceptionHandler extends ExceptionHandler {

    // Application exceptions that are dealt with by aspects
    declare soft: RemoteException:
       ServiceBeanPointcuts.serviceBeanMethods();
    declare soft: IOException:
       ServiceBeanPointcuts.serviceBeanMethods();
    declare soft: QSIException:
       ServiceBeanPointcuts.serviceBeanMethods();

    public pointcut serviceBeanMethods() :
       ServiceBeanPointcuts.serviceBeanMethods();

}
```

Listing C.16: SXI Exception Handler

```
package nz.govt.nzqa.sper.infrastructure.exceptionhandling
                                                .servicebeans;

import nz.govt.nzqa.sper.sxi.exception.SXIException;
import nz.govt.nzqa.sper.sxi.exception.*;
import nz.govt.nzqa.sper.sxi.exception.*;
import nz.co.solnet.infrastructure.bof.helpers.*;
import nz.co.solnet.infrastructure.security.*;
import nz.govt.nzqa.sper.infrastructure.common.servicebeans.*;

public aspect SXIExceptionHandler {

    declare soft: Exception: SXIServicePointcuts.mutators()
                           || SXIServicePointcuts.accessors();

    Object around() throws SXIException
                    : SXIServicePointcuts.mutators()
                    || SXIServicePointcuts.accessors() {

        Object result = null;
        try {
            result = proceed();
        }
        catch(AuthorisationException e) {
            throw new SXIAuthorisationException(e);
        }
        catch(OptimisticConcurrencyException e) {
            throw new SXIOptimisticConcurrencyException(e);
        }
        catch(SXIException e) {
            throw e;
        }
        catch(Exception e) {
            throw new SXIException(e);
        }
        return result;
    }
```

```
}
```

Listing C.17: Sper Aspect Precedence

```
package nz.govt.nzqa.sper.infrastructure.common.servicebeans;

import nz.govt.nzqa.sper.infrastructure.exceptionhandling
                                        .servicebeans.*;
import nz.govt.nzqa.sper.infrastructure.servicewrapper
                                        .servicebeans.*;
import nz.govt.nzqa.sper.infrastructure.transactionhandling
                                        .servicebeans.*;

// Aspect to ensure that we apply aspects to the service beans
// in an appropriate order for SPER
public aspect AspectPrecedence {

    declare precedence: ExceptionRollback,
                        SperExceptionHandler,
                        SperServiceWrapper;

    declare precedence: SXIServiceTransactionHandling,
                        SXIExceptionHandler,
                        SXIServiceWrapper;
}
```

Listing C.18: Sper Pointcuts

```
/**
 * Provides common pointcuts for all users needing access to
 * the sper service beans.
 * @author Chris Elgar
 */
package nz.govt.nzqa.sper.infrastructure.common.servicebeans;

import nz.co.solnet.infrastructure.bof.ServiceBaseBean;
import nz.govt.nzqa.sper.sxi.service.impl.SXIServiceBean;

import nz.govt.nzqa.sper.learner.common.dto.LearnerKeysDto;
import nz.govt.nzqa.sper.learner.common.dto.LearnerDto;
import nz.govt.nzqa.sper.entry.common.dto.EntryDto;
import nz.govt.nzqa.perorg.provider.common.dto.ProviderDto;
import nz.co.solnet.infrastructure.bof.oids.OID;
import nz.govt.nzqa.sper.common.codetable.BatchType;

import java.util.List;

public aspect ServiceBeanPointcuts {

  public static pointcut serviceBeans():
     (within(nz.govt.nzqa.sper..*)
                      || within(nz.govt.nzqa.perorg..*))
                     && within(ServiceBaseBean+)
                     && !within(SXIServiceBean);

  public static pointcut serviceBeanMethods(): serviceBeans()
                       && execution(public !static * *.*(..));

  public static pointcut excludedServiceBeanMethods():
     excludedMethod1()
                             || excludedMethod2()
                             || excludedMethod3()
                             || excludedMethod4();

  public static pointcut excludedMethod1() : execution(public
     OID saveEntry(EntryDto,OID,OID,boolean));
```

```
    public static pointcut excludedMethod2() : execution(public
        void processInput(List,String,ProviderDto,Long,BatchType));
    public static pointcut excludedMethod3() : execution(public
        void preProcessInput(List,OID));
    public static pointcut excludedMethod4() : execution(public
        LearnerKeysDto saveLearner(LearnerDto,boolean));

}
```

Listing C.19: SXI Pointcuts

```
package nz.govt.nzqa.sper.infrastructure.common.servicebeans;

import java.util.Collection;
import nz.govt.nzqa.sper.sxi.service.impl.SXIServiceBean;

// Pointcuts for SXI interface
public aspect SXIServicePointcuts {

    public static pointcut servicemethods() : mutators() ||
       accessors();

    public static pointcut ejbs() : within(SXIServiceBean);

    public static pointcut mutators() : ejbs()
                              && (
                                      updatemethods()
                                      || savemethods()
                                      || storemethods()
                                      || createmethods()
                                      || includedmutators()
                              )
                              && !excludedmutators();

    public static pointcut accessors() : ejbs()
                              && (getmethods() ||
                                  includedaccessors())
                              && !excludedaccessors();

    public static pointcut getmethods() : execution(public
       !static * *.get*(..));
    public static pointcut includedaccessors():
       execution(public void doQualCheck(..))
                                          || execution(public
                                             boolean ping());
    public static pointcut excludedaccessors(): execution(public
        void generateLearnerChangeDetailsReport(..));
```

```
    public static pointcut updatemethods() : execution(public
       !static * *.update*(..,String));
    public static pointcut savemethods() : execution(public
       !static * *.save*(..,String)) ;
    public static pointcut storemethods() : execution(public
       !static * *.store*(..,String));
    public static pointcut createmethods() : execution(public
       !static * *.create*(..,String));
    public static pointcut excludedmutators() : execution(public
        Collection
       storeCompassionateEntries(Collection,String));
    public static pointcut includedmutators() : execution(public
        * withdrawResult(..));


}
```

C.3 EOS Aspects

Listing C.20: EOS Service Wrapper

```
package nz.govt.nzqa.eos.infrastructure.servicewrapping;

import nz.co.solnet.infrastructure.bof.ServiceBeanWrapper;
import nz.govt.nzqa.eos.infrastructure.common.*;

// Aspect links EOS to library aspects
// for service wrapping
public aspect EOSServiceWrapper extends ServiceBeanWrapper {

    public pointcut serviceBeanMethods():
        ServiceBeanPointcuts.serviceBeanMethods();
    public pointcut excludedServiceBeanMethods();
}
```

Listing C.21: EOS Exception Handler

```
package nz.govt.nzqa.eos.infrastructure.exceptionhandling;

import nz.co.solnet.infrastructure.exceptionhandling.*;
import nz.govt.nzqa.eos.infrastructure.common.*;
import nz.co.solnet.infrastructure.bof.*;
import nz.govt.nzqa.eos.service.common.ejb.*;

// Aspect that specifies the EOS specific
// exception handling components
public privileged aspect EOSExceptionHandler extends
    CustomExceptionHandler {

    public pointcut serviceBeanMethods():
        ServiceBeanPointcuts.serviceBeanMethods();

    public void exceptionHandler(ServiceBaseBean bean, Exception
        e) {
        EosBaseSessionFacade eosbean = (EosBaseSessionFacade)
            bean;
        throw eosbean.convertException(e);
    }

}
```

Listing C.22: EOS Aspect Precedence

```
package nz.govt.nzqa.eos.infrastructure.common;

import nz.govt.nzqa.eos.infrastructure.exceptionhandling.*;
import nz.govt.nzqa.eos.infrastructure.servicewrapping.*;
import nz.govt.nzqa.eos.infrastructure.tracing.*;

// Ensure aspects are applied to EOS
// in the correct order
public aspect AspectPrecedence {

    declare precedence: EOSTracing,
                        EOSExceptionHandler,
                        EOSServiceWrapper;
}
```

Listing C.23: EOS Pointcuts

```
package nz.govt.nzqa.eos.infrastructure.common;

// Pointcuts for the EOS application
public aspect ServiceBeanPointcuts {

    public static pointcut eosServiceBeans() :
        within(nz.govt.nzqa.eos.service.*.ejb.bean.*);

    public static pointcut serviceBeanMethods() :
        eosServiceBeans() && execution(public !static * *(..));
}
```

Listing C.24: EOS Tracing

```
package nz.govt.nzqa.eos.infrastructure.tracing;

import nz.co.solnet.infrastructure.tracing.PTWLog4jTracing;
import nz.govt.nzqa.eos.infrastructure.common.*;

// Aspect to link EOS to tracing library aspect
public aspect EOSTracing extends PTWLog4jTracing {

    protected pointcut tracingScope() :
        ServiceBeanPointcuts.eosServiceBeans();

}
```