Evaluation of Aspect-Oriented Software Development For Distributed Systems

Cormac Driver

A dissertation submitted to the University Of Dublin in partial fulfilment of the requirements of the degree of Master of Science in Computer Science

September 16, 2002

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: ___________________

Date: September 16, 2002

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _________________

Date: September 16, 2002

Acknowledgments

I would like to thank my supervisor, Dr. Siobhán Clarke, for all her guidance and assistance, and especially for taking such a dedicated interest in my work.

Thanks and appreciation go to those classmates who helped me out along the way - Howard Kim for his insights on my field of study and Arkaitz Bitorika for his enlightening critiques and technical support.

Much thanks goes to my family for putting me where I am, and to my friends for all their inspiration.

Finally, gratitude in abundance to Annie for doing what she does.

Abstract

Problems relating to the issue of modularity result in the underperformance of the object-oriented software development paradigm. Aspect-oriented software development (AOSD) is a new technology for the separation of crosscutting concerns in computer software. A crosscutting concern arises in a software system when the implementation of a system requirement impacts on more than one implementation module. It is proposed that AOSD techniques can modularise crosscutting concerns that are inherent in object-oriented software systems, resulting in software with greater reusability, evolvability and comprehensibility.

The objective of this dissertation is to evaluate the performance of the current AOSD state of the art when applied to distributed systems. Aspect-oriented design and implementation languages/tools are used to re-implement an existing object-oriented application called AppTrack. AppTrack is a web-based information system implemented using Java, an object-oriented programming language.

The re-implementation process required the identification, design and implementation of crosscutting concerns in the AppTrack codebase. These concerns were re-implemented using two of the most prominent aspect-oriented programming (AOP) languages currently available – AspectJ and Hyper/J. From the information gathered throughout this process it is concluded that indeed AOSD does deliver significantly on its major promises, the most important being the increase in the degree of modularity evident in a software system. However, it is far from a silver bullet solution to the problem of crosscutting concerns in object-oriented software, and much future work on the area is necessary.

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

Software engineers have long been aware of the need to make software less complex. Reducing complexity can result in such benefits as maintainability, evolvability, tracability and comprehensibility. A key concept in reducing the overall complexity levels in computer software is *modularity*. By keeping related modules together and separating them from modules addressing unrelated issues, software systems can evolve freely without establishing restrictive dependencies. The practice of dividing different areas of interest into separate, independent modules is widely referred to as *separation of concerns* [1,2]. It has been established that total separation of concerns is not possible with the current standard programming paradigms [3].

Object-oriented programming is the current standard paradigm for software development. Promoters of the paradigm claim that it can fundamentally aid software engineering by creating architectures that better fit with domain models. Whilst this is partially true, the whole story is not being told. There are many software development problems to which a suitable solution cannot be achieved with object-orientation. It is problems relating to the issue of modularity that result in the underperformance of the object-oriented model. There is a need for a new concept to address these shortcomings.

Aspect-oriented software development (AOSD) is a new technology for separation of concerns in software development [4]. The AOSD community propose that it is possible to modularise crosscutting aspects of a system using AOSD techniques. In this context a crosscutting aspect can be thought of as a requirement in a software system that involves affecting more than one implementation module (most commonly the object-oriented *class* or *method*) during its implementation. Applying AOSD techniques can lead to a system that exhibits cleanly captured concerns within its codebase and possesses the beneficial properties mentioned earlier. The aim of this dissertation is to evaluate aspect-oriented software development techniques when applied to a distributed information system called AppTrack.

1.1 AppTrack

AppTrack is a web-based distributed information system that was developed by a group from the TCD MSc. Networks and Distributed Systems class of 2002, including myself. The main requirement of the system was to automate the postgraduate applications system, making it possible for students and course staff to conduct the administration of the applications process online. The system was written using Sun Microsystems' object-oriented programming language, Java.

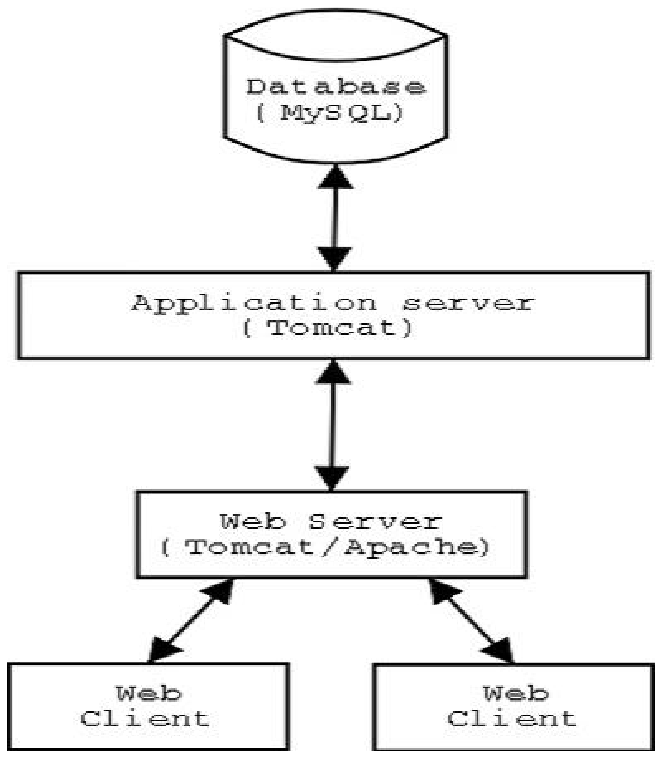

Figure 1: AppTrack application architecture

Figure 1 illustrates a high-level overview of the AppTrack application architecture. The application code residing on the Tomcat server was written using the Struts framework. Struts is an open-source framework from the Jakarta project for building web applications. The framework encourages application architectures based on the Model 2 approach, a variation of the classic Model-View-Controller (MVC) design pattern. The use of the Struts framework enforces rules on the application code i.e. classes must implement specific interfaces. Whilst conformance to the Struts framework leads to a good degree of modularity, there remain some concerns (non-MVC related) that are not cleanly separated. AppTrack is composed of 81 Java classes and 20 Java Server Pages.

1.2 Project Goals

The major goal of this dissertation is the evaluation of aspect-oriented software development as a technique for achieving better separation of concerns in distributed systems. Achieving this goal required the following actions:

- Gaining a full understanding of the need for separation of concerns
- Understanding how AOSD can help increase modularity
- Re-implementing the existing AppTrack system using AOSD techniques
- Assessing AOSD in relation to current standards

1.3 Dissertation Roadmap

The remaining chapters of this dissertation are laid out as follows:

Chapter 2 – State of the Art

This chapter provides an overview of the current leading technology in the areas of separation of concerns and aspect-oriented software development. It is elements of this technology that are used during the implementation phase. This chapter also introduces and discusses other research that is relevant to this project.

Chapter 3 – Design

This chapter details the identification of crosscutting concerns in the AppTrack codebase and the design of an aspect-oriented software solution for each concern.

Chapter 4 – Implementation

This chapter explains the implementation of the aspects designed in the preceding chapter. Each aspect is implemented using two markedly different AOSD techniques.

Chapter 5 – Evaluation

The implementation phase is evaluated in order to determine the advantages and disadvantages of the chosen solution(s) relative the existing solution. This chapter records the evaluation process.

Chapter 6 – Conclusions

The final chapter draws conclusions regarding the success of the completed work in relation to the original stated goals of the project. The future and worth of AOSD are discussed, as well as possible future work on the project.

2 STATE OF THE ART

AOSD is a new technology for the separation of concerns in software development [4]. Separation of concerns in software has long been the goal of many notable computer scientists such as Dijkstra, Parnas and Jackson. Promoters of AOSD claim that by using this new technology it is possible to fully modularise the crosscutting concerns (or aspects) of a software system.

A crosscutting concern arises in a software system when the implementation of a system requirement impacts on more than one implementation module (most commonly the *class* or *method* in Object-Oriented Programming). Crosscutting concerns are also known as *aspects* [3]. More often than not, aspects tend not to be implementations of requirements that come straight from the initial requirements document but rather properties that affect the system in a non-functional way. Classic examples of aspects include codebase tracing and synchronisation of concurrent objects.

Object-Oriented code suffers from two phenomena known as *scattering* and *tangling* [3]. Scattering is evident when similar code is distributed throughout many system modules, with the risk of misuse and inconsistencies at each point of use. Changes to the system's implementation may require finding and editing all scattered code belonging to an aspect. Tangling occurs when two or more concerns are implemented in the same implementation module (class/method), making the module harder to understand and change. The word crosscutting is used to characterise a concern that spans multiple units of Object-Oriented modularity. Crosscutting concerns inevitably evolve through the use of current programming methodologies, resulting in the development of systems that are harder to design, understand, implement, maintain and evolve. AOSD aims to allow system design documents (usually diagrams) and implementation code to be structured to reflect the way developers want to think about a system [5]. There are currently three main approaches being promoted as the solution to clearly expressing programs that involve crosscutting concerns. These are:

1. The Aspect Approach [3]
2. The Hyperspace approach (Multi-Dimensional Separation of Concerns) [6]
3. The Composition Filters [7] Approach

Each of the areas above is supported by numerous tools and languages that combine to establish a state of the art for AOSD. There are both major and minor players in each area and these are discussed in the following subsections.

2.1 Approaches To Aspect-Oriented Software Development

Three main approaches to AOSD have emerged in recent years. This section will discuss the prominent technologies supporting Aspects, Hyperspaces and Composition Filters.

2.2 Aspects

The Aspect approach was first presented in [3]. A system property or requirement that must be implemented is an aspect if it cannot be cleanly encapsulated in a component that is well localised, easily accessed and composed. The goal of aspect-oriented programming (AOP) as stated in [3] is to support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system. This goal is achieved by adding an *aspect language* and an *aspect weaver* to the set of standard tools used in the implementation of a software system. The aspect language is a separate language to the component language (Java, C++ etc.) and is used to program the aspects. The aspect weaver is used to integrate aspects written in the aspect language with the rest of the program code, which is written in the component language. The aspect weaver is a tool that accepts both component and aspect language as input and outputs the combination of the two in component language. The most prominent language supporting the aspect approach is AspectJ [8], which is discussed below.

2.2.1 AspectJ

AspectJ is an example of an aspect language. It is an aspect-oriented extension to Java that supports general-purpose aspect-oriented programming [9]. The word *aspect* means two things in AspectJ, depending on the context in which it is stated. At the design level it is a concern that crosscuts (as previously described). At the implementation level it is a programming construct that enables crosscutting concerns to be implemented in a modular fashion. This programming level aspect is the dominant unit of decomposition in the AspectJ programming language.

2.2.1.1 Language Constructs

AspectJ is based on a concept referred to as the *dynamic joinpoint.* Dynamic joinpoints are points in the execution of a Java program such as method calls and access to class attributes. The joinpoint is the main new concept that AspectJ adds to Java, with most of the rest of the language being made up of simple constructs based around this notion. *Pointcuts* and *advice* dynamically affect program flow whereas *introduction*, another new concept, statically affects a program's class hierarchy. Pointcuts select certain joinpoints and values at those points. Advice defines code that is executed when a joinpoint is reached. This is the dynamic behaviour of AspectJ. AspectJ uses introduction to modify a program's static structure, the members of its classes and the relationships between those classes by either introducing new member variables and methods into a class or by defining new parents for an existing class/defining that a class implements another class. The last new construct in AspectJ is the aspect itself. Aspects are AspectJ's unit of modularity for crosscutting concerns and they are defined in terms of pointcuts, advice and introduction.

Below is a simple aspect class that shows how using AspectJ it is possible to log method calls at runtime and print this information to the terminal.

```
aspect MethodLogger {
      private String name;
      private String kind;
      private String signature;

      pointcut trace() : within(application..*) && execution(* *(..))
         && !within(application.aspect.log.*);

      before() : trace() {
         name = thisJoinPoint.getSignature().getName();
         kind = thisJoinPoint.getKind();
         signature = "" + thisJoinPoint.getSignature();
         System.out.println.setName(name);
         System.out.println.setKind(kind);
         System.out.println.setSignature(signature);
      }
}
```

The pointcut `trace()` defines a pointcut that picks out the execution of every method within every package including and contained within the specified package but not within the package containing the logging aspect itself. On the execution of each method a piece of code, known as advice, is executed. In this case the code is executed before the execution of the method currently picked out by the joinpoint. The body of this `before()` advice uses the static variable `thisJoinPoint` to retrieve information about the method currently executing. The information is then printed to the terminal screen. The `thisJoinPoint` variable offers context specific information about the currently executing joinpoint as identified by the pointcut. It can be thought of as similar to the `this` variable in standard Java.

What this simple aspect shows is that using the constructs outlined previously it is possible to get the same end result by writing one aspect class that you would get by scattering code among many classes. Of course you also get the added benefits that come with separating concerns in your codebase such as maintainability, evolvability, pluggability and reusability.

2.2.1.2 Tools

The AspectJ development team have developed three main tools to support the use of the language. `ajc` is the compiler that combines Java and AspectJ source files to create standard Java class files. This means that a program written with AspectJ can run on a standard JVM and ensures that any AspectJ program is a legal Java program and vice versa. `ajdb` is a debugger for AspectJ programs. This tool has all the capabilities of a standard Java debugger. In addition, the AspectJ working directory can be specified and the breakpoint command, stop on, has been extended to allow the setting of a breakpoint on an aspect source file line. `ajdoc` generates HTML API documentation for pointcuts, advice and intertype declarations, as well as the Java constructs that javadoc renders. `ajdoc` links both advices from members affected by the advice and inter-type declarations for members declared from aspects.

2.2.1.3 GUI

The AspectJ browser is a GUI tool for compiling programs with the `ajc` and navigating the crosscutting structure. It is essentially a development tool that supports AspectJ's aspects and can be used to edit source files and configuration build files.

The AspectJ team have developed plug-ins for various widely used IDEs including Eclipse, JBuilder, Forte and Emacs. These plug-ins all offer similar functionality such as AspectJ compilation from within the development environment, browsing of the aspect structure and establishment of a compile configuration for specifying which files are to be passed to the compiler. This allows for the simple integration of pluggable aspects with your codebase. The learning curve when adopting the language is greatly reduced through the use of these tools because they allow you to see the areas of the codebase that your pointcuts affect. This means you can ascertain (to a certain extent) whether or not you are achieving the desired affects without running the application and observing its behaviour.

2.2.1.4 Performance

In many cases the use of AspectJ's aspects will reduce the number of lines of code needed to implement a system. This is a side effect and was not an initial goal of the language. The major benefit of using aspects is not the reduced number of lines but the improved modularity. Aspects gather code into a single module that would otherwise be redundantly duplicated throughout the codebase. This lack of duplication is the reason for the reduction in code size.

There is currently no benchmark against which aspect-oriented programming languages can be compared. It is too early for the development of important benchmarks as languages are still maturing and no one coding style is prominent. In the absence of benchmarks it can be said that AspectJ displays acceptable performance for everything except non-static advice (this is because the `ajc` compiler must insert more checks into the weaved code to cater for the this type of advice). Introduction and static advice have an extremely small overhead compared to doing the same thing by hand.

It has been suggested that AspectJ violates encapsulation [12]. Whilst this is definitely true the developers have answered this criticism by stating that they have given more power to the language initially (in the spirit of Smalltalk [11]) to let the user community experiment and discover what is right. Currently this strategy has led to the creation of many useful aspects that crosscut the internal state of an object but the developers have not ruled out future restrictions [12].

2.2.2 Other Tools And Languages

Whilst AspectJ is without doubt the most popular and widely adopted aspect language, this section will outline some of the others currently available.

2.2.2.1 Aspect Languages

AspectR [13] is an AOP implementation for the Ruby programming language, letting you wrap code around existing methods in your classes. AspectS is an early prototype that enables AOP in the Squeak/Smalltalk environment. Apostle [15] is another extension of the Smalltalk language with the aim of providing AOP functionality. It is a port of AspectJ to Smalltalk, sporting a similar language model with joinpoints, pointcuts, advice and aspects. MixJuice [14] is an extension of the Java language that adopts difference-based modules instead of Java's original module mechanism. This means that crosscutting concerns can be modularised, developed and tested independently. ArchJava [17] is an extension to the Java language with explicit software architecture constructs, separating out structural concerns that would otherwise crosscut the system. This language is significantly different to AspectJ. Pythius [18] is an open-source project to support aspect-oriented programming techniques in the Python programming language. Pythius is a set of tools to assess the quality of Python code by using AOP to implement simple metrics on Python source code. AspectC [19] and AspectC++ [20] are prototype AOP implementations for C and C++ respectively. AspectC is a simple extension to C based on AspectJ. AspectC++ is a more substantial language, also heavily based on the constructs introduced by AspectJ. The preceding two AOP languages are of some importance to the success of AOP and AOSD as the majority of software systems are still being written in C and C++.

2.2.2.2 Aspect Tools

The most important aspect related tools that have been developed thus far support the identification of aspects in existing codebases. Many programmers find the idea of AOSD interesting but don't know where or how to start actually applying the techniques. The Aspect Mining Tool (AMT) [21] and the Feature Exploration and Analysis Tool (FEAT) [22] are two tools that have a similar function. Both tools provide a graphical interface for identification, location and analysis of aspects in a codebase. This graphical functionality makes concern location a lot easier than trawling through large amounts of source code, making AOSD more accessible to beginners.

Despite their worth, these tools were not employed during for the aspect discovery work on the project being described in this dissertation. After some experimentation with these tools it was decided that their involvement was not necessary. The main reasons for arriving at this decision are that the codebase was reasonably compact and I was quite familiar with it having been on the initial development team. This negated the necessity for such tools.

2.3 Multi-Dimensional Separation of Concerns and Hyperspaces

Multi-Dimensional Separation of Concerns (MDSOC) and the hyperspace model are presented in [6]. This approach tackles the problems with existing popular software development techniques but in a markedly different way to the aspect approach.

Initial attempts at separation of concerns led to increased modularisation. These efforts were carried out with the intention of alleviating expensive change as well as obstacles to reuse and component integration. The key reason that past modularisation efforts such as object orientation did not fully achieve these goals is the fact that you need different decompositions according to different concerns at different times. However, most languages and modularisation approaches support only one kind of modularisation. This unit of modularisation is referred to as the *dominant decomposition*. Once a system has been decomposed by the dominant decomposition, extensive refactoring is needed to remodularise it. MDSOC and hyperspaces seeks to address the problems caused by the use of a dominant decomposition in system modularisation. MDSOC allows for simultaneous separation according to multiple, arbitrary kinds (dimensions) of concern, with *on-demand remodularisation*.

Concerns are the reason for organising software into modules that are understandable and manageable. Many different kinds of concerns can be relevant to different developers in different roles, at different stages of the software lifecycle. Data and classes can be concerns, as can features such as persistence and aspects such concurrency control. In MDSOC terminology, a kind of concern is referred to as a *dimension* of concern. Hence, separation of concerns involves decomposition of software according to one or more dimensions of concern. Separation along one dimension of concern may promote some goals whilst impeding others. It is difficult to discover the relevant set of concerns to separate as they are context sensitive and vary over time. This means that any set of criterion for decomposition and integration will be appropriate for some set of requirements but not for all. Additionally, multiple dimensions of concern may be relevant simultaneously, and they may overlap and interact, as features and classes do. Thus, modularisation according to different dimensions of concern is needed for different purposes: sometimes by class, sometimes by feature, sometimes by aspect or other criterion. These considerations imply that developers must be able identify, encapsulate, modularise and manipulate multiple dimensions of concern simultaneously, and to introduce new concerns and dimensions at any point during the software lifecycle without suffering the effects of invasive modification and rearchitecure. As mentioned earlier, modern languages suffer from a problem termed the "tyranny of the dominant decomposition" [23] - they permit separation and encapsulation of only one kind of concern at a time. It is therefore impossible to obtain the benefits of different decomposition dimensions throughout the software lifecycle. Changing the unit of modularisation during the evolution of a system can have disastrous effects, some languages don't even let you change your dominant decomposition e.g. you must use classes in Object-Oriented programming.

The term MDSOC is used to denote separation of concerns involving the following:

- Multiple, arbitrary dimensions of concern
- Separation along these dimensions simultaneously
- The ability to handle new concerns, and new dimensions of concern, dynamically
- Overlapping and interacting concerns[1]

[1] It is possible to get the impression that most concerns are independent, this is often not the case in practice

Full support for MDSOC opens the door to *on-demand remodularisation*, allowing a developer to choose at any time the best modularisation, based on any or all of the concerns, for the development task at hand.

2.3.1 Hyperspaces

Hyperspaces permit the explicit identification of any concerns of importance, encapsulation of those concerns, identification and management of relationships among those concerns, and integration of concerns. The primary goal of hyperspaces is to limit the impact of change and simplify evolution.

A concern space is made up of all units in some body of software, where units can be *primitive* or *compound*. A primitive unit is considered to be atomic e.g. an instance variable, a method or a performance requirement. A compound unit is composed of primitive units e.g. a class or collaboration diagram. The job of the concern space is to organise the units in a body of software so as to separate all important concerns, describe relationships between concerns and to indicate how they can be composed together. There are three distinct elements to separation of concerns: identification, encapsulation and integration.

A hyperspace is a concern space characterised by a number of unique features. Firstly, all its units are organised in a multi-dimensional matrix. Each axis represents a dimension of concern, and each point on an axis represents a concern in that dimension. This structure makes explicit all the dimensions of interest, the concerns that belong to each dimension, and which concerns are affected by which units. The coordinates of a unit indicate which concerns it affects; the structure clarifies that each unit affects exactly one concern in each dimension. Each dimension can be viewed as a partition of the set of units - a particular software decomposition.

Identification of concerns is achieved in practice by declaring the packages that contain the concerns in a hyperspace file. Any package or class within this file becomes part of the conceptual hyperspace and is available to be manipulated when separating concerns.

A hyperslice is a set of concerns that is declaratively complete, that is, it must explicitly declare everything to which it refers. Essentially, a hyperslice will capture a concern that exists in the system codebase. A concern mapping is used to declare the elements of the concern such as the classes, features and variables. Usually, each package named in the hyperspace specification file will itself become a distinct hyperslice.

A set of hyperslices cannot exist in isolation; concerns need to interact to make up a working system. The hypermodule comprises a set of hyperslices that are to be integrated and it contains a set of relationships used to bind the hyperslices together. The hypermodule essentially acts as an integrator of the separated concerns in your codebase.

2.3.2 Hyper/J

Hyper/J is a tool that implements hyperspaces. It currently supports the definition of units in only one language - Java. Hyper/J permits the identification, encapsulation and integration (through composition) of multiple dimensions of concern, and it realises the model of hyperspaces. Hyper/J supports visual, WYSIWYG specification of concerns in a project codebase. Hyper/J is a tool, there is no need for developers to learn a new language or use special-purpose compilers or virtual machines. The only new syntax that developers are required to learn is that of the hypermodule, hyperspace and concern mapping specifications. Hyper/J works on and generates standard Java. The current version of the tool does not implement the full hyperspace specification making the results achievable with Hyper/J somewhat limited. For example, to date the developers have worked only with units at the granularity of declarations (methods, classes) rather than lower-level constructs such as statements and expressions. Finer granularity is required to suitably capture some of the concerns identified in the AppTrack system. As well as this issue, some relationships required to implement crucial delegation functionality have not been implemented. This prohibits the implementation of AppTrack concerns that require delegation from existing methods to newly created proxy methods.

2.4 Composition Filters

The composition filters approach to AOSD was first proposed in [7]. Essentially, this approach seeks to solve the problems inherent in current techniques by using composition filters to abstract communications among objects. The basic object model is extended modularly by introducing input and output composition filters that affect the sent and received messages respectively. The main claim that this approach is making is that it is more beneficial to modularly extend the object-oriented model rather than build increasingly complex object structures [24].

Each message that arrives at an object is subject to evaluation and manipulation by the filters associated with that object. Filters support functions such as delegation, pre/post conditions, synchronisation and error detection. They have well-defined semantics and are open-ended, two properties that support the quality factors of verifiability and evolvability respectively. Filters can be added/attached to objects defined in object-oriented programming languages such as Java and Smalltalk without modifying the languages. This means that a non-language extension approach, similar to that used in Hyper/J, is adopted. A composition filters definition language is used to create the filters and an appropriate tool is used to map the filters onto the object-oriented language that being used as the primary development language.

2.4.1 ComposeJ

The result of combining the composition filters object model with Java is a language called ComposeJ [25]. The current implementation provides a command-line tool and a programming environment, which includes a user interface that allows the user to edit and translate Java files and composition filter definitions. The ComposeJ system is an add-on to a standard Java compiler.

2.4.2 Other Composition Filter Languages and Tools

ConcernJ [26] is a tool based on the composition filters model that allows for the super-imposition of composition filter specifications over a Java system. The work arose out of the need for a systematic way to apply filters application-wide. The current prototype demonstrates the transformation of language independent concern specifications and Java classes into executable composition filters and Java files. This concept is built upon the initial concept of composition filters.

2.5 Other Approaches to Aspect-Oriented Software Development

There are of course other approaches to AOSD. Another less prominent but still substantial approach is Adaptive Programming [27]. Essentially, adaptive programming is concerned with facilitating the expression of elements (classes, methods, variables) that are essential to an application by avoiding making a commitment on the particular structure of the application. Demeter/Java [28] is a high-level interface to Java that allows developers to write Java programs using the adaptive programming methodology. The approach is based on the Law of Demeter, hence the name. The Law of Demeter is a simple style rule for designing object-oriented systems. "Only talk to your immediate friends" is the motto.

2.6 Composition Patterns: Designing For Separation of Concerns

The above languages and tools address the issue of implementing systems free of code scattering and tangling. Before implementation must of course come design. Design is important for many reasons, including management of complexity, gaining a greater understanding and improving evolvability. Systems implemented using aspect-oriented development techniques are on the surface no different to systems developed with more widely used paradigms. However, when it comes to designing aspect-oriented systems the modelling techniques used for these standard paradigms are not sufficient. System requirements that suggest a crosscutting implementation can be separated at the design level using composition patterns [29]. Composition patterns were developed as a way of capturing reusable patterns of crosscutting behaviour at the design level. These patterns, an extension to the subject-oriented model, use extensions to the UML to define modelling language elements that can be used together to model requirements for aspect-oriented systems. The model seeks to be generic in that it can model requirements for aspect-oriented systems, regardless of the chosen implementation technique i.e. AspectJ, Hyper/J, Composition Filters etc.

There are three important models in the composition patterns specification. As a whole, these three models fully define a reusable concern. The models are the following:

1. Templates
2. Cross-Cutting Behaviour
3. Binding

A template is a parameterised model element that cannot actually be used directly in a concrete design model but acts as a placeholder for future design elements related to the template models. Actual units of modularity such as classes and methods can be bound to the template to generate usable models by the use of a special binding relationship. The binding relationship takes as parameters the actual parameters that are to replace the parameters in the template.

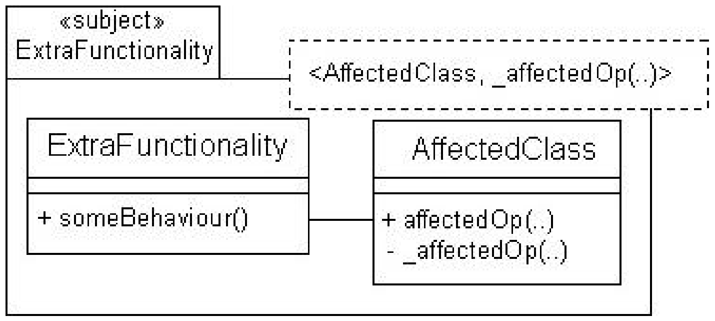

Figure 2: An example composition patterns template

Figure 2 is an example of a composition pattern. Each template is given a name and the parameters are contained within the dotted box. The presence of the `AffectedClass` class illustrates that any bound class can be augmented with extra functionality. The operation preceded with the underscore represents an operation that can be used to delegate to additional behaviour as defined in the composition pattern. In the example this additional behaviour is represented by the `someBehaviour()` method of the `ExtraFunctionality` class. The operation preceded with the plus sign represents the combination of both the standard method (underscored method) and the extra behaviour. Note that the `affectedOp(..)` method can in practice accept any parameters; this is denoted by the double dot notation.

Using AOSD techniques, concerns present in software applications can be untangled and separated. Using some aspect-oriented technique these separated concerns must be *weaved* back together, most usually during compilation. This weave relationship, the collaboration of untangled concerns, corresponds to the delegation of a method to some appropriate additional behaviour (as described above). The collaboration between the modules involved in the interaction can be shown with the composition patterns model using a diagram similar to a standard UML sequence/collaboration diagram.

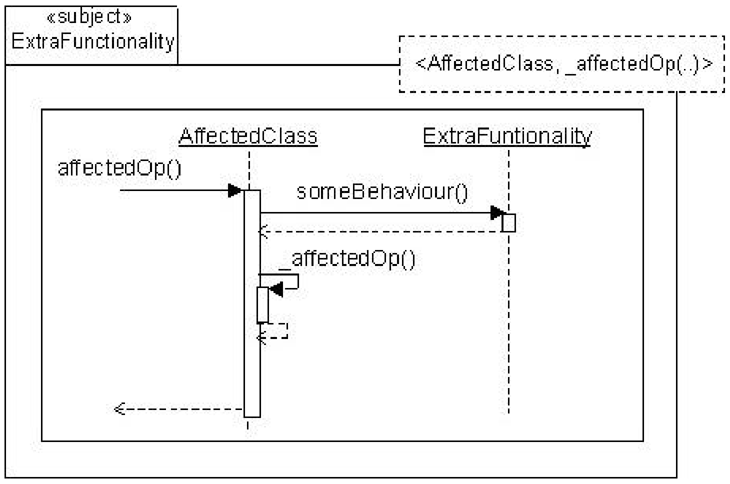

Figure 3: An example composition patterns collaboration diagram

Figure 3 shows the sequence of events that actually occur when the `affectOp()` method is executed.

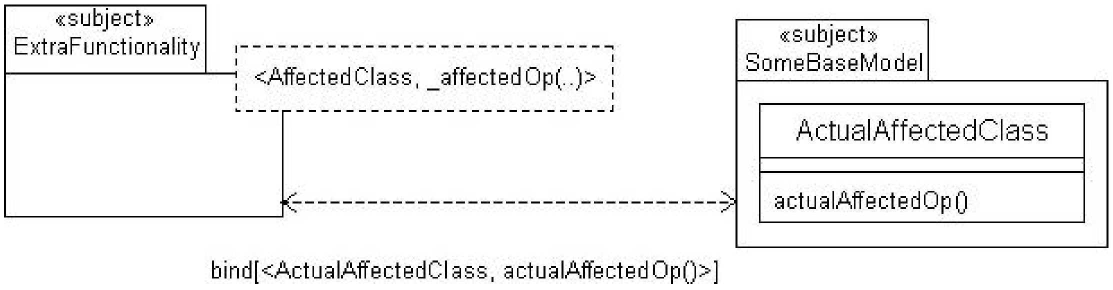

Figure 4: An example composition patterns composition relationship

Figure 4 shows that the class `ActualAffectedClass` and the operation `actualAffectedOp()` are the concrete class and method that will be augmented with the extra functionality. The bind attachment can define more than one method to be affected or more than one {class, method} pair.

Separation of concerns at the design level, and indeed throughout the software development lifecycle as a whole, has not received as much attention as the implementation level thus far. The Composition patterns model represents just one emerging effort (see also [38, 39]) to eradicate scattering and tangling at the design level, hence easing the path to aspect-oriented implementation and maintenance phases.

2.7 Related Research

This section will outline some recent research that is considered important in relation to this dissertation. The papers discussed in this section describe research with varying degrees of similarity to the work I conducted. These papers provided a good basis on which to commence this work.

2.7.1 A Case Study in Building a Web-Based Learning Environment using AOP

The Advanced Teaching and Learning Academic Server (Atlas) is a software system that supports web-based learning. The system was developed using aspect-oriented software development techniques and the results of the implementation are described in [30]. The fact that the paper describes experiences with applying aspect-oriented techniques to the development of web-based systems makes it quite relevant to this dissertation.

The paper describes the identification of aspects and how they were implemented using an early version of AspectJ. They found that development aspects such as tracing and debugging proved to be not as useful as they thought. This is because the production aspect they implemented (configuring different network contexts) made things a lot less complicated than before and there was less need for debugging and tracing. However, because these concerns had been modularised, they could be easily plugged out and back in at will. My findings from re-implementing the AppTrack system do not completely align theirs in this area. Although complexity is reduced in some areas there was still a need for the tracing concern throughout the aspect-oriented development process. The log files generated by the tracing concern proved useful not only for debugging but also for learning how the aspect-oriented languages were working at runtime. It was found that the implementation of a tracing concern could go some way towards helping to flatten the learning curve associated with AOSD and AOP languages. I do however totally agree with the conclusions of the researchers as regards the pluggability of the concerns.

The paper describes some guidelines and techniques for designing a system using the aspect approach. The *knows-about* relationship is a key factor in determining what should be an aspect. An aspect knows about another class when the aspect names the class. A class knows about an aspect if it relies on the aspect to provide it with state or functionality. Using this relationship, aspects can be divided into four categories – *open* (both the aspect and the class know about each other), *closed* (neither the aspect nor the class know about the other), *class directional* (aspect knows about the class but not vice-versa) and *aspect-directional* (class knows about the aspect but not vice-versa). Thinking of aspects as belonging to a certain category makes it easier to understand how the aspect affects your codebase and how these aspects may behave in the future.

During the development of Atlas the researchers found it beneficial to establish a set of aspect coding style guidelines. These guidelines make good sense and it is useful to consider them when starting any project using aspect-oriented programming. The paper notes that their use of the guidelines lead to increased comprehensibility and ease of debugging and testing.

A major finding of the paper is that choosing what should be an aspect is difficult and there is no set formula to follow. Their approach was to begin with an initial object model and to incrementally consider crosscutting additions as aspects, using the concept of the knows-about relationship and the aspect style rules to help them. Overall they found the application of AOSD techniques to the development of a web-based system resulted in a fast, well-structured system in a reasonable amount of time.

2.7.2 A Study on Exception Detection and Handling Using AOP

Exception handling is a source of both scattering and tangling and I had recorded this as a potential aspect in the AppTrack system. In [31], Lippert and Lopes describe their investigation into the ability of AOSD to ease tangling and scattering related to exception handling. The paper describes the re-engineering of an existing system and the results and conclusions they derived from the exercise.

The detailed implementations and results tables presented in the paper prove their main conclusion that the use of AOP and specifically AspectJ can drastically reduce the portion of the code related to exception detection and handling. On one occasion exception related code was reduced by a factor of four. Along with this reduction in code they found that AspectJ provided better support for different configurations of exceptional behaviour with respect to standard Java. Also achieved was support greater reuse, incremental development, automatic enforcement of contracts and cleaner program texts.

Some weaknesses with AspectJ were highlighted in the paper and these weaknesses were said to hinder the work. The main issue was the lack of expressiveness of the pointcut designators. The issue has since been satisfactorily addressed and did not hinder my re-implementation of the exception handling concern in the AppTrack system. However, there were other issues with the language that I encountered during the implementation of the exception handling concern that were a cause for some concern (these are discussed in section 6.1.1).

2.7.3 Initial Assessment of Aspect-Oriented Programming

A number of experiments were carried out at the University of British Columbia to assess the capabilities of AOP in two main areas. The results of these experiments and the general findings resulting from the work are presented in [32].

The experiments involved participants undertaking programming exercises spread over six sessions – three sessions using AspectJ and three using a control language. Each session began with training on both the languages being used and the topic of the concern being modularised.

The experiments compared the performance and experience of the participants working on two common programming tasks: debugging and change. During the first experiment those developers using AspectJ were able to finish the task notably faster than the participants using the non-aspect-oriented control language (in this case Java). The Java participants were noted to be performing more semantic analysis of the code and they switched the file they were viewing more than the AspectJ participants.

In the second experiment it was the AspectJ participants who required more time to finish the change task. However, in two out of three cases this time was spent on coding their solution in AspectJ rather than repeatedly switching files or conducting analysis of the semantic content of the control language files. The participant who did spend a significant amount of time analysing the control files was the only one to successfully complete the task. This suggests that an in-depth analysis of the control language is required to carry out a change task using AspectJ.

The results of the experiments led the researchers to two key insights. The first of these is that programmers may be better able to understand an aspect-oriented program when the effect of the aspect code has a well-defined scope. Debugging can be easily understood as a concept but a change task is more complex which involves analysing and understanding an unknown quantity of classes. The second insight is that the presence of aspect code may alter the strategies programmers use to address tasks perceived to be associated with aspect code.

2.7.4 Challenges of Aspect-oriented Technology

Alexander and Bieman of Colorado State University have recently turned to exploring the costs of AOSD in terms of its impact on software engineering. Their findings are described in [33]. The paper seeks to understand both the strengths and weaknesses of AOSD and to raise awareness of the potential negative side effects of its use. The following areas for disquiet are put forward for consideration:

- Understandability – is there of a loss of understandability?
- Emergent Properties and Fault Resolution – more testing and debugging required?
- Implicit Changes in Syntactic Structure and Semantics – structure not fully defined until after a weave has occurred?
- Effects on Cognitive Burden – how do you know you won’t cause undesirable emergent properties after weaving?

The paper also poses the following questions?

- How to measure the complexity that results from the weaving process?
- How do we maintain aspect-oriented programs?
- How do we effectively test aspect-oriented programs?
- How do we analyse aspect-oriented programs?

The paper raises some interesting points about development using aspect-oriented techniques, some of which have been partially answered in previous research but definitive answers to many of the questions are still needed. The points made in this paper are worth thinking about both when designing aspect-oriented systems and evaluating the usefulness of aspect-oriented programming in general.

I encountered a number of the issues highlighted in this paper during my analysis of the re-implementation of the AppTrack system. Whilst the points made by authors are certainly valid, my conclusions are generally not as pessimistic. Given a sufficient understanding of both AOSD and the AOP language(s) used in a software system, I don't believe there to be a significant change in either understandability or cognitive burden. I found that maintenance of the AppTrack system is much improved following the aspect-oriented re-implementation, transforming from an awkward, time consuming process into a well-defined and efficient task. There are however some cases that give credence to their views.

2.7.5 AOP: Does it Make Sense? The Case of Concurrency and Failures

Researches from the Swiss Federal Institute of Technology have recently investigated the area of applying AOSD/AOP techniques to the problems of concurrency and failures in distributed computing. Their findings are described in [45].

The authors begin by highlighting that they feel it does not make sense to use AOP techniques to separate concurrency control and failure management concerns from the other parts of a distributed system. They believe that currency control should be dealt with as a full part of the application semantics. They state that whilst AOP can be used to achieve some level of separation, the separation achieved is purely syntactic rather than semantic. The base their opinions on the results of three experiments conducted as part of their research.

The aim of the first experiment was to figure out the extent to which it is possible to *aspectise transaction semantics*. That is, they attempted to figure out the extent to which it is possible to completely hide transactional semantics from the programmer and have these semantics implicitly associated with the relevant modules in an automatic manner. They concluded that this is impossible for a number of reasons (discussed in full in the paper).

The second experiment is aimed at analysing the extent to which it is possible to *aspectise transaction interfaces*. That is, the authors attempted to figure out the extent to which one can completely separate transactional interfaces (begin, commit, abort) from the main object methods, and have these encapsulated writin code invoked through specific aspects. The results show that such separation is possible, but that in certain cases it may be artificial, and that it leads to rather confusing code.

The third and final experiment aims to figure out the extent to which it is possible to *aspectise transaction mechanisms*. That is, they attempted to figure out the extent to which it is possible to completely separate the mechanisms needed to ensure the ACID properties of transactions (i.e. concurrency control and failure management) from the main codebase and have these encapsulated within code invoked through specific aspects. They concluded from the experiment that, syntactically speaking, an AOP language like AspectJ provides a nice way of separating these mechanisms from the core codebase. However, they state that this separation should be handled with care and that the programmer must remain aware that the physical separation does not imply a semantic decoupling.

Experiments one and two relate closely to the work on transactions in this dissertation. Whilst my findings show a certain level of similarity to those presented in [45], there are some areas in which they differ.

3 DESIGN

This chapter details the design phase of the AppTrack re-engineering project. The design phase consisted of two major tasks, both of which would have a considerable impact on the later stages of the project and so consequently were of great importance.

The first undertaking was the identification of concerns in the code that were to be separated to achieve greater modularity. As discussed in chapter 2, there are tools for the exploration of codebases for the purpose of aspect discovery. However after some experimentation with these tools it was decided that their involvement was not necessary. There were two main reasons for arriving at this decision. Firstly, the codebase was small enough to navigate through without the use of a tool. Secondly, I was extremely familiar with the codebase after being part of the original development team. In a situation where both these factors are not present I believe the aspect identification tools currently available would add considerable value to an AOSD re-engineering effort.

The second task was the design of the aspects identified in the first phase of the design. Normally when designing a software system the Unified Modelling Language is used to graphically represent the objects that interact to satisfy the requirements. As mentioned previously, the UML is not suitable for designing systems based on AOSD concepts. For this reason, the UML-based composition patterns modelling language is used to visually describe the separation of concerns that was conceived.

The following aspects were identified in the AppTrack codebase and the remainder of this chapter deals with the identification and design of each one individually:

- Tracing
- Transactions
- Enforcing Factory Design Pattern
- Database Compatibility
- Design By Contract (Preconditions)
- Exception Handling
- Recording Bean Properties Modification

3.1 Tracing

During the implementation of the AppTrack system my colleagues and I, the original development team, predictably found ourselves in situations where our expectations had not been met by a certain piece of functionality that we had implemented. Non-conformance to requirements manifested itself as errors ranging from minor bugs to major deficiencies. Although we were following a test-first [46] approach to the development of the system, we sometimes lapsed back into the classic scenario in which you find yourself inserting random `System.out.println` method calls into the bodies of various methods so you can observe a) whether or not a method is being executed at all at runtime or b) the values of any variables/parameters used within a method. This approach to tracing is at best haphazard. Both scattering and tangling are manifest in the implementation of this concern. Scattering because the same piece of tracing code was cut and paste into many different methods across many different classes in many different packages. Tangling because to fully implement the tracing concern, numerous classes not designed to cater for tracing behaviour were required to encapsulate tracing logic, leading to the implementation of at least two concerns within one implementation module. As mentioned, the tracing concern was implemented on as "as needed" basis. When some difficulty arose and we needed to trace certain methods we would implement some crosscutting tracing behaviour. Full implementation of the tracing concern in this manner would have required adding approximately five lines of code to every method of every class. This would have been a huge undertaking for only limited rewards as tracing is a development concern and this code would have had to be removed from the production release of the system. An aspect-oriented design for this concern would in theory provide pluggable tracing functionality, implemented without affecting any other core concern.

Using the composition patterns model and aspect-oriented software development techniques a tracing concern was designed. There were a number of requirements for the concern:

- It had to be standalone – its implementation must not establish any dependency relationships (in either direction) with the core AppTrack codebase
- It must record context specific information. This is more of an implementation issue but it must be kept in mind during design that context must be available without creating dependencies
- Any method must be eligible for tracing
- The tracing must happen before the execution of the method being logged

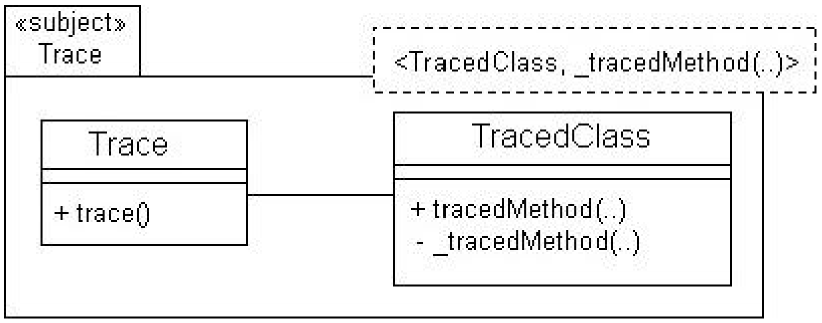

Figure 5: Tracing template

Figure 5 illustrates that a traced method, encapsulated within a traced class, can be augmented with tracing behaviour. Any method is available for tracing; hence there are no parameter restrictions on the `tracedMethod(..)` operation defined in the template. The tracing functionality is encapsulated in a separate module, `Trace`, and the association relationship is established by the AOP language rather than by an explicit programmer action.

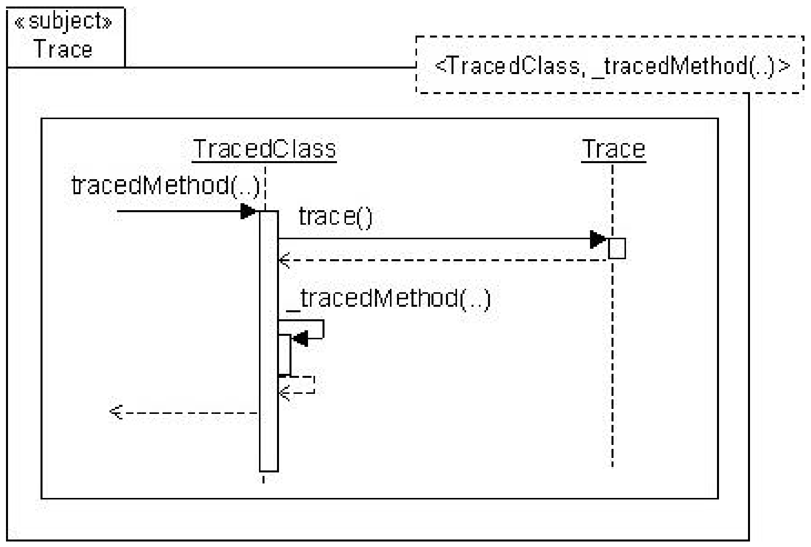

Figure 6: Tracing collaboration

Figure 6 shows the object interaction at runtime. The invocation of a traced method will delegate to an invocation of the tracing concern's functionality before the behaviour of the actual traced method is performed.

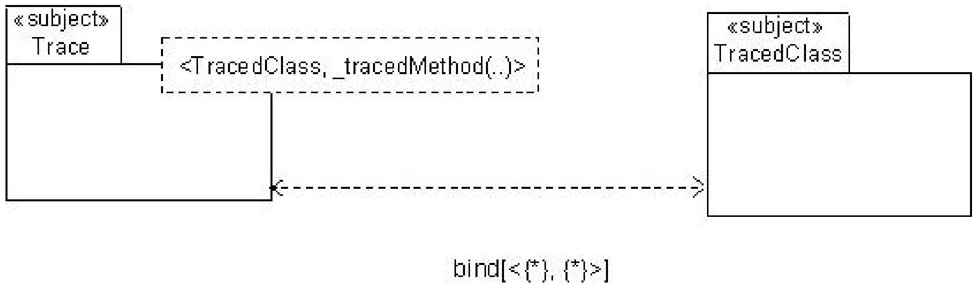

Figure 7: Tracing composition relationship

Figure 7 illustrates the composition relationship with bind attachment. From the asterisks within the braces of the bind attachment we can see that any method from any class can be traced.

The three models presented in this subsection describe the design of the tracing concern that was implemented for the AppTrack system. The models show that the requirements of every method being traceable and the tracing happening before the method execution have been satisfied. Satisfaction of the other requirements will be proven during the implementation phase.

3.2 Transactions

AppTrack is a web-based information system. Like most systems of this genre, it has an underlying database and can cater for multiple simultaneous users. It is necessary to employ a transaction service in order to retain data integrity whilst serving multiple users. Data access is a functional requirement whilst a transaction service affecting data access is considered to be a non-functional requirement. This distinction is cause for their separation. In AppTrack they are implemented together and the result is extremely tight coupling.

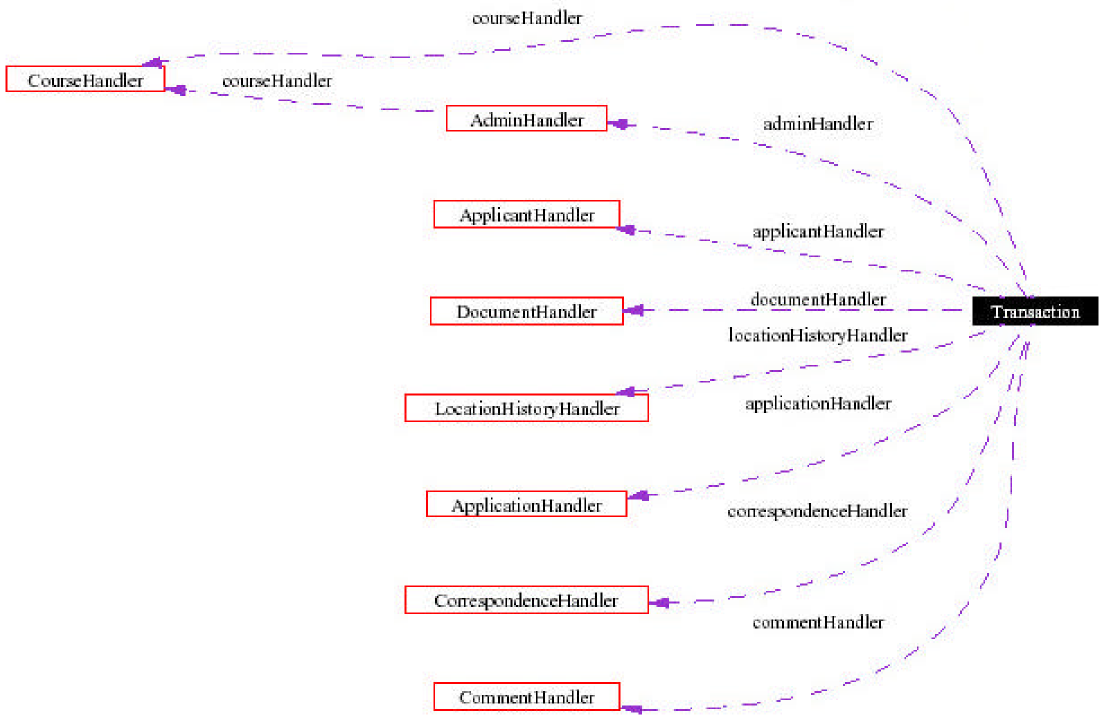

Figure 8: AppTrack transaction and data access architecture

Figure 8 shows the relationship between the transactions concern and the data access concern. Each key object in the system has a `Handler` class to cater for its data access requirements. The key objects are derived from the requirements specification. As can be seen from the diagram, any class that wants to use data access functionality can only gain access to classes that will deliver this functionality by going through the `Transaction` class. What this means in practice is that you must declare an instance of the `Transaction` class and obtain the instance of the `Handler` you desire from the `Transaction` instance. This makes data access and transactions inseparable. It could be argued that you should never provide data access functionality in a system such as AppTrack without a transaction service and that they are two logically entwined concepts. Whilst coupling the two concerns tightly does enforce the use of transactions it also diminishes the reusability and evolvability of the system. An aspect-oriented architecture would make it possible to the implement the two concerns independently and weave transactions onto the data access at compile time. Whilst this makes it easier to create a system without transactional data access I feel these risks are offset by the benefits of an AOSD-based solution. There is however other higher-level issues with the separation of this concern. These are discussed further on in this dissertation (section 5.2).

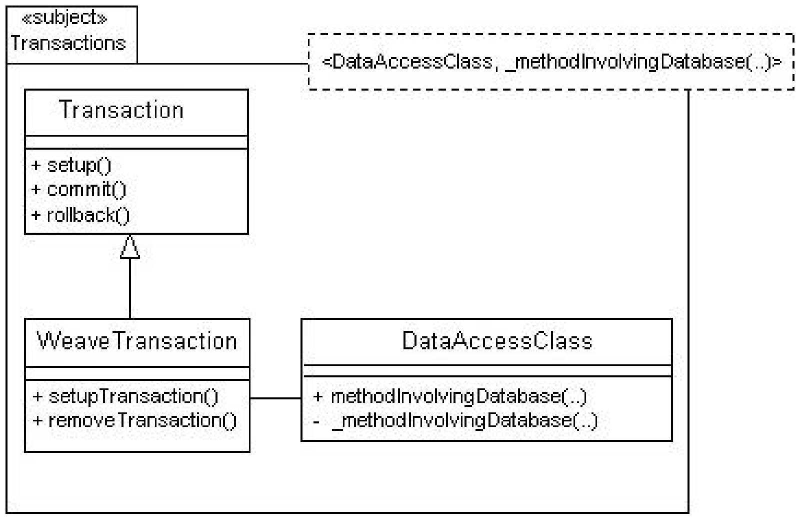

Figure 9: Transactions template

Figure 9 shows the composition patterns structure for the transactions concern. The Struts framework dictates that certain classes must implement certain interfaces and perform particular duties. `Action` classes contain the code for the actual core behaviour of a requirement such as, for example, registering a potential applicant with the system. `Action` classes (and supporting bean and form classes) are the only classes that perform data access and hence it is only these classes that can be enhanced with transactional behaviour. An instance of one of these classes is referred to as a `DataAccessClass` in the template model. There are no parameter restrictions on methods that can have their data access functionality supplemented with transactions.

The transactional behaviour resides in the `Transaction` class. This class contains core database connection management methods such as `setup(), commit()` and `rollback()`. The behaviour implemented in the `Transaction` class is weaved into the methods in the data access classes by the `WeaveTransaction` class. This class contains simple methods for designating which methods in the data access classes need to be augmented with transaction setup and removal behaviour.

Figure 10 shows the sequence of interactions that occur between the transactions concern and the `DataAccessClass` that is making use of it. When a method involving data access is invoked, a transaction will be setup. The database will be accessed and the rest of the method body will execute. Following the completion of the method involving the data access, the transaction will be committed and all appropriate resources will be freed. This behaviour is encapsulated in the `removeTransaction()` method.

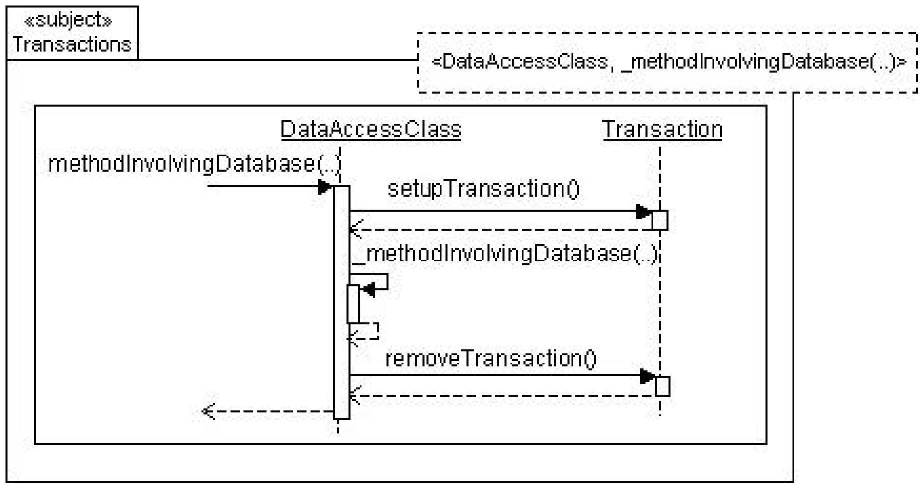
«subject»
Transactions
<DataAccessClass, _methodInvolvingDatabase(..)>
DataAccessClass
Transaction
methodInvolvingDatabase(..)
setupTransaction()
_methodInvolvingDatabase(..)
removeTransaction()

Figure 10: Transactions collaboration

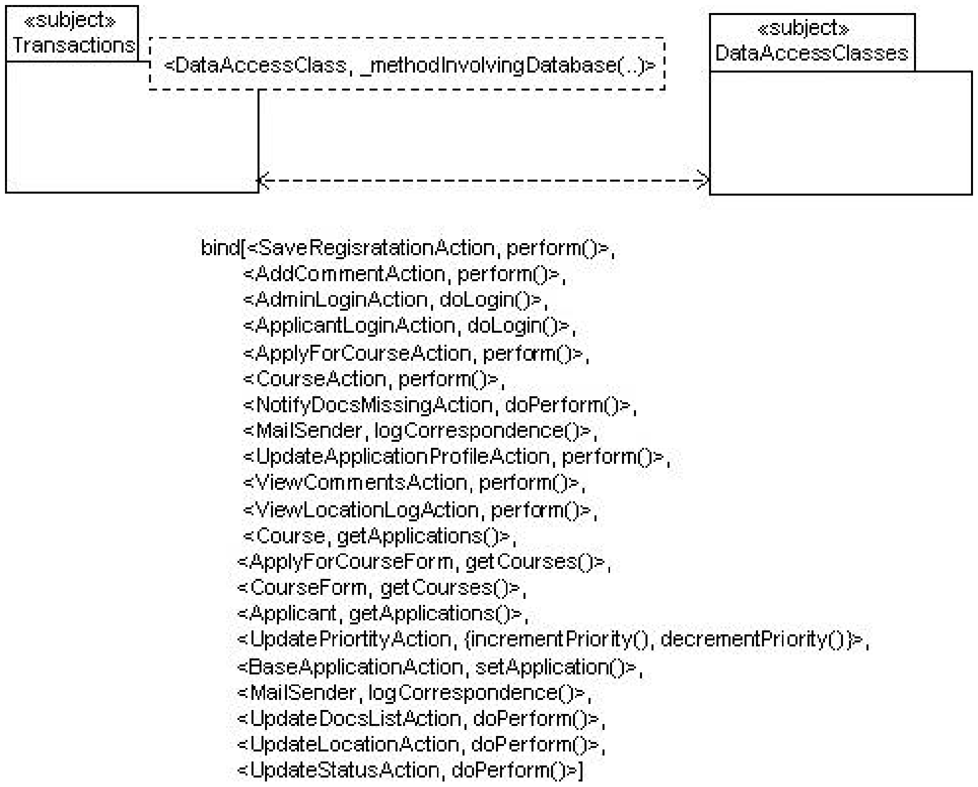
«subject»
Transactions
<DataAccessClass, _methodInvolvingDatabase(..)>
«subject»
DataAccessClasses
bind[<SaveRegisratationAction, perform()>,
<AddCommentAction, perform()>,
<AdminLoginAction, doLogin()>,
<ApplicantLoginAction, doLogin()>,
<ApplyForCourseAction, perform()>,
<CourseAction, perform()>,
<NotifyDocsMissingAction, doPerform()>,
<MailSender, logCorrespondence()>,
<UpdateApplicationProfileAction, perform()>,
<ViewCommentsAction, perform()>,
<ViewLocationLogAction, perform()>,
<Course, getApplications()>,
<ApplyForCourseForm, getCourses()>,
<CourseForm, getCourses()>,
<Applicant, getApplications()>,
<UpdatePriortityAction, {incrementPriority(), decrementPriority()}>,
<BaseApplicationAction, setApplication()>,
<MailSender, logCorrespondence()>,
<UpdateDocsListAction, doPerform()>,
<UpdateLocationAction, doPerform()>,
<UpdateStatusAction, doPerform()>]

Figure 11: Transactions composition relationship

Figure 11 presents the composition relationship for the transactions concern. Any method involving data access can have that data access execute within the context of a transaction. The bind attachment to the composition relationship shows the actual methods in the AppTrack codebase that use the transactions concern. Note the use of the parenthesis to add transactional behaviour to two methods in the same class, `UpdatePriorityAction`.

The models presented in this subsection describe the design of the transactions concern that was implemented for the AppTrack system. The models show that any appropriate method that needs to access the database can have this data access execute within a transaction. This behaviour can be achieved without tightly coupling the concerns as tightly as they were before. This is discussed further in the section that evaluates the re-implementation of this concern (section 5.2).

3.3 Enforcing Factory Design Pattern

The concept of object factories is presented in [34]. This concept forms the basis of the Factory Method and Abstract Factory design patterns. These two patterns are object-oriented software design patterns. The core intent of the patterns is to formalise structures for object creation. The Factory Method lets a class defer instantiation to subclasses. The pattern is a class creational pattern. The Abstract Factory pattern provides an interface for creating families of related or dependent objects, without specifying their concrete classes. A factory class based on the general concept (controlling object creation) of these two design patterns is used in AppTrack for the implementation of the application status requirement. Each application for a postgraduate course has an associated status. There are eleven possible statuses that an application can have. Each status from the requirements specification has a status object to represent it. Each of these objects implements the `Status` interface.

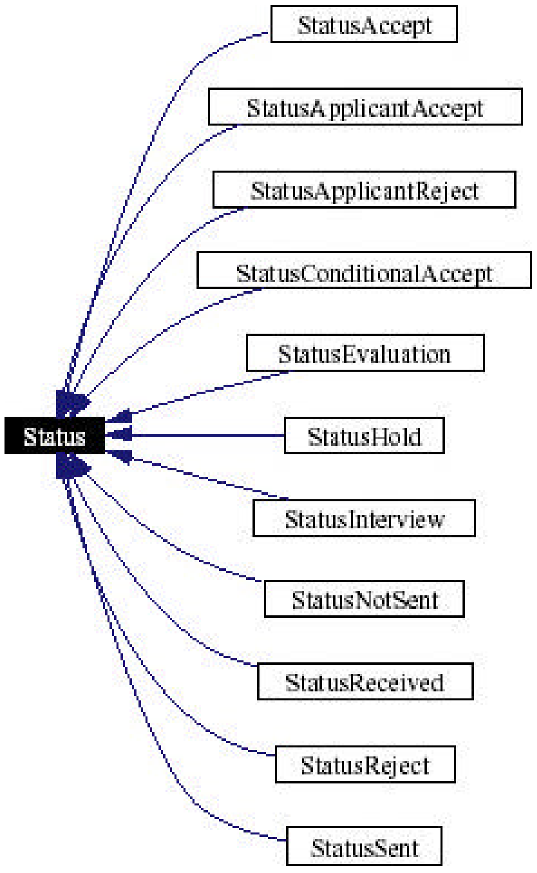

Figure 12: The `Status` hierarchy

A factory-style pattern is needed to put tighter controls on how status objects can be created and manipulated. It would not be desirable to have unrestrained creation of status objects as only one instance of each one needed. This instance can be shared amongst all clients of the application and the factory enforces this because it is a Singleton [34] itself. The Singleton pattern is used to ensure that only one instance of a class is created, and that a global point of access is provided to it.

The inclusion of a factory in the AppTrack architecture required slightly more development resources than a less elegant solution involving non-regulated creation of `Status` objects (making it harder to enforce the high-level requirements of the system that place restrictions on status transition e.g. you may only go from status A to status B, not status C). The pattern makes it possible to easily extend or modify the existing base of status objects and to integrate any new statuses seamlessly. It is desirable to protect this investment in the architecture so that any future maintainers of the system follow the pattern. Although the pattern is in place, it would be possible to bypass it when making future modifications to the system. It is possible to enforce the use of the pattern by ensuring that any class that implements the `Status` interface is only instantiated from within the `StatusFactory` class. An object-oriented solution to this enforcement requirement would necessitate somehow (possibly using Java Reflection) checking which class was calling the constructor of each concrete status object. If this client class wasn't the `StatusFactory` then an exception would be raised or some other appropriate action would be taken. There are two main problems with an object-oriented solution (if one is possible at all). Firstly, the implementation would be overly complex; there is no easy way to do what we want to with the object-oriented approach. Complexity should always be avoided wherever possible. Secondly, including design pattern enforcement code in the constructor of a concrete status object leads to the tangling of concerns. The standard status behaviour should be separate to any enforcement of the factory pattern. An aspect-oriented solution to this dilemma proves to be superior to its object-oriented counterpart.

The requirement, once more, is to ensure that concrete status objects are only instantiated from within the `StatusFactory` class. Any call to the constructor of a concrete status object, denoted in AspectJ by the method `someStatus.new(..)` will cause the execution of the `enforceFactory()` method. The `FactoryEnforcer` is not explicitly associated with any class in the system and vice versa.

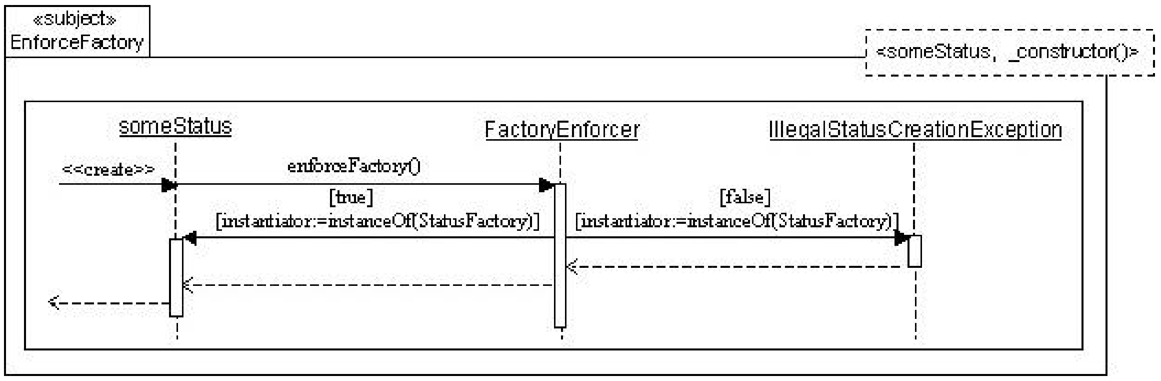
«subject»
EnforceFactory
<someStatus, _constructor()>
someStatus
FactoryEnforcer
IllegalStatusCreationException
<<create>>
enforceFactory()
[true]
[instantiator:=instanceOf(StatusFactory)]
[false]
[instantiator:=instanceOf(StatusFactory)]

Figure 13: Enforce factory pattern collaboration

Figure 13 describes the sequence of events that occur when the constructor of a concrete status object is invoked. Before the constructor executes, the precondition-style `enforceFactory()` method executes. This method checks to see if the `Status` object was invoked from a valid location. If it has then the object is created successfully, otherwise an exception is thrown before the creation of the object. The actual implementation of `enforceFactory()` method is not overly important, the key point is that the method runs whenever a status object is created and it contains appropriate behaviour so as to enforce the correct use of the status factory pattern.

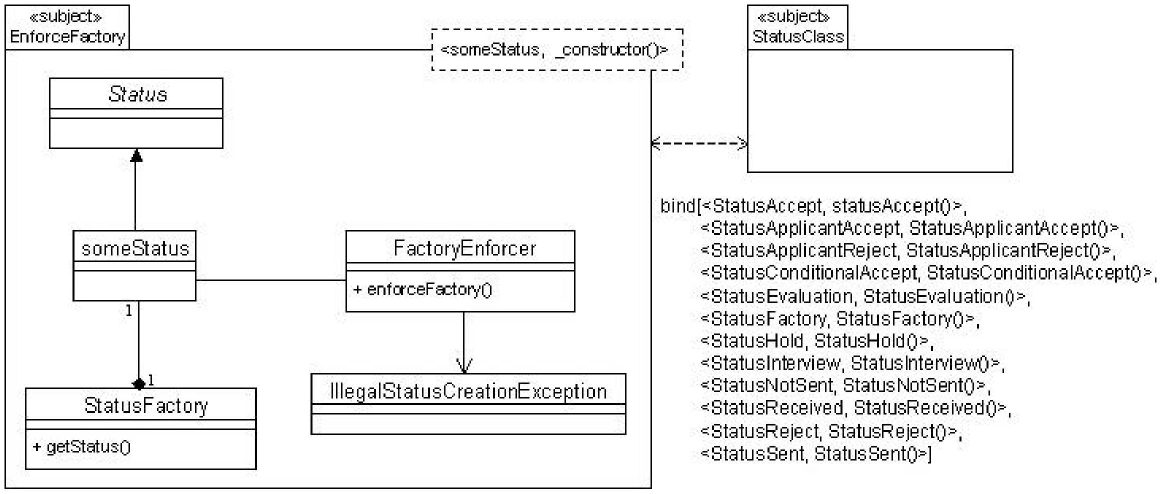
«subject» EnforceFactory
<someStatus, _constructor()>
«subject» StatusClass
Status
someStatus
FactoryEnforcer
+ enforceFactory()
1
1
StatusFactory
+ getStatus()
IllegalStatusCreationException
bind[<StatusAccept, statusAccept()>,
<StatusApplicantAccept, StatusApplicantAccept()>,
<StatusApplicantReject, StatusApplicantReject()>,
<StatusConditionalAccept, StatusConditionalAccept()>,
<StatusEvaluation, StatusEvaluation()>,
<StatusFactory, StatusFactory()>,
<StatusHold, StatusHold()>,
<StatusInterview, StatusInterview()>,
<StatusNotSent, StatusNotSent()>,
<StatusReceived, StatusReceived()>,
<StatusReject, StatusReject()>,
<StatusSent, StatusSent()>]

Figure 14: Enforce factory composition relationship

Figure 14 illustrates the composition relationship for the enforce factory concern. The `StatusFactory` class contains one instance of each `Status` class, represented here by the `someStatus` class. In the implementation of this model, the `someStatus` class is replaced by the actual `Status` classes in the AppTrack codebase. These classes are listed in the bind attachment. Therefore, the instantiation of any of the listed classes will cause the execution of the enforce status factory behaviour.

The models presented in this subsection explain the design of the status factory enforcing concern that was implemented for the AppTrack system. The models show that any call to the constructor of a class implementing the `Status` interface will be checked to see if it is legal. Using AOSD techniques this behaviour can be realised without tangling the precondition with the status constructor.

3.4 Database Compatibility

AppTrack uses the MySQL relational database management system (RDBMS) as its underlying data source. MySQL is an open-source database designed for speed, power and precision rather than, for example, user-friendliness. The database suited our needs on this project satisfactorily and most importantly, it was free. However, the DBMS displays some curious behaviour when a string containing an apostrophe (‘) is added to a field of a table. If an applicant registers with the system using a name like “John O’ Leary” the name will be added and read back as “John OLeary”. This is clearly unsuitable behaviour. The MySQL workaround for this issue is to double up on apostrophes when they are required. If the same applicant registers with the system then the name will be parsed and all apostrophes will be replaced with double apostrophes (''). The user's name will now be read back as "John O' Leary".

There exists a `Handler` class for each key object in the system (see figure 8). These database interface classes contain a method, `setQueryParams()`, that sets the parameters for any database queries that the `Handler`' classes make. We implemented a method to parse any parameters that were set inside this method. This solution worked fine and there were never any problems adding and displaying names or indeed any text string containing an apostrophe. However, because we had included the fix to this database issue inside our core application codebase there were two problems. The first of these was that we had coded our system into a state where it was database specific. Due to the parsing and manipulation of strings containing apostrophes we could not port seamlessly to another DBMS. From a software engineering perspective, the second problem is that database fix code was tangled within the `Handler` classes. A greater degree of separation of these two related but distinct concerns was definitely desirable. It must be noted here that standard object-oriented software development techniques could have been used to reduce the degree of scattering and tangling caused by the implementation of this concern. The parsing behaviour could have been encapsulated within a class of its own (as opposed to appearing in each `Handler` class) and then invoked from the database handler classes. This would reduce the number of scattered lines of code in the system that related to this concern but the concern would still be tangled within the `Handler` classes.

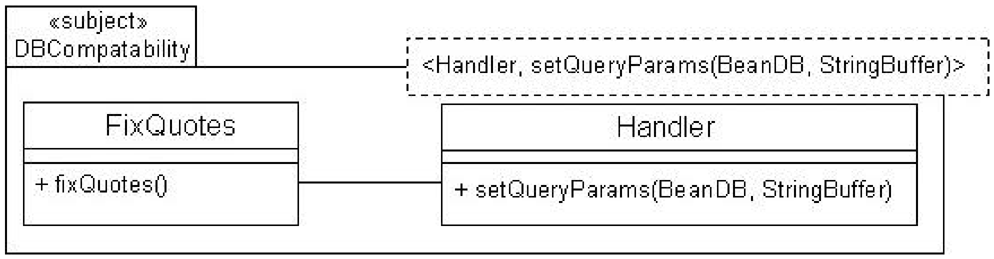
«subject»
DBCompatability
<Handler, setQueryParams(BeanDB, StringBuffer)>
FixQuotes
+ fixQuotes()
Handler
+ setQueryParams(BeanDB, StringBuffer)

Figure 15: The database compatibility template

Figure 15 shows the composition patterns structure for the database compatibility concern in the AppTrack system. Each `Handler` class has a `setQueryParams` method, which is the method that sets the parameters to be sent to the database. The `fixQuotes()` method of the `FixQuotes` class is run in place of `setQueryParams` method. This is because the `setQueryParams` method should be implemented for the perfect case i.e. it should be implemented for a standard database that doesn't have the apostrophe issue. The pluggable AOSD solution should run in place of the standard behaviour until such a time as an issue-free database becomes available.

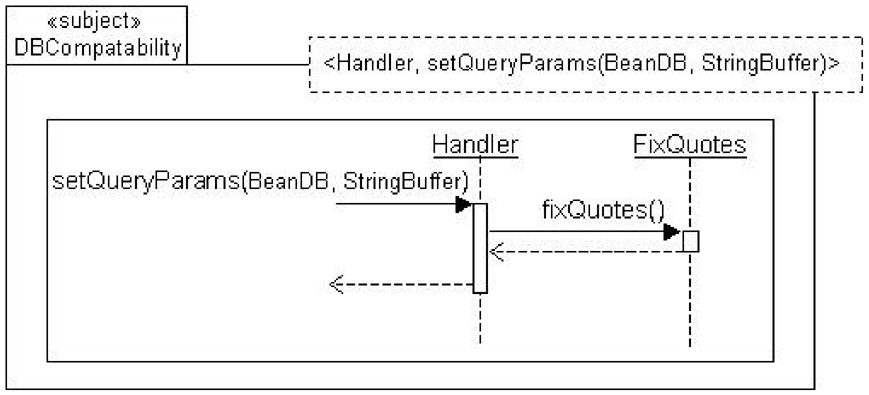
«subject»
DBCompatability
<Handler, setQueryParams(BeanDB, StringBuffer)>
Handler
FixQuotes
setQueryParams(BeanDB, StringBuffer)
fixQuotes()

Figure 16: Database compatibility collaboration

Figure 16 illustrates the behaviour that occurs at runtime when a user attempts to add a text string to the database. When a `setQueryParams` method from an affected `Handler` class is invoked, it will delegate is execution to the `fixQuotes` method in the `FixQuotes` class. This method parses parameters for apostrophes and fixes them as appropriate. When a more suitable database becomes available then the apostrophe handling concern can be unplugged and the standard implementation will be sufficient.

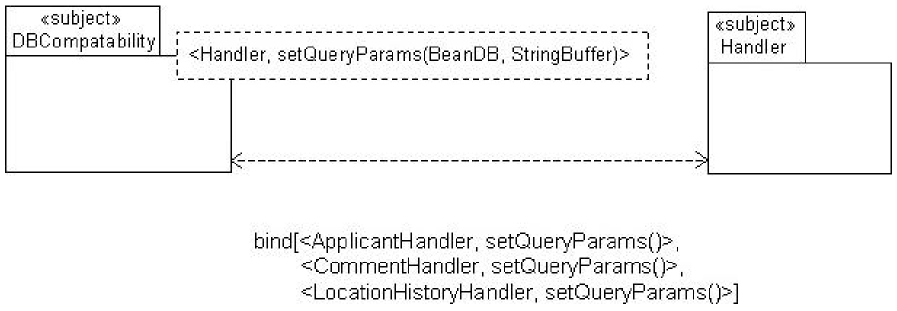
«subject»
DBCompatability
<Handler, setQueryParams(BeanDB, StringBuffer)>
«subject»
Handler
bind[<ApplicantHandler, setQueryParams()>,
<CommentHandler, setQueryParams()>,
<LocationHistoryHandler, setQueryParams()>]

Figure 17: Database compatibility composition relationship

In AppTrack, there are three bean objects that have text information obtained from the user stored about them in the database. This information is acquired via web forms. Anywhere a text box on a web form is used for retrieving information, that information must be parsed for apostrophes. Figure 17 shows the composition relationship with the bind attachment for the database compatibility concern. The three objects containing user input values are `Applicant`, `Comment` and `LocationHistory`. It is the handlers for these classes that are bound to the template diagram.

This subsection presented three models that describe the design of the separation of a concern that deals with a database compatibility issue. The models show that any call to `setQueryParams` that involves dealing with text strings that may contain the apostrophe character can be delegated to a separate method call, `fixQuotes`. Using the aspect-oriented paradigm, no explicit relationship is established between the two method calls so the delegation associated can be removed when appropriate without the need for any invasive change.

3.5 Design By Contract (Preconditions)

A precondition expresses the constraints under which a routine will function properly [35]. A precondition applies to all calls to the method possessing the precondition and a correct system will never execute a method with a precondition unless the precondition is satisfied. Preconditions support the concept of design by contract, which is defined in [35] as “viewing the relationship between a class and its clients as a formal agreement, expressing each party’s rights and obligations”. By taking a design by contract approach to the relationship between collaborating classes, a significant degree of confidence in the correctness of software systems can be attained.

Design by contract was not widely used in the development of the AppTrack system. An import reason for this was the lack of explicit language support in Java for expressing assertions (such support has since been partially added following the introduction of the `assert` keyword into the Java language [47]). Another reason is that the test-first approach to development was being used and this was our method for attempting to ensure the correctness of our system. However, during my examination of the AppTrack codebase I came across some examples of precondition-style code segments at the beginning of several methods. The methods were related to the sending of email as part of certain system features. Before email can be sent from the system, the status of the objects used to compose the email requires verification. Preconditions are used to carry out this verification process before the email sending methods execute.

Preconditions are a source of tangling in object-oriented systems. The code that enforces the contract concern is grouped together, in the same method, with the code that implements the functionality. This can hinder both evolvability and reusability. AOSD techniques can be used to extract preconditions from the core codebase and implement them separately. The preconditions can then be merged back as needed, achieving the same effect as before but without the drawbacks. However, given that preconditions are such an important element of a method, there is an argument to suggest that they should not be separated, as it may confuse clients of the method. Whilst this argument certainly has some validity, I chose to separate this concern based on my previous experience working with preconditions in the Eiffel programming language [36], where the precondition to a method is encapsulated within a language-defined *require* construct. This allows for simple identification of precondition code, aiding maintenance. This was the core benefit sought from the aspect-oriented re-implementation of this concern. Whilst I appreciate that the degree of separation in my implementation is greater than that evident in Eiffel’s assertion mechanism, and that the precondition and method affected are a greater distance apart, I do not believe that the degree of separation achieved is a cause of confusion.

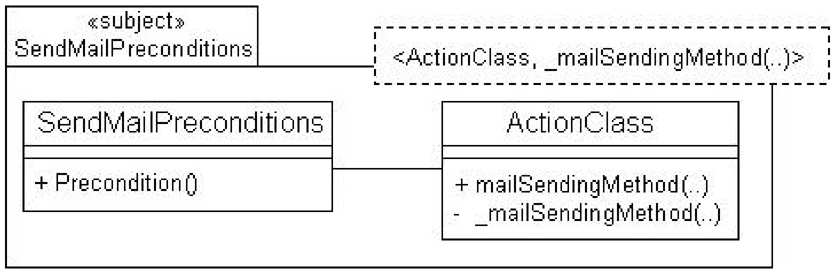

Figure 18: Mail sending preconditions template

Figure 18 shows the mail sending preconditions template. Before the execution of any method that composes an email message (`mailSendingMethod(..)`) from any `Action` class, the parameters passed to this method are verified by supplementing the `mailSendingMethod(..)` with the `Precondition()` method from the `SendMailPreconditions` class.

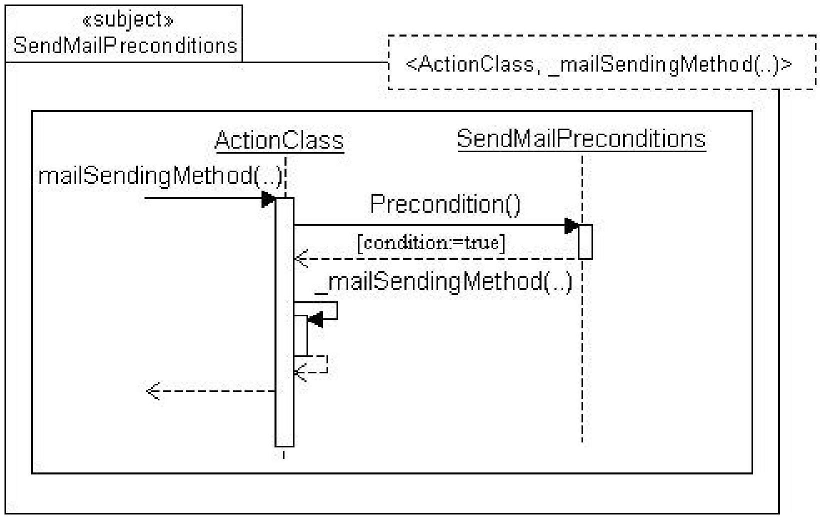

Figure 19: Mail sending preconditions collaboration

Figure 19 shows the collaborations between the preconditions concern and the modules that it affects. On invocation of a mail sending method, control is passed to the `SendMailPreconditions` class. This class executes a method to evaluate the parameters passed to the `mailSendingMethod(..)`. If the final result of the Boolean expressions that make up the precondition is true then the `mailSendingMethod(..)` will execute. If the result is false then the precondition will execute certain appropriate behaviour, for example an exception could be thrown.

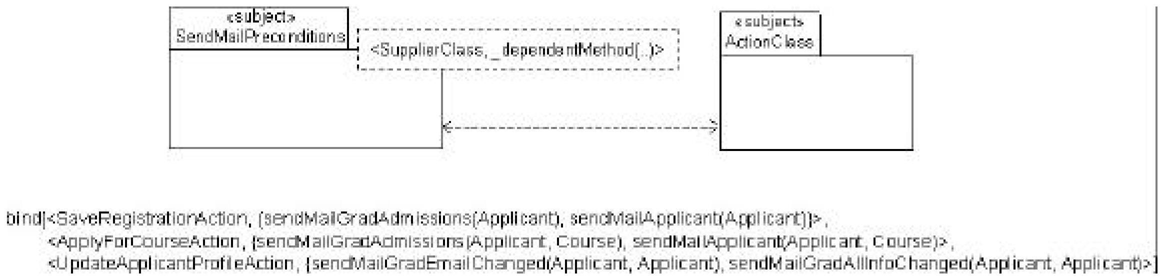
«subject»
SendMailPreconditions
<SupplierClass, _dependentMethod(..)>
«subject»
ActionClass
bind[<SaveRegistrationAction, {sendMailGradAdmissions(Applicant), sendMailApplicant(Applicant)}>,
<ApplyForCourseAction, {sendMailGradAdmissions(Applicant, Course), sendMailApplicant(Applicant, Course)}>,
<UpdateApplicantProfileAction, {sendMailGradEmailChanged(Applicant, Applicant), sendMailGradAllInfoChanged(Applicant, Applicant)}>]

Figure 20: Mail sending preconditions composition relationship

Figure 20 illustrates the composition relationship with the bind attachment for the preconditions concern. There are currently six methods from three different classes bound to the template model. The bind attachment is quite specific as precondition concerns are generally quite specialised, as opposed to a concern like tracing.

The models that are presented in this section explain the design of the mail sending preconditions concern. The models show that certain methods from `Action` classes (and supporting classes) that send emails can be augmented with preconditions before their execution. The implementation of these preconditions is contained separately not only from the method being verified but also from the class that contains that method. This separation eradicates the previously described contract tangling.

3.6 Exception Handling

Exception handling code is a major cause of code tangling in object-oriented systems. Ideally, the core system codebase would contain code to handle the ordinary case rather than both the ordinary and exceptional cases. As well as the tangling, exception handling code is also a large source of scattering. Regularly, an exception is handled in the same manner no matter how many times it may be thrown in different locations. Replicating this code can lead to inconsistencies and the risk of misuse at each point of replication. AppTrack contains 29 `catch` blocks in the `ie.tcd.cs.mscnds.apptrack.action` package. These 29 `catch` blocks only throw four different types of exception. Each exception type has only one piece of exception handling behaviour associated with it. This piece of code is used each time that particular exception is thrown. AOSD techniques can help produce an exception handling architecture that will eradicate this scattering and tangling.

As noted in chapter 2, there has been work done on exception detection and handling using aspect-oriented programming. This work sought to remove all exception handling code from the core codebase and separate it into a totally detached module. What this approach does is remove all `try` and `catch` blocks, as well as the exception handling code, from the core codebase and weaves them back in as appropriate at compile time. I opted not to take this approach during my re-engineering of the AppTrack system. The main reason for this is that I feel the removal of `try` and `catch` blocks adversely affects the readability and comprehensibility of the source files in the AppTrack system. The presence of `try` and `catch` blocks in Java code has become so familiar to developers that its absence would result in a significant loss of comprehensibility. Therefore, my design will focus on separating specifically the exception handling code rather than exception designation code.

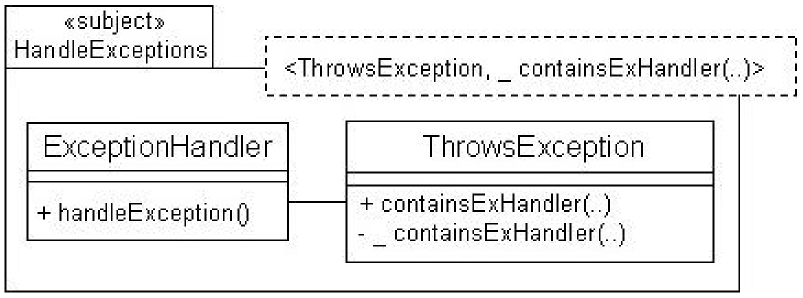

Figure 21: Exception handling template

Figure 21 shows the composition patterns structure that describes the relationship between the separated exception handling concern and the core system classes. Whenever an exception handler for an exception that is catered for by the `handleException()` method is invoked, the `ExceptionHandler` class will provide the correct exception handling functionality before handing control back to the method that contains the exception handler. This means that the throwing of an exception and the handling of that exception are kept separate. Exception handling functionality that is used in many different code base locations need only be written once. This affords greater changeability of system's exception handling behaviour.

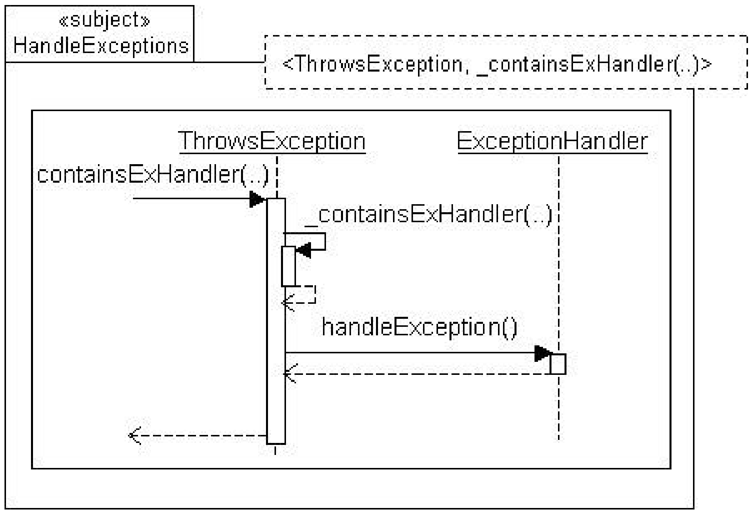

Figure 22: Exception handling collaboration

Figure 22, the exception handling collaboration diagram, describes the sequence of object interaction at runtime. If a method containing an exception handler catered for by the `ExceptionHandler` class throws an exception then the handling of this exception is delegated to the `handleException()` method. Control returns to the class that raised the exception following the execution of the exception handling behaviour.

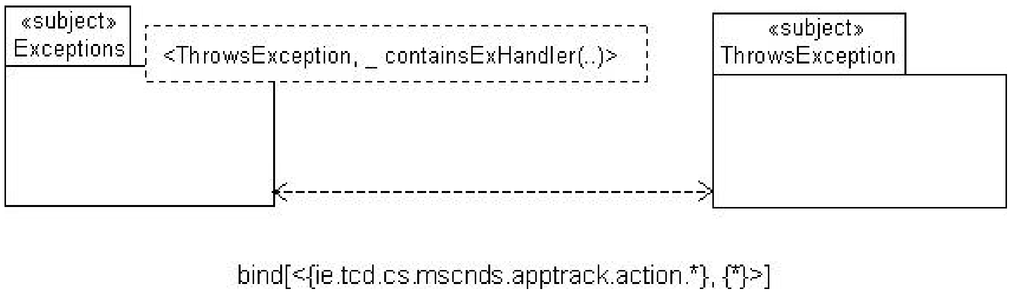
«subject»
Exceptions
<ThrowsException, _ containsExHandler(..)>
«subject»
ThrowsException
bind[<{ie.tcd.cs.mscnds.apptrack.action.*}, {*}>]

Figure 23: Exception handling composition relationship

Figure 23, the composition relationship with the bind attachment, illustrates that any method within any class from the `ie.tcd.cs.mscnds.apptrack.action` package can have its exception's handled by the exception handling concern.

This subsection describes the design of the exception handling concern that was separated during the re-implementation of AppTrack. The models show that the invocation of any exception handler within the `ie.tcd.cs.mscnds.apptrack.action` package will have its behaviour delegated to the exception handling concern. The encapsulation of exception handling behaviour using AOSD techniques allows for greater reuse and increased changeability/extensibility of ways in which exceptions are handled.

3.7 Recording Bean Properties Modification

The `Applicant` class is a bean class representing system users who apply for postgraduate positions in via the AppTrack system. A common problem with the old paper-based system is that it is hard to keep track of applicants when they change address or other contact details. The new system aims to make it easier for applicants to notify the college of any changes in their profile information i.e. contact address, email address and telephone number. This requirement is realised in the form of an "Update Profile" function. When college staff are viewing an application, a summary of the applicant's profile information is presented. The date (including time) that the applicant last updated their details is also presented along with their profile information. In order to present this "last update" time information, whenever an applicant modifies a property of the `Applicant` class (via the update profile function), the variable `addressLastUpdateTime` in the `Applicant` class must be set. The current implementation is a source of both scattering and tangling, albeit to a lesser extent than any previously described concerns. It is interesting to note here that this isn't the type of concern that AspectJ purposes to be the solution for. AspectJ aims to solve issues involving crosscutting concerns whereas the MDSOC approach to separation of concerns aims to eradicate any scattering and tangling, whether it is part of a crosscutting concern that affects many implementation modules or not. As long as a concern is considered to be a logically separate, the MDSOC approaches states that it should be separated.

The `Applicant` class contains methods and variables to implement the functionality that keeps track of applicant property updates. Whilst the implementation of this functionality is only found in the bean class, it is not really applicant behaviour and so it is logically misplaced. So even though only one module is involved in the implementation there is still a degree of concern tangling. The scattering occurs because there are seven different accessor methods that contain a call to the `setAddressLastUpdateTime()` method. Accessor methods should deal with whatever variable they are intended to; they should not be concerned with other requirements. Also, with this approach to setting the update time variable there is a chance that the scattered code will be omitted from an accessor method or used incorrectly. An aspect-oriented design to this concern proves to be rather more elegant than the object-oriented solution.

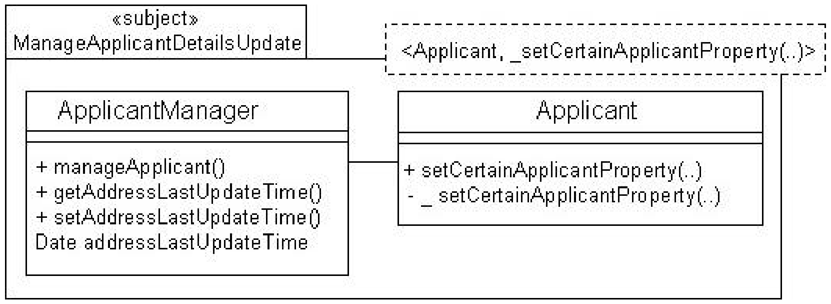
«subject»
ManageApplicantDetailsUpdate
<Applicant, _setCertainApplicantProperty(..)>
ApplicantManager
+ manageApplicant()
+ getAddressLastUpdateTime()
+ setAddressLastUpdateTime()
Date addressLastUpdateTime
Applicant
+ setCertainApplicantProperty(..)
- _ setCertainApplicantProperty(..)

Figure 24: Record property modification template

Figure 24 shows the composition patterns structure for the record property modification concern. Every property in the `Applicant` class has a set of accessor methods. These accessor methods either modify or return the associated property. The template model shows that when certain setter methods are run, their standard behaviour can be augmented with behaviour from the `ApplicantManager` concern. This behaviour is the `manageApplicant()` method. The other methods and variables listed in the `ApplicantManager` concern are merged into the `Applicant` class so that they can be accessed by the Java server pages that display applicant information to the system users.

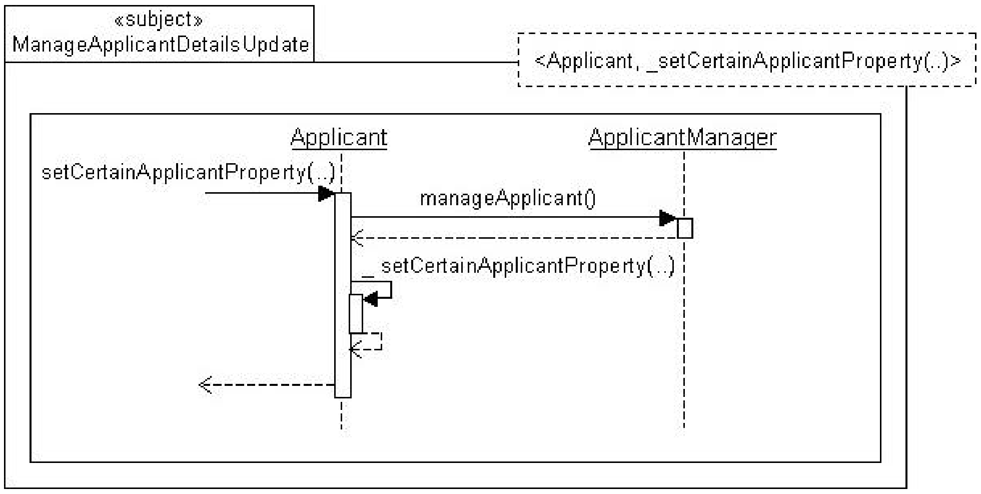
«subject»
ManageApplicantDetailsUpdate
<Applicant, _setCertainApplicantProperty(..)>
Applicant
ApplicantManager
setCertainApplicantProperty(..)
manageApplicant()
_ setCertainApplicantProperty(..)

Figure 25: Record property modification collaboration

The collaboration diagram, figure 25, shows that when a method setting a certain applicant property is invoked, control will be passed to the `ApplicantManager` class, which will record that the property in the `Applicant` class has changed. Following this event the setter method will execute as normal.

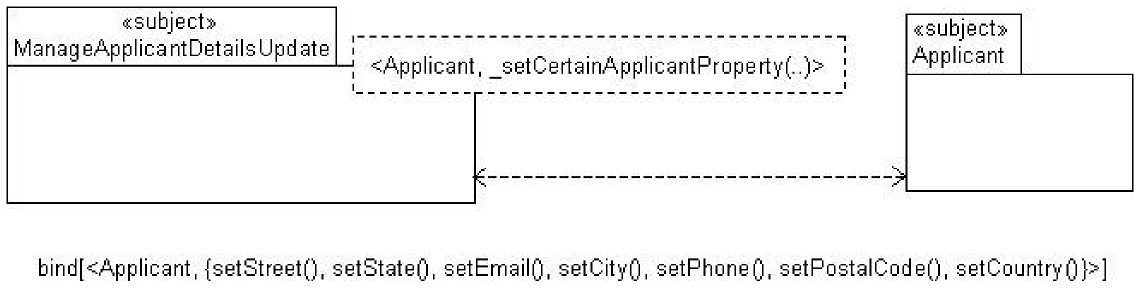
«subject»
ManageApplicantDetailsUpdate
<Applicant, _setCertainApplicantProperty(..)>
«subject»
Applicant
bind[<Applicant, {setStreet(), setState(), setEmail(), setCity(), setPhone(), setPostalCode(), setCountry()}>]

Figure 26: Record property update composition relationship

Figure 26 shows the composition relationship with the bind attachment for the record property modification concern. Currently there are seven methods from the `Applicant` class being supplemented with behaviour to record their invocation by applicants.

This subsection describes the design of the concern that records the date and time that a property of the `Applicant` class was updated. The design shows the removal of the tangling and scattering from the `Applicant` class and the encapsulation of the concern within a more logical module. The resulting re-engineered architecture retains the availability of all applicant properties to the view layer of the system (the Java server pages).

4 IMPLEMENTATION

This chapter describes the work carried out during the implementation phase of the project. The core activity during this phase was the implementation of the architectures designed during the design phase (described in chapter three). This chapter describes the implementation of the concerns only. Evaluation of this work is contained in the evaluation chapter (chapter 5).

The primary Aspect-oriented programming language used for the implementation was AspectJ. AspectJ is the most widely adopted and highly supported AOP language currently available[2]. It supports the aspect-approach to separation of concerns. A secondary implementation was undertaken using the Hyper/J tool. Hyper/J has not evolved as much as AspectJ since its inception in 1999 and did not have as much of an impact on the implementation process as AspectJ. Hyper/J supports the multi-dimensional separation of concerns and the hyperspaces model.

For each concern identified, the AspectJ implementation is described, followed by the Hyper/J implementation. Due to limitations with the current version of the Hyper/J tool (it failed to cope with the package depth in the AppTrack codebase), I was unable to apply Hyper/J to the existing AppTrack codebase. For this reason, the Hyper/J implementation was carried out on purpose-built codebase.

4.1 Implementation Environment

Version 1.0.5 of AspectJ was used for the aspect approach implementation. No version number was associated with the Hyper/J release that was employed. The Java Development Kit (JDK) and Java Runtime Environment (JRE) were both Sun's 1.3.1 release. The Integrated Development Environment used was Borland's JBuilder 4 professional. The AspectJ JBuilder extension tool version 1.0.5 was added to JBuilder 4 to assist with the use of AspectJ. The web server and application server used were Apache's Tomcat version 3.3. The database used was MySQL version 3.23.49. The operating system was Microsoft Windows 2000 Professional Edition.

4.2 Tracing

This section describes the implementation of the tracing concern with both AspectJ and Hyper/J.

4.2.1 Tracing with AspectJ

The first concern implemented was the tracing concern. A new package to contain the aspect code was created within the existing AppTrack codebase. Within this package a new package to contain the tracing concern was created.

[2] AspectJ are the only AOP language/tool to publish adoption rates and host active mailing lists

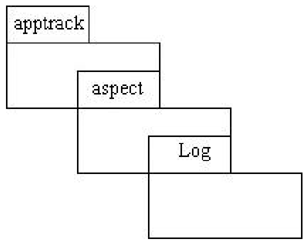

Figure 27: The package structure with the `aspect` and `log` packages added

Figure 27 shows part of the AppTrack package structure. Note the newly added packages to cater for the tracing concern. The full path of the log package is: `ie.tcd.cs.mscnds.apptrack.aspect.log`. Three classes reside within the aforementioned package, implementing the tracing concern. The following is a list of these classes:

- `LogEntry`
- `Logger`
- `TracingAspect`

The `LogEntry` class is a simple bean class. The class contains a variable and a set of accessor methods for each piece of information that we want to log about each method call made during the execution of the system. The table below lists the variables contained in the class:

Type	**Name**	**Description**
`int`	`ID`	A unique numerical identifier assigned to each entry in the log file
`String`	`name`	The name of the method being executed
`String`	`kind`	The type of the joinpoint that identified this point in the systems execution
`String`	`signature`	The full signature of the method being executed
`String`	`args`	The actual arguments supplied to the currently executed method

Table 1: Properties of the `LogEntry` class

The `Logger` class contains the main low-level logging functionality. The constructor of this class initialises a `PrintWriter` object. This object takes a `FileOutputStream` object as a parameter, which in turn takes a `String` argument, the name of the log file on disk. The `println` method of the `PrintWriter` class is now used to send log entries to the log file. The `makeEntry(LogEntry)` method of the `Logger` class accepts as input a `LogEntry` object and it extracts the properties from this object one-by-one, inputting them into the log file. The `makeEntry` method is executed for each method encountered during an execution of the system.

The two classes described combine to provide method tracing functionality. However, they do not separate the tracing concern from the AppTrack codebase that it traces. For each method that is invoked when the system is in operation, a `LogEntry` must be created and the `makeEntry(LogEntry)` method must be invoked. This behaviour is encapsulated within the `TracingAspect` class. It is this class that separates the tracing concern from the rest of concerns in the codebase.

The `TracingAspect` class is an `aspect` class, the new class type introduced by the AspectJ language. This class declares a pointcut to capture the execution of every method not within the `ie.tcd.cs.mscnds.apptrack.aspect.log` package. The reason that method calls within the package containing the tracing concern are not logged is to prevent an infinite sequence of logging calls. The pointcut capturing the method calls is implemented as follows:

```
pointcut trace() : within(ie.tcd.cs.mscnds.apptrack..*) && execution(*
*(..)) && !within(ie.tcd.cs.mscnds.apptrack.aspect.log.*);
```

The above pointcut, named `trace()`, specifies that the execution of any method in any class within any package within the `ie.tcd.cs.mscnds.apptrack` package and not within any class in the `ie.tcd.cs.mscnds.apptrack.aspect.log` package is a joinpoint. Once a joinpoint is identified, advice can be run on it. In this case I chose to run the advice before the execution of method identified by the joinpoint. Below is the advice that is run on each joinpoint picked out by the `trace()` pointcut:

```
before() : trace() {
   name = thisJoinPoint.getSignature().getName();
   kind = thisJoinPoint.getKind();
   signature = "" + thisJoinPoint.getSignature();
   args = printParameters(thisJoinPoint);
   logEntry.setID(ID);
   logEntry.setName(name);
   logEntry.setKind(kind);
   logEntry.setSignature(signature);
   logEntry.setArgs(args);
   logger.makeEntry(logEntry);
   ID++;
}
```

The aspect contains five class scope variables that correspond to those in the `LogEntry` class. These variables are initialised with the appropriate context specific behaviour, which is obtained from the `thisJoinPoint` variable. The arguments to the currently executing method are returned from the `printParameters(JoinPoint)` method. This method uses elements of the `thisJoinPoint` variable to generate a suitably formatted string containing the number of arguments, the type of each argument and the actual values of the arguments for the current execution of the method. All properties of the `LogEntry` class (bar `ID`) are now set to values obtained from the current joinpoint. The information about the current method is then written to the log by passing the `LogEntry` object containing the relevant information as an argument to the `makeEntry(LogEntry)` method of the `Logger` class. The following is an example of the output created by the tracing concern:

```
********** Logging Commenced: Fri Jul 26 12:28:30 BST 2002 **********

ID: 0
Method Name: getPassword
JP Kind: method-execution
Signature: String
ie.tcd.cs.mscnds.apptrack.form.AdminLoginForm.getPassword()
Args: 0

ID: 1
Method Name: setPassword
JP Kind: method-execution
Signature: void
ie.tcd.cs.mscnds.apptrack.form.AdminLoginForm.setPassword(String)
Args: 1
      0. password: java.lang.String -> cormac
```

4.2.2 Tracing with Hyper/J

A test application which involves simple interactions between a person object and a message object, resulting in a message being sent to the terminal, is used as the codebase for the Hyper/J evaluation. It is the method executions of this application that are traced.

Because Hyper/J does not define a language extension like AspectJ, there is no one construct that is used to encapsulate crosscutting behaviour. The `LogEntry` class and `Logger` class can be reused from the AspectJ implementation but a new class, `LoggedClass`, must be created. This class contains a method that will have a similar function as the `before()` advice used in the AspectJ implementation. If only one method from either the `LogEntry` class or the `Logger` class needs to be called to obtain tracing behaviour then this method call could be made from the hypermodule. However, as a set of method invocations must be made, the `LoggedClass` defines a method to encapsulate the method calls necessary. This method is then invoked from the hypermodule. Below is the `LoggedClass`:

```
public class LoggedClass {

   public void beforeMethodInvokation(String methodName) {
      Logger logger = Logger.getInstance();
      LogEntry logEntry = new LogEntry();
      Class currentClass = this.getClass();
      logEntry.setClassName(currentClass.getName());
      logEntry.setMethod(methodName);
      logger.makeEntry(logEntry);
   }
}
```

This class encapsulates the behaviour that needs to be run before the invocation of each method that is run during the execution of the system. Due to the existence of a minor Hyper/J bug when declaring the `logger` variable at the class level, the `Logger` class was modified so that is behaves as a singleton. It can now be declared and instantiated within the method body but still only one instance will be created. The `LogEntry` class was modified to contain only those properties that it was possible to access using Hyper/J's context support.

The package containing the logging functionality and the packages containing the test application are declared to be concerns via a concern mapping; this means that they can be merged together by use of a hypermodule. All concerns in the application make up the hyperspace and they are declared in a hyperspace file. The following is the contents of the hyperspace for the messaging application with the logging concern:

```
hyperspace TestHyperspace
   composable class code.messages.*;
   composable class code.messages.people.*;
   composable class code.logger.*;
```

The bracket relationship of the hypermodule is used to associate a method execution with some behaviour. In this case we bracket any method call with tracing behaviour in the form of the `beforeMethodInvokation` method. The following is the contents of the hypermodule that merges the tracing concern with the rest of the functionality-related concerns in the messaging application:

```
hypermodule TestHyperModule
   hyperslices:
      Feature.Message,
      Feature.People,
      Feature.Logging;
   relationships:
      mergeByName;

      bracket "*"."*"
      before
      Feature.Logging.LoggedClass.beforeMethodInvokation($OperationName
      );

end hypermodule;
```

By composing the concerns listed in the hypermodule using the featured relationships, tracing functionality is achieved without the developer having to create relationships between the concerns at the source code level. The `$OperationName` variable that is supplied as a parameter to the `beforeMethodInvokation` method is a Hyper/J defined variable that resolves to the current operation name at runtime. There is also a `$ClassName` variable that could have been used to acquire the class name but this only returns the short class name, no package information is included.

4.3 Transactions

This section describes the implementation of the transactions concern with both AspectJ and Hyper/J.

4.3.1 Transactions with AspectJ

The implementation of the transactions concern began with the creation of a new package within the `aspect` package to accommodate any new classes implemented as part of the transactions concern. The full path of this package is: `ie.tcd.cs.mscnds.apptrack.aspect.transaction`.

The first stage of the implementation involved deconstructing the existing transaction architecture and removing the dependency between the class providing transactional behaviour and the database handler classes. After this task had been completed the codebase was left with database handler classes for each major object in the system but no way of getting a database connection for these classes to use. It is the establishment and management of these database connections that is the main responsibility of the transactions concern. A description of the way in which the transaction fulfils this responsibility makes up the remainder of this subsection.

The `Transaction` class contains all the transactional functionality. It is an abstract aspect class that is extended by the class `WeaveTransaction`. The `Transaction` class contains the following methods:

- `synchronized setup(JoinPoint)`
- `synchronized commit()`

- `synchronized rollback()`
- `freeConnection(Connection)`

Along side these methods are two abstract pointcuts:

- `abstract standardTransaction()`
- `abstract transactionForUpdate()`

The following pieces of advice run on the joinpoints identified by the pointcuts and their behaviour involves calling the methods listed previously:

- `before() : standardTransaction()`
- `after() : standardTransaction()`
- `before() : transactionForUpdate()`
- `after() : transactionForUpdate()`

Database handler classes require a valid database connection to be passed to their constructor. This connection must come from the class that wants to access the database (classes from the `Action` or `Bean` package). The methods in the `Transaction` class rely on the existence of a class level variable of type `java.sql.Connection` named `connection` within any class affected by the transactions concern. It is this variable that is manipulated by the transactions concern. All uses of this variable from within the `transaction` package are achieved via the use of Java Reflection mechanisms.

The `setup(JoinPoint)` method establishes a connection to the database and sets the `connection` variable in the affected class to have the same value as this new database connection. The `JoinPoint` argument to the method is used to discover the type of the current class and the Java's Reflection mechanisms are used to retrieve and manipulate the `connection` variable of the current class. A call to the `setup(JoinPoint)` method makes up the behaviour of the two advices: `before() : standardTransaction()` and `before() : transactionForUpdate()`.

The `commit()` method invokes the `commit()` method from the `java.sql.Connection` class on the `connection` variable and uses the `freeConnection(Connection)` method to return the connection to the AppTrack connection pool. A call to the `commit()` method makes up the behaviour of the advice: `after : standardTransaction()`.

The `rollback()` method invokes the `rollback()` method from the `java.sql.Connection` class on the `connection` variable and uses the `freeConnection(Connection)` method to return the connection to the AppTrack connection pool. The `rollback()` method is used as part of the behaviour of the advice: `after : transactionForUpdate()`. This advice first attempts to commit the database operation. If the committal of the operation's effects is unsuccessful then the effects are rolled back.

Still unexplained are the two abstract pointcuts listed previously. These pointcuts must be given a concrete definition by the class that specifies the crosscutting behaviour i.e. the class that denotes where transactions are to be used. Once these pointcuts have been given a concrete implementation, any joinpoint identified by them will have the advice defined in the `Transaction` class run before and after it. It is the `WeaveTransaction` class that provides a concrete implementation for the abstract `Transaction` class, hence implementing the abstract pointcuts. The subsequent code segment shows the implementation of the abstract pointcut, `transactionForUpdate()`:

```
   pointcut transactionForUpdate() :
      execution(ActionForward UpdateDocsListAction.doPerform(..)) ||
      execution(ActionForward UpdateLocationAction.doPerform(..)) ||
      execution(ActionForward UpdateStatusAction.doPerform(..));
```

The pointcut picks out three method execution joinpoints. These are the last three methods named in the bind attachment for this concern in figure 11. They are all method executions that require data access to carry out their functionality correctly. The `||` operator indicates that the advice will be run if any of the execution joinpoints are reached. The remaining pointcut, `standardTransaction` is implemented in the same manner, specifying a collection of joinpoints to be advised.

4.3.2 Transactions with Hyper/J

To implement the separation of the transactions concern with Hyper/J the test application was extended to contain some data access operations that were invoked when the application executed. A concern mapping was created for the transactions concern and the transaction package was included in the hyperspace specification. The `Transaction` class was modified to work with a standard Java system. This meant that all the AspectJ specific code was removed. The standard methods remained and they still manipulated the `connection` variable of the class accessing the database. The class is no longer abstract.

To achieve the merging of the transactions concern with the main application codebase, the four pieces of advice from the AspectJ implementation needed to be recreated using Hyper/J and standard Java. The pieces of advice that only involved one method call could be implemented directly in the hypermodule. The following hypermodule specification shows the augmentation of a method with transactional behaviour:

```
hypermodule TestHyperModule
   hyperslices:
      Feature.Message,
      Feature.People,
      Feature.Transaction;

      relationships:
         mergeByName;

      bracket "Message"."printMessage"
         before
            Feature.Transaction.Transaction.setup($ClassName),
         after
            Feature.Transaction.WeaveTransaction.afterSpecial;

end hypermodule;
```

Please note that the logging concern has been removed for clarity

The hypermodule shows that transactional behaviour is added to the data access functionality in the `printMessage()` method of the `Message` class. The `before` section of the bracket relationship can directly access the `Transaction` class's `setup` method, passing the name of the current class (obtained using the reflective `$ClassName` variable) as opposed to a `JoinPoint`. Because the data that this operation accesses could potentially be accessed by several users simultaneously, the `after` section of the bracket relationship must perform similar duties to the `after` advice for the `transactionForUpdate()` pointcut. This pointcut involves several method calls and so a method must be created to encapsulate these method calls. The `WeaveTransaction` class in the Hyper/J implementation is now a standard Java class that extends `Transaction` and contains only one method. This method is the `afterSpecial()` method, which attempts to commit a database operation. If the commit is not possible then a rollback is carried out. It is this method that is invoked by the `after` section of the bracket relationship in the hypermodule presented previously.

4.4 Enforcing Factory Design Pattern

This section describes the implementation of the enforce factory design pattern concern with both AspectJ and Hyper/J. Due to limitations in the current implementation of the Hyper/J tool, this concern could not be successfully implemented with Hyper/J.

4.4.1 Enforce Factory Design Pattern with AspectJ

The implementation of this concern did not involve the re-engineering of any existing AppTrack classes. It was added to the system during the re-engineering process and was intended to illustrate how this kind of concern can be conveniently implemented using aspect-oriented software development techniques. Firstly a new package, enforceFactory was added within the aspect package. This package contains the classes needed to implement the factory enforcing concern. With AspectJ, the implementation requires only one aspect class and one custom exception class:

```
aspect EnforceFactory {

   pointcut illegalStatusCreation() :
      !within(ie.tcd.cs.mscnds.apptrack.bean.status.StatusFactory) &&
      call(Status*.new());

   before() : illegalStatusCreation() {
      try {
        throw new IllegalStatusCreationException("Illegal Status
           Creation in " +
           thisJoinPoint.getSourceLocation().getFileName()
           + ". <error message & recommendation>");
      }
      catch(IllegalStatusCreationException isce){}
   }
}
```

The previous code segment is taken from the main body of the aspect class EnforceFactory. The pointcut, illegalStatusCreation(), identifies the creation, by means of the new() operation, of any Status object that did not occur within the StatusFactory class. The * property is used to avoid listing every Status class that we want to affect. These are classes listed in the bind attachment in figure 14. Any class whose name begins with the word Status (as all Status classes do) will be affected by this pointcut. The creation of Status* objects within the StatusFactory is permitted because the status factory design pattern used specifies that this is the correct place for the creation of these objects.

Advice is declared for the illegalStatusCreation() pointcut. This before() advice throws an IllegalStatusCreationException(String) exception and uses the thisJoinPoint variable to pass the file name of the current class to the constructor of the exception. Also passed to the exception are some recommendations about the use of the status factory pattern and arbitrary creation of Status* objects throughout the AppTrack codebase (these recommendations are not shown for clarity reasons). If a developer is working on the AppTrack system and misuses any Status class, this concern will identify this behaviour and alert the developer to the coding standard i.e. the status factory design pattern.

4.4.2 Enforcing Factory Design Pattern with Hyper/J

A new package was added to the test application to implement the enforcement of the status factory design pattern. This package contains two classes, `DummyA` and `DummyB` that are basically dummy objects; they both contain only a simple constructor. These classes are analogous to the `Status` classes in the AppTrack codebase. Another class, `EnforcePattern`, contains a method intended to act in a similar manner to the `before()` advice in the AspectJ implementation. A concern mapping was created for the new package and the package was added to the hyperspace specification. Below is the hypermodule containing the merge declarations for the concern in question:

```
hypermodule TestHyperModule
   hyperslices:
      Feature.Message,
      Feature.People,
      Feature.EnforceDesignPattern;

   relationships:
      mergeByName;

      bracket "*"."*"
      before
          Feature.EnforceDesignPattern.EnforcePattern.beforeInit
            ($ClassName, $OperationName);

end hypermodule;
```

Please note that the logging and transactions concerns have been removed for clarity

The hypermodule shows that every method call made in the system is bracketed with the `beforeInit(String, String)` method. This method is required to use the parameters passed to it, the name of the current class and the name of the current method, to discover if the dummy classes are being invoked from within the appropriate package – the package they reside in (this corresponds to the `StatusFactory`). Using the method name it is possible to see if a constructor has been called (the `$OperationName` variable resolves to `<init>`) and using the class name it is possible to see which class's constructor was invoked and hence which package it is in. However, it is not possible using Hyper/J to deduce which class invoked the constructor. Therefore the concern cannot be successfully implemented.

4.5 Database Compatibility

This section describes the implementation of the database compatibility concern with both AspectJ and Hyper/J. Due to limitations in the current implementation of the Hyper/J tool, this concern could not be successfully implemented with Hyper/J.

4.5.1 Database Compatibility with AspectJ

The implementation of this concern began with the removal of the existing scattered and tangled code from the database handler classes affected by the concern. Once these classes had been cleansed of any unwanted code, they were implemented for the perfect case i.e. an environment with a database free from issues accepting the apostrophe character (‘) available for use as AppTrack’s data source. The AppTrack codebase was now ready for the addition of a newly implemented concern to cater for the current database issues.

The `DBCompatiability` package was created inside the `aspect` package. One class was added to this package, `FixQuotes`, which acts as a proxy for database handler methods that potentially have to insert string data containing apostrophes. The class contains proxy code for each method that requires delegation to this class and also a method called `fixQuotes(String)` that transforms string variables so that they are suitable for insertion to the MySQL database. Below is one such proxy method implementation:

```
aspect FixQuotes {

   pointcut fixApplicant(ApplicantHandler handler, BeanDB bean,
      StringBuffer sql) : execution(void
      ApplicantHandler.setQueryParams(BeanDB, StringBuffer)) &&
      target(handler) && args(bean, sql);

   void around(ApplicantHandler handler, BeanDB bean, StringBuffer sql)
      : fixApplicant(handler, bean, sql) {
      Applicant applicant = (Applicant)bean;
      sql.append("email='" + fixQuotes(applicant.getEmail()) + "', ");
      sql.append("first_name='" + fixQuotes(applicant.getFirstName()) +
      "', ");
      sql.append("last_name='" + fixQuotes(applicant.getLastName()) +
      "', ");
      sql.append("password='" + fixQuotes(applicant.getPassword()) +
      "', ");
      sql.append("telephone='" + fixQuotes(applicant.getPhone()) + "',
      ");
```

```
        sql.append("street='" + fixQuotes(applicant.getStreet()) + "',
        ");
        sql.append("city='" + fixQuotes(applicant.getCity()) + "', ");
        sql.append("state='" + fixQuotes(applicant.getState()) + "', ");
        sql.append("country='" + fixQuotes(applicant.getCountry()) + "',
        ");
        sql.append("post_code='" + fixQuotes(applicant.getPostalCode()) +
        "'");
    }
    …
}
```

The pointcut `fixApplicant()` isolates the execution of the `setQueryParams()` method of the `ApplicantHandler` class (this class is named in the bind attachment in figure 17). The parameters passed to the pointcut are an instance of the class that the executing method resides in and the parameters that are passed to the method. Because these parameters are parameters of the pointcut they can be accessed from the advice.

The key to implementing proxy style delegation behaviour in AspectJ is the `around()` advice construct. Around advice runs in place of the method picked out by the pointcut. The advice declaration accepts the same parameters as the pointcut and it then uses those parameters when indicating that it is related to the pointcut. The actual runtime parameters are now accessible within the body of the advice. Using these parameters, the `setQueryParams()` method is given a MySQL specific implementation, with the `fixQuotes` method filtering any strings containing apostrophes and amending them as appropriate. Following the execution of the `around()` advice, control is passed back to the AppTrack codebase method that made the initial call to the `setQueryParams()` method. For any database handler class to acquire this proxy behaviour for its `setQueryParams()` method, an entry (relevant pointcut and advice) similar to the one displayed must be made in the `FixQuotes` class.

4.5.2 Database Compatibility with Hyper/J

According to [36] "the `override` composition relationship indicates that one unit overrides one or more units with which it corresponds". This `override` relationship is similar to the `around()` advice specified in AspectJ. The relationship specifies that one method's implementation is to be overridden by that of another so that essentially one method is acting as a proxy for another. It would seem that this `override` relationship provides the basis for a good implementation of the database compatibility concern with Hyper/J. However, [36] also states that the override composition relationship does not work at present i.e. it has not been implemented in the current version of the Hyper/J tool. As a result this concern cannot be suitably implemented with the current version of Hyper/J.

4.6 Design By Contract (Preconditions)

This section describes the implementation of the design by contract (preconditions) concern with both AspectJ and Hyper/J. Due to limitations in the current implementation of the Hyper/J tool, this concern could not be successfully implemented with Hyper/J.

4.6.1 Design By Contract (Preconditions) with AspectJ

Implementation of the preconditions concern began with the removal of precondition-style code from several mail sending methods in various `Action` classes. This left the mail sending methods free of tangling, encapsulating only their logical behaviour. Preconditions could now be added to these methods without directly affecting the internal body of the methods.

The `dbc` package was created within the `aspect` package. This new package contains the aspect class that implements the preconditions on the mail sending methods. The class, `SendMailPreconditions`, uses pointcuts to identify where a precondition needs to be added, and `around()` advice to achieve the same behaviour as the code removed from the mail sending methods in the `Action` classes. The following is a section of the `SendMailPreconditions` class that implements a precondition for methods of the `SaveRegistrationAction` class:

```
aspect SendMailPreconditions {

   pointcut registrationMails(SaveRegistrationAction sra, Applicant
      applicant) : call(void
      SaveRegistrationAction.sendMail*(Applicant)) &&
      target(sra) && args(applicant);

   void around(SaveRegistrationAction sra, Applicant applicant) :
      registrationMails(sra, applicant) {
      if (applicant == null) {
         handlePreconditionFailure(thisJoinPoint);
         return;
      }
      else {
         proceed(sra, applicant);
      }
   }
…
}
```

The `registrationMails()` pointcut identifies calls to the mail sending methods in the `SaveRegistrationAction` class (this is illustrated in the bind attachment in figure 20). The names of these methods begin with the words `sendMail`, so the `*` property is used to advise any method in the specified class that matches the pattern `sendMail*`. Two arguments are passed to the pointcut, an instance of the class that the advised methods belong to and an instance of the `Applicant` bean object. As with the previous concern, these parameters are supplied to the pointcut so that they will be available for access in the advice related to the pointcut.

The pointcut is advised with `around()` advice. This advice runs in place of the actual method but unlike in the previous concern it does not act as a pure proxy for the method. The instance of the `Applicant` class is evaluated to asses whether it is null or not. If the instance is valid (not null) then a call to the AspectJ–specific `proceed()` method is issued. The `proceed()` method takes as parameters an instance of the class that contains the method being advised as well as the parameters to the same method. The basic behaviour of the `proceed()` method is to invoke the method being advised. So, the mail sending methods do not delegate totally to the advice, they simply delegate their precondition behaviour. If the precondition holds then the rest of the method is executed as normal (via a call to `proceed()`). If however the precondition does not hold (if the `Applicant` object has not been initialised) then the appropriate behaviour is taken (defined in the `handlePreconditionFailure` method) and a `return` statement is issued. The advised method does not then proceed with its standard behaviour.

For any other class to acquire preconditions on its mail sending methods, an entry (pointcut and advice) similar to the one displayed previously must be made in the `SendMailPreconditions` class. There are currently three different types of mail sending precondition implemented in the re-engineered version of AppTrack. These different precondition implementations are needed because the methods being advised accept different combinations of runtime parameters.

4.6.2 Design By Contract (Preconditions) with Hyper/J

Similar to the database compatibility concern, a suitable implementation of the preconditions concern requires the use of the `override` relationship. As we know this relationship has been left unimplemented in the current version of Hyper/J. For this reason an appropriate implementation cannot be achieved with Hyper/J. There is another problem, apart from the non-implementation of the override relationship, which hinders the implementation of this concern with Hyper/J. There does not appear to be a way to access the parameters of the method that is being overridden. This functionality is essential for the implementation of the preconditions concern.

4.7 Exception Handling

This section describes the implementation of the exception handling concern with both AspectJ and Hyper/J. Due to limitations in the current implementation of the Hyper/J tool, this concern could not be successfully implemented with Hyper/J.

4.7.1 Exception Handling with AspectJ

The `Action` package contains the classes that implement the main functionality requirements of the AppTrack System. The implementation of the exception handling concern began with the identification of any commonality amongst a) the types of exceptions thrown by these `Action` classes and b) how these exceptions are handled. It was discovered that there are four different types of exception thrown throughout the classes of the `Action` package. Associated with each of these exceptions is some handling behaviour, repeated at each point that the exception is handled. This is a great source of scattering.

The Struts framework defines an object of type `ActionErrors`. When an error occurs within an `Action` class, a new `ActionError` object is created to refer to the relevant error message in the `ApplicationResources.properties` file. This `ActionError` object is then added to the `ActionErrors` collection, which is accessed by the end-user interface layer to display the errors to the user. Each `Action` class has an `ActionErrors` object associated with it and the exception handling code in these classes manipulates this object. The separated exception handling concern needs to interact with this object without scattering exception handling code throughout the classes of the `Action` package.

A package, `exceptionHandler`, was created within the `aspect` package to contain the classes making up the exception handling concern. The `ExceptionHandler` class, the parent of all other classes in the package, has three key constructs:

- An abstract pointcut: `exceptionHandler`
- `after()` advice for the `exceptionHandler` pointcut
- An abstract method: `ActionErrors handleException(ActionErrors errors)`

Aspect classes that extend the `ExceptionHandler` class provide a concrete implementation for the pointcut. These concrete pointcuts specify where the exception handling behaviour is to be weaved with the `Action` classes. Below is an example of a concrete implementation of the `exceptionHandler` pointcut:

```
pointcut exceptionHandler() : within(ie.tcd.cs.mscnds.apptrack.action.*)
   && handler(java.sql.SQLException);
```

This implementation of the abstract pointcut isolates any pieces of exception handling code that handle an exception of type `java.sql.SQLException`. A piece of exception handling code is identified as such by the presence of a `catch(Exception)` statement. This pointcut is defined in the `SQLExceptionHandler` class.

The `after()` advice on the abstract pointcut in the `ExceptionHandler` class contains the code that manipulates the `ActionErrors` object of the `Action` classes. Some codebase preparation was necessary to support this concern. To ensure that the `ActionErrors` variable is present at class level in each `Action` class, each `Action` class extends a class called `ErrorsAction`, which contains a public `ActionErrors` variable. Java's dynamic binding (via casting) mechanism can now be used to cast the object returned by the `getThis()` call of the `thisJoinPoint` variable (which is of type `Object`) to an object of type `ErrorsAction`, representing the object currently executing. The `after()` advice retrieves the `ActionErrors` variable from the current `Action` class and gives it a new value. This new value is the result of adding the new `ActionError` to the existing collection of action errors. The advice achieves the addition of a new error to the existing collection by invoking the abstract method `handleException()`. The following code illustrates the concepts just described:

```
after() : exceptionHandler() {
      ErrorsAction action = (ErrorsAction)thisJoinPoint.getThis();
      ActionErrors errors = action.getErrors();
      errors = this.handleException(errors);
   }
```

The `handleException()` method should contain the behaviour that was once scattered across the `Action` classes that catch the exception. The method is given a concrete implementation in each class that extends the `ExceptionHandler` class. Below is the implementation contained in the aspect from which the concrete implementation of the `exceptionHandler()` pointcut shown previously was taken:

```
protected ActionErrors handleException(ActionErrors errors) {
   errors.add("generalDBError", new
      ActionError("error.database.general"));
   return errors;
}
```

This is an example of the type of behaviour that is executed when an exception handler is identified by the pointcut.

The `exceptionHandler` package contains four concrete implementations of the `ExceptionHandler` class, one for each of the exception types identified at the start of the re-engineering implementation process.

4.7.2 Exception Handling with Hyper/J

Implementation of the exception handling concern involves isolating units in the code at a finer level of granularity than that required for the implementation of any previous concern. Up until now the method construct has been the finest unit of granularity dealt with. The exception handling concern requires the identification of specific lines of code, as opposed to method or classes. The current version of Hyper/J does not provide support for this kind of behaviour. The Hyper/J manual [36] states “To date, we have worked only with units at the granularity of declarations (e.g., methods, functions, classes, UML diagrams) rather than lower-level constructs, such as statements or expressions”. The `throws()` statement that is used in the implementation of exception handling concerns in Java is an example of a lower-level construct not yet catered for by the Hyper/J tool. It is for this reason that the exception handling concern cannot be suitably implemented with the current version of Hyper/J.

4.8 Recording Bean Properties Modification

This section describes the implementation of the recording bean properties modification concern with both AspectJ and Hyper/J. Due to limitations in the current implementation of the Hyper/J tool, this concern could not be successfully implemented with Hyper/J.

4.8.1 Recording Bean Properties Modification with AspectJ

The first task in the implementation of this concern was the removal of the tangled property and associated accessor methods from the `Applicant` bean class. Following the completion of the first task, the usages of the accessor methods throughout the `Applicant` class were removed. The class was now ready to have this concern managed by the new aspect-oriented implementation.

A new package, `applicantManager`, was created within the `aspect` package to accommodate the classes created during the implementation. The `ManageApplicantDetailsUpdate` class is an aspect class that has two main purposes. The first of these is the introduction of a variable and its associated accessor methods into the `Applicant` class. These correspond to the previously removed property and accessor methods. They are re-introduced so that they are available to the user interface layer (JSPs). The following code segments show the syntax for variable and method introduction in AspectJ:

```
private Date Applicant.lastAddressUpdateTime;

public Date Applicant.getLastAddressUpdateTime() {
   if (this.lastAddressUpdateTime == null) {
        this.lastAddressUpdateTime = new Date();
    }
    return lastAddressUpdateTime;
}

public void Applicant.setLastAddressUpdateTime(Date
    lastAddressUpdateTime) {
    this.lastAddressUpdateTime = lastAddressUpdateTime;
}
```

The first line of code illustrates the introduction of a variable of type `java.util.Date` into the `Applicant` class. The only difference in the declaration of this variable that distinguishes it as an introduction statement is that the name of the variable is preceded by name of the class that the variable is to be introduced into. This class name is followed by the dot (`.`) operator. Interface layer clients of the `Applicant` class can access the introduced variable and methods as if they were standard members of the class.

The second purpose of the `ManageApplicantDetailsUpdate` class is to define a pointcut that identifies all the points in the `Applicant` class should make calls to the setter method of the `lastAddressUpdateTime` variable. This pointcut should be advised with `before()` advice that invokes the setter method. The subsequent code segments illustrate the implementation of these requirements:

```
pointcut timestampUpdateRequired() :
        execution(* Applicant.setStreet(..))     ||
        execution(* Applicant.setCity(..))       ||
        execution(* Applicant.setState(..))      ||
        execution(* Applicant.setCountry(..))    ||
        execution(* Applicant.setPostalCode(..)) ||
        execution(* Applicant.setPhone(..))      ||
        execution(* Applicant.setEmail(..));

    before() : timestampUpdateRequired() {
        Applicant applicant = (Applicant)thisJoinPoint.getThis();
        applicant.setLastAddressUpdateTime(new Date());
    }
```

The pointcut `timestampUpdateRequired()` identifies any methods in the `Applicant` class whose invocation affects the important properties of the applicant profile from the point of view of the postgraduate administrators. The `before()` advice specified for this pointcut uses the introduced accessor methods of the `Applicant` class to update the introduced `lastAddressUpdateTime` variable. This information is now available to the postgraduate administration staff using the system.

4.8.2 Recording Bean Properties Modification with Hyper/J

The implementation of this concern requires obtaining a reference to the currently executing `Applicant` object and invoking a method on it to update the `lastAddressUpdateTime` variable. In AspectJ, the `thisJoinPoint` variable is used to obtain the currently executing object instance. This instance is then cast to an object of type `Applicant`. Hyper/J does not provide a facility for accessing the currently executing object. For this reason the concern cannot be implemented using the current version of Hyper/J.

5 EVALUATION

This, the penultimate chapter, seeks to provide a detailed evaluation of the use of aspect-oriented software development techniques in the re-implementation of AppTrack, an object-oriented distributed information system. For each concern designed and implemented, a separate evaluation subsection will be presented.

Thus far, the most widely used metric in the evaluation of aspect-oriented re-engineering efforts has been lines of code (LOC). This metric measures the increase or reduction in the number of lines of code required to implement a concern using aspect-oriented techniques as opposed to the original implementation methodology. Whilst this metric is useful, it does not address the primary aims of AOSD. A comprehensive evaluation of AOSD should seek to address the intangible, less measurable properties of software systems that are of greater importance to a comprehensive evaluation of AOSD than LOC. For this reason the evaluation is more subjective than scientific. The concerns described in the preceding two chapters will be evaluated under the following headings:

- Separation Achieved
- Changeability/Extensibility
- Reusability
- Pluggability
- Complexity
- Productivity
- Lines of Code

The separation achieved sections will address the degree of separation of concerns achieved through the use of AOSD during the re-implementation process. This can be measured exactly by assessing the lines of code relating to a particular concern that remain scattered or tangled following the separation process.

Changeability/Extensibility deals with the ease of change and extension of the re-implemented AOSD-based concern and the system in general. This is quite a subjective area. The changeability of AOSD systems was examined in [32]. For their experiments they videotaped participants working on change tasks and analysed such factors as the time spent studying the code, time spent coding the solution, general patterns of activity (e.g. switching between classes) and the code written. Whilst this would seem like the ideal way to analyse the changeability/extensibility of the AppTrack system, I wasn't in a position to conduct such a study due to the nature of the environment in which the original AppTrack implementation was developed. Also, scientific analysis is made harder by the fact that the original development team consisted of four members and I was the sole developer working on the re-implementation. Therefore, the information contained in the changeability/extensibility sections is based on my personal experience in refactoring code in both the original implementation and the AOSD re-implementation, tasks that I have had vast experience with.

The reusability sections will consider whether or not an aspect-oriented concern can be reused in another environment without further significant manipulation. I discovered no prior work relating to the measurement of the reuse properties of either AspectJ or Hyper/J concerns. In general, a loosely coupled implementation represents the key to higher code reuse. The information in the reuse sections of the evaluation are based on my experience with both a) attempting to reuse and b) considering the reuse of the concern implementations in both the original application and the aspect-oriented re-implementation.

The pluggability sections will discuss how closely tied to the application a concern is, and assess whether or not it can be easily removed and added again. With no publications on techniques for measuring the pluggability of AOP systems, again I relied on my experience in attempting to unplug both the original concern implementations and the aspect-oriented re-implementations.

The complexity section considers any changes in the complexity of the system since the introduction of the re-implemented aspect-oriented concern. The authors of [40] seek to measure complexity by comparing the number of classes, methods, LOC etc. required in both the object-oriented and aspect-oriented implementations. Whilst this formed part of my approach to measuring complexity I feel that this is not an entirely accurate method of analysis. In my experience, it is often the case that a solution with less code is more complex than a solution with more code. This is because numerous events may occur as a result of the execution of one line of code, resulting in rather complex code. My experiences in coding both the object-oriented and aspect-oriented solutions formed the main basis for analysis of the complexity of the concerns. This takes into account factors such as readability, comprehensibility, separation achieved and level of familiarity with the AppTrack system and AOSD necessary to understand the solution.

The productivity section discusses whether or not the new solution furthers the productivity of the developers working on the system. My experiences in developing both solutions form the basis of the analysis presented in this section.. I estimated how long tasks that took two pairs of object-oriented pair programmers [46] would have taken one of my team-mates or I on our own, and then compared that to that time that it took me to implement the aspect-oriented solution. Although this seems extremely subjective, my findings appear to concur with those of other noted developers on AOSD mailing lists where this subject has been discussed at length [41].

Finally, the LOC section compares the LOC required for the object-oriented and aspect-oriented solutions. This can be scientifically measured. The lines of code removed from the object-oriented solution were counted as the re-implementation process advanced, and the lines of aspect-oriented code added to the system were calculated at the end of the re-implementation process.

The remainder of this chapter will be composed of an evaluation subsection for each concern. Each subsection will contain a discussion of the concern under each of the evaluation criteria described above. A summary discussion of the evaluation is then presented. An evaluation of the composition patterns model is also contained in this chapter.

5.1 Tracing

The following section contains an evaluation of the use of aspect-oriented software development techniques in the re-implementation of the tracing concern for the AppTrack System.

5.1.1 Separation Achieved

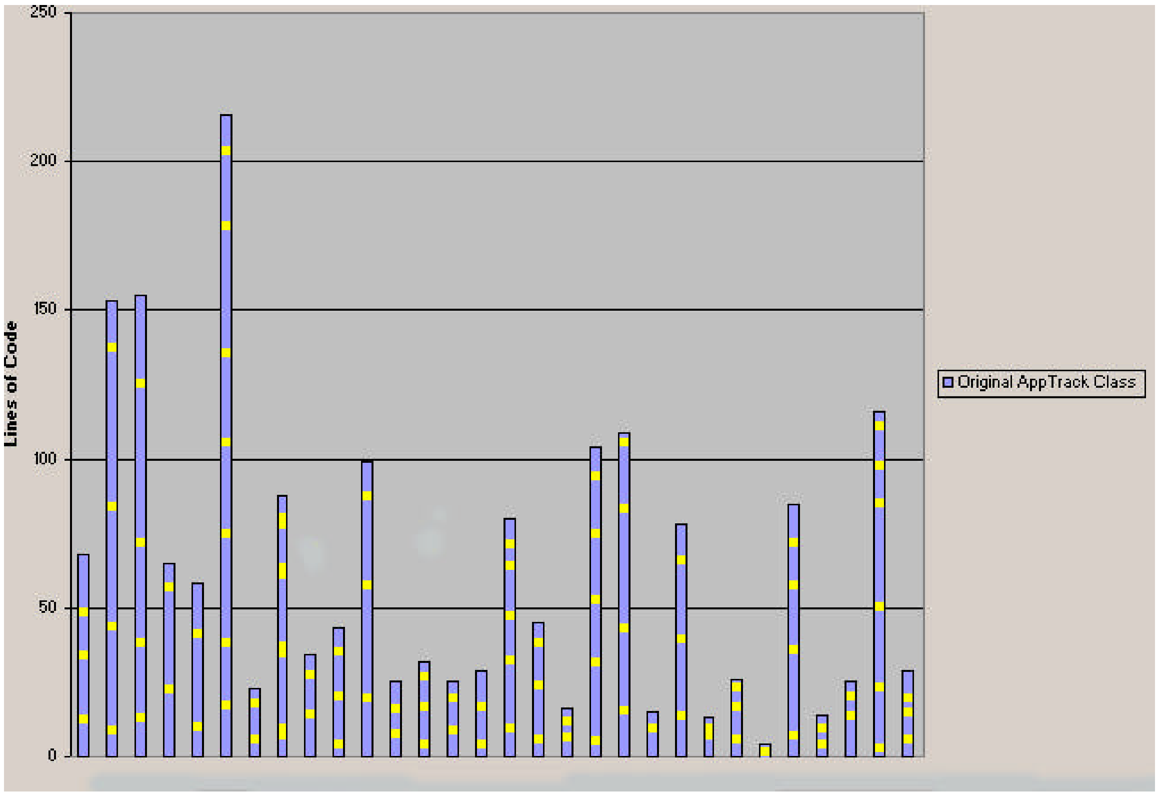

Figure 28: Scattering in original tracing concern

Figure 28 illustrates the scattering of code relating to the tracing concern across a randomly selected cross-section of the classes in the original AppTrack implementation. Each bar on diagram represents an object-oriented class. The light coloured sections within each class represent code that relates to the logging concern. It is obvious that there is a huge degree of both scattering and tangling.

Figure 29 illustrates the total separation of code relating to the tracing concern from the main AppTrack codebase. The three classes to the right-hand side of the diagram are the three new classes added during the re-implementation of the concern using AOSD techniques with both AspectJ and Hyper/J. These classes encapsulate the tracing concern and the remainder of the AppTrack codebase is oblivious to their existence.

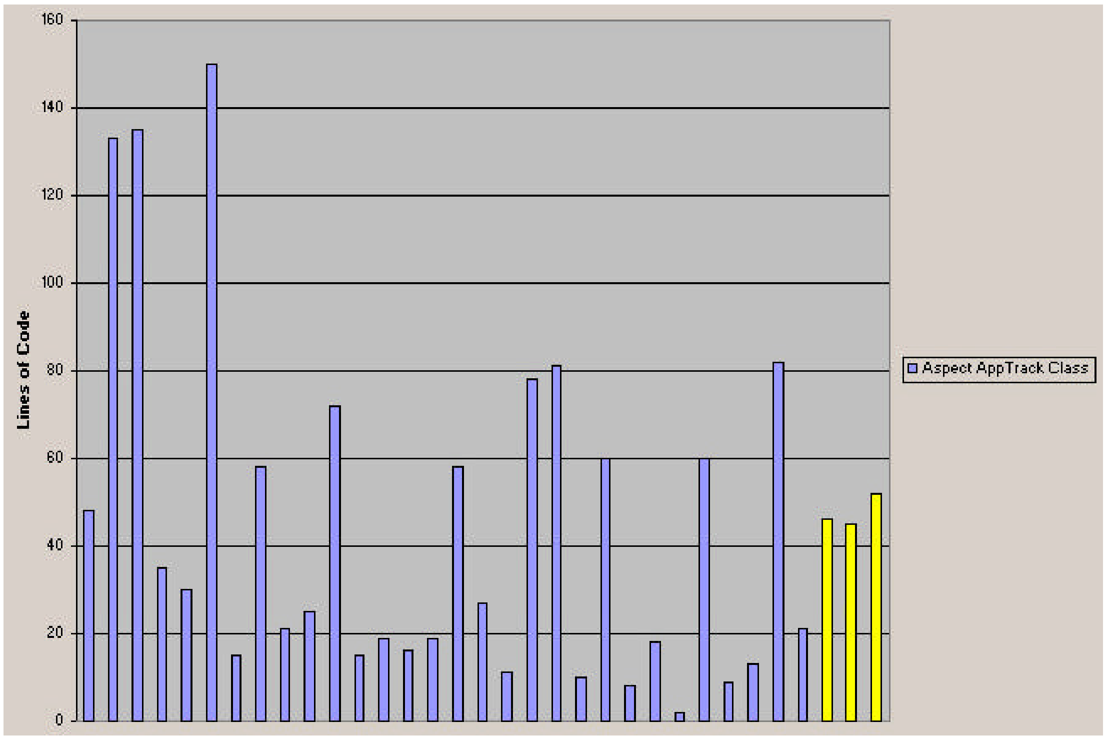

Figure 29: Separation of tracing concern

5.1.2 Changeability/Extensibility

The implementation of the logging concern in both AspectJ and Hyper/J can be easily modified to alter the information that is recorded in the log file at runtime. This requires making minor, logically related, alterations to each of the three classes involved in the implementation of the concern. In AspectJ for example, to log a new property about each method you would need to add that property to the `LogEntry` bean class, set the property in the `TracingAspect` class and write the property to the log file in the `Logger` class. To make such a change in the standard object-oriented implementation would require making changes to every method that requires logging, as well as the `LogEntry` and `Logger` classes.

5.1.3 Reusability

To reuse the AspectJ implementation of the tracing concern in a different environment requires only a straightforward modification to the pointcut that selects the methods to be logged. Reuse of the Hyper/J implementation requires a change to the tracing bracket relationship in the hypermodule. This is because the current implementation of the pointcut/bracket relationship references methods specific to the AppTrack application. However, the AspectJ implementation it could be made completely reusable by creating an abstract class containing an abstract pointcut (the rest of the aspect would say as it is). The developer could then define a class that extends this abstract class and provides a concrete implementation of the abstract pointcut.

5.1.4 Pluggability

The tracing concern is completely pluggable. The three concern classes have no dependencies with any class outside of the package in which they reside. This means that a production release of the code can be generated simply by compiling the system without the inclusion of the classes making up the tracing concern. This is far more convenient than manually deleting scattered and tangled tracing code from every method in the codebase.

5.1.5 Complexity

The aspect-oriented tracing concern reduces the overall complexity of the AppTrack system. Although there is learning curve involved to get up to a sufficient level of understanding and competence with AspectJ and Hyper/J, the modularity achieved by the new implementation of the tracing concern makes the remainder of the codebase more streamlined, readable and generally more comprehensible. The new implementation enforces consistent behaviour. All methods are treated equally i.e. the same information is logged for each method. This was not necessarily the case with the old solution (due to scattering), a phenomenon that often caused confusion. The benefits (in terms of time saving) of the new solution definitely negate the disadvantage of the learning curve involved to understand the solution.

5.1.6 Productivity

A full implementation of the tracing concern across the whole codebase using the standard object-oriented approach is less productive in terms of developer time than the aspect-oriented implementation. The aspect-oriented solution took less time to develop and the real gain is in the area of future maintenance. The new tracing concern requires minimal modification to be removed completely, augmented or scaled down. Implementing such changes in the object-oriented solution requires much effort on behalf of the developers due to the scattered nature of the implementation.

5.1.7 Lines of Code

Approximately 360 methods are logged by the aspect-oriented implementation of the tracing concern, which accounts for 105 LOC. If the object-oriented tracing concern was implemented across the whole codebase then each method would be supplemented with approximately 5 lines of tracing code. As well as this, the two supporting classes required, `LogEntry` and `Logger`, make up 60 LOC. This brings the total number of LOC needed for a full object-oriented implementation to 1860. The aspect-oriented solution requires only 5.65% of the code necessary for an object-oriented solution. Figure 30 illustrates the vast difference in LOC required to implement the tracing concern using the two methodologies.

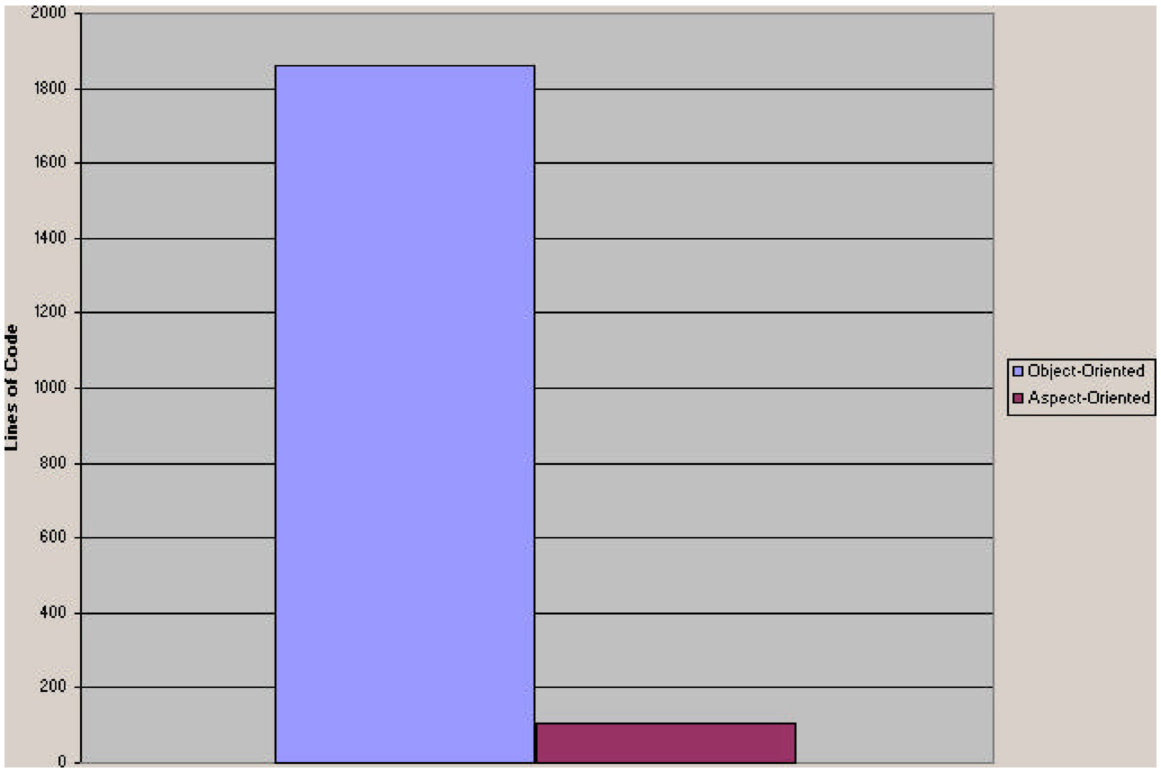

Figure 30: Tracing concern LOC comparison

5.1.8 Conclusion

The aspect-oriented implementation of the tracing concern resulted in maximum separation. The separation achieved, owing to the suitability of the two languages used, resulted in benefits under each of the evaluation criteria. There was a significant reduction in the size of the codebase following the re-implementation of this concern.

5.2 Transactions

The following section contains an evaluation of the use of aspect-oriented software development techniques in the re-implementation of the database transactions concern for the AppTrack System.

5.2.1 Separation Achieved

Figure 31 illustrates the tangling caused by the original implementation of the transactions concern within the core AppTrack codebase. The light coloured sections represent code that directly related to transaction management. The light coloured bar on the far right-hand side of the diagram is the `Transaction` class that all the other classes are referencing. All access to database handler classes occurs via the `Transaction` class, this is the reason for the large amount of tangling.

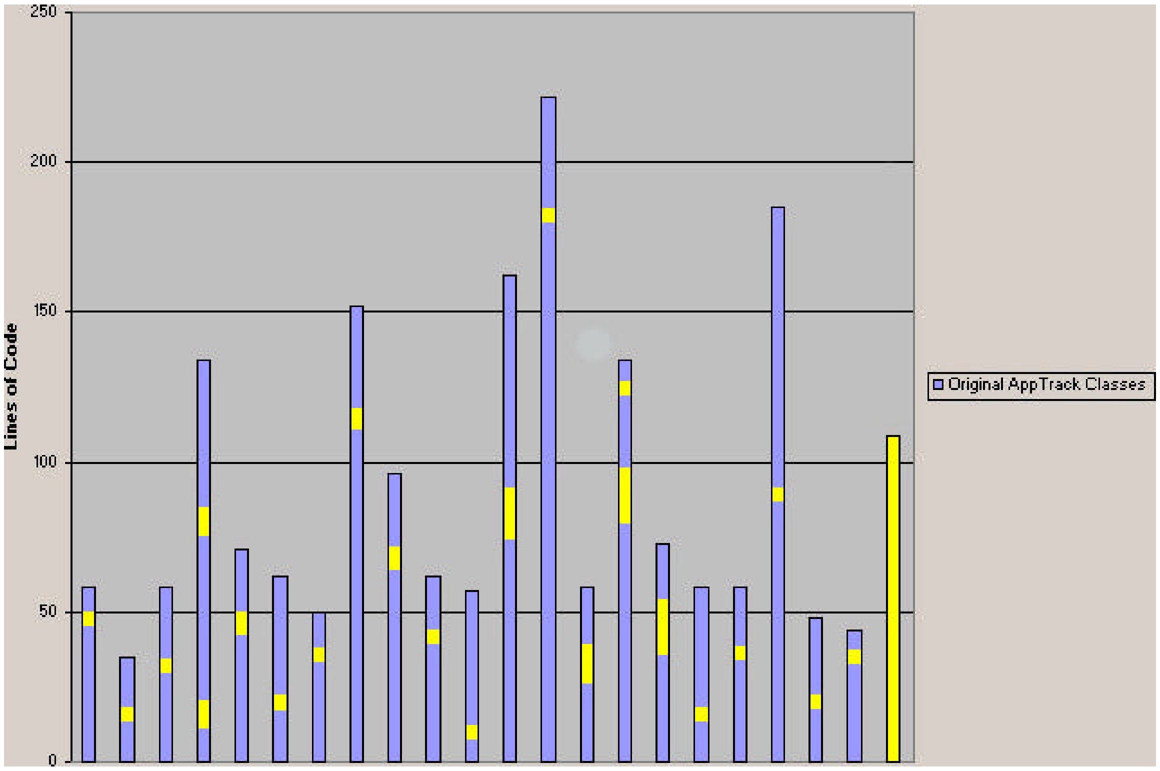

Figure 31: Tangling in original transactions concern

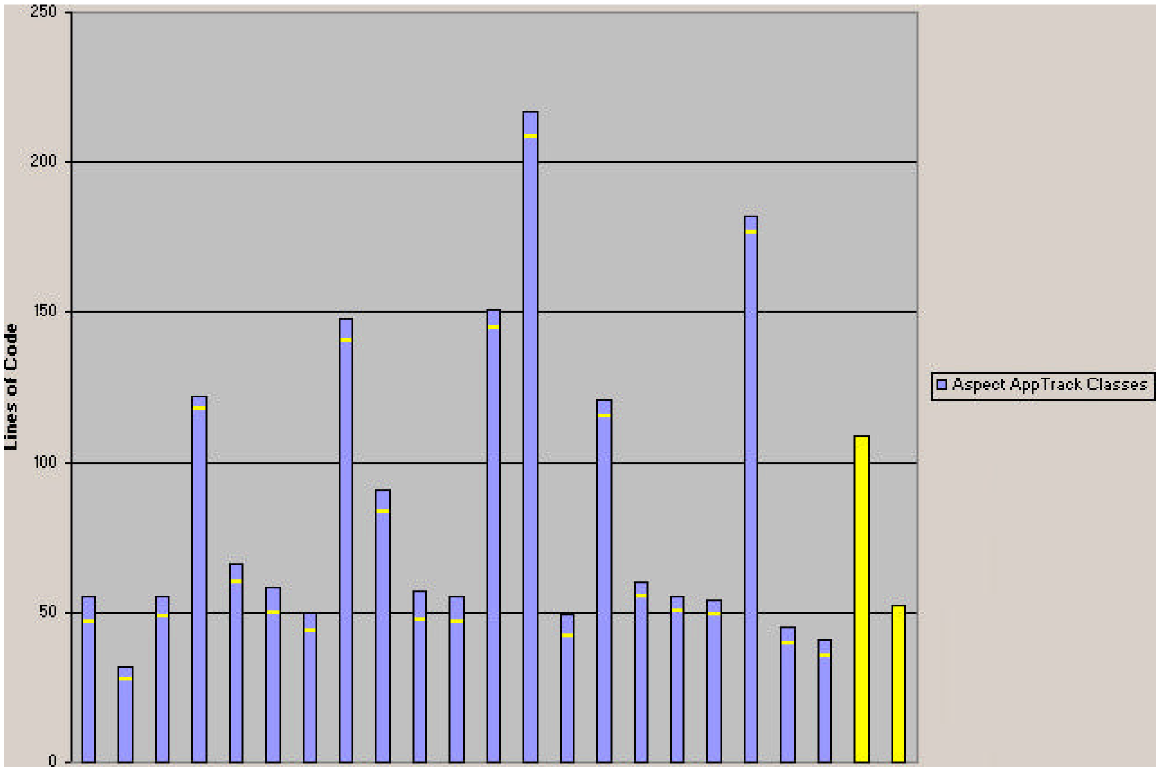

Figure 32: Separation of transactions concern

Figure 32 shows the resulting reduction in tangling following the aspect-oriented implementation of the database transactions concern. The number of transaction-related LOC evident in each non-transaction class has significantly reduced. Each class now contains only one line of code related to the transactions concern. This line declares the `java.sql.Connection` variable that is manipulated by the `Transaction` aspect. The two bars to the right-most side of the figure 32 represent the two aspect classes the implement the aspect-oriented transactions concern. Whilst total separation of transactions concern has not been achieved, a significant level undoubtedly has.

5.2.2 Changeability/Extensibility

The actual methods that provide transactional behaviour to the system in the aspect-oriented implementation are, at a high level, basically the same as those in the object-oriented implementation. Hence there no real gain regarding the evolvability of these methods. However, significant difference appears in the area of extending the system as a whole. A new class that requires database access now has the option whether or not this database access should be transactional. No longer is the developer locked into using transactions. Of course, it is generally a good idea to use transactions when accessing data in a distributed environment. All that is required to obtain transactional behaviour for a method containing standard database access is a one-line entry in the `WeaveTransaction` aspect or hypermodule, naming the method containing the data access behaviour. The class that this method resides in must of course contain the `java.sql.Connection` variable necessary for the operation of the transactions concern. The increased encapsulation of the transactions concern makes general extension of the AppTrack system less complex and time consuming, as programming for transactions is no longer a major coding issue. However, the increased complexity of the solution as a whole means that changing/extending this area of the system requires a greater knowledge of the implementation of the concern than before. This is discussed in detail below.

5.2.3 Reusability

The transactions concern is reusable. In fact, it was reused in the Hyper/J implementation without any modification of the core methods (the AspectJ specific code had to be removed). To use the `Transaction` class, you must create a class the implements the abstract members of the class. This means that the class can be reused in any context. In the current AspectJ and Hyper/J implementations the `WeaveTransaction` class provides an extension of the `Transaction` class. However, the `Transaction` class does have dependencies with classes outside of the package in which it resides. The class relies on the existence of a `java.sql.Connection` variable in each class that is named within the concrete implementation of its abstract pointcuts. This does not limit the concern's reusability but it does put some undesirable preconditions on it.

5.2.4 Pluggability

The aspect-oriented implementation of the transactions concern is not cleanly pluggable in the manner of the tracing concern. The solution is designed to work with the database connection used by the classes requiring transactional data access. The `Transaction` aspect is responsible for initialising, manipulating and closing connections to the database. Unplugging the aspect and recompiling the system would result in a loss of database connectivity due to connections not being initialised. However, the amount of effort required to gain data access capabilities following the removal of the aspect-oriented implementation of the transactions concern is minimal when compared to removing the use of transactions in the object-oriented version and using standard, non-transactional data access.

5.2.5 Complexity

Despite the reduction in tangling and the increased separation of the transactions concern, the aspect-oriented version of the AppTrack system is considered more complex than its object-oriented counterpart. This complexity comes about for a number of reasons.

Java reflection is used to manipulate the `java.sql.Connection` variable of the classes requiring transactional data access. Reflection techniques allow for the creation of a generic version of a method that caters for all classes, eradicating the need to list all the classes affected by the concern in the aspect classes. This leads to greater system extensibility and concern reusability. However, reflection code can be quite abstract and complex. The methods of the `Transaction` class require significant study before a comprehensive understanding is achieved.

Each class that contains methods requiring transactional data access must declare a variable of type `java.sql.Connection`. As described previously, this variable is manipulated by the `Transaction` class and used by the client class when making use of database handler classes. For an observer of the code looking at a class containing data access code this can be quite confusing. The system appears to work without the initialisation or committal of the database connection held by the class. The data access portions of the system can only be understood with full knowledge of the relatively complex, reflection-based, transactions concern.

According to [45], due to the nature of the concern, separation of transactionality is syntactic rather than semantic. This, they argue, is because transactions should be implemented with the rest of the application semantics. They state that AOSD and specifically the AspectJ programming language can be used to achieve some level of syntactical separation, but that the developer should be aware of its very *syntactic-only* nature. It appears that this is the case with the re-implementation of the transactions concern in the AppTrack system. This adds to the complexity of the new implementation.

5.2.6 Productivity

The development of the aspect-oriented version of the transactions concern required considerably more developer effort than the object-oriented version. This increase in effort required is a result of both the learning curve of the AOP languages and the heightened complexity of the new solution. The increase in development effort is somewhat offset by the benefits described in the changeability/extensibility section. The addition of transactionality to any new methods involving data access is now a trivial task given sufficient knowledge of the new, more complex solution. This will ensure a saving on development effort on future system maintenance in this area providing that developers have sufficient knowledge of the relevant technologies.

5.2.7 Lines of Code

The total number of LOC cut from the original code base as a result of the reduction in tangling is 33 tangled lines and one whole class, the original `Transaction` class. However the aspect-oriented implementation of the class providing the transactional behaviour is 16 lines longer than the object-oriented version it replaces. This means that a total saving of 17 LOC was achieved with the aspect-oriented implementation of the transactions concern.

5.2.8 Conclusion

This concern differed greatly from the tracing concern. Total separation of the transactions concern was not achieved using AOSD techniques. As a result, the implementation of the concern is partially aspectised, resulting in a concern that is more difficult to understand. However, the reusability and pluggability of the concern are enhanced. Also, with a sufficient understanding of AOSD and the concern implementation, maintenance is more productive. There was a minimal reduction in the size of the codebase following the re-implementation of this concern, vastly differing from the tracing concern.

5.3 Enforce Factory Design Pattern

The following section contains an evaluation of the use of aspect-oriented software development techniques in the re-implementation of the status factory enforcement concern for the AppTrack System. This discussion refers specifically to the AspectJ implementation, as the concern could not be implemented using Hyper/J (section 4.4.2).

5.3.1 Separation Achieved

Figure 33 illustrates the total separation of the enforce factory concern that was achieved using AOSD techniques. The right-most bar represents the class containing the code that enforces the use of the status factory. This one class contains a pointcut that can identify the instantiation of any `Status` object anywhere in the system and the dark coloured bars are representative of a random cross-section of the AppTrack codebase. The concern cannot be implemented using object-oriented techniques (see design section 3.3); hence a diagram illustrating the object-oriented concern scattering is not presented. If it were possible to implement the concern, factory enforcing code would reside within the constructor of the each status class. This would cause a high degree of both scattering and tangling. The aspect-oriented solution is an extremely elegant solution to a complex problem.

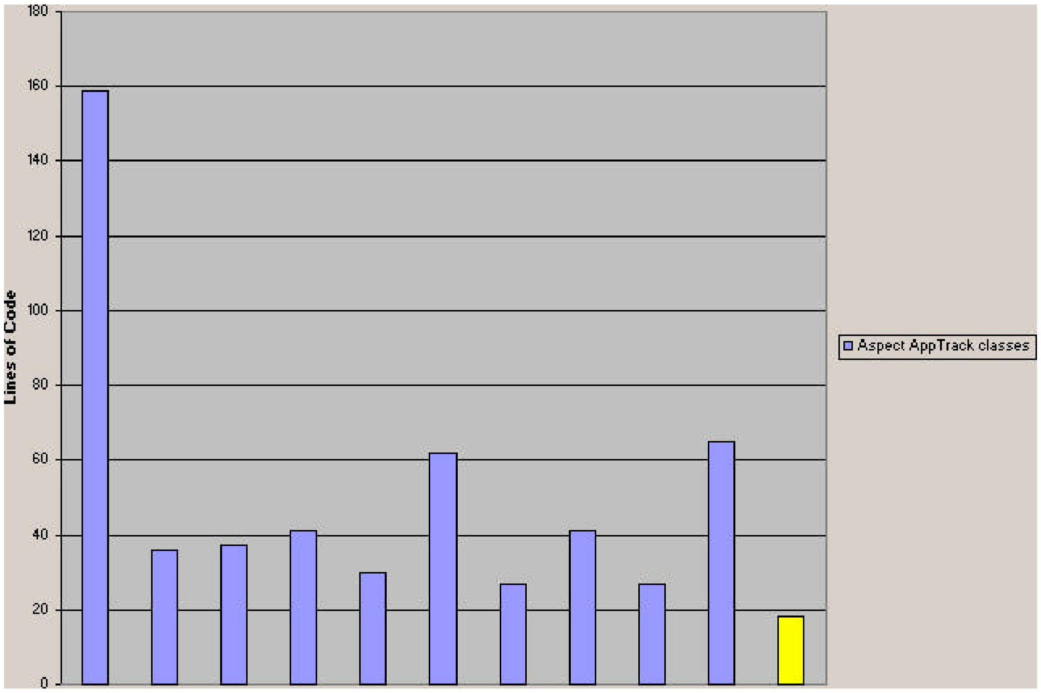

Figure 33: Separation of enforce factory concern

5.3.2 Changeability/Extensibility

The implementation of the factory enforcing concern is highly changeable and extensible. The factory enforcing code is confined to one location, the `EnforceFactory` aspect class. This means that any changes or extensions only need to be made once. The implementation has no dependencies with the rest of the AppTrack codebase and so any future changes will not propagate in a negative manner.

5.3.3 Reusability

The factory enforcing concern is reusable in the same manner as the tracing concern. The only piece of application specific code in the implementation is the pointcut specification. This pointcut lists numerous classes specific to the AppTrack application. To reuse the concern the pointcut must be modified to refer to a set of classes relevant to the system it is being reused in. However, to make a totally reusable version of the concern an abstract version of the aspect could be created. The only difference with this version would be the existence of an abstract version of the pointcut. The developer reusing the concern would then be required to write a class that inherits the abstract concern and provides a concrete implementation for the pointcut.

5.3.4 Pluggability

The factory enforcing concern is completely pluggable. This is because the implementation defines no dependency relationships with any class outside of its own package. Pluggability is an especially important property for this concern. The concern enforces the use of a design pattern that was used during the development of the first version of the AppTrack system. Of course change is inevitable, this is one of the major reasons that we seek to separate concerns in the first place. It would be considered rather poor design to enforce the use of a design architecture that no longer held any value. An aspect-oriented solution does not tie you down in any way. It you want to lift design restrictions, they can be lifted by simply compiling the system without the inclusion of the aspect class that contains the implementation of this concern.

5.3.5 Complexity

The solution to this seemingly complex problem is relatively simplistic in reality. The expressive power of AspectJ means that the core concern can be captured with one simple pointcut. I believe it is not possible to achieve such behaviour with an infinite amount of object-oriented Java code. The pointcut and the associated advice that make up the implementation of this concern can be understood with minimal knowledge of Java, AspectJ and AOSD.

5.3.6 Productivity

Following the design phase, the concern took a matter of minutes to implement using AspectJ. I envisage any modification of the concern in the future being similarly straightforward. This is due to the simplistic implementation achieved with AspectJ. The concern will increase productivity in two ways. Firstly, the aspect-oriented solution saves time that could have been spent attempting to solve the seemingly impossible predicament of implementing this concern using the standard Java. Secondly, because this concern catches misuses of the `Status` objects throughout the entire system, it promotes quality software development practices. This eradicates the possibility of poor code being produced during future extensions of the system in this area. This will cut down on the re-engineering work required to fix bugs.

5.3.7 Lines of Code

The implementation of the `EnforceFactory` aspect class accounts for only 18 LOC. There is no object-oriented comparison for this figure, as the concern cannot be implemented with standard Java. If a solution were possible using Java I estimate each `Status` class constructor would be augmented with approximately 7 LOC. However, regardless of the lack of a firm object-oriented LOC figure to compare the aspect-oriented solution with, it is clear the 18 LOC is quite inexpensive and efficient.

5.3.8 Conclusion

The expressive power of AspectJ provides a fully separated solution to a design that was not possible to implement using standard Java. Similar to the tracing concern, the total separation achieved in the aspect-oriented implementation of this concern affords benefits under each of the evaluation criteria. Thus far, the development concerns, tracing and enforce status factory, have benefited more from the aspect-oriented re-implementation when compared to the transactions concern, a complicated production concern. There was a slight increase in the size of the codebase following the re-implementation of this concern, this differs to both previous concerns, tracing and transactions, but it is of little consequence.

5.4 Database Compatibility

The following section contains an evaluation of the use of aspect-oriented software development techniques in the re-implementation of the database compatibility concern for the AppTrack System. This discussion refers specifically to the AspectJ implementation of the concern, as it could not be implemented with Hyper/J (section 4.5.2).

5.4.1 Separation Achieved

Total separation of the database compatibility concern was achieved. Figure 34 illustrates the scattering and tangling in the original implementation of the concern. This scattering, illustrated by the presence of the light coloured section occupying the bottom 15 lines of each class, arises from the need to include a method to parse database parameters for apostrophes. The tangling occurs because methods that create database query strings need to use the parsing method to parse the parameters they accept when invoked.

Figure 35 demonstrates the separation of the database compatibility concern from the core database handler classes. The scattering and tangling evident in figure 34 have been completely eradicated. The light coloured bar on the right-hand side of the diagram is a new aspect class that was created during the implementation stage. This class now caters for any database compatibility issues that are present in the AppTrack system. Leaving the core codebase free to cater for the standard case.

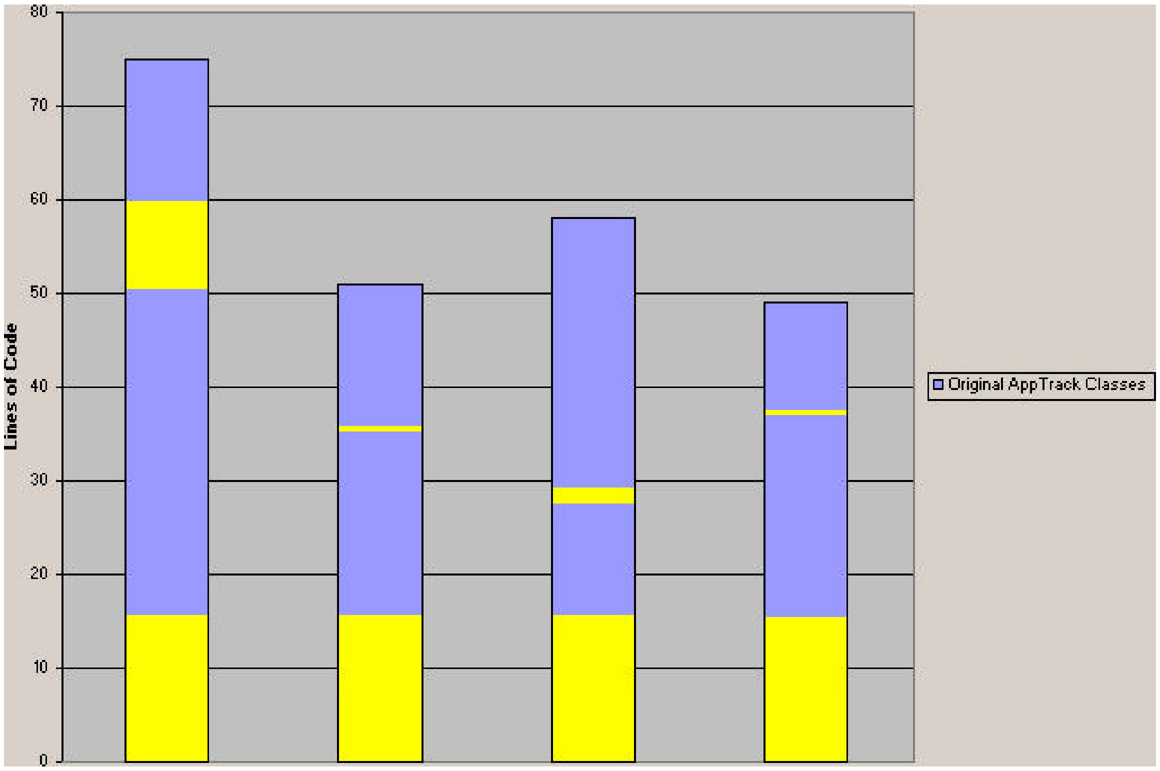

Figure 34: Scattering and tangling in original database compatibility concern

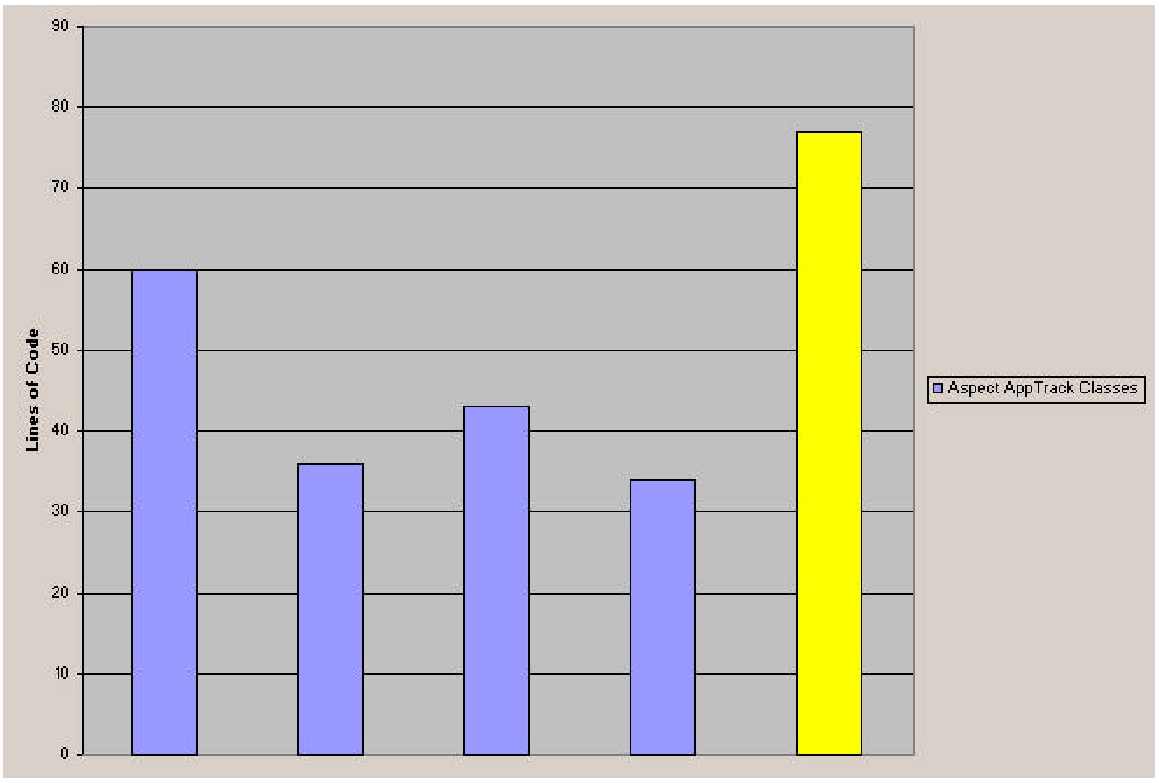

Figure 35: Separation of the database compatibility concern

5.4.2 Changeability/Extensibility

As illustrated by figure 35, the implementation of the database compatibility concern is completely self-contained. The implementation defines no dependency relationships with any classes in the system. This means that any changes to the implementation of the concern can be made without affecting any other system classes. The concern was in fact extended during the implementation stage as it was discovered that several more database handler classes needed to be affected by the concern than was previously envisaged. All that was required for the extension was the addition of a relevant pointcut and corresponding advice to the concern implementation and the removal of the scattered and tangled code from the object-oriented class affected by the concern. No core codebase preparation, such as that required in the transactions concern, is necessary for the implementation of this concern. However, any change in the signature of a method being overridden by a proxy implementation must be reflected in the pointcuts and advice that compose the proxy implementation. This is because the proxy implementation requires access to the runtime parameters of the overridden method. This issue is also pointed out in [48].

5.4.3 Reusability

The database compatibility concern cannot be totally reused as is, nor would there be a desire for such reuse. The nature of the concern dictates that its capacity for reuse is quite low. The concern handles a common problem with the MySQL database in a manner specific to the AppTrack system (i.e. the solution applies the MySQL fix to specific locations in the AppTrack codebase). Therefore, only the section of the implementation that deals with the MySQL limitation would be of interest to developers of future applications, as opposed to the concern as a whole. The implementation could be made reusable by extracting the code related to the MySQL issue and including it in an abstract aspect class. Future implementations of similar concerns could then inherit this functionality and define their own pointcuts and advice to make use of the database parameter parsing method that handles the MySQL apostrophe issue.

5.4.4 Pluggability

The database compatibility concern works as a proxy for the standard implementation. Until an environment in which the use of the standard implementation becomes accessible, the database compatibility concern provides a proxy for the standard implementation to delegate to at runtime. Hence, a major requirement for the implementation of the concern was that it should facilitate a smooth change over to use of the standard implementation. It was therefore imperative that this concern be seamlessly pluggable. The concern does not define any dependency relationships that result in propagation of concern modifications. To replace the database compatibility concern with the standard implementation the system can simply be recompiled without the inclusion of the `FixQuotes` aspect.

5.4.5 Complexity

Conceptually, the AspectJ implementation of this concern is on a similar level to that of the standard object-oriented implementation. However, the implementation is made slightly less comprehensible by its reduced readability. The pointcut definitions for identifying each method that requires delegation to a proxy method are quite complex. The parameters for the method being overridden by the proxy implementation must be made available to the proxy method. AspectJ provides a mechanism for doing this but it can make the declaration of pointcuts and advice related to the method being overridden quite large. A reasonably good understanding of AspectJ is required to understand the re-engineered solution. However, this reduction in comprehensibility is offset by the benefit of increased modularity[3].

5.4.6 Productivity

Given a reasonable understanding of AOSD and AspectJ, this concern can be implemented quite efficiently (it is estimated that the implementation times for both the object-oriented and aspect-oriented versions of the concern were quite similar). A large gain in productivity is experienced in the area of concern maintenance, specifically extension, pluggability and changeability.

The concern can be extended by adding a new pointcut (and corresponding advice) to cater for each method requiring delegation to a proxy. The concern was extended periodically throughout the re-implementation of the AppTrack system as new database handler methods that required delegation were identified. Extending the concern in this manner requires far less work than extending the object-oriented version. Compiling the system without the inclusion of the `FixQuotes` class can seamlessly unplug the concern. Removing the concern from the object-oriented version would require removing code from numerous classes. Changing the parsing techniques in the aspect-oriented version requires affecting only one method as opposed to numerous scattered versions of the same method scattered throughout the object-oriented version. The general reduction in maintenance effort related to this concern increases overall developer productivity.

5.4.7 Lines of Code

Currently four classes are affected by the database compatibility concern. Prior to the separation of the concern, sum LOC count for these four classes was 277. Following the removal of the scattered and tangled code from the affected classes, their combined LOC count is 173. This represents a reduction of 60 LOC or 37.5%. However, the implementation of the new aspect class to cater for the concern adds 77 lines of code to the system. This sees an increase of 13 LOC to the total number of LOC in system that relate to this concern.

[3] Assuming that modularity is a priority of the developer

5.4.8 Conclusion

This was the first concern to use the delegation mechanism offered by AspectJ to create a proxy implementation for an existing object-oriented module. This facility allowed for an implementation that, similar to the tracing and enforce factory concerns previously, achieved total separation. Whilst this affords many benefits it must be noted that a change in the affected method will likely result in a change in the aspect as the pointcuts are closely tied to affected method signatures.

It is interesting to note that although this is a production concern, it did not suffer from the problems experienced during the re-implementation of the transactions concern. There was a slight increase in the size of the codebase following the re-implementation of this concern, similar to the enforce factory concern. The many benefits introduced by the aspect-oriented implementation of the concern more than adequately compensate for the addition of a rather negligible number of LOC.

5.5 Design By Contract (Preconditions)

The following section contains an evaluation of the use of aspect-oriented software development techniques in the re-implementation of the design by contract (preconditions) concern for the AppTrack System. This discussion refers specifically to the AspectJ implementation, as the concern could not be implemented with Hyper/J (section 4.6.2).

5.5.1 Separation Achieved

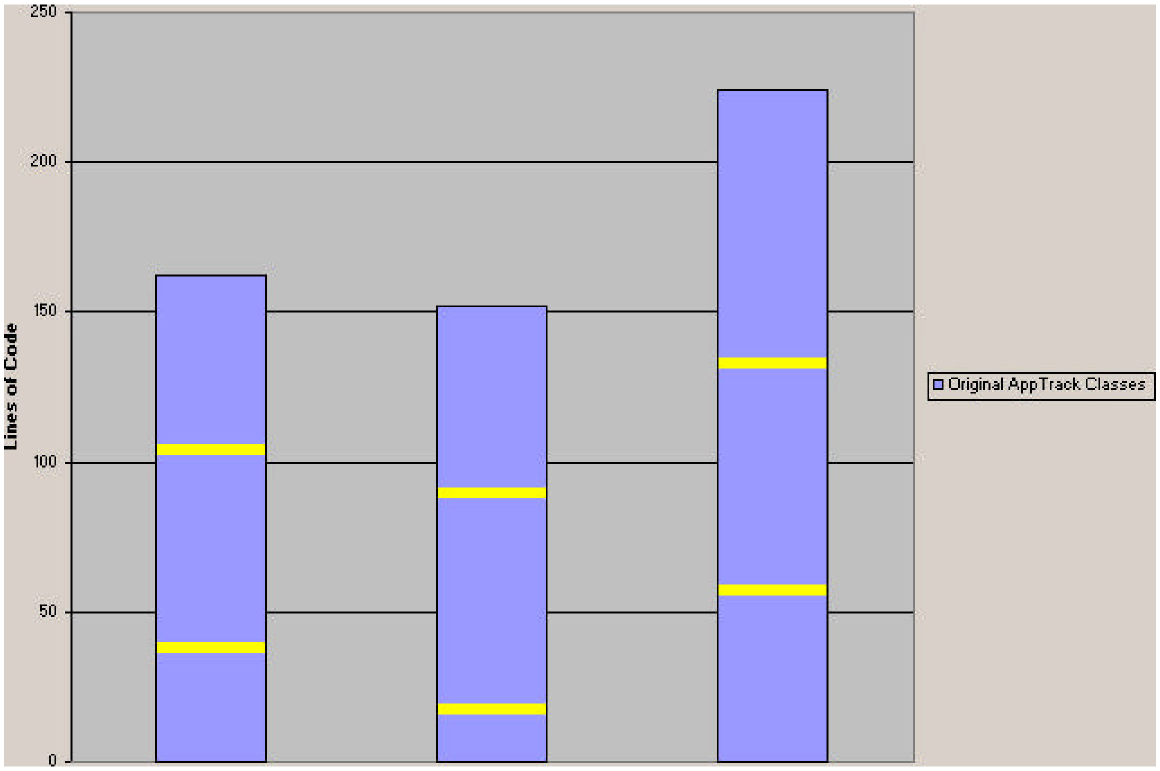

Figure 36: Tangling in original design by contract (preconditions) concern

Figure 36 illustrates the tangling produced by the implementation of the design by contract (preconditions) concern in the original AppTrack system. There are currently three classes affected by the concern, each with two methods that contain code to check the runtime parameters passed by their clients. The light coloured sections of each bar represent the tangled code in each affected method. The precondition code is tangled with the code that implements the core behaviour of each method.

Figure 37 illustrates that total separation of the concern has been achieved. No longer are the methods requiring preconditions subject to the tangling illustrated above. All precondition code has been untangled and located in a class created during the re-implementation process. The right-most bar on the diagram represents this new concern encapsulating class.

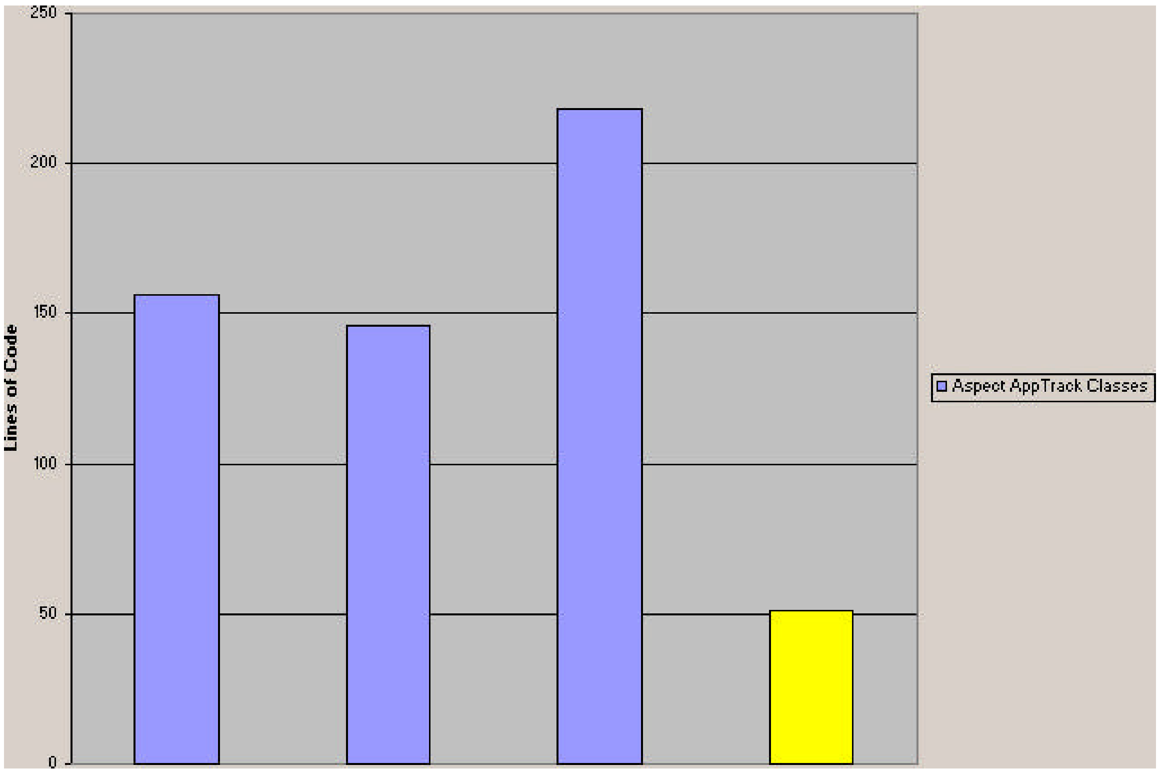

Figure 37: Separation of the design by contract (preconditions) concern

5.5.2 Changeability/Extensibility

The aspect-oriented implementation of the design by contract (preconditions) concern is more changeable and extensible than the previous implementation. Precondition handling has been encapsulated separately to the code it affects and changing the implementation of the preconditions requires changes to code within this encapsulation only. However, changes to the signature (specifically the parameters) of the contracted methods must be reflected in the implementation of the preconditions. This is because preconditions must validate the runtime arguments of the contracted methods.

The handling of precondition failure, which was scattered throughout the tangled precondition implementations, has been modularised within the aspect-oriented version of the concern. One method has been defined to handle precondition failures. Therefore a general policy exists for dealing with precondition failure. Changes can be made to the way all precondition failures are managed by appropriately modifying this one method.

This concern is extensible in a manner similar to that of the database compatibility concern. Addition of a new precondition requires the implementation of a new pointcut and corresponding advice for the precondition in the class containing the existing preconditions. The method that deals with precondition failure can be reused in the implementation of the advice. The contracted method is not made aware of the existence of the precondition.

5.5.3 Reusability

As mentioned previously, the method that caters for precondition failure can be reused by future extensions to the preconditions concern. However, the concern as a whole is not reusable, nor can it be expected to be. The implementation of the concern is made up of a series of pointcuts and `around` advice on these pointcuts. These pointcuts share little or no commonality (the `*` property was used to capture any commonality that existed between methods affected by a single pointcut). The pointcuts are specific to the AppTrack application and hence would not fit cleanly into the implementation of another application. Commonality in the advice of each pointcut has been extracted and encapsulated in the method that handles pointcut failure. In spite of the lack of concrete reusability of this concern, the general model employed provides a good basis for future implementations of design by contract concerns. This suggests that the solution may the basis of an AOP design pattern for method preconditions. This is discussed in the future work section.

5.5.4 Pluggability

The implementation of the design by contract (preconditions) concern defines no dependency relationships with any classes in the AppTrack system. This means that it can be seamlessly plugged and unplugged. All that is required to unplug the concern is recompilation of the system without the inclusion of the preconditions aspect. This is quite important because the preconditions used in the AppTrack system are a development concern rather than a production concern. Before an application is deployed, development concerns are removed. As we know this is trivial with an aspect-oriented implementation, however with an object-oriented implementation of the concern any method that contains a precondition must be modified. The increased pluggability eliminates the disadvantages of implementing preconditions, giving developers fewer excuses not to employ quality software engineering techniques.

5.5.5 Complexity

Given a reasonable level of familiarity with AOSD and AspectJ, the re-implementation of the design by contract (preconditions) concern is no more complex than the original implementation. Pointcuts are defined to identify the methods that must be affected by preconditions and around advice is used to provide the implementation of the precondition. The readability and comprehensibility of the code is similar to that of the original implementation. The tangling in the object-oriented implementation adversely affected the readability of the code. The precondition declarations in the aspect-oriented implementation are quite complex, as they require access to the parameters of the contracted methods. This affects the readability of the aspect-oriented implementation. The implementation of the preconditions themselves is similar in both solutions hence there is no increase in complexity. The main reduction in complexity comes in the areas of changeability/extensibility and pluggability as described earlier.

5.5.6 Productivity

Whilst the aspect-oriented solution required significantly more time to design and implement than the object-oriented original, there is a definite increase in productivity related to the area of future maintenance. As described earlier, changing and extending the concern is considerably less time consuming and complex when compared to the original implementation. Any modifications are restricted to the encapsulated concern, and the core codebase is not affected. Complete removal of the concern requires much less effort than the removal of the object-oriented concern. The concern can be removed by compiling the system without the inclusion of the package containing the aspect-oriented implementation. Removal of the original implementation required visiting as many classes as were affected by the concern and manually editing the methods that were contracted by the preconditions. The increased initial development effort is compensated for by the increased future productivity[4].

5.5.7 Lines of Code

The aspect-oriented implementation of the design by contract (preconditions) concern facilitated the removal of 3 LOC from each method that contained tangled object-oriented precondition code. This resulted in the reduction of the AppTrack codebase by 18 LOC. The aspect-oriented implementation of the concern requires 56 LOC. This represents an increase of 250% on the original implementation.

5.5.8 Conclusion

Like the database compatibility concern, the preconditions concern used AspectJ’s delegation mechanism for its implementation. This resulted in the total separation of the concern. However, this concern also suffers from the same tight coupling to the affected methods as the database compatibility concern. An interesting issue arose in this evaluation that has not be seen previously. It was noted that the general

[4] Assuming future work is conducted on the aspect-oriented implementation of the AppTrack system

architecture used in the preconditions concern implementation would be applicable to future implementation of similar concerns. This suggests that it may form the basis of an AOP design pattern for preconditions.

There was a significant increase in the size of the codebase following the re-implementation of this concern. However, this is of little consequence when considered in the context of the benefits afforded by the re-implementation.

5.6 Exception Handling

The following section contains an evaluation of the use of aspect-oriented software development techniques in the re-implementation of the exception handling concern for the AppTrack System. This discussion refers specifically to the AspectJ implementation, as the concern could not be implemented with Hyper/J (section 4.7.2).

5.6.1 Separation Achieved

Figure 38 illustrates both the scattering and tangling evident in the original object-oriented implementation of the exception handling concern. The light coloured sections represent exception handling code.

A class should deal with the ordinary case and not be concerned about the exceptional. The Java language is constructed in such a way that a class must cater for both. This leads to the tangling of two distinct concepts in the implementation code. The scattering arises because in the object-oriented version of the system, four types of exception class are used throughout the `Action` classes. Each time an exception of a certain type is declared as thrown, it is handled in the same manner. This leads to duplication of exception handling code.

Figure 39 shows the separation of the exception handling concern that was achieved using aspect-oriented techniques. The addition of five new classes during the re-implementation process eradicated the scattering completely and minimised the tangling as much as was possible. The tangling could not be removed completely because of the Struts framework specification. Basically, Struts defines a custom collection object for storing runtime exceptions. This object must be manipulated by exception handling code and the object must reside in the `Action` class implementing the system requirement that has thrown the exception. This is fully discussed in the implementation section (4.7.1).

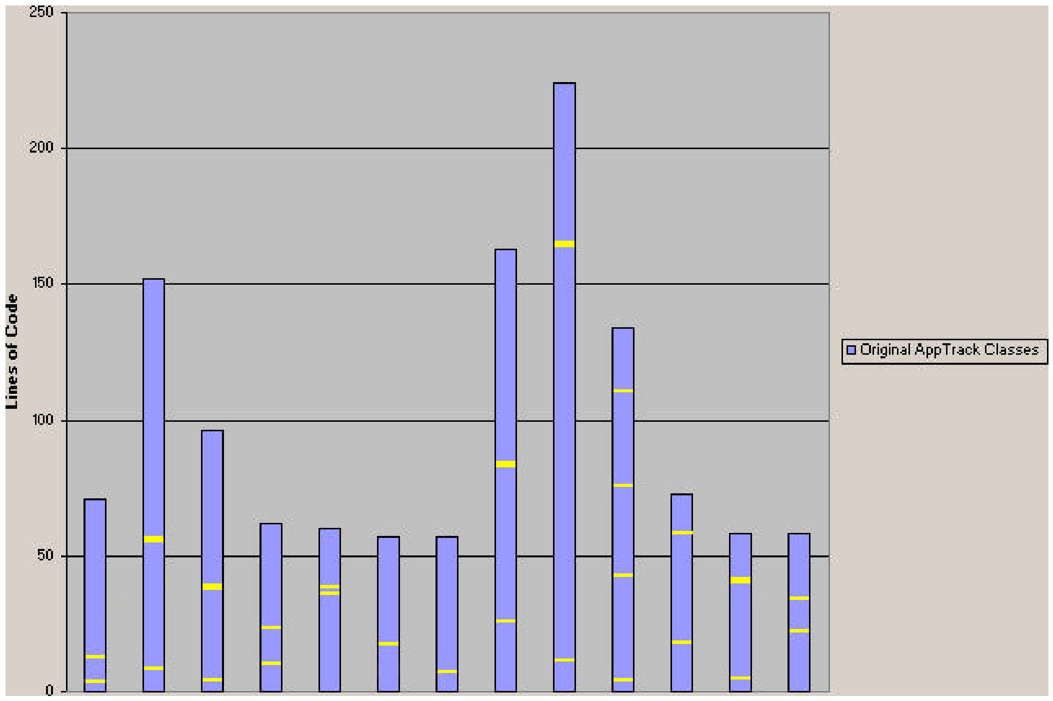

Figure 38: Scattering and Tangling in original exception handling concern

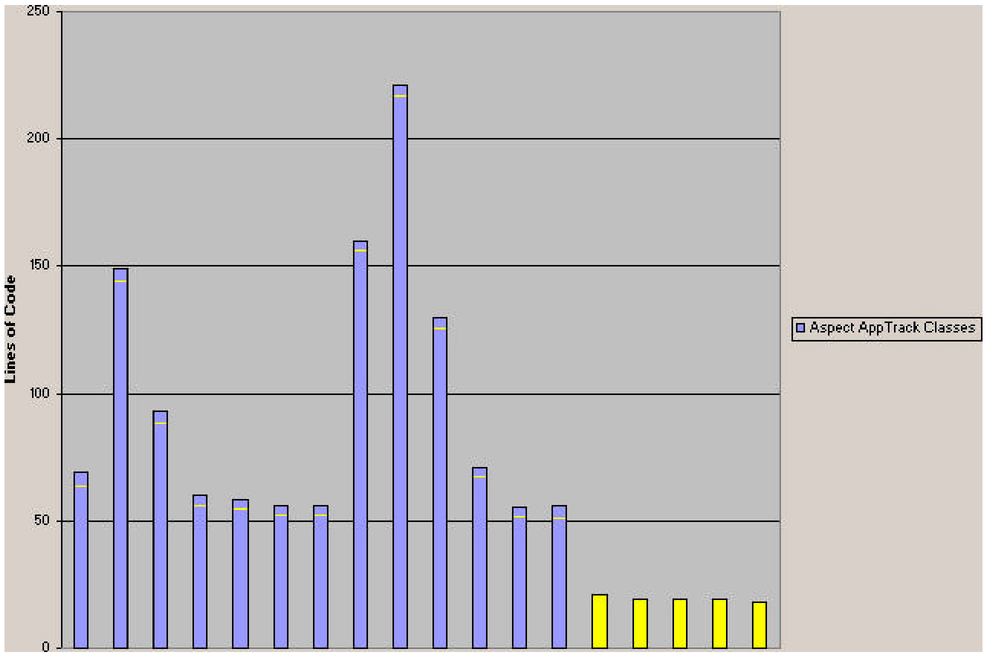

Figure 39: Separation of the exception handling concern

5.6.2 Changeability/Extensibility

The exception handling concern is both highly changeable and extensible. Changing the manner in which an exception type is handled across the `action` package is a matter of altering only one location in the code (the `handleException` method). This property leads to the appliance of consistent behaviour for each type of exception thrown. It is impossible to confidently enforce consistent behaviour, such as that in the aspect-oriented implementation of the concern, with a scattered object-oriented implementation. Changing the exception handling policy for a certain type of exception in the object-oriented version required visiting the code body at each location an exception of the type in question is declared as thrown. This increases the chances of a location being erroneously omitted or misused.

Extensibility is enhanced in the aspect-oriented implementation of the concern. There are two distinct areas that see improvements in this area. The first of these is the extension of the system as a whole. Adhering to the Struts framework means that any new higher-level requirements will result in the creation of an `Action` class. It is highly likely that any new `Action` classes will throw exceptions of a type currently catered for by the aspect-oriented version of the concern. If this is the case, developers only have to declare the exceptions; their handling will be automatically dealt with following a recompilation (re-weave) of the system.

The extensibility of the concern itself is also improved following the new implementation. As described in the implementation section, each exception that is thrown must have a corresponding aspect class. These aspect classes extend a predefined abstract aspect class that acts partly as a template and partly to provide implementation for concrete aspect classes. This means that all exception aspects are created in a uniform manner with some implementation automatically written, making extension of this area of the system rather trivial.

5.6.3 Reusability

Within the AppTrack application, the aspect-oriented implementation of the exception handling concern is quite reusable. Each class that wishes to provide behaviour for a specific type of exception extends the `ExceptionHandler` class. This class captures commonality amongst all exception handling aspects, enforcing consistent behaviour and saving on development time. The reuse of this class makes the handling of new exception types for the `action` package quite a trivial issue.

The concern as a whole is quite specific to both the AppTrack application and applications that adhere to the Struts framework. This limits its capabilities for widespread reuse. The Struts framework requires that exception handling code be related to variables defined in the `Action` classes. This creates inter-concern relationships that hinder reusability. Therefore, this concern cannot be plugged directly into an unrelated, non-Struts application. However, the model that the solution adheres to is definitely worthy of reuse as it provides a good solution to the exception handling requirement and achieves the maximum separation possible in a Struts environment. This suggests that the solution may form the basis of an AOP design pattern for exception handling. This is discussed in the future work section.

5.6.4 Pluggability

The exception handling concern implements a production requirement. Therefore, unlike the tracing concern, it should be included in release builds of the software. As with the transactions concern, unplugging the exception handling concern will remove vital system functionality. Therefore the concern should not be unplugged unless it is being replaced by a suitable implementation.

Leaving this issue aside, unplugging the concern from the application is trivial when compared to the task of unplugging the object-oriented implementation. As with previous concerns, unplugging the exception handling concern from the AppTrack system is simply a matter of compiling the application codebase without the inclusion of the `exceptionHandler` package. This ease of pluggability exists because scattering has been removed completely. Removal of the object-oriented implementation currently involves visiting twenty-nine separate `try/catch` blocks spread across thirteen different `Action` classes and deleting the exception handling code from each one. Unplugging the exception handling concern would be convenient if you wished to plug in another implementation of the same concern.

5.6.5 Complexity

This concern has led to various changes in the complexity of the system, both positive and negative. The positive changes come in the areas of reusability, pluggability, changeability, extensibility and readability. The first four properties have been discussed in the preceding subsections. Readability is enhanced because exception handling logic has been removed from classes whose primary intent is the implementation of the standard case. Future students of the code will be better able to understand its behaviour as they can focus on its primary intent, rather than having to deal with the exceptional case (in addition to the standard case).

The aspect-oriented implementation of the exception handling concern does add some complexity to the system. The Struts framework requires that its custom collection variable be used to store exceptions that are generated in `Action` classes at runtime. These collection variables must therefore have class scope so that the aspect classes can access them. This requires a slight deviation from the general Struts model, altering the hierarchy of the classes in the `action` package so that each action class inherits the appropriate variable with class scope. Such class hierarchy manipulation is not necessary with the object-oriented implementation. This adversely affects the system as a whole as Java only provides single inheritance and extension of `Action` classes is now limited in relation to future inheritance.

It could be argued that the readability of the code is actually reduced in the aspect-oriented solution. Java has a well-defined model for dealing with the exceptional case and all experienced Java developers understand the model. The removal of exception handling code from the `try/catch` blocks of classes implementing core behaviour may lead developers without significant knowledge of AOSD to believe that no exception handling behaviour exists. This can increase the time required by new developers to become familiar with the system. However, this situation can easily be avoided by sufficient training of developers.

The increases in reusability, pluggability and changeability/extensibility all aid complexity reduction by simplifying concern maintenance tasks. However, the changes to the `Action` class hierarchies lead to a significant reduction in comprehensibility.

5.6.6 Productivity

Exception handling with Java and the Struts framework is quite straightforward as it follows the standard Java exception handling pattern. There was certainly more development effort required during the initial development of the aspect-oriented implementation of the concern compared to the original object-oriented version.

Productivity in the area of maintenance is much improved with the aspect-oriented implementation. Changing how a particular type of exception is handled requires changing one method only, as opposed to many scattered `try/catch` blocks. The addition of a new exception type and its corresponding behaviour has become a trivial issue. This is due to the reusability of the class that all exception handling aspects must extend, and also to the clarity of the aspect-oriented exception handling model. The model used is specifically designed to promote extension of the exception handling concern whilst maintaining the maximum separation possible at all times. This not only leads to rapid development of new exception handling behaviour but also promotes quality software development at the same time. It is not possible to achieve such a benefit with object-oriented techniques.

5.6.7 Lines of Code

The aspect-oriented implementation of the exception handling concern caused the removal of a total of 37 LOC, taken from 13 classes in the `action` package. Five new aspect classes were added during the implementation of the concern. These classes account for a total of 96 LOC. This denotes a 260% increase in the amount of code required to handle exceptions in the AppTrack system. However, because five new classes are added to the system most of the additional code is basic class set-up code.

5.6.8 Conclusion

Like the transactions concern, the other major production concern in the system, the exception handling concern was not fully separated. This would suggest that the capabilities of AspectJ are more suited to development concerns such as tracing and basic, delegation-based, production concerns. However, the exception handling concern is not as complex as the transactions concern and it benefits more from the separation in terms of extensibility and changeability. Similar to the preconditions concern, it was noted that the general architecture used in the exception handling concern implementation would be applicable to future implementations of similar concerns. There was a significant increase in the size of the codebase following the re-implementation of this concern, similar to the preconditions concern. However, this increase in code size is offset by the benefits afforded by the solution.

5.7 Recording Bean Properties Modification

The following section contains an evaluation of the use of aspect-oriented software development techniques in the re-implementation of the recording bean properties modification concern for the AppTrack System. This discussion refers specifically to the AspectJ implementation, as the concern could not be implemented with Hyper/J (section 4.8.2).

5.7.1 Separation Achieved

Figure 40 illustrates the scattering and tangling evident in the object-oriented implementation of the `Applicant` class. The presence of these phenomena is a direct result of the object-oriented implementation of the record bean properties modification concern. The thick, light coloured lines towards the bottom of the class represent the methods created in the `Applicant` class during the original implementation of concern. The uppermost thin line represents the declaration of the `java.util.Date` variable that is manipulated by the aforementioned methods. The remainder of the thinner lines represent scattered calls to these methods.

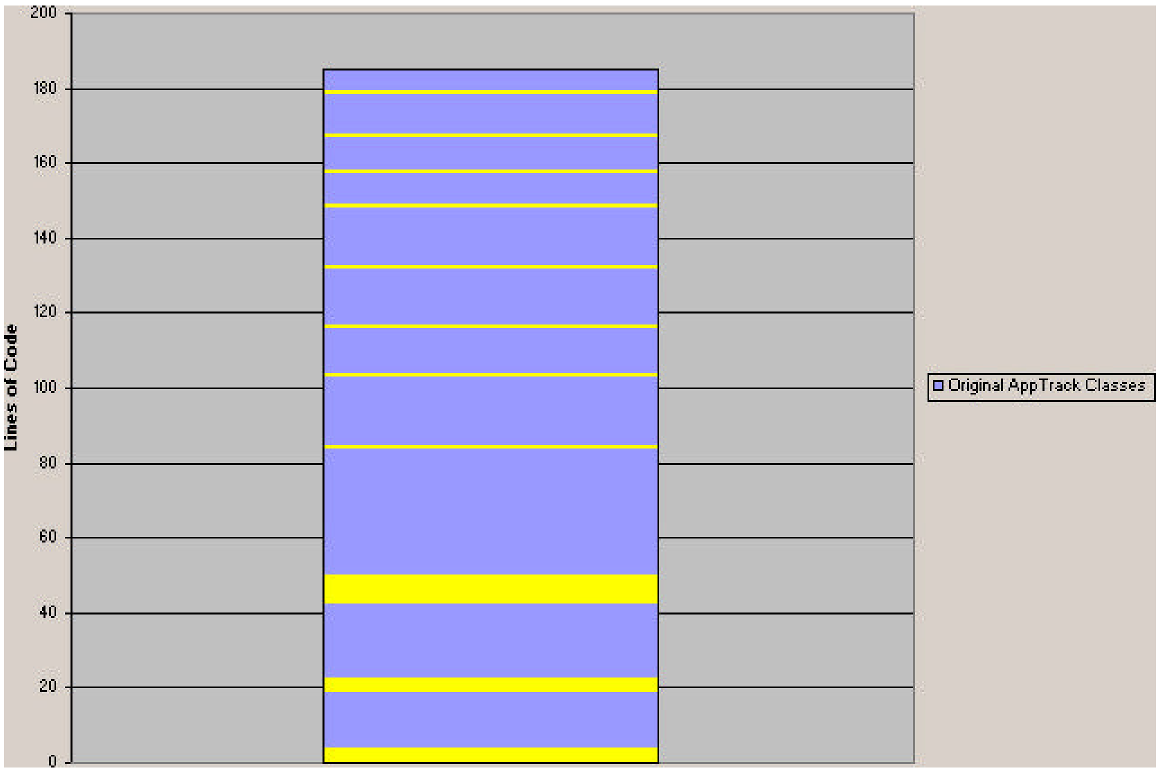

Figure 40: Scattering and Tangling in original record bean properties modification concern

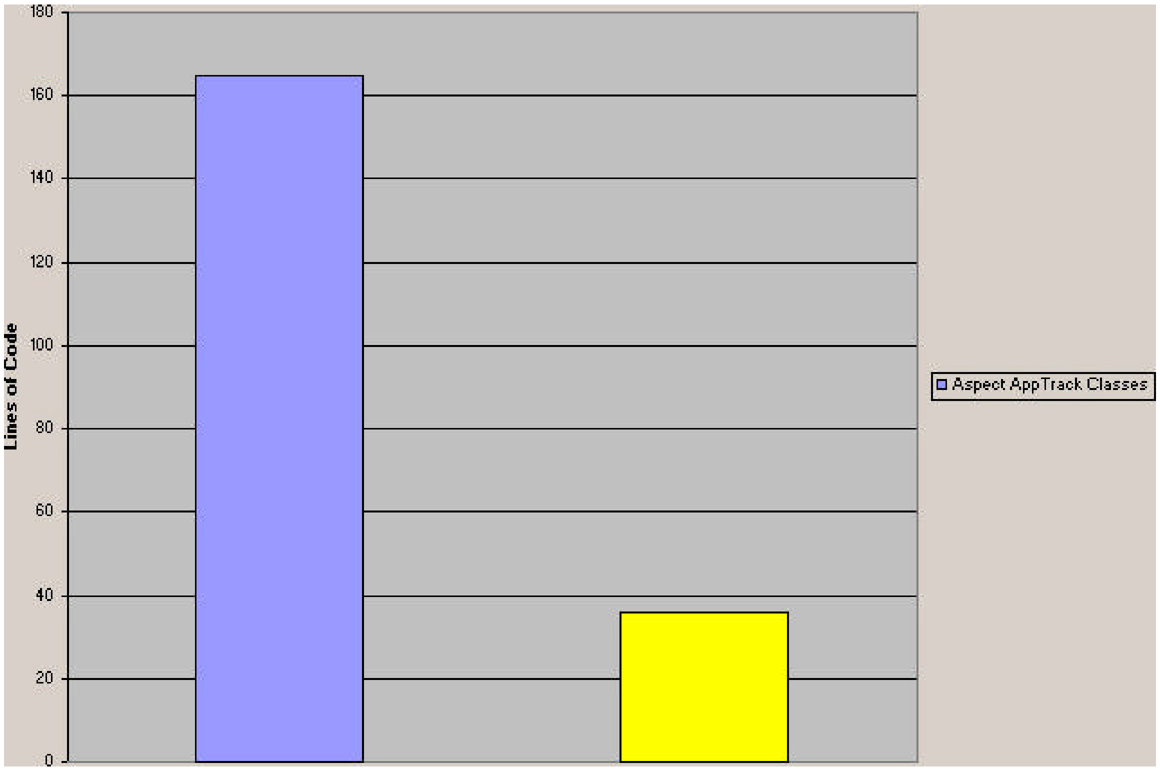

Figure 41: Separation of the record bean properties modification concern

Figure 41 shows that total separation of the concern was achieved. The aspect-oriented implementation removes any code relating to the concern from the `Applicant` class and encapsulates it, along with newly defined functionality, in the `applicantManager` aspect class (right-hand bar). Although the aspect-oriented implementation of the concern re-introduces concern-related code back into `Applicant` class, at compile time the code is totally separate.

5.7.2 Changeability/Extensibility

The changeability and extensibility of this concern are greatly enhanced following the aspect-oriented re-implementation. Due to the encapsulation of the scattered code (via the implementation of a joinpoint and corresponding advice), changing the number of methods affected by the concern is a matter of simply modifying the pointcut that specifies this relationship. No longer does each accessor method need to be augmented with a call to the appropriate method(s). Extending the coverage of the concern is also managed by this pointcut.

Changing the behaviour that is executed each time a property is modified is as it was before, only the code is located in another class. The main drawback of the object-oriented implementation of this behaviour was its location i.e. it was tangled within the `Applicant` class. This has been now resolved.

5.7.3 Reusability

The implementation of this concern was written with the `Applicant` class in mind. In AspectJ you must introduce variables and methods into a specific class, there is no abstract facility for the `introduction` construct. This makes the implementation AppTrack-specific and so it cannot be cleanly reused in another environment. The implementation introduces variables and methods into the `Applicant` class and the joinpoint and advice that use these introduced constructs both repeatedly refer to the `Applicant` class. However, references to the `Applicant` class are the only barrier to reuse, and the basis of the concern can be reused in implementation of similar concerns involving classes other than `Applicant`. The current aspect-oriented implementation of the concern defines a good model for future implementations of closely related concerns. This suggests that the solution may the basis of an AOP design pattern for monitoring and recording property modification in Java beans. This is discussed in the future work section

5.7.4 Pluggability

Unplugging the aspect-oriented implementation of the concern from the AppTrack application is quite easy when compared to attempting the same task in the object-oriented version of the application. Removal of the object-oriented version requires deleting multiple code segments from the `Applicant` class. Unplugging the aspect-oriented version only requires a recompilation of the system without the inclusion of the concern code. However, because the concern is a production concern its removal will result in the loss of vital application functionality. Therefore, the concern should not be unplugged unless a new implementation is being plugged in to replace it. In this situation the advantages over the object-oriented version are that the removal of the concern requires only a recompilation of the codebase as opposed to the manual removal of scattered and tangled code.

5.7.5 Complexity

The aspect-oriented implementation affects the readability of the `Applicant` class in two ways. By removing the scattered code it makes the class easier to comprehend. However, because a runtime property of the class is not actually coded within the class (but within the concern implementation code), a good understanding of AOSD and AspectJ is required to understand the `Applicant` class.

The implementation of the concern is no more complex than that of the original object-oriented version; it has simply changed location to eliminate the tangling. Because of the removal of the scattering and tangling, the complexity of future concern modification has been reduced.

5.7.6 Productivity

Whilst the re-implementation process required significantly more developer effort than the initial solution, maintenance of the concern is a more productive process following the aspect-oriented implementation. The removal of the scattering means that modifying the coverage of the concern requires changing only one construct, rather than visiting each accessor method that should (or does) make a call to the method implementing the functionality to record the time the property was accessed at. The significantly reduces developer effort.

5.7.7 Lines of Code

The aspect-oriented implementation of the concern allows for the removal of 20 LOC from the `Applicant` class. The new aspect class added to cater for the concern accounts for 36 LOC. This represents an increase of 14 LOC or 80% on the object-oriented implementation.

5.7.8 Conclusion

This was the first and only concern that saw the separation of a concern that was encapsulated within one class only. The introduction facility in AspectJ made this possible. Like all but two of the concerns (transactions and exception handling) total separation was achieved. This is interesting as this represents the total separation of another, less complex, production concern. Similar to the preconditions and exception handling concerns, it was noted that the general architecture used in the exception handling concern implementation would be applicable to future implementation of similar concerns. There was a slight increase in the size of the codebase following the re-implementation of this concern, similar to the database compatibility and enforce factory concerns.

5.8 Evaluation of Composition Patterns

The process of identifying and designing the concerns that were dealt with in this project was arguably more important than the actual re-implementation of the concerns. More and more frequently design is being recognised as the key activity in the development of object-oriented software. This belief in the power of significant design upfront has carried over to the aspect-oriented world. However, due to the relatively infancy of the field, none of the design languages presented to the AOSD community thus far have emerged as the de facto standard (like UML has in the object-oriented world) for graphically representing aspect-oriented software architectures. For this reason it is constructive to briefly evaluate the chosen design language for this project and to suggest some areas for improvement.
Key information about each concern must be captured at the design level. These core properties were identified as:

- *Separation* of the concern from the remainder of the AppTrack codebase
- *Interaction* of the concern with the AppTrack codebase
- *Integration* of the concern with the appropriate modules in the AppTrack codebase

The composition patterns specification provides a distinct model to cater for each of concern-to-core codebase relationships identified above. These models are used in the design chapter of this dissertation to describe the aspect-oriented design of the existing concerns identified in the AppTrack system. The models, as described in the State of the Art section on composition patterns (section 2.6), effectively describe the modularisation achieved by as aspect-oriented solution and how the implementation modules composing the solution interact in a working system to provide the appropriate behaviour. The three models (template, crosscutting behaviour, and binding) used to graphically represent the design of the aspect-oriented version of the AppTrack system fit quite satisfactorily with the logical model of aspect-oriented separation of concerns, and from studying the models produced during the design phase an excellent understanding of the concerns can be acquired. The composition patterns model was found be adequate for the design of both the development and production aspects implemented during this project. Despite the capabilities of the model there are a number of areas that may require future consideration by the designers.

Not all concerns will apply to all methods in a certain class or package i.e. they do not have global scope over the application. When defining the binding the relationship for such concerns the star (*) property cannot be used to capture commonality (as none exists). In such cases, the fully defined binding relationship can grow quite large and as it does so, its readability diminishes rapidly. This is not so much a failing of the composition patterns model but rather a result of its current lack of tool support. An innovative tool to support the creation of composition patterns could alleviate some of the readability issues associated with complex bind relationships.

The composition patterns model caters for the general case. The model is not intended to be implementation language/tool specific and does not provide mappings for any of the new constructs introduced by AOP languages/tools such as AspectJ, Hyper/J and Composition Filters. This non-recognition of specifics produces a modelling language that is suitable for the design of systems implemented with any AOP language. However, because there are new concepts introduced by most of the leading AOP languages, designing concerns with the composition patterns model leads to a general loss of information. In AspectJ, aspect classes are seen as distinct logical entities from standard Java classes. In Hyper/J, hypermodules, concern mappings and hyperspaces are all crucial artefacts in the implementation of a software system. No provision is made for these key implementation modules in the composition patterns model, with standard object-oriented style classes being the main unit of decomposition supported (although some work has been done on mapping the compositions patterns model to AspectJ [48, 49] and Hyper/J [49]). If a system is designed using composition patterns and implemented using AspectJ, then it is not immediately obvious to the reader of the models that there are aspect classes present in the design. The hypermodules, concern mappings and hyperspaces are omitted completely from any design models intended for a Hyper/J implementation. Whilst inclusion of such concepts would reduce the generality of the composition patterns model, hence reducing the value of the language immensely, I consider the loss of information to be something of a disadvantage.

5.9 Summary of Evaluation

This section presents a summary of the evaluation contained in the preceding sections of this chapter. The findings under each of the evaluation headings for all concerns are amalgamated and conclusions are drawn from results.

5.9.1 Separation Achieved

Seven concerns were designed and implemented using aspect-oriented techniques. Of these, five were completely separated from the remainder of the AppTrack codebase. Although five concerns were separated completely, only two had the property of obliviousness [50] ("whether the writer of the main code has to be aware that aspects will be applied to it"), which allows developers to write the core system code without knowing aspects will be applied to it. These two concerns were the development concerns - tracing and enforce factory.

Two of the concerns could not be fully separated from the core AppTrack codebase. The two concerns in question are the most important production concerns. Both the transactions concern and the exception handling concern could not be entirely separated due to their level of intimacy with the core codebase that they affect. Both concerns require access to data in the classes they are weaved onto, meaning the concerns cannot be totally separated. Although the record bean properties modification concern and the database compatibility concern were separated satisfactorily, most success in the area of separation was achieved when re-implementing the development aspects – the tracing, enforce factory design pattern and design by contract (preconditions) concerns.

5.9.2 Changeability/Extensibility

None of the concerns are affected aversely in this area following the aspect-oriented re-implementation. There are two main areas that are much improved. Firstly, as a result of the eradication of scattered code, it is easier[5] to alter the behaviour of a concern. This requires changing only one method/construct as opposed to changing code scattered across many locations. This property applies to most of the concerns, but mostly notably to the tracing, exception handling, database compatibility and design by contract (preconditions) concerns.

The second area for improvement is in extending the coverage of the concern, that is, the number of methods/classes affected by the concerns behaviour. All concerns can be easily modified to affect more methods or classes. This is usually achieved by editing a pointcut declaration. Some concerns will automatically affect newly created methods and classes as appropriate. It is not possible to achieve such effects with object-oriented software development.

[5] This assumes a sufficient knowledge of AOSD and the relevant AOP languages/tools

5.9.3 Reusability

Whilst some of the aspect-oriented implementations yielded concerns with good reuse properties, none could be deployed in another context without modification. However, the modifications required to make concerns such as tracing and enforce status factory reusable are minimal. Other concerns such as transactions and exception handling were not totally separated and hence cannot be reused as standalone concerns. The reuse of these concerns places additional conditions on the application reusing them. The remainder of the concerns were more application specific than those mention previously in this subsection and hence they are less reusable. In spite of the lack of reusability of some concern implementations, many of the concerns adhere to models that can themselves be quite reusable. Concerns such as the design by contract (preconditions) and exception handling define models that can be applied to future applications to achieve a beneficial degree of separation of concerns.

5.9.4 Pluggability

The pluggability of all concerns was greatly enhanced following their aspect-oriented re-implementation. The majority of the concerns can be removed from the system by simply by excluding them from a fresh recompilation of the AppTrack codebase. This represents a major difference in behaviour over the object-oriented version where removing a concern always involved visiting and editing many implementation modules.

5.9.5 Complexity

The changes in complexity resulting from the aspect-oriented re-implementation could generally not be considered too extreme, but they were significant. A greater reduction in complexity was achieved from development of concerns such as tracing and enforce factory - development concerns with global scope that affect the whole codebase. Complexity was heightened by the re-implementation of production concerns such as transactions and exception handling. These concerns are intimate with the classes they affect and have local scope, affecting only a subset of system modules only. The complexity levels of the remaining concerns remained similar to those possessed before re-implementation. Generally, the increases in complexity arose mainly from loss of readability and comprehensibility stemming from the use of reflection, the failure to fully separate concerns or the preparation of the core codebase to facilitate the re-implementation of a concern.

5.9.6 Productivity

The majority of concerns required more developer effort during the re-implementation phase than was necessary for the initial implementation. This was primarily due to the learning curve associated with both AOSD in general and specifically the AOP languages employed. Two exceptions to the general case however are development concerns - tracing and enforce factory. The development of these concerns was more productive because no longer were they being simultaneously implemented along with multiple other concerns. This practice shifts the focus of the developer from the main concern to the peripheral, lowering productivity [37].

Significant productivity increases in the area of future maintenance of all concerns are clearly evident. Maintenance of concerns involves varying tasks such as changing general behaviour, extending coverage, reusing completely or partially, and unplugging and re-plugging. These tasks are made far more efficient by aspect-oriented software development techniques, generating considerable timesavings over the execution of the same task on an object-oriented implementation.

5.9.7 Lines of Code

Leaving aside the huge reduction in LOC brought about by the aspect-oriented implementation of the tracing concern, there was generally an increase in the number of lines of code needed to implement an aspect-oriented version of a concern. I believe that this can largely be attributed to the fact that scattered and tangled code is removed from existing classes and encapsulated in newly defined aspect classes. This requires the creation of a lot of new classes, hence a lot of standard class setup code i.e. import statements, class declarations etc. Also, because scattered code is generally encapsulated into one reusable unit, a new method definition is required to enclose the behaviour. The increase in code implementing actual behaviour is minimal.

5.10 Table-based Evaluation Summary

The following table summarises the evaluation. Each concern implementation (in both AspectJ and Hyper/J where applicable) is rated on a scale of 1 to 5, with a value of 3 meaning the concern performs relatively similarly to the object-oriented implementation under a particular criterion. A value of 1 indicates that a concern performs positively in a particular area, with a value of 5 indicating the underperformance of a re-implemented concern. Please note that the concerns are listed in abbreviated form due to space limitations.

	Separation Achieved	**Changeability/ Extensibility**	**Reusability**	**Pluggability**	**Complexity**	**Productivity**	**Lines of Code**
Tracing (AJ)	1	1	2	1	1	1	1
Tracing (H/J)	1	1	2	1	1	1	1
Trans (AJ)	2	2	3	2	5	3	2
Trans (H/J)	2	2	3	2	5	3	2
Enforce Pattern	1	1	2	1	2	1	3
DB Compat’	1	2	2	1	3	2	4
Preconditions	1	2	2	1	3	2	4
Ex Handling	2	2	2	2	4	2	4
Recording Property Modifications	1	1	2	1	3	2	3

Table 2: Summary of evaluation

6 CONCLUSIONS

The final chapter will draw some conclusions from the large volume of information presented during the design, implementation and evaluation chapters. The discussion will focus primarily on the advantages and disadvantages of aspect-oriented software development in relation to object-oriented software development. The current state of the AOSD field is considered and a predication on where it is going in the near future is offered. Finally, future areas for related research are identified.

6.1 General Conclusions

Following the evaluation of the design and implementation of the concerns identified in the AppTrack system, a number of conclusions about AOSD (and the AOP languages employed) were arrived at. These conclusions share some similarities with conclusions of related work. No radically different results were discovered in this dissertation that dramatically contradict the conclusions of related work. The general conclusions are the following:

- AOSD techniques can greatly improve modularity in software systems
- The degree of separation achievable is related to the type of concern being implemented
- No great change in the complexity of the system as a whole following an AOSD re-implementation
- AOSD techniques can greatly aid the maintenance of concerns

Applying AOSD techniques to a software development effort can greatly increase the overall modularity of the application that is produced. The separation of both development and production concerns that is possible with AOSD is much greater than that possible with traditional approaches such as object-oriented software development. The greatest separation was achieved when implementing development concerns with global scope that required no core codebase preparation to cater for their addition to the application. The core codebase is oblivious to the existence of these concerns, even though the concerns affect this codebase. It must be noted however, that the separation of production concerns may not always be purely semantic; rather it may in some circumstances (transactions concern) be simply syntactic [45].

Generally, no vast changes in the complexity levels of the AppTrack application were observed. Whilst fluctuations (in both directions) were apparent following the implementation of each concern, in most cases the changes in complexity were not of major importance. These conclusions are supported by the findings of [40]. The authors found that the complexity of applications implemented using AspectJ was not markedly reduced or enhanced. The authors of [40] also state that the main reductions in complexity come from the simplicity that results when concerns are separated, and the main increases come from the difficulty in reasoning about the concerns to integrate a subset of them to form a coherent system. This was found to be the case in my work. It is interesting however to note that certain types of aspects cause the most notable fluctuations. Development aspects such as tracing produced the greatest reduction in complexity whilst production aspects such as transactions that require codebase intimacy caused the greatest increase in complexity. This increase in complexity can be attributed to the nature of the concerns. According to [45], the nature of some production concerns (such as transactions) leads to very intricate aspect-oriented implementations. As mentioned above, the separation of certain production concerns can be considered more syntactic than semantic [45]. This is a factor that contributes to the lack of satisfactory separation of concerns, which in turn leads to an increase in complexity. This is discussed further in section 6.1.2.

Maintenance of concerns is greatly improved following their re-implementation with AOSD techniques. The ease of maintenance afforded by an aspect-oriented implementation is a source of heightened productivity. While initial development of complex production concerns takes longer than the development of the object-oriented version of the same concern, the benefits of the aspect-oriented implementation are clearly evident in the later stages of the software development lifecycle. Changeability, pluggability and extensibility, important properties in relation to maintenance efforts, were greatly enhanced in all concerns. There was no notable distinction between development and production concerns in this area. Statements made in [41] support my conclusions regarding maintenance and productivity. The author asserts that it is not only true that the real benefits of AOSD are seen in the later stages of the software development lifecycle, but that this behaviour is essential. Without the benefit of simplified maintenance and increased productivity, AOSD would fail to deliver on some on its key promises. This dissertation shows that it does not fail to deliver. However, I must note that it is likely that the experienced increase in maintenance productivity (experienced during my constant refactoring the re-implemented concerns) owes somewhat to my intimate knowledge of the aspect-oriented implementation of the AppTrack system rather than just to the power of AOSD. The heightened complexity of some concerns may actually serve as a hindrance to maintenance productivity levels if the maintainer is not familiar with either the AppTrack application or AOSD and the relevant AOP mechanisms.

6.1.1 Intimate Concerns

The aspect-oriented re-implementation of the AppTrack system requires some concerns to affect data in the core codebase. These concerns require some codebase preparation. Such concerns are referred to as being intimate [50]. Two extremely intimate concerns were identified in the AppTrack system (transactions and exception handling). To implement these concerns whilst maintaining the current state of the core codebase would have required the aspect classes to access local variables declared in methods that the aspect classes affect. Such functionality is not present in the current implementations of the AOP languages used, nor does it seem likely that it will ever be present. For this reason, significant codebase preparation is required to cater for the implementation of the intimate concerns. Of course, a language that provides access to local variables would be heavily criticised for breaking encapsulation. This is a situation that the designers of AspectJ find themselves in at present due to their provision for `privileged` aspects. An aspect, when defined as being `privileged`, can access the private data members of another class. This constructed has been the source of much criticism directed at the AspectJ language. It is unlikely that the designers will change the language to allow access to local variables, but a solution to this dilemma of codebase preparation is clearly necessary.

Related to this discussion of codebase preparation is the argument of dynamic binding/polymorphism vs. reflection. The two intimate concerns mentioned above both require some codebase preparation so that all classes affected by the concerns will have a variable available for manipulation by the relevant aspect classes. This variable can be added to a set of classes by having them extend another class that contains the variable (as was the case in the exception handling concern). At runtime, the aspect class can use dynamic binding (via Java casting) to attach the currently executing object to an object whose type is that of the parent class of the currently executing object. All classes affected by the concern have a common parent class. This removes the need to specify the type of all affected classes within the aspect and produces quite an elegant aspect implementation. However, as Java comes with single inheritance only there is a large cost associated with this solution i.e. you cannot extend the class via inheritance in the future as the sole inheritance relationship has been consumed.

An alternative to using inheritance to place the required variable in the classes affected by a concern is to simply insert the variable into each class manually (as was the case in the transactions concern). This leaves the classes free to extend another class as required. However, to avoid exhaustively naming all the classes affected by this concern, reflection must be used in the aspect implementation. This is because there is no common parent class binding the set of classes affected by the concern together. A reflective solution is considerably more complex than one using dynamic binding/polymorphism but it does not use up the precious inheritance relationship.

The above discussion illustrates the need for further consideration to be given to the area of variable/method access from an unrelated collection of classes. Even if AOP languages were equipped with the ability to access local variables from aspect classes, the matter of dynamically accessing these variables without declaring all the possible classes that the currently executing class would still be restrictively complicated.

6.1.2 Remembering Object-Oriented Programming

The primary aim of AOSD is the separation of crosscutting concerns. This involves removing the scattering and tangling that is inherent in object-oriented systems. During the process of identifying crosscutting concerns in the AppTrack system, numerous instances of scattered and tangled code were exposed. This scattered and tangled code was then removed during the aspect-oriented re-implementation process. However, it was observed that some of the scattering could have been removed through the application of superior object-oriented software development techniques (section 3.4). Object-oriented attempts to eradicate code scattering still however leave a certain, though significantly reduced, amount of scattering and do not alleviate tangling. Whilst AOSD gives more in terms of separation of concerns than object-oriented software development, good OO practices should be exploited fully, regardless of whether AOSD is being used on a project or not. AOSD should not become an excuse for poor object-oriented software development.

The growing popularity of AOSD should be no justification for developers to rush to "aspect everything". This issue is highlighted and discussed in [45]. The paper concludes that it may appear that non-functional concerns (such as transactions) would benefit from a greater degree separation, but that in reality this is not actually the case. The paper states that for non-trivial distributed applications, concurrency control (together with the associated failure management), should be dealt with as a full part of the application semantics i.e. it should be contained along side the rest of the core codebase. The paper serves as a warning to AOSD newcomers with high expectations. The findings on the transactions concern implemented in this dissertation share some similarities with the conclusions of [45]. Whilst a full understanding of the concern affords benefits such as increased evolvability and maintenance productivity, it also has its disadvantages.

6.2 The Value of Re-engineering with AOSD

The re-implementation of the AppTrack system, and the conclusions drawn from its evaluation, indicate that there is a definite value associated with re-implementing/re-engineering an existing codebase using AOSD techniques. For all concerns, the concern separation afforded by AOSD leads to benefits in the maintenance stage of the software lifecycle. However, these benefits may not be offset by other properties that arise as a result of the re-implementation. I am referring here to the heightened complexity and general loss of comprehensibility that that arose as a result of the AOSD implementation of complex production concerns such as transactions and exception handling. I conclude that the value afforded by an AOSD re-implementation can indeed be significant (and worth the re-implementation cost), however, it is directly related to the genre of the concerns that are re-implemented.

6.3 Future of AOSD

It has been demonstrated that AOSD, along with its supporting implementation languages and tools, goes a good distance towards fulfilling its core responsibilities as a new technology for the separation of crosscutting concerns in computer software. AOSD represents a significant leap in the historically heavily explored area of separation of concerns and it can help in constructing more maintainable code. Developers across the globe are beginning to recognise the importance of AOSD, as modularisation of crosscutting concerns is an important factor in the development of quality software. The recent advances made by the field augur well for its future. There are a number of factors that I consider important for AOSD's future success.

Tool and user support are of the utmost importance for the success of the languages and approaches that are supporting the field. It is no surprise that the language most recognised as the face of AOP is the language with the most sophisticated tool and user support.

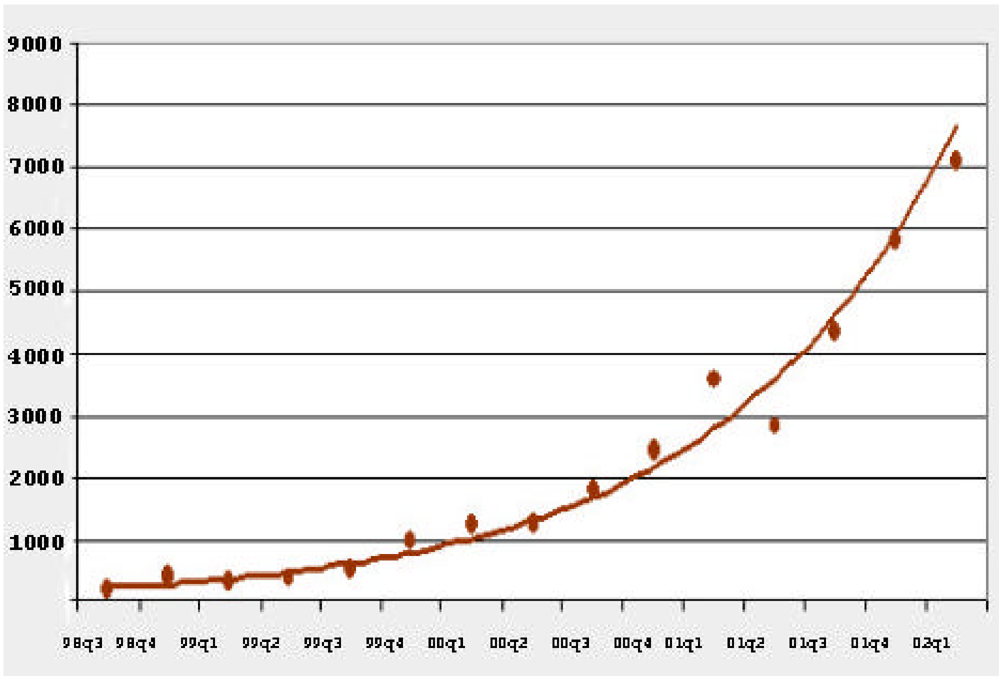

Figure 42: The adoption of AspectJ - 1998 to 2002

Figure 42 [51] shows the adoption rates that AspectJ has experienced since the third quarter of 1998 up until the first quarter of 2002. The y-axis shows the number of users adopting the language and the x-axis represents the adoption timescale. This impressive adoption curve is no doubt a direct result of the tool and user support. AspectJ has implemented IDE plug-ins for JBuilder, Forte and Eclipse, as well as support for Emacs and Ant. AspectJ also maintains several very active mailing lists to connect users from around the world. Regardless of the merits of the model implemented by AspectJ when compared to the opposing models such as hyperspaces and composition filters, the model with the most support will be the one that is adopted by the masses. This is proved by the fact that tools and languages supporting other approaches to AOSD have not experienced as much success as AspectJ. During the re-implementation of the AppTrack system, tool support was a major factor in the success or otherwise of a concern implementation. The future success of languages/tools and the field in general relies heavily on the provision of high quality tool and user support. This feeling is echoed by [45].

AOSD is, relatively speaking, in its infancy. I envisage continued research being another key factor in its future success. Researchers need to build upon the previously presented assessment-based studies with broader, more in-depth studies. The commercial success of a technology depends greatly on its capabilities in large multi-developer projects. Thus far, assessments of AOSD and its associated AOP technologies have not been conducted in this environment, but rather in smaller, more controlled environments. AOSD must prove itself capable of catering for the types of problems encountered in these environments. The work presented in this dissertation suggests that this is an area that must be addressed.

It is clear from this work that AOSD techniques are more applicable to certain types of concern than others. The genre of concern that AOSD is most effective in separating (this research strongly suggests that it is development concerns with global scope) must be clearly defined and publicised. AOSD is not the perfect solution to everything and selling itself as such will do more harm than good.

By undertaking further significant research, defining its core intentions more clearly, and by showing an all round improvement in tool and user support, AOSD should continue to grow at its current rate of progression. This would suggest that AOSD could become a major software development methodology in the near-to-mid future.

6.4 Future Research

This section highlights some areas for future work related to this dissertation. One piece of work relates to the separation of concerns in code whereas the other is related to the design of systems implemented using AOSD techniques.

6.4.1 Aspect-Oriented Unit Testing

A test-first development approach was adopted as part of the eXtreme Programming (XP) methodology for the development of the initial object-oriented version of the AppTrack system. As the proportion of the codebase related to automated testing increased, so did the complexity of the test code. The tangling of concerns in this test codebase was phenomenal, possibly greater than that evident in the main application. Testing functionality written towards the end of the project became next to impossible due to the intricate web of relationships that existed between the various unit test modules. The implementation and execution of a new test had the potential to adversely affect the results of an existing test. This was clearly an untenable situation and minor efforts (due to time constraints) to improve the situation resulted in the successful completion of the application.

Despite the amendments made towards the end of the initial development cycle, there is still a massive need to refactor the whole test codebase. An AOSD approach to this refactoring effort would make for some interesting and worthwhile future work on this project as it would expose a possible complimentary relationship between AOSD and XP. Such work would tie in with recent research in this area such as that mentioned in [42]. The concept of supporting XP unit tests by using AOSD techniques has been discussed by both the AOSD community [43] and the XP community [44].

6.4.2 Aspect-Oriented Design Patterns

As highlighted in the evaluation section (specifically sections 5.5.3, 5.6.3 and 5.7.3), some AOSD concern implementations, whilst not extremely reusable, do follow patterns of implementation that appear to provide a good basis for the future implementation of similar concerns. Whilst work [52] has been done in implementing the GOF [34] design patterns, this future work would seek to apply aspect implementation architectures (presented in this dissertation) in various environments, to asses whether their reuse is both feasible and beneficial. If it is concluded that the reuse of an aspect architecture is applicable in many environments, then a formal AOSD design pattern could be created based on this architecture.

BIBLIOGRAPHY

[1] E W. Dijkstra. “A Discipline of Programming”. Prentice-Hall, 1976.

[2] D. L. Parnas. “On the Criteria to be used in Decomposing Systems into Modules. Communications of the ACM”. Vol. 15. No. 12. December 1972, pp. 1053-1058.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin. "Aspect-Oriented Programming". ECOOP, 1997

[4] www.aosd.net - The Aspect-Oriented Software Development website: May 13th, 2002

[5] G. Kiczales. “Discussing Aspects of AOP”. Communications of the ACM. October 2001 - Volume 44, Number 10. P 33.

[6] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach". ECOOP 2000

[7] M. Askit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa. "Abstracting Object Interactions Using Composition Filters". ECOOP 1993

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. "An Overview of AspectJ". ECOOP 2001

[9] E. Hilsdale, J. Hugunin. Introduction to Aspect-Oriented Programming with AspectJ. Tutorial 3, AOSD 2002

[10] www.aspectj.org - The AspectJ website: May 13th, 2002

[11] A. Goldberg, D. Robson. "Smalltalk-80: the language and Its Implementation". Addison-Wesley, 1983

[12] References www.aspectj.org/doc/dist/faq.html#q:encasulationviolations - The AspectJ FAQ webpage: May 14th, 2002

[13] aspectr.sourceforge.net - The AspectR website: May 14th, 2002

[14] www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS - The AspectS website: May 14th, 2002

[15] www.cs.ubc.ca/labs/spl/projects/apostle - The Apostle website: May 14th, 2002

[16] staff.aist.go.jp/y-ichisugi/mj - The MixJuice website: May 14th, 2002

[17] www.cs.washington.edu/homes/jonal/archjava - The ArchJava website: May 14th, 2002

[18] pythius.sourceforge.net - The Pythius website: May 14th, 2002

[19] www.cs.ubc.ca/labs/spl/projects/aspectc.html - The AspectC website: May 14th, 2002

[20] www.aspectc.org - The AspectC++ website: May 14th, 2002

[21] www.cs.ubc.ca/~jan/amt - The AMT website: May 14th, 2002

[22] www.cs.ubc.ca/~mrobilla/feat - The FEAT website: May 14th, 2002

[23] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton Jr. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". ICSE 1999

[24] M. Askit and B. Tekinerdogan. "Solving the Modelling Problems of Object-Oriented Languages by Composing Multiple Aspects Using Composition Filters". AOP 98 Workshop

[25] J. Wichman "ComposeJ: The Development of a Preprocessor to Facilitate Composition Filters in the Java Language". Masters Thesis, University of Twente, 1999

[26] P. Caro. "Adding Systematic Crosscutting and Super-Imposition to Composition Filters”. Masters Thesis, University of Twente, 2001

[27] K. Lieberherr, I. Silva-Lepe, C. Xiao. "Adaptive Object-Oriented Programming using Graph-Based Customization". Communications of the ACM, 1994

[28] K. Lieberherr, D. Orleans. "Preventive Program Maintenance in Demeter/Java". ICSE 1997

[29] S. Clarke, R. Walker. "Composition Patterns: An Approach to Designing Reusable Aspects". ICSE 2001

[30] M. Kirsten, G. Murphy. “A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming”. OOPSLA 1999

[31] M. Lippert, C. Lopes. “A Study on Exception Detection and Handling Using Aspect-Oriented Programming”. ICSE 2000

[32] R. Walker, E. Baniassad, G. Murphy. “An Initial Assessment of Aspect-Oriented Programming”. ICSE 1999

[33] R. Alexander, J. Bieman. “Challenges of Aspect-oriented Technology”. Colorado State University 2002 – not yet published.

[34] E. Gamma, R. Helm, R. Johnson, J. Vlissides. “Design Patterns. Elements of Reusable Object-Oriented Software”. Addison Wesley, 1995

[35] B. Meyer. “Object-Oriented Software Construction”. Prentice Hall PTR, 1997

[36] P. Tarr, H. Ossher. “Hyper/J User and Installation Manual”. 2000. http://www.research.ibm.com/hyperspace

[37] R. Laddad. “I Want My AOP! (Part 1) - Separate software concerns with aspect-oriented programming”. Java World, January 18th, 2002. http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect-p1.html

[38] D. Stein, S. Hanenberg, R. Unland. “An UML-based Aspect-Oriented Design Notation”. AOSD, 2002

[39] W. Ming, J-M Jezequel, F. Pennaneac’h, N. Plouzeau. “A Toolkit for Weaving Aspect Oriented UML Designs”. AOSD, 2002

[40] J Andrés Díaz Pace, M. Campo. “Analyzing The Role Of Aspects In Software Design”. Communications of the ACM. October 2001 - Volume 44, Number 10. P 71

[41] J. Memmert. “AOP and Evidence of Improvements”. Thread on the aosd-discuss mailing list. February 20th, 2002

[42] M. Kircher, P. Jain, A. Corsaro. “XP + AOP = Better Software?”. XP 2002

[43] S. Toft. “AOP and Unit Testing (JUnit)”. Thread on the aosd-discuss mailing list. February 26th, 2002

[44] http://www.xpdeveloper.com/cgi-bin/wiki.cgi?XPmeetsAOP - Wiki discussion board on AOP and XP hosted on the XP developer website, April 16th 2001

[45] J. Kienzle, R. Guerraoui. "AOP: Does it Make Sense? The Case of Concurrency and Failures". ECOOP, 2002

[46] R. Jeffries, A. Anderson, C. Hendrickson, K. Beck. "Extreme Programming Installed". Addison-Wesley, 2000

[47] http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html - Programming with assertions in Java, the addition of the *assert* keyword, September 3rd 2002

[48] S. Clarke, R. Walker. "Towards a Standard Design Language for AOSD". AOSD, 2002

[49] S. Clarke, R. Walker. "Mapping Composition Patterns to AspectJ and Hyper/J". ICSE, 2001

[50] T. Elrad, R. Filman, A. Bader. "Aspect-Oriented Programming". Communications of the ACM. October 2001 - Volume 44, Number 10. P31.

[51] R. Bodkin. "Aspect-Oriented Programming with AspectJ". Slides from a talk given at SDWest, 2002

[52] J. Hannemann, G. Kiczales. "Design Pattern Implementation in Java AspectJ". OOPSLA, 2002