

# **Design Language for Aspect Oriented Software Development & Design Pattern Extensions**

**THESIS**

Submitted in partial fulfillment  
of the requirements for the degree of  
**DOCTOR OF PHILOSOPHY**

by

**DEEPAK DAHIYA**

Under the Supervision of  
**Prof. R.K. Sachdeva**



**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE  
PILANI (RAJASTHAN) INDIA  
2007**

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE  
PILANI (RAJASTHAN) INDIA**

**CERTIFICATE**

This is to certify that the thesis entitled **Design Language for Aspect Oriented Software Development and Design Pattern Extensions** which is submitted for award of Ph.D Degree of the Institute, embodies original work done by him under my supervision.

Signature in full of Supervisor: \_\_\_\_\_

Name in Capital block letters: \_\_\_\_\_

Designation: \_\_\_\_\_

Date:

## **ACKNOWLEDGEMENTS**

Although my name is the sole name on the front page, this thesis has only become a reality due to the countless contributions of many people whom I want to thank here:

I would like to express my sincere thanks to my supervisor Professor R.K. Sachdeva for his professional guidance and great support throughout all my studies and interactions with him at Indian Institute of Public Administration (IIPA), Delhi. He has taught me much about academia and beyond – for which I am very thankful.

I would like to thank the colleagues at BITS Pilani for their prompt reply to my series of queries during this period regarding work deliverables. The whole system at conducting this Ph.D programme for experienced professionals from academia and industry is just the best.

I am also very grateful to my colleagues of Institute for Integrated Learning in Management (IILM) at Delhi for all their inspiration and assistance.

Special thanks also go to Professor Radhakrishna Pillai at Indian Institute of Management (IIM) kozhikhode where I lecture as visiting faculty during summer. During surrounding discussions various comments flowed on the thesis topic and related work. Finally, thanks be to God who has been a great source of strength and motivation throughout my life.

I want to dedicate this thesis to my parents who have given me the chance of a good education and so much love and support over the years.

My spouse Sudha and daughter Mahima were especially accommodating in the times of separation that preceded this thesis work and at other times in our career after marriage.

I probably owe them much more than they think.

## **ABSTRACT**

Recently there has been growing interest in propagating the aspect paradigm to the activities in the earlier phases of the software development lifecycle. There is a need to study various approaches in the use of object-oriented design patterns and aspect oriented design approach in enterprise systems for architecture and its implementation. Aspect Oriented Software Development (AOSD) is a step towards further enhancing the popular object oriented software development framework by embedding the underlying design elements without disturbing the base design. The development of aspect oriented requirements gathering approach, design notation and environment for development of enterprise systems needs to be further refined in the context of software applications and industry. As part of the general research trend, this thesis focuses on the design and development of a general purpose design language and the path to future work is highlighted for aspect oriented software development.

This thesis starts off with a general introduction into the research area and a description of the basic concepts and rationale behind aspect oriented software development and aspect oriented programming (AOP). A critical examination of existing AOP language features and aspect oriented design languages is provided. These observations together with previous experiences of developing enterprise wide software applications based on object oriented methodologies lead to a set of fundamental design elements in aspect oriented software development. The core of the thesis presents the prototype for the design language for aspect oriented software development based on the so called 'aspect oriented software development design language'(AOSDDL). This aspect oriented design language with its graphical notation helps developers to design and comprehend aspect-oriented programs and would facilitate the perception of aspect-orientation. This design notation will help developers to assess the crosscutting effects of aspects on their base classes. Its application carries over the advantages of aspect-orientation to the design level and facilitates adaption and reuse of existing design constructs.

Extensive computer laboratory work on software applications using existing AOP languages and AOSD design notations was carried out. Next, work on the issues related to the mapping of AOP languages to existing legacy code / OO code and design patterns was carried. In the next stage, work related to mapping of AOSD design notations to existing AOP languages was taken forward. Finally, the success of the design language and its prototype graphical notation is evaluated by means of a few

concrete software applications. This shows that AOSDDL offers sufficient flexibility and extensibility to augment software development in ways that suits today's environment of providing a platform for developing enterprise wide distributed applications, and the prototype implementation confirms that the research platform offers acceptable design elements (constructs) to map aspect oriented programming to aspect oriented software development design language.

ACKNOWLEDGEMENTS .....	3
ABSTRACT .....	4
LIST OF FIGURES .....	9
LIST OF ABBREVIATIONS / SYMBOLS .....	10
CHAPTER 1 .....	11
Introduction .....	11
1.1 Overview .....	11
1.2 Evolution of Software Programming Methodology .....	11
1.3 Aspect Oriented Programming And Design .....	13
1.4 Need for Aspect Oriented Design in Software Development? .....	15
1.5 Aspect Oriented Software Development Design Language .....	16
1.6 Who are the beneficiaries? .....	16
1.7 Research Challenges .....	16
1.8 Thesis Structure .....	18
CHAPTER 2 .....	21
Aspect Oriented Programming .....	21
2.1 Overview .....	21
2.2 Background .....	21
2.3 Why do we need AOP? .....	22
2.4 Evolution of programming methodologies .....	23
2.5 Managing system concerns .....	24
2.6 Identifying system concerns .....	25
2.7 A one-dimensional solution .....	26
2.8 Modularizing .....	27
2.8.1 Implementing crosscutting concerns in non modularized systems .....	30
2.8.2 Symptoms of nonmodularization .....	31
2.8.3 Implications of non modularization .....	33
2.9 Aspect Oriented Programming (AOP) appears on the Scene .....	34
2.10 Background of AOP .....	35
2.11 The AOP methodology .....	36
2.12 Anatomy of an AOP language .....	37
2.13 Benefits of AOP .....	37
2.14 Summary .....	39
CHAPTER 3 .....	41
Related Work .....	41
3.1 Overview .....	41
3.2 UMLAUT: Weaving UML Designs .....	43
3.2.1 General Architecture and Core Engine .....	43
3.2.2 The Extendible Transformation Framework .....	44
3.3 Theme/UML .....	45
3.3.1 Design Approaches .....	48
3.3.2 Implementation Approaches .....	48
3.3.3 Lifecycle Impact .....	49
3.4 Approach To Aspect Oriented Programming And Design .....	50
3.5 Impact Of Requirements Engineering On AOSD .....	51
3.6 Aspect Oriented Requirements Engineering (AORE) .....	52
3.7 Software Architecture View Of Aspects .....	53
3.8 Aspect Oriented Software Development Design Language .....	54
3.9 Summary .....	55

CHAPTER 4 .....	56
Design Language Requirements .....	56
4.1 Overview .....	56
4.2 Requirements .....	56
4.3 Evolving of Concerns in Early Requirements Phase .....	58
4.4 Separation of Concerns .....	58
4.5 Multiple Perspectives to Concern Requirements .....	59
4.6 Architectural Design Enforcement.....	62
4.6.1 Enforcing Architectural Regularities.....	62
4.7 Avoiding Design Incompatibility .....	63
4.8 Requirements Validation .....	65
4.9 AOSDDL Requirements.....	66
4.10 Summary.....	67
CHAPTER 5.....	68
The AOSDDL.....	68
5.1 Overview .....	68
5.2 Tools Environment .....	68
5.3 AOSDDL Notations .....	69
5.3.1 Symbols for AOSDDL Notations .....	70
5.4 Other Design Notations.....	74
5.5 Mapping AOP to Aspect Oriented UML Extensions.....	74
5.5.1 Problem Identification.....	75
5.5.2 Basic Notations for AspectJ .....	75
5.5.3 Weaving Mechanism of AspectJ .....	82
5.5.4 Weaving Advice .....	82
5.5.5 Weaving Introductions .....	84
5.5.6 Weaving Relationship .....	85
5.6 Identifying Software Concerns .....	86
5.6.1 Concern Modeling Schema Framework.....	87
5.6.2 Applications .....	87
5.7 Design Language Issues for Component Based software Development .....	88
5.8 Mapping UML extensions through composition patterns to Aspects.....	90
5.8.1 Mapping To AspectJ.....	91
5.9 AspectJ Extensions for Distributed Computing.....	91
5.9.1 Implications on Network Processing.....	92
5.10 Summary.....	93
CHAPTER 6.....	95
Implementation and Evaluation .....	95
6.1 Overview .....	95
6.2 Processing and Test Environments .....	95
6.3 Mapping Learning Resource Center (LRC) Design to Aspects .....	95
6.3.1 Functional Decomposition .....	96
6.3.2 Design Diagrams for Learning Resources Center (LRC) .....	96
6.3.3 Aspects Modularization and Dependency Effects of Aspects .....	97
6.3.4 Designing Aspects.....	99
6.3.5 Crosscutting Requirements: Aspects .....	100
6.3.6 Designing LRCObserver Aspect .....	102
6.4 Testing.....	105
6.5 Defining Validation Parameters for Aspects .....	106
6.6 Detecting Aspect Faults .....	109
6.7 Evaluation.....	109
6.8 Qualitative Evaluation: Mapping To AspectJ .....	109

6.8.1 Synchronize in AspectJ .....	110
6.9 AspectJ Extensions for Distributed Computing .....	112
6.9.1 Implications on Network Processing .....	113
6.10 Summary .....	117
<b>CHAPTER 7 .....</b>	<b>119</b>
<b>Conclusion .....</b>	<b>119</b>
7.1 Overview .....	119
7.2 Thesis Summary .....	119
7.3 Concluding Remarks .....	121
<b>CONTRIBUTIONS .....</b>	<b>123</b>
Natural Extension to UML .....	123
CASE Tool Support .....	123
Extension of Architectural framework for design constructs .....	123
Enforcing Architectural Regularities .....	124
Implementation Support .....	124
Software Development .....	124
<b>FUTURE SCOPE OF WORK .....</b>	<b>125</b>
Code Generators for Aspect modeling .....	125
Tools Integration .....	125
Notation deployment .....	126
Support for Hyper/J .....	126
Testing of Aspect Oriented Requirements .....	126
Summary .....	126
<b>REFERENCES .....</b>	<b>127</b>
<b>BIBLIOGRAPHY .....</b>	<b>132</b>
<b>LIST OF PUBLICATIONS AND PRESENTATIONS .....</b>	<b>133</b>
<b>APPENDICES .....</b>	<b>135</b>
A. Installation and Configuration of Eclipse Platform .....	135
A.1 Running Eclipse .....	135
A.2 Importing an existing AspectJ project .....	136
A.3 Running AspectJ project in Eclipse .....	136
<b>BRIEF BIOGRAPHY OF THE CANDIDATE .....</b>	<b>137</b>
<b>BRIEF BIOGRAPHY OF THE SUPERVISOR .....</b>	<b>138</b>



## LIST OF FIGURES

Figure 2.1... Viewing a system as a composition of multiple concerns ...	Page 24
Figure 2.2... Concern decomposition .....	Page 25
Figure 2.3... Mapping the N-dimensional concern space .....	Page 25
Figure 2.4... Implementation of a logging concern .....	Page 28
Figure 2.5... Implementation of a logging concern using AOP techniques	Page 28
Figure 2.6... Code tangling .....	Page 29
Figure 2.7... Code scattering caused by identical code blocks .....	Page 30
Figure 2.8... Code scattering caused by complementary code blocks ...	Page 31
Figure 2.9... AOP development stages .....	Page 34
Figure 5.1... Using Package and Connector .....	Page 68
Figure 5.2... Connector Syntax .....	Page 72
Figure 5.3... Representing Join Points in Sequence Diagrams .....	Page 73
Figure 5.4... Mapping of Pointcuts to Operations .....	Page 74
Figure 5.5... Mapping of Advice to an Operation .....	Page 75
Figure 5.6... Design of Introductions .....	Page 76
Figure 5.7... Weaving Advice with UML Use Cases .....	Page 79
Figure 5.8... Specifying Weaving Order .....	Page 79
Figure 5.9... Weaving Introductions with UML Use Cases .....	Page 81
Figure 6.1... Learning Resource Center (LRC) Design .....	Page 93
Figure 6.2... Hierarchy Diagram for LRC .....	Page 94
Figure 6.3... AOSDDL's Authentication functionality .....	Page 94
Figure 6.4... Effect of aspects with pointcut-advise on dependencies .....	Page 95
Figure 6.5... Effect of aspects with introduction on dependencies .....	Page 95
Figure 6.6... Design rules for Aspect Oriented Modularization .....	Page 96
Figure 6.7... LRC Synchronization for Aspect Design .....	Page 97
Figure 6.8... Synchronization of Learning Resource Center (LRC).....	Page 98
Figure 6.9... Aspect Design for LRCAbsObserver .....	Page 100
Figure 6.10... Design Aspect for LRCAbsObserver .....	Page 101
Figure 6.11... Design Aspect for Authentication (LRCLogger).....	Page 101
Figure 6.12... The State Model of Class Journal .....	Page 107
Figure 6.13... The Overdue Aspect .....	Page 108
Figure 6.14... The Testing Code in AspectJ .....	Page 107

## LIST OF ABBREVIATIONS / SYMBOLS

AO	Aspect Oriented
AODM	Aspect Oriented Design Model
AOP	Aspect-oriented Programming
AOSDDL	Aspect-Oriented Software Development Design Language
API	Application Programming Interface
ASOC	Advanced Separation of Concerns
UML	Unified Modeling Language
CASE	Computer Aided Software Engineering
CBSD	Component Based Software Development
EJB	Enterprise Java Beans
OO	Object Oriented
OOD	Object Oriented Design
OOP	Object Oriented Programming
OOSD	Object Oriented Software Development
QoS	Quality of Service
RMI	Remote Method Invocation
SOC	Separation of Concerns
SOCMS	Separation of Concern Modeling Schema
XML	Extensible Markup Language

# **CHAPTER 1**

## **Introduction**

### **1.1 Overview**

This introduction sets the stage for the research carried out for this thesis. It introduces the concept of “aspect oriented programming” and outlines the general path of research that has been taken.

An analysis of the evolution of object oriented design methodology shows that the original object or class architecture was not designed for the requirements of today’s enterprise wide distributed environment. This chapter outlines how the novel paradigm proposed by aspect oriented design language could advance the current design architecture and overcome its main design flaws. A discussion of the applications of aspect oriented programming and its advantages highlights the potential beneficiaries of this new design methodology, namely third party tool developers, software developers, software vendors and most importantly the end users.

This chapter concludes with an overview of the main research challenges that are targeted by this research effort, followed by an outline of the thesis structure.

### **1.2 Evolution of Software Programming Methodology**

In the early days of computer science, developers wrote programs by means of direct machine-level coding[1]. Unfortunately, programmers spent more time thinking about a particular machine's instruction set than the problem at hand. Slowly, we migrated to higher-level languages that allowed some abstraction of the underlying machine. Then came structured languages, we could now decompose our problems in terms of the procedures necessary to perform our tasks. However, as complexity grew, we needed better techniques. Object-oriented programming (OOP) let us view a system as a set of collaborating objects. Classes allow us to hide implementation details beneath interfaces. Polymorphism provided a common behavior and interface for related concepts, and allowed more specialized components to change a particular behavior without needing access to the implementation of base concepts.

Programming methodologies and languages define the way we communicate with machines. Each new methodology presents new ways to decompose problems: machine code, machine-independent code, procedures, classes, and so on. Each new methodology allowed a more natural mapping of system requirements to programming constructs. Evolution of these programming methodologies let us create systems with ever increasing complexity. The converse of this fact may be equally true: we allowed the existence of ever more complex systems because these techniques permitted us to deal with that complexity.

There is a well documented problem in the software engineering field relating to a structural mismatch between the specification of requirements for software systems and the specification of object-oriented software systems. The structural mismatch happens because the units of interest during the requirements phase (for example, feature, service, capability, function etc.) are different to the units of interest during object-oriented design and implementation (for example, object, class, method, etc.)[2]. The structural mismatch results in support for a single requirement being scattered across the design units and a single design unit supporting multiple requirements - this in turn results in reduced comprehensibility, traceability and reuse of design models. Currently, OOP serves as the methodology of choice for most new software development projects. Indeed, OOP has shown its strength when it comes to modeling common behavior. However, OOP does not adequately address behaviors that span over many -- often unrelated -- modules. Separation of concerns is a basic engineering principle that is also at the core of object-oriented analysis and design methods in the context of UML [3]. Separation of concerns can provide many benefits: additive, rather than invasive, change; improved comprehension and reduction of complexity; adaptability, customizability, and reuse.

In contrast, AOP [4] methodology fills this void. AOP quite possibly represents the next big step in the evolution of programming methodologies. However, for aspect-oriented software development (AOSD) [5] to live up to being a software engineering paradigm, there must be support for the separation of crosscutting concerns across the development lifecycle including traceability from one lifecycle phase to another. Concerns that have a crosscutting impact on software (such as distribution, persistence, etc.) present well-documented difficulties for software

development. Since these difficulties are present throughout the development lifecycle, they must be addressed across its entirety.

Although a lot has been done to study the aspect oriented design approach in enterprise systems for architecture and its implementation, work on a general-purpose design language for aspect-oriented software development is attracting a lot of attention. The development of aspect oriented requirements gathering approach, design notation and environment for development of enterprise systems needs to be further refined in the context of software applications and industry.

This discussion has shown a range of design methodologies related to object oriented and aspect oriented software development that augment the current software industry scene and practices. Ongoing efforts in this area suggest that this trend of incorporating aspect elements inside any object oriented software design is far from over.

The majority of these designs are implemented as individual ad-hoc extensions – all with the goal of improving the software design to account for today's requirements such as logging, caching, persistence and distribution. However, the fundamental problem, namely that the programming methodology provides no architectural support for flexible extensibility, remains.

This thesis therefore investigates traceability between developing a standard and general purpose AOSD design language with existing UML features and extensions to map AOSD design notations to AOP language. The aim is to provide a uniform design interface to add new extensions (for example, logging, caching, security etc) with a view towards eventually developing a standard design language for a broad range of AOSD approaches – independent of the programming language in hand.

### **1.3 Aspect Oriented Programming And Design**

A gap exists between requirements and design on one hand, and between design and code on the other hand. Aspect oriented programming (AOP) extended to the modeling level where aspects could be explicitly specified during the design process will make it possible to weave these aspects into a final implementation model. Another step could be extension of AOP to the entire software development cycle. Each aspect of design and implementation should be declared during the design phase so that there is a clear traceability from requirements through source

code thus using UML as the design language to provide an aspect-oriented design environment.

The separation and encapsulation of crosscutting concerns has been promoted as a means of addressing these difficulties; the standard object-oriented paradigm does not suffice. In order to overcome the difficulties for crosscutting concerns throughout the lifecycle, an approach is required that provides a means to separate and encapsulate both the design and the code of crosscutting behaviour. It is important to work towards a general purpose AOSD design language that meets certain goals including the following:

- *Implementation language independent*: The final form of AOP language may vary from that of any current one. Thus, any design language that simply mimics the constructs of a particular AOP language is liable to fail to achieve implementation language independence.
- *Design-level composability*: Design level composability is a desirable property for two reasons. First designers may check the result of composition prior to implementation, for validation purposes. Second, some projects will continue to require the use of a non-aspect-oriented implementation language because of pragmatic constraints, such as the presence of legacy code written in languages without aspect-oriented extensions; these projects could still benefit from separating the design of crosscutting concerns.
- *Compatibility with existing design approaches*: An AOSD design-level language should also build existing design languages such as UML, to provide a bridge from old techniques to new, so that software engineering realities such as incremental adoption and legacy support are possible.

The construction of complex, evolving software systems requires a high-level design model. This model should be made explicit, particularly the part of it that specifies the principles and guidelines that are to govern the structure of the system. In reality, however, implementators tend to overlook the documented design models and guidelines, causing the implemented system to diverge from its model. Reasoning about a system whose models and implementation diverge is

error prone – the knowledge we gain from these models is not of the system itself, but of some fictitious system, the system we intended to build. The system's comprehensibility is impeded, and so using software engineering techniques goes against our intended goals – quality, maintainability and cost minimization. The essence of the problem of implementing higher-level principles and guidelines lies in their globality. These principles cannot be localized in a single module, they must be observed everywhere in the system, which means that they crosscut the system's architecture.

#### **1.4 Need for Aspect Oriented Design in Software Development?**

The identification of the mapping and influence of a requirement level aspect promotes traceability of broadly scoped requirements and constraints throughout system development, maintenance and evolution. The improved modularization and traceability obtained through early separation of crosscutting concerns can play a central role in building systems resilient to unanticipated changes hence meeting the adaptability needs of volatile domains such as banking, telecommunications and e-commerce. These crosscutting concerns are responsible for producing tangled representations that are difficult to understand and maintain. Examples of such concerns at the requirements level are compatibility, availability and security requirements that cannot be encapsulated by a use case and are typically spread across several of them.

With increasing support for aspects at the design and implementation level, the inclusion of aspects as fundamental modeling primitives at the requirements level and identification of their mappings also helps to ensure homogeneity in an aspect oriented software development project.

The main drive behind aspect oriented design language research is the idea of developing design constructs (elements) that exhibit a degree of flexibility and customizability that is only known from programmable end systems. While new design language constructs based on aspect oriented programming are being designed they are still tied to a particular platform whereby the vendor provides both the software tool and the design language tool as a complete package with additional proprietary tools. Thus, new design language aspect constructs can only be tested or utilized to individual specific requirements after the vendor has released a software upgrade. The development of new functionality is typically

preceded by a long and awkward standardization process. These different paradigms have created an increasing gap between the functions and capabilities of these constructs in an aspect oriented development environment.

Reconsidering the system architecture of object oriented software applications is therefore a crucial step in aspect oriented software development.

### **1.5 Aspect Oriented Software Development Design Language**

AspectJ [6, 7, 8] is a popular and well established AOP language that provides support for specifying and composing crosscutting code into a core system. It supports the AOP paradigm by providing a special unit, called “aspect”, which encapsulates crosscutting code. Other compositional implementation languages and mechanisms also exist [9, 10]. At the design level, an AOSD design language with extensions to UML [1, 11, 12, and 13] in its capabilities relating to decomposition and modularization is required that would map to a particular AOSD implementation. Further, a standard AOSD design language must be capable of supporting many of these aspect programming languages. A graphical notation helps developers to design and comprehend aspect-oriented programs. Further, it would facilitate the perception of aspect-orientation. A design notation helps developers to assess the crosscutting effects of aspects on their base classes. Its application carries over the advantages of aspect-orientation to the design level and facilitates adaption and reuse of existing design constructs.

### **1.6 Who are the beneficiaries?**

The advantages of a flexible and extensible aspect oriented design language are expected to benefit the software community at various levels.

### **1.7 Research Challenges**

The main aim of this work is to investigate flexible and extensible mechanisms that enable dynamic introduction of new functionality into an existing operational design. This endeavor is pursued from the endpoint of the programmer and the design team as both has a great interest in implementation and / or processing of individual elements.

The key challenge of this thesis therefore is to design a novel design language architecture that provides the basis for flexible extensibility of design functionality.



In order to verify the practicality of this architecture, prototyping an application according to the new design elements will be a major part of this undertaking.

The challenges of the architectural design language are as follows:

- *Generic platform (not tied to a specific application)*

The design goal is to develop a generic programmable design language platform to support the diversity of today's and future design specifications. The idea is to replace the numerous ad hoc approaches to provide specific design elements inside the language that allows users (such as programmers or systems analyst) to extend the design capabilities in a uniform way.

Unlike most existing design language architectures, which are tied to a specific application domain, the goal here is to start with a requirement analysis of a wide range of software applications and design specifications in order to consider the multitude of requirements in the architectural design.

- *Modular component-based architecture*

Another key objective is to design a design language architecture that is truly component-based taking advantage of component features such as modularity, extensibility, and reusability. The design elements can hence be programmed into aspects or classes called components. These components will typically provide a new specification or simply extend an existing specification.

The component architecture allows complex technical and design specifications to be split into simply and easy-to-develop functional components. This 'divide and conquer' approach eases the design and development of specifications. Moreover, it improves the granularity of design specification extensibility and reusability of components among specifications.

- *Compatibility and transparency*

The introduction of aspect oriented programming in current design methodologies, such as object-oriented, depends largely on how easily it can be integrated with existing technologies. It is therefore a major objective to design the design language architecture in a way that enables

seamless transitioning towards the aspect based programming paradigm. Most early design proposals, for example, did not consider the crosscutting concerns, a vital requirement, and hence, ended up with solutions that rely on a design consisting only of objects and classes. Such software systems are obviously very hard to introduce in a distributed environment where security, caching and logging are major concerns. Consequently, an important goal here is to design an aspect based architecture that allows transparent, and hence seamless, application of design elements to the software components. No change to the domain specific functional components, systems and applications, or the intermediate modules that are not directly involved should be required. Such transparent solutions have the advantage that a partial transitioning from object oriented design to aspect oriented design – where the common but the more important concerns reside are most effective – is possible.

- *Commercial feasibility*

Another important factor for the success of aspect oriented design language is its commercial viability. Many great technologies have failed in the past simply due to a weak business model. As a result, this work focuses on a solution that has evident beneficiaries and a likely commercial perspective.

The challenge is to develop an active design language that enables third party development of aspect based software applications. Breaking the tight coupling between the design language and the software development environments decouples the role of the systems analyst from the software vendor and thus opens up a new competitive market for third party aspect oriented design software. This is particularly promising as unhindered competition typically maximizes the cost-performance ratio of products and specifications.

## **1.8 Thesis Structure**

This first chapter of the thesis has introduced the concepts of aspect oriented programming and software development. It outlines how the new methodology has emerged from traditional object oriented methodologies as a result of the growing demands of today's software practitioners and applications. Furthermore, it

provides the motivation for this line of research along with the main research challenges of this study. The remainder of this thesis is structured as follows.

Chapter two continues with an introduction of the general background and issues of aspect oriented software development. It defines the basic methodology and introduces the main concepts. These include different design language approaches towards aspect oriented programming, various programming models and other important issues such as crosscutting concerns and system security and integrity.

Chapter three provides a comprehensive overview of the current state-of-the art in the field by introducing related work that is or has been under investigation at other research institutions and universities. A special focus is placed on research into aspect oriented software design methodologies and enabling technologies. Chapter three concludes with an overview of current work on aspect oriented applications and design language specifications.

Chapter four continues with a requirements analysis for aspect oriented systems. The requirements are derived from past experiences in object oriented and aspect oriented programming paradigms of working in the software industry and academics in my previous work places and a thorough study of related work as well as other influencing factors, for example commercial aspects such as the deployment of new technologies. From these general requirements a subset of requirements that form the basis for the design of the AOSDDL design language architecture and implementation is drawn.

Chapter five presents the AOSDDL design language. This central part of the thesis describes in detail how AOSDDL operates and how the component based design architecture enables handling of crosscutting concerns through flexible integration and extensibility of design functionality. In addition to the basic language design, special focus is placed on the following key aspects: components, distribution and weaving.

Chapter six then describes the ongoing implementation efforts of developing prototype design constructs of the AOSDDL design language architecture. Due to the considerable extent of the AOSDDL architecture, this chapter focuses primarily on validating the key aspects of the design through a 'proof-of-concept' implementation.

It continues with a qualitative and quantitative evaluation of AOSDDL and its prototype implementation. It evaluates how the AOSDDL architecture satisfies the

objectives and requirements identified in chapter four based on a case study and several example applications.

Finally, chapter seven concludes the thesis by drawing together the main arguments of this work and summarizing the contributions that have been made. It also describes future work that could be carried out based on this line of research.

## **CHAPTER 2**

### **Aspect Oriented Programming**

#### **2.1 Overview**

This chapter provides a general background on the field of aspect oriented programming. It looks back to the initial developments of this trend in the 1990's and shows how the field has evolved since.

The main focus however is to introduce the core concepts [14] and issues of aspect oriented programming as a basis for further discussions throughout the thesis. As such, this chapter defines the basic methodology for aspect oriented programming and describes various approaches towards aspect oriented software development. Although the idea of adopting the appropriate design methodology to software development is not revolutionary (for example, object oriented approach is also a design methodology), identifying common concerns in a software design and separating this functionality requires architectural changes to the design and implementation of current aspects. This chapter introduces several architectural approaches to the design of aspect oriented systems and defines various programming models for aspect oriented software development.

Furthermore, the fact that aspects allow software practitioners to program the software places more responsibility and functionality concerns on such architectures. This chapter examines solutions within the context of aspect oriented software development.

#### **2.2 Background**

While object-oriented programming (OOP) is the most common methodology employed today to manage core concerns, it is not sufficient for many crosscutting concerns, especially in complex applications. A typical OOP implementation [15] creates a coupling between the core and crosscutting concerns that is undesirable, since the addition of new crosscutting features and even certain modifications to the existing crosscutting functionality require modifying the relevant core modules. AOP is a new methodology that provides separation of crosscutting concerns by

introducing a new unit of modularization—an *aspect*—that crosscuts other modules. AOP implements crosscutting concerns in aspects instead of fusing them in the core modules. An *aspect weaver*, which is a compiler-like entity, composes the final system by combining the core and crosscutting modules through a process called *weaving*. The result is that AOP modularizes the crosscutting concerns in a clear-cut fashion, yielding a system architecture that is easier to design, implement, and maintain.

In this chapter, we examine the fundamentals of AOP, the problems it addresses. Perhaps the most commonly asked question in today's software engineering is, how much design is too much? Good system architecture considers present and potential future requirements. Failing to take into account the potential future requirements of a crosscutting nature may eventually require changing many parts of the system or perhaps even re-implementing them. On the other hand, including low-probability requirements may lead to an over designed, hard-to-understand, bloated system. There is a demand to create well-designed systems that can meet future needs without compromising quality. Then again, inability to predict the future and time-to-market pressure simply suggests going with what you need today. Further, since requirements are going to change anyway, why bother considering them? The question that pops up is: Is it under design / over design?

### **2.3 Why do we need AOP?**

The usual approach is to build the system, profile it, and retrofit it with optimizations to improve performance. This approach calls for potentially changing many parts of the system using profiling. Further, over time, new bottlenecks may need to be addressed due to changes in usage patterns. The architects of reusable libraries have an even more difficult task because it is a lot harder to imagine all the usage scenarios of a library. Today's fast-changing technology makes it even more difficult since technological changes may make certain design decisions useless. Table 2.1 enumerates the forces on an architect that are at the root of the architect's dilemma.

When software projects turn out to be insufficient for future business requirements, it is common to blame the problem on the design decisions. However, what is often believed to be insufficient design effort or design shortcomings may be simply a limitation of the design methodologies used and the language implementation. With

current design and implementation techniques, there is a limit to what we can do to produce a system that satisfies the current and potential future requirements in a balanced way, and even that limit may not be acceptable when considering the ever-increasing pressure on time-to-market and quality requirements of feature-rich products.

The architect's dilemma, then, is the perennial problem of achieving balance throughout the software process; you are always aiming for that balance, though you know you can never achieve it. One point needs to be made explicitly clear: AOP is not an antidote for bad or insufficient design. In fact, it is very tough to implement crosscutting concerns in a poorly designed core system. There will still be a need to create a solid core architecture using traditional design methodologies, such as OOP. What AOP offers is not a completely new design process, but an additional means that allows the architect to address future potential requirements without breaking the core system architecture, and to spend less time on crosscutting concerns during the initial design phase, since they can be woven into the system as they are required without compromising the original design.

## 2.4 Evolution of programming methodologies

From machine-level languages to procedural programming to OOP, software engineering has come a long way; we now deal with the problems at a much higher level than we did a few decades back. We no longer worry about the machine instructions but rather view a system as a symbiosis of the collaborating objects. However, even with the current methodologies there is a significant gap between knowing the system goals and implementing them. The current methodologies make initial design and implementation complex and evolution hard to manage.

**Table 2.1 Forces behind the architect's dilemma**

Benefits of Under design	Benefits of Over design
Reduced short-term development cost	Better long-term system manageability
Reduced design bloat	Easy to accommodate new requirements
Reduced time-to-market	Improved long-term product quality

This is ironic given the world we live in, which demands a faster implementation cycle and where the only constant is change.

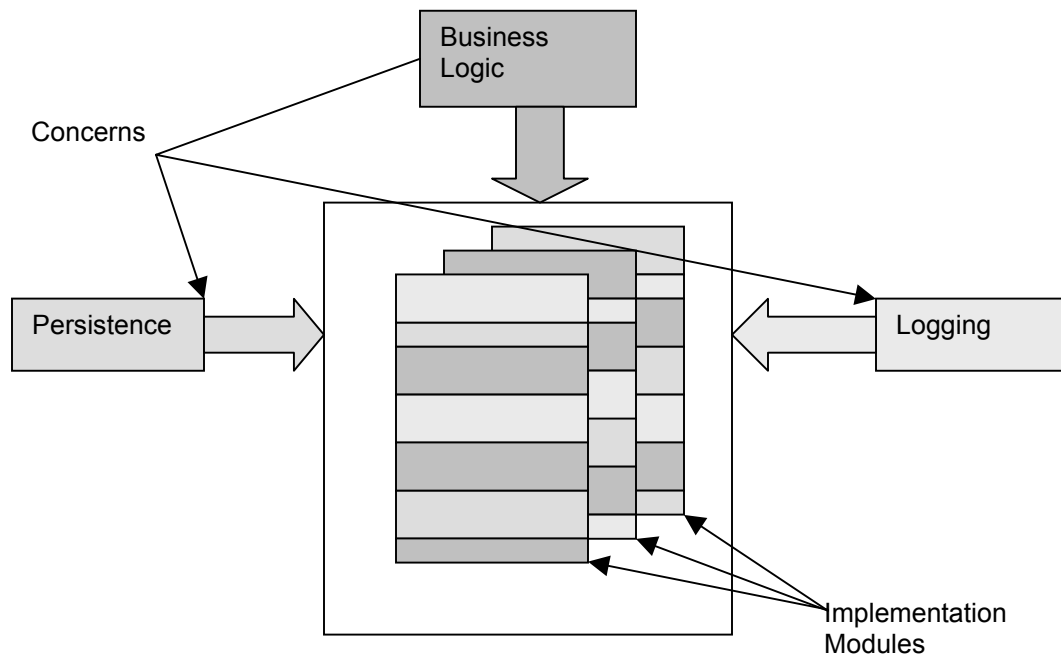
In the evolutionary view of programming methodology, procedural programming introduced functional abstraction, OOP introduced object abstraction, and now AOP introduces concern abstraction [16]. Currently, OOP is the methodology of choice for most new software development projects. OOP's strength lies in modeling common behavior [17]. However, it does not do as good a job in addressing behaviors that span many, often unrelated, modules. AOP fills this void.

## **2.5 Managing system concerns**

A *concern*[18, 19] is a specific requirement or consideration that must be addressed in order to satisfy the overall system goal. A *software system* is the realization of a set of concerns [20]. In addition to system concerns, a software project needs to address process concerns, such as comprehensibility, maintainability, traceability, and ease of evolution.

A concern can be classified into one of two categories: core concerns capture the central functionality of a module, and crosscutting concerns capture system-level, peripheral requirements that cross multiple modules. A typical enterprise application may need to address crosscutting concerns, such as authentication, logging, resource pooling, administration, performance, storage management, data persistence, security, multithread safety, transaction integrity, error checking, and policy enforcement, to name just a few. All of these concerns crosscut several subsystems. For example, the logging concern affects every significant module in the system, the authorization concern affects every module with access control requirements, and the storage-management concern affects every stateful business object. Figure 2.1 shows how these concerns often interact in a typical application. This figure shows how the implementation modules in a system each address both system-level and business concerns. This view portrays a system as a composition of multiple concerns that become tangled together by the current implementation techniques; therefore the independence of concerns cannot be maintained.





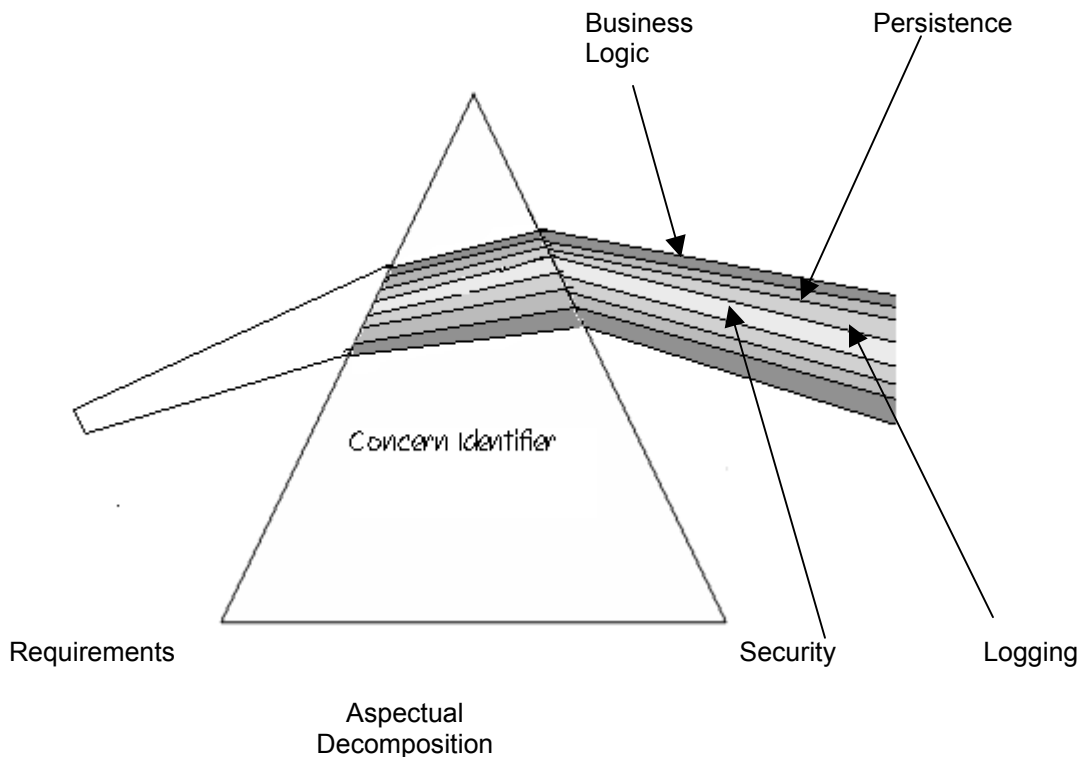
**Figure 2.1** Viewing a system as a composition of multiple concerns. Each implementation module addresses some element from each of the concerns the system  
 (Source: <http://www.javaworld.com>)

## 2.6 Identifying system concerns

Identifying the core and crosscutting concerns of a system focuses on each individual concern separately and reduce the overall complexity of design and implementation. In order to do this, the first step is to decompose the set of requirements by separating them into concerns. Figure 2.2 illustrates the process of decomposing the requirements into a set of concerns. While each requirement initially appears to be a single unit, by applying the concern identification process, we can separate out the individual core and crosscutting concerns that are needed to fulfill the requirement. The significance of this kind of system view is it shows us that each concern in a multidimensional space is mutually independent and therefore can evolve without affecting the rest.

For example, changing the persistence requirement [21] from a relational database to an object database [22] should not affect the business logic or security requirements. Separating and identifying the concerns in a system is an important exercise in the development of a software system, regardless of the methodology used. Once we have done so, we can address each concern independently,

making the design task more manageable. The problem arises when we implement the concerns into modules. Ideally, the implementation will preserve the independence of the concerns, but this doesn't always happen.



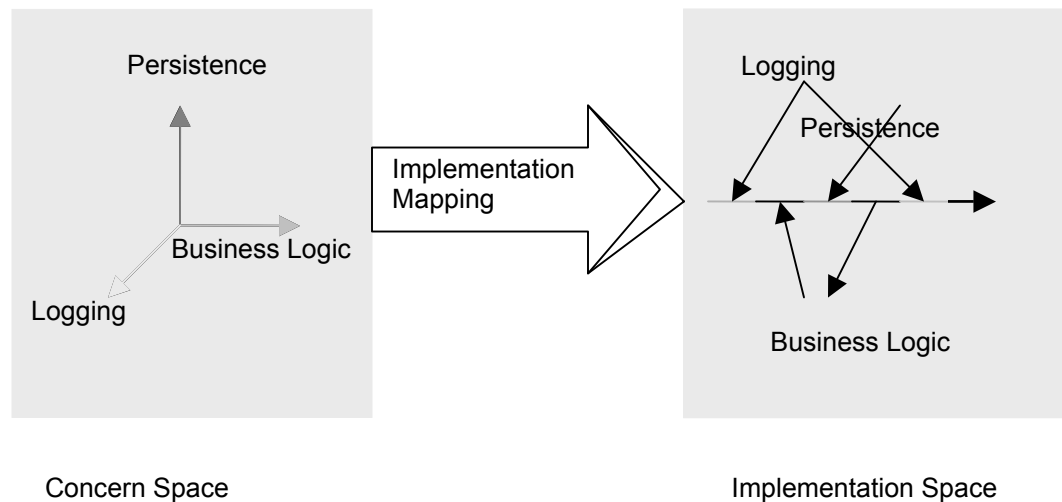
**Figure 2.2** Concern decomposition. While the requirement initially appears as a single requirement, after passing it through the concern identification mechanism, you can see the constituent concerns separated out.  
(Source: <http://www.javaworld.com>)

## 2.7 A one-dimensional solution

Crosscutting concerns, by their nature, span many modules, and current implementation techniques tend to mix them into the individual core modules. To illustrate this, figure 2.3 shows a three-dimensional concern space, whereas the code that implements the concerns is a continuous flow of calls, and in that sense is one-dimensional. Such a mismatch results in an awkward mapping of the concerns to the implementation.

Since the implementation space is one-dimensional, its main focus is usually the implementation of the core concern, and the implementation of the crosscutting concerns is mixed in with it. While we may naturally separate the individual requirements into mutually independent concerns during the design phase, current

programming methodologies do not allow us to retain the separation in the implementation phase.



**Figure 2.3** Mapping the N-dimensional concern space using a one-dimensional language. The orthogonality of concerns in the concern space is lost when it is mapped to one-dimensional implementation space.

(Source: <http://www.javaworld.com>)

## 2.8 Modularizing

In software design, the best way of simplifying a complex system is to identify the concerns and then to modularize them. In fact, the OOP methodology was developed as a response to the need to modularize the concerns of a software system. The reality is, though, that although OOP is good at modularizing core concerns, it falls short when it comes to modularizing the crosscutting concerns. The AOP methodology was developed to address that shortfall. In AOP, the crosscutting concerns are modularized by identifying a clear role for each one in the system, implementing each role in its own module, and loosely coupling each module to only a limited number of other modules.

In OOP, the core modules can be loosely coupled through interfaces, but there is no easy way of doing the same for crosscutting concerns. This is because a concern is implemented in two parts: the server-side piece and the client-side piece. (The terms *server* and *client* are used here in the classic OOP sense to mean the objects that are providing a certain set of services and the objects using those services.)

OOP modularizes the server part quite well in classes and interfaces [23]. However, when the concern is of a crosscutting nature, the client part, consisting of the requests to the server, is spread over all of the clients.

As an example, let's look at a typical implementation of a crosscutting concern in OOP: an authorization module that provides its services through an abstract interface. The use of an interface loosens the coupling between the clients and the implementations of the interface. Clients who use the authorization services through the interface are for the most part oblivious to the exact implementation they are using. Any changes to the implementation they are using will not require any changes to the clients themselves. Likewise, replacing one authorization implementation with another is just a matter of instantiating the right kind of implementation. The result is that one authorization implementation can be switched with another with little or no change to the individual client modules. This configuration, however, still requires that each client have the embedded code to call the API. Such calls will need to be in all the modules requiring authorization and will be mixed in with their core logic.

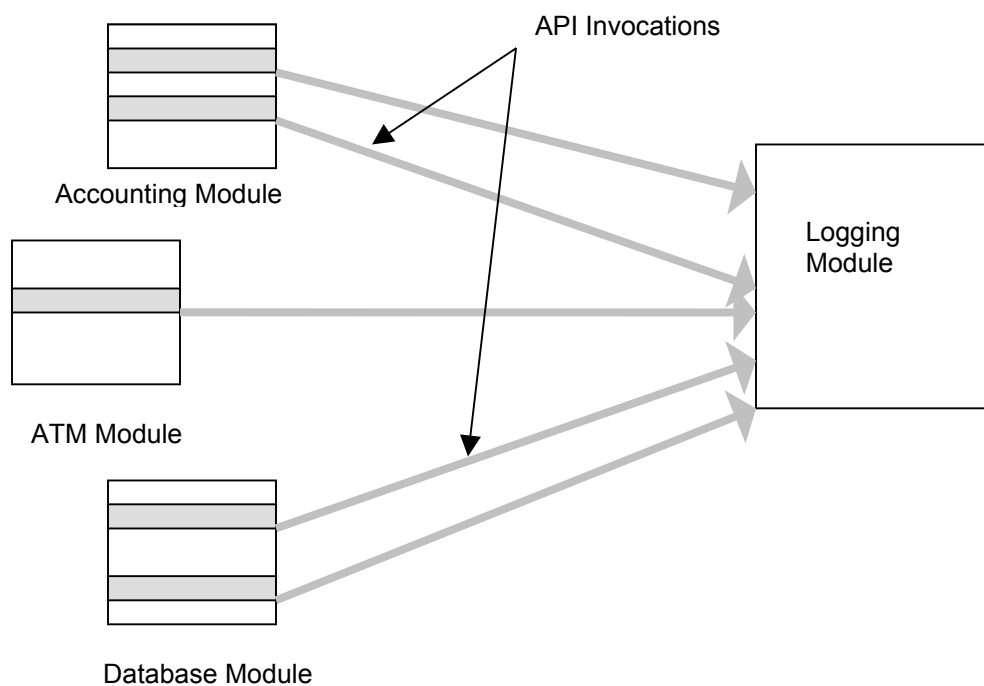
Figure 2.4 shows how a banking system would implement logging using conventional techniques. Even when using a well-designed logging module that offers an abstract API and hides the details of formatting and streaming the log messages, each client—the accounting module, the ATM module, and the database module—still needs the code to invoke the logging API. The overall effect is an undesired tangling between all the modules needing logging and the logging module itself. Each coupling is represented in the figure by a gray arrow.

This is where AOP comes into the picture. Using AOP, none of the core modules will contain calls to logging services—they don't even need to be aware of the presence of logging in the system. Figure 2.5 shows the AOP implementation of the same logging functionality shown in figure 2.4. The logging logic now resides inside the logging module and logging aspect; clients no longer contain any code for logging. The crosscutting logging requirements are now mapped directly to just one module—the logging aspect. With such modularization, any changes to the crosscutting logging requirements affect only the logging aspect, isolating the clients completely.

Modularizing crosscutting concerns is so important that there are several techniques to achieve it. For example, the Enterprise JavaBeans (EJB)

architecture [24, 25,26] simplifies creating distributed, server-side applications, and handles the crosscutting concerns, such as security, administration, performance, and container-managed persistence. To implement the crosscutting concern of persistence in EJB the bean developers focus on the business logic, while the deployment developers focus on the deployment issues, such as mapping the bean data to the database. The bean developers, for the most part, are oblivious to the storage issues. The EJB framework achieves the separation of the persistence concern from the business logic through use of a deployment descriptor—a file in XML format—that specifies how the bean’s fields map to database columns.

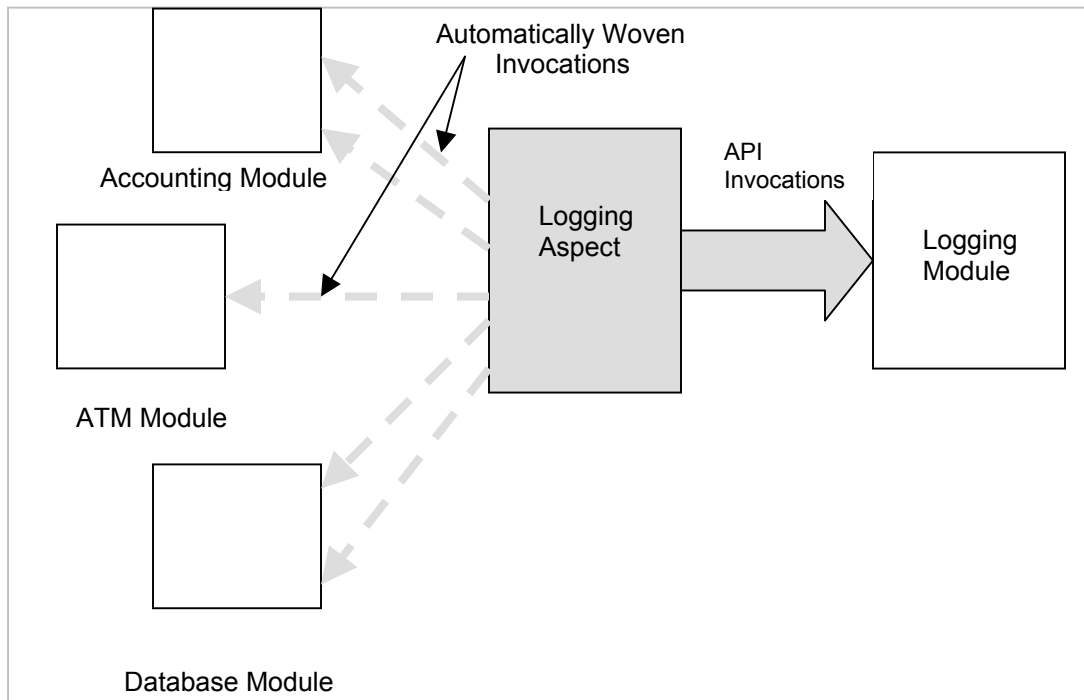
Similarly, the framework separates other crosscutting concerns such as authentication and transaction management by managing their specifications in the deployment descriptor.



**Figure 2.4** Implementation of a logging concern using conventional techniques: The logging module provides the API for logging. However, the client modules—Accounting, ATM, and Database—each still need to embed the code to invoke the logging API.  
 (Source: <http://www.javaworld.com>)

Another technique for handling crosscutting concerns is to use dynamic proxies, which provide language support for modularizing the proxy design pattern.

However, this feature offers a reasonable solution to modularize crosscutting



**Figure 2.5** Implementation of a logging concern using AOP techniques: The logging aspect defines the interception points needing logging and invokes the logging API upon the execution of those points. The client modules no longer contain any logging-related code. (Source: <http://www.javaworld.com>)

concerns, as long as they are simple. The very existence of frameworks like EJB and language features like dynamic proxies confirms the need for AOP. The advantage of AOP is that it is not limited to a single domain in the way that EJB is limited to distributed server-side computing [27], and that AOP code is simpler than that of dynamic proxies when they are used alone.

### 2.8.1 Implementing crosscutting concerns in non modularized systems

The implementation of crosscutting concerns often becomes complicated by tangling it with the implementation of core concerns. A real world system would consist of many classes. Many would address the peripheral concerns such as authorization, authentication, multithread safety, contract validation, cache management and logging. Therefore, while we may have had a good understanding of different crosscutting concerns and their separation during the design phase, the implementation [28] paid almost no attention to preserving the

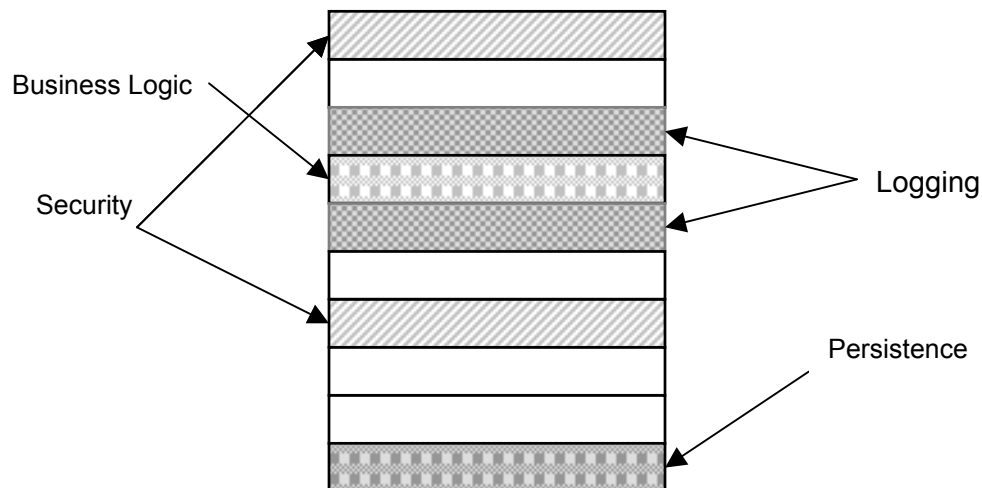
separation.

## 2.8.2 Symptoms of nonmodularization

The symptoms of non modularization can be modularized into into two categories: *code tangling* and *code scattering*.

### **Code tangling**

Code tangling is caused when a module is implemented that handles multiple concerns simultaneously. A developer often considers concerns such as business logic, performance, synchronization, logging, security, and so forth while implementing a module. This leads to the simultaneous presence of elements from each concern's implementation and results in code tangling. Figure 2.6 illustrates code tangling in a module.

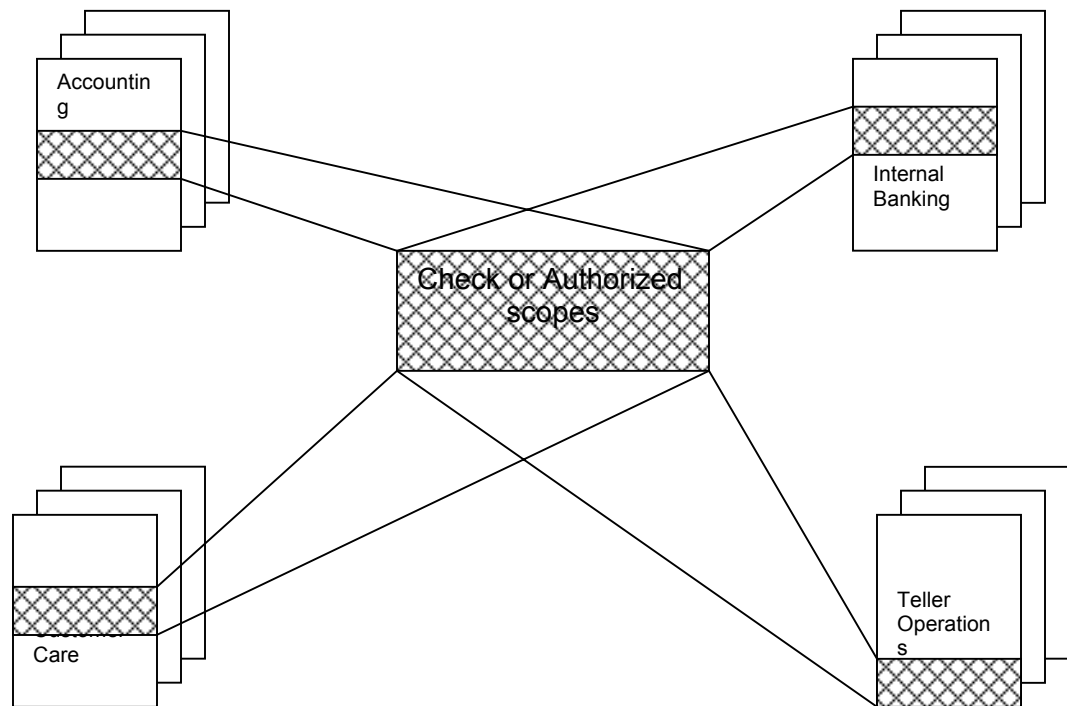


**Figure 2.6** Code tangling caused by multiple simultaneous implementations of various concerns. The figure shows how one module manages a part of multiple concerns. (Source: <http://www.javaworld.com>)

### **Code scattering**

Code scattering is caused when a single issue is implemented in multiple modules. Since crosscutting concerns, by definition, are spread over many modules, related implementations are also scattered over all those modules. For example, in a system using a database, performance concerns may affect all the modules accessing the database.

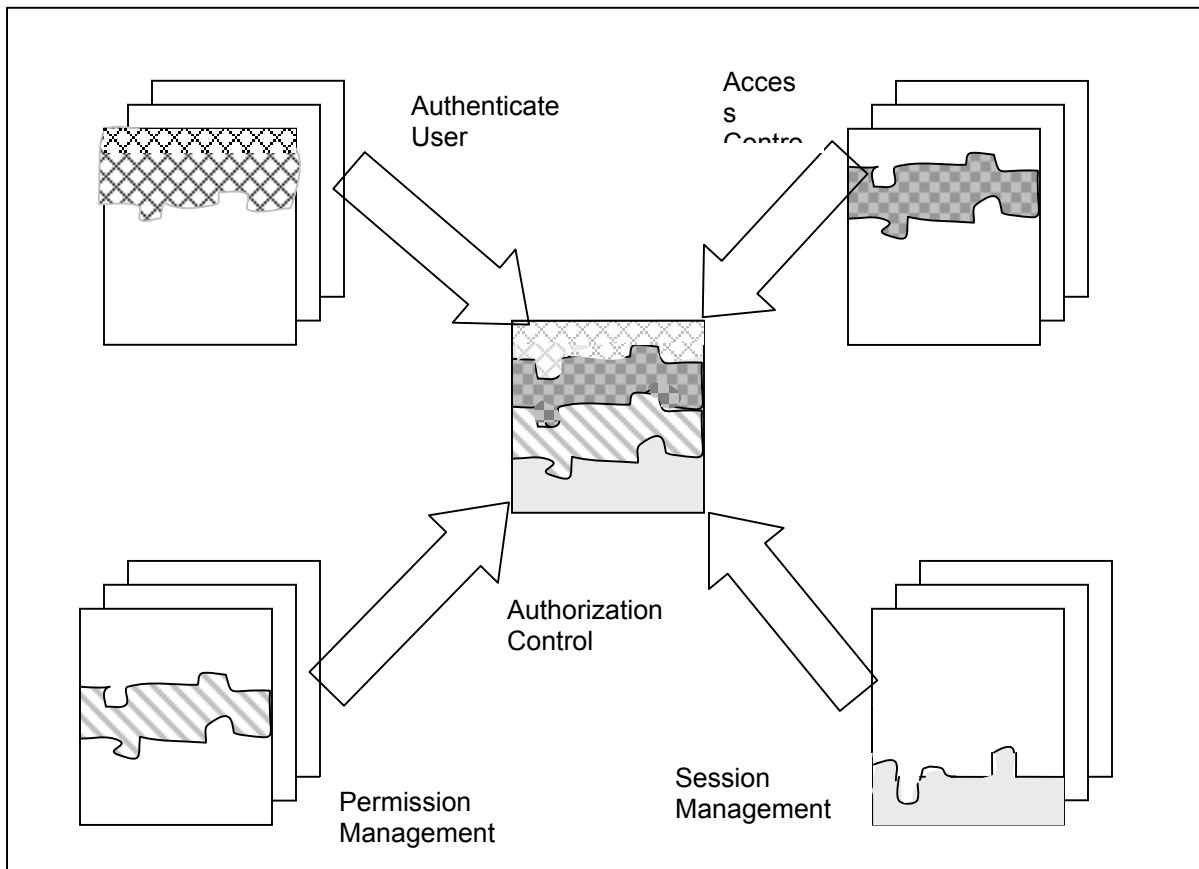
Code scattering can be classified into two distinct categories: duplicated code blocks and complementary code blocks. The first kind is characterized by repeated code of a nearly identical nature. For example, resource pooling will typically involve adding nearly identical code to multiple modules to fetch a resource from a pool and return the resource back to the pool. Figure 2.7 illustrates the scattered duplicated code blocks.



**Figure 2.7** Code scattering caused by the need to place nearly identical code blocks in multiple modules to implement a functionality. In this banking system, many modules in the system must embed the code to ensure that only authorized users access the services. (Source: <http://www.javaworld.com>)

The second kind of code scattering happens when several modules implement complementary parts of the concern. All these pieces must be carved to fit together perfectly to implement the functionality, as shown in figure 2.8 In figure 2.8, multiple modules include code for authentication logic and access checking; they must work together to correctly implement the authorization. For example, before you can check the credentials of a user (access control), you must have verified that user's authenticity (authentication).





**Figure 2.8** Code scattering caused by the need to place complementary code blocks in multiple modules to implement a functionality  
 (Source: <http://www.javaworld.com>)

### 2.8.3 Implications of non modularization

Code tangling and code scattering together impact software design and development in many ways: poor traceability, lower productivity, lower code reuse, poor quality, and harder evolution.

*Poor traceability*—Simultaneous implementation of several concerns obscures the mapping of the concern to its implementation. This causes difficulty in tracing requirements to their implementation, and vice versa. This is evident when doing the tracing of implementation of an authentication concern, examination of all modules is required.

*Lower productivity*—Simultaneous implementation of multiple concerns also shifts the focus from the main concern to the peripheral concerns. The lack of focus then leads to lower productivity as developers are sidetracked from their primary objective in order to handle the crosscutting concerns.

Further, since different concern implementations may need different skill sets, either several people will have to collaborate on the implementation of a module or the developer implementing the module will need knowledge of each domain. The more concerns you implement together, the lower your probability of focusing on any one thing.

*Lower code reuse*—If a module is implementing multiple concerns, other systems requiring similar functionality may not be able to readily use the module due to a different set of concerns they might need to implement.

Consider a database access module. One project may need one form of authentication to access the database, another project may need a different form, and still another may need no authentication at all. The variation of crosscutting requirements may render an otherwise useful module unusable.

*Poor quality*—Code tangling makes it more difficult to examine code and spot potential problems, and performing code reviews of such implementations is harder. For example, reviewing the code of a module that implements multiple concerns will require the participation of an expert in each of the concerns. Often not all of them are available at the same time, and the ones who are may not pay sufficient attention to the concerns that are outside their area of expertise.

*Difficult evolution*—An incomplete perspective and limited resources often result in a design that addresses only current concerns. When future requirements arise, they often require reworking the implementation. Because implementation is not modularized, this may mean modifying many modules. Modifying each subsystem for such changes can lead to inconsistencies. It also requires spending considerable testing effort to ensure that this implementation change does not introduce regression bugs.

All of these problems lead to a search for better approaches to architecture, design, and implementation. Aspect-oriented programming offers one viable solution.

## **2.9 Aspect Oriented Programming (AOP) appears on the Scene**

AOP builds on top of existing methodologies such as OOP and procedural programming, augmenting them with concepts and constructs in order to modularize crosscutting concerns. With AOP, the core concerns are implemented using the chosen base methodology. If OOP is the base methodology, classes are implemented as core concerns. The aspects in the system encapsulate the

crosscutting concerns; they stipulate how the different modules in the system need to be woven together to form the final system.

The most fundamental way that AOP differs from OOP in managing crosscutting concerns is that in AOP, the implementation of each concern is oblivious to the crosscutting behavior being introduced into it. For example, a business logic module is unaware that its operations are being logged or authorized. As a result, the implementation of each individual concern evolves independently.

## **2.10 Background of AOP**

For years now, many theorists have agreed that the best way to create manageable systems is to identify and separate the system concerns. This general topic is referred to as “separation of concerns” (SOC). In a 1972 paper, David Parnas proposed that the best way to achieve SOC is through modularization—a process of creating modules that hide their decisions from each other. In the ensuing years, researchers have been studying various ways to manage concerns. OOP provided a powerful way to separate core concerns. However, it left something to be desired when it came to crosscutting concerns. Several methodologies—generative programming, meta-programming, reflective programming, compositional filtering, adaptive programming, subject-oriented programming, aspect-oriented programming, and intentional programming—have emerged as possible approaches to modularizing crosscutting concerns. AOP is the most popular among these.

Much of the early work that led to AOP today was done in universities all over the world. Cristina Lopes and Gregor Kiczales of the Palo Alto Research Center (PARC), a subsidiary of Xerox Corporation, were among the early contributors to AOP. Gregor coined the term “AOP” in 1996. He led the team at Xerox that created AspectJ, one of the first practical implementations of AOP, in the late 1990s.

Xerox recently transferred the AspectJ project to the open source community at eclipse.org, which will continue to improve and support the project. AspectJ is an implementation of AOP, just as Java and SmallTalk are implementations of OOP. AspectJ is based on Java, but there are implementations of AOP for other languages, ranging from AspectC for C to Pythius for Python, that apply the same concepts that are in AspectJ to other languages. Further, there are a few Java implementations of AOP other than AspectJ, such as Java Aspect Component

(JAC) from AOPSYS. These implementations differ in the ways they express the crosscutting concerns and translate those concerns to form the final system.

## **2.11 The AOP methodology**

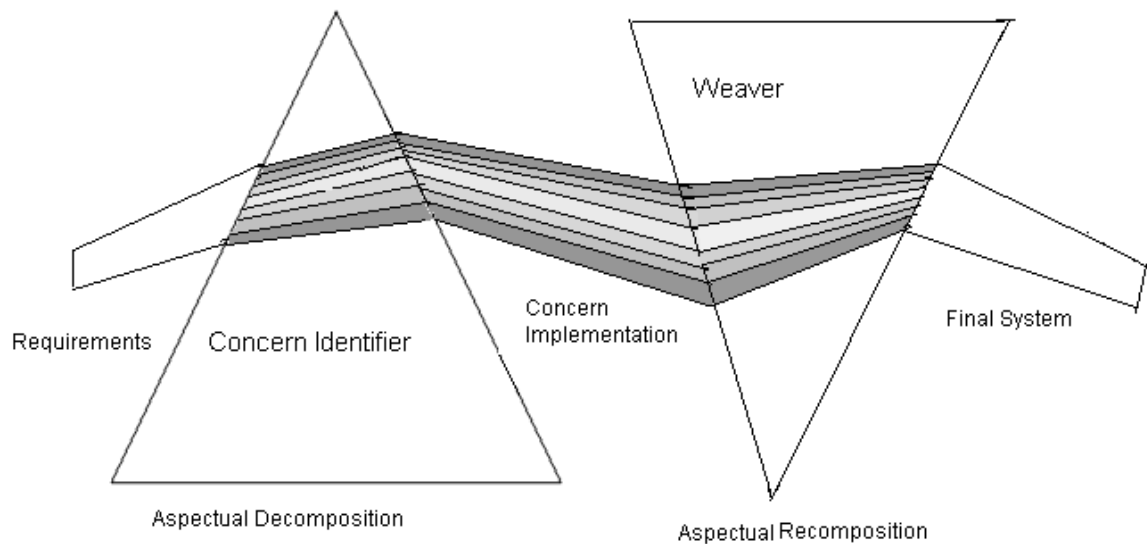
In many ways, developing a system using AOP is similar to developing a system using other methodologies: identify the concerns, implement them, and form the final system by combining them. The AOP research community typically defines these three steps in the following way:

1 *Aspectual decomposition*—In this step, the requirements are decomposed to identify crosscutting and core concerns. This step separates core-level concerns from crosscutting, system-level concerns.

2 *Concern implementation*—In this step, each concern is implemented *independently*.

3 *Aspectual recomposition*—In this step, you specify the recomposition rules by creating modularization units, or *aspects*. The actual process of recomposition, also known as weaving or integrating, uses this information to compose the final system.

The fundamental change that AOP brings is the preservation of the mutual independence of the individual concerns when they are implemented. The implementation can then be easily mapped back to the corresponding concerns, resulting in a system that is simpler to understand, easier to implement, and more adaptable to change. In the AOP development stages as shown in Figure 2.9, in the first stage, decompose the system requirements into individual concerns and implement them independently. The weaver takes these implementations and combines them together to form the final system.



**Figure 2.9** AOP development stages. In the first stage, you decompose the system requirements into individual concerns and implement them independently. The weaver takes these implementations and combines them together to form the final system. (Source: <http://www.javaworld.com>)

## 2.12 Anatomy of an AOP language

The AOP methodology is just that—a methodology. In order to be of any use in the real world, it must be implemented, or realized. As with any methodology, it can be implemented in various ways. For example, one realization of the OOP methodology specification consists of the Java language and tools such as the compiler. In a similar manner, each realization of AOP involves specifying a language and offering tools to work with that language. Like any other programming methodology, an AOP implementation consists of two parts:

The language specification describes the language constructs and syntax that will be used to realize both the logic of the core concerns and the weaving of the crosscutting concerns.

The language implementation verifies the code's adherence to the language specification and translates the code into an executable form. This is commonly accomplished by a compiler or an interpreter.

## 2.13 Benefits of AOP

The benefits of AOP are:

*Cleaner responsibilities of the individual module*—AOP allows a module to take responsibility only for its core concern; a module is no longer liable for other crosscutting concerns. For example, a module accessing a database is no longer responsible for pooling database connections as well. This results in cleaner assignments of responsibilities, leading to improved traceability.

*Higher modularization*—AOP provides a mechanism to address each concern separately with minimal coupling. This results in modularized implementation even in the presence of crosscutting concerns. Such implementation results in a system with much less duplicated code. Because the implementation of each concern is separate, it also helps avoid code clutter. Modularized implementation results in an easier-to-understand and easier-to-maintain system.

*Easier system evolution*—AOP modularizes the individual aspects and makes core modules oblivious to the aspects. Adding a new functionality is now a matter of including a new aspect and requires no change to the core modules. Further, when we add a new core module to the system, the existing aspects crosscut it, helping to create a coherent evolution. The overall effect is a faster response to new requirements.

*Late binding of design decisions*—Architects in general are faced with underdesign / overdesign issues. With AOP, the architect can delay making design decisions for future requirements because it is possible to implement those as separate aspects. Architects can now focus on the current core requirements of the system. New requirements of a crosscutting nature can be handled by creating new aspects.

*More code reuse*—The key to greater code reuse is a more loosely coupled implementation. Because AOP implements each aspect as a separate module, each module is more loosely coupled than equivalent conventional implementations. In particular, core modules aren't aware of each other—only the weaving rule specification modules are aware of any coupling. By simply changing the weaving specification instead of multiple core modules, you can change the system configuration. For example, a database module can be used with a different logging implementation without change to either of the modules.

*Improved time-to-market*—Late binding of design decisions allows a much faster design cycle. Cleaner separation of responsibilities allows better matching of the module to the developer's skills, leading to improved productivity. More code reuse leads to reduced development time. Easier evolution allows a quicker response to

new requirements. All of these lead to systems that are faster to develop and deploy.

*Reduced costs of feature implementation*—By avoiding the cost of modifying many modules to implement a crosscutting concern, AOP makes it cheaper to implement the crosscutting feature. By allowing each implementer to focus more on the concern of the module and make the most of his or her expertise, the cost of the core requirement's implementation is also reduced. The end effect is a cheaper overall feature implementation.

## **2.14 Summary**

The most fundamental principle in software engineering is that the separation of concerns leads to a system that is simpler to understand and easier to maintain.

Various methodologies and frameworks exist to support this principle in some form. However, for crosscutting concerns, OOP forces the core modules to embed the crosscutting concern's logic. While the crosscutting concerns themselves are independent of each other, the use of OOP leads to an implementation that no longer preserves the independence in implementation.

The current most common response to the difficulties of crosscutting concerns is to develop new domain-specific solutions, such as the EJB specification for enterprise server-side development. While these solutions do modularize certain crosscutting concerns within the specific domain, their usefulness is restricted to that domain. The cost of using these pre-wired solutions is reflected in the time and effort that is required to learn each new technology that, in the end, is useful only within its own limited scope.

Aspect-oriented programming will change this by modularizing crosscutting concerns in a generic and methodical fashion. With AOP, crosscutting concerns are modularized by encapsulating them in a new unit called an aspect. Core concerns no longer embed the crosscutting concern's logic, and all the associated complexity of the crosscutting concerns is isolated into the aspects. AOP marks the beginning of new ways of dealing with a software system by viewing it as a composition of mutually independent concerns. By building on the top of existing programming methodologies, it preserves the investment in knowledge gained over the last few decades. In the short-term future, it is highly likely that we will see AOP-based solutions providing powerful alternatives to domain-specific solutions.

AOP, being a brand-new methodology, is not the easiest to understand. The learning curve involved is similar to transitioning from procedural to OOP. The payoff, however, is tremendous. Most developers who are exposed to AOP are amazed by its power once they get over the initial learning curve.



## CHAPTER 3

### Related Work

#### 3.1 Overview

This chapter outlines important contributions already made in the area of aspect oriented software development and shows the multitude of research areas that have been followed. Since the focus of the work presented in this thesis concerns the development of an aspect oriented design language, this chapter focuses primarily on completed and ongoing work in aspect oriented design language and enabling programming technologies. A number of research projects that design and develop aspect oriented design languages and specifications are studied. This chapter concludes with an overview of current work on aspect oriented applications and design language specifications.

Since the beginning of research into aspect oriented design, many research groups have tried to develop aspect oriented design languages. The diversity of research groups has led to a large variety of different approaches[29, 30, 31].

Recently there has been growing interest in propagating the aspect paradigm to the activities in the earlier phases of the software development lifecycle. A number of approaches to aspect-oriented design have been proposed e.g. [5,32]. One approach proposes an extension to UML to support aspects. Composition Patterns is another approach to handle crosscutting concerns at the design level [5, 32].

The Unified Modeling Language (UML) [2, 3] is an object-oriented design notation that provides basic building blocks to model software-intensive systems, such as *abstractions* that represent structure and behavior of a system, *relationships* that state how the abstractions relate to each other, and *diagrams* that show interesting excerpts of a set of abstractions and relationships. The most important characteristic of UML in respect to the issue tackled in this work is its extension mechanisms [33]. UML's extension mechanisms provide standardized means to

extend existing UML building blocks with new properties, called *tagged values*, or with new semantic, called *constraints*. Besides the alteration of existing building blocks, the UML may be extended with completely new building blocks that are derived from existing ones. The new building blocks, called *stereotypes*, have the same structure (attributes, associations, operations) as the base building block they are derived from. One such approach is referred to as the “*aspect-oriented design model*”, or *AODM* for short [33] that extends the Unified Modeling Language with the aspect-oriented design concepts as specified in *AspectJ* [34, 35, 36].

Design Patterns became popular after the “Gang of Four” book of the same name (Gamma et al, 1995) [15, 16, 17, 20, 25, 25, 26, 37] showed how design patterns could be used in object-oriented software development. Design patterns are a method of encapsulating the knowledge of experienced software designers in a human readable and understandable form. They provide an effective means for describing key aspects of a successful solution to a design problem and the benefits and tradeoffs related to using that solution. Using design patterns help produce good design, which helps produce good software. The ability to work with design patterns in conjunction with Unified Modeling Language (UML) is a major benefit. UML is now a standard for OO modeling and is an industry standard now. Compatibility with UML makes design patterns more palatable for many programmers and designers as they are already familiar with UML. For the implementation of design patterns, the design policy to consider the patterns as concerns is important. At the same time, it is to be desired that we have effective languages and tools supporting the advanced separation of concerns [20, 38]. The new implementation technologies that support the advanced separation of concerns such as *Hyper/J* [9, 10, 39] and *AspectJ* [34, 35, 36] help with coding this kind of design.

Separation of concerns[18, 40] is a basic engineering principle that is also at the core of object-oriented analysis and design methods in the context of UML. Separation of concerns can provide many benefits: additive, rather than invasive, change; improved comprehension and reduction of complexity; adaptability, customizability, and reuse. With its nine views that can be thought of as projections of a whole multi-dimensional system onto separate plans, UML [3]

provides the designer with an interesting separation of concerns called the 4+1 view model (Design view, Component view, Process view, Deployment view, plus Use Case view). In turn, each of these views has two dimensions, one static and one dynamic. Furthermore the designer can add non-functional information (e.g. persistency requirements) to a model by “*stamping*” model elements, for instance with design pattern occurrences [33], stereotypes or tag values. It is appealing to think of many concerns as being independent or “orthogonal”, but this is rarely a case in practice. It is essential to be able to support interacting concerns, while still achieving useful separation. An aspect-oriented approach to design can help to express these concerns explicitly. Frameworks that provide methodological support for building and manipulating UML models with aspects have been proposed. One such framework is the *UMLAUT* [11] (*UML All pUrpose Transformer*) framework which allows the engineer to program the “weaving” of the aspects at the level of the UML meta-model.

### **3.2 UMLAUT: Weaving UML Designs**

UMLAUT [41] is a framework dedicated to the manipulation of UML models where the weaving process can be adapted and extended: new weavers can be constructed simply by changing the weaving rules. The framework takes care of the weaver implementation. In UMLAUT toolbox, a weaving process is implemented as a model transformation process: each weaving step is a transformation step applied to a UML model. Hence the final output is a UML model too (endomorphing transformation). The model transformation engine is itself designed as a configurable and extendible framework.

#### **3.2.1 General Architecture and Core Engine**

UMLAUTs architecture [42] is a three-layered one. The input front end consists of a graphical user interface for interactive editing; another interface deals with importing UML models described in various formats (XMI, Rational Rose TM MDL, Eiffel source, Java source). The middle core engine is made up of the UML meta-model repository and the extendible transformation engine. Finally, the output back end contains various generators (including code generators and an XMI generator). The design concept of UMLAUT is a basic core (the middle layer) that communicates with its surroundings *via hot spots* (*i.e.* interfaces). Functional

modules can be plugged in order to specialize the tool's behavior and to meet specific requirements.

### **3.2.2 The Extendible Transformation Framework**

The transformation engine of UMLAUT is responsible for the weaving process. A weave operation is described as a transformation of an initial model to a final one. A designer specifies the required transformation by explicitly composing a set of operators from the UMLAUT transformation library. Since the transformation engine is an open framework, users may add new operators and extend the existing library to support new weaving operations. The framework is designed to cater for three different kinds of user:

Model designers are interested in performing a set of weaving operations. Their main concern is what transformation operators are available and useful to the model, and how they should be used.

Transformation architects are responsible for defining how to implement a given transformation for a given implementation requirement. They extend the transformation library by adding new transformation operators.

Framework implementor's aim at enhancing the weaver framework to support specific needs of the previous two groups of users.

The transformation framework uses a mix of object-oriented and functional programming paradigms. The object-oriented paradigm allows us to encapsulate our operators as discrete entities, and the functional paradigm provides us with a composition mechanism for these operators. The main architecture consists of three major components:

1. A core structure that provides the logic for operator composition and implicit control flow when a transformation is initiated.
2. A library of iterators for traversing a UML model. An iterator builds a path through a UML model graph so that lazy list operations can be applied.
3. A library of primitive operators for querying, modifying and creating UML model elements.

Each of these components can be augmented and enhanced. In particular, the operator library is likely to be extended by a transformation architect whereas the iterator library will more likely be extended by a framework implementer knowledgeable about the UML meta-model.

For aspect-oriented software development (AOSD) to live up to being a software engineering paradigm, there must be support for the separation of crosscutting concerns across the development lifecycle including traceability from one lifecycle phase to another. Concerns that have a crosscutting impact on software (such as distribution, persistence, etc.) present well-documented difficulties for software development [40]. Since these difficulties are present throughout the development lifecycle, they must be addressed across its entirety. One such work done is the investigation of traceability between one particular AOSD design-level language, *Theme/UML* [5, 42] and one particular AOSD implementation-level language, *AspectJ*. This provides for a means to assess these languages and their incompatibilities, with a view towards eventually developing a standard design language for a broad range of AOSD approaches.

### 3.3 Theme/UML

**Theme/UML** [43, 44, 45] presents an approach to designing systems based on the object-oriented model, but extending this model by adding new decomposition capabilities. The new decomposition capabilities support a way of directly aligning design models with individual requirements. Each model contains its own **theme**, or design of an individual requirement, with concepts from the domain (which may appear in multiple requirements) designed from the perspective of that requirement. Standard UML is used to design the models decomposed in this way. Extensions to the UML are required for the composition of the thematic design models. This is achieved with a *composition relationship*. A composition relationship specifies how models are to be composed by identifying overlapping concepts in different models and specifying how models should be integrated.

It is the nature of crosscutting behaviour that it has an impact on multiple, different elements within software. In order to design such behaviour in standard UML, it is necessary to explicitly specify crosscutting behaviour for each of the particular elements it affects; the designs of crosscutting behaviour cannot be separated and encapsulated with existing UML constructs. These limitations result in design models with scattering and tangling properties comparable to those in code. Theme/UML mitigates these problems by supporting the design of crosscutting concerns as separated, encapsulated design models. Composition of these separate design models is specified with a *composition relationship*, detailing

which elements are to be combined, and how to integrate them. *Merge* is one strategy for integration that includes all the elements from the input design models in the composed design, reconciling conflicts where appropriate.

*The composition pattern* (CP) construct of Theme/UML, based on an extension to UML templates, permits a crosscutting design model to be independent of any base design model, allowing it to be reused. The composition of concrete design models with CPs is based on the semantics of the merge integration strategy.

Template parameters on a composition pattern may represent operations. A key feature of CPs is that they may define *supplementary behaviour* on such template operations. When a template operation with supplementary behaviour is bound to a concrete operation, the supplementary behaviour is merged with the original behaviour of that concrete operation. Any calls to the original operation result in delegation to some ordered combination of the supplementary behaviour and original behaviour, as prescribed within the specification of that CP.

Thematic design with **Theme/UML** has two important implications:

- **Overlapping specifications supported:** Different requirements may exist that have an impact on the same core concepts (for example, objects) of the system. It is this level of overlapping of requirements that is one of the causes of the problems with comprehensibility, extensibility and reuse discussed previously. The Theme/UML model recognises and explicitly caters for overlap in the different design models for each requirement. This is achieved by allowing a separate design model to include the specification of any core concepts only as suits the requirement under design by that design model. Composition capabilities supported by this new approach cater for identifying overlapping concepts, integrating them, and handling any conflicts.
- **Crosscutting specifications supported:** There are also many kinds of requirements that will have an impact across the full design of a software system. For example, a requirement for distributed objects has an impact on a potentially large proportion of the objects of a computer system. Such requirements are referred to as *crosscutting*, since support for such requirements must be included across many different objects in a system. With the approach to decomposition described here, crosscutting

requirements may also be designed separately, with composition capabilities handling their integration with other system objects as appropriate.

Decomposition in this manner also requires corresponding composition support, as object-oriented designs still must be understood together as a complete design. The **Theme/UML** model supports a new kind of design construct, called a *composition relationship* that supports the specification of how design models should be composed. With composition relationships, a designer can:

- **Identify and specify overlaps:** Where decomposition allows overlaps in different design models, corresponding composition capabilities must support the identification of where those overlaps are. In order to integrate separate design models, overlapping design elements (or elements which correspond and therefore should be integrated into a single unit) are specified with composition relationships.
- **Specify how models should be integrated:** Design models may be integrated in different ways, depending on why they were modularised in a particular way. For example, if different design models were designed separately to support different requirements, a composed design where all requirements are to be included might be integrated with a merge strategy - that is, all design elements are relevant to the composed design. Alternatively, if a design model contains the design of a requirement that is a change to a requirement previously designed (for example, a business process has changed), then that design model might replace the previous design. In this case, integration with an override strategy is appropriate, where existing design elements are replaced by new design elements.
- **Specify how conflicts in corresponding elements are reconciled:** For some integration strategies, where some corresponding elements are integrated into a single design element, (merge integration is an example of such a strategy) conflicts between the specifications of those corresponding elements must be reconciled. Composition relationships support the specification of different kinds of reconciliation possibilities - for example, one design model may take precedence over another, or default values should be used.

In addition, for design models that support **crosscutting** requirements, (i.e., those requirements that have an impact on potentially multiple classes in the design), composition of those models with other models is likely to follow a pattern. In other words, a crosscutting requirement has behaviour that will affect multiple classes in different design models in a uniform way. For these kinds of requirements, the **Theme/UML** model defines a mechanism whereby this common way of composing the crosscutting requirement may be defined as a *composition pattern*.

Other possible dimensions to Theme/UML that the authors highlight include:

### **3.3.1 Design Approaches**

One of the primary contributions of Theme/UML is its capabilities relating to decomposition and modularization of UML models. The UML itself provides modularization mechanisms such as packages and subsystems, upon which Theme/UML builds its additional composition capabilities. These are largely related to modularization and generic composition of crosscutting design elements. Theme/UML, provides a more generic approach, including support for both functional separation (like roles) and separation of patterns of crosscutting behaviour.

Collaboration-based design or role modeling is a compositional design approach that concentrates on decomposing designs on the basis of the roles that objects play in particular collaborations. Other approaches to providing design support for crosscutting concerns appear more tied to the AspectJ model of AOSD exclusively. Theme/UML has taken the more independent route in extending the UML to provide just those *constructs* required to support the decomposition (and subsequent composition specification) of design models based on requirements specifications. These requirements may be functional or crosscutting, and new design constructs are focused on how to compose the separate models, not on providing constructs to map to any particular implementation paradigm. This approach makes the model more concern centric, not implementation-paradigm centric.

### **3.3.2 Implementation Approaches**



Other compositional implementation languages and mechanisms exist. Multidimensional separation of concerns [46, 47] with its associated Hyper/J language arose from the subject-oriented programming paradigm as has Theme/UML. Composition filters are a means of intercepting and rerouting messages as they arrive at objects; they can be used as separate crosscutting concerns such as synchronization, and have been described as an aspect-oriented technique. Adaptive programming [48, 49] has also been described as a (special case) aspect-oriented technique [50]. It provides a means to separate the algorithms on data from the structure of that data, allowing the structure of the data to change without requiring related changes to the algorithms. Implicit context is a structuring mechanism and design philosophy concentrating on removing knowledge of the large scale from smaller-scale components: it is related to AOSD. Others have looked to mixins [51] and mixin layers as a means of realizing compositional implementations of collaboration-based designs. Mixin layers are useful for product-line architectures, where features are understood from conception to be optional between different configurations of a product.

### **3.3.3 Lifecycle Impact**

There has been some recognition of the need for separating crosscutting concerns throughout the lifecycle. For example, Griss [5, 52] has proposed a development process for component based product-lines that draws together high-level analysis and design composition techniques with supporting implementation composition techniques. But this process does not advise how to map the differing constructs within the combination of approaches that may be used.

The difficulties reported in re-engineering implementations to take advantage of compositional implementation techniques highlights the importance of separating crosscutting concerns across the lifecycle. Being forced to manually untangle and un-scatter the concerns that were identified was a difficult and error-prone process; if the systems discussed in that work had been designed with their crosscutting concerns separated in the first place, porting the implementations between the different compositional techniques studied could have been more tractable.

### 3.4 Approach To Aspect Oriented Programming And Design

A gap exists between requirements and designs on one hand, and between design and code on the other hand. Aspect oriented programming (AOP) extended to the modeling level where aspects could be explicitly specified during the design process will make it possible to weave these aspects into a final implementation model. Another step could be extension of AOP to the entire software development cycle. Each aspect of design and implementation should be declared during the design phase so that there is a clear traceability from requirements through source code thus using UML as the design language to provide an aspect-oriented design environment.

The separation and encapsulation of crosscutting concerns has been promoted as a means of addressing these difficulties; the standard object-oriented paradigm does not suffice. In order to overcome the difficulties for crosscutting concerns throughout the lifecycle, an approach is required that provides a means to separate and encapsulate both the design and the code of crosscutting behaviour. It is important to work towards a general purpose AOSD design language that meets certain goals [33], including the following:

- *Implementation language independent*: The final form of AOP language may vary from that of any current one. Thus, any design language that simply mimics the constructs of a particular AOP language is liable to fail to achieve implementation language independence.
- *Design-level composability*: Design level composability is a desirable property for two reasons. First designers may check the result of composition prior to implementation, for validation purposes. Second, some projects will continue to require the use of a non-aspect-oriented implementation language because of pragmatic constraints, such as the presence of legacy code written in languages without aspect-oriented extensions; these projects could still benefit from separating the design of crosscutting concerns.

- *Compatibility with existing design approaches:* An AOSD design-level language should also build existing design languages such as UML, to provide a bridge from old techniques to new, so that software engineering realities such as incremental adoption and legacy support are possible.

The construction of complex, evolving software systems requires a high-level design model [53]. This model should be made explicit, particularly the part of it that specifies the principles and guidelines that are to govern the structure of the system. In reality, however, implementators tend to overlook the documented design models and guidelines, causing the implemented system to diverge from its model. Reasoning about a system whose models and implementation diverge is error prone – the knowledge we gain from these models is not of the system itself, but of some fictitious system, the system we intended to build. The system's comprehensibility is impeded, and so using software engineering techniques goes against our intended goals – quality, maintainability and cost minimization. The essence of the problem of implementing higher-level principles and guidelines lies in their globality. These principles cannot be localized in a single module, they must be observed everywhere in the system, which means that they crosscut the system's architecture.

### **3.5 Impact Of Requirements Engineering On AOSD**

The identification of the mapping and influence of a requirement level aspect promotes traceability of broadly scoped requirements and constraints throughout system development, maintenance and evolution [54]. The improved modularization and traceability obtained through early separation of crosscutting concerns can play a central role in building systems resilient to unanticipated changes hence meeting the adaptability needs of volatile domains such as banking, telecommunications and e-commerce. One such generic model is the *AORE (Aspect Oriented Requirements Engineering)* [55, 57] model and its concrete realization with viewpoints [56] and XML. The focus of this model is on modularization and composition of requirements level concerns that cut across other requirements. These crosscutting concerns are responsible for producing

tangled representations that are difficult to understand and maintain. Examples of such concerns [58, 59] at the requirements level are compatibility, availability and security requirements that cannot be encapsulated by a use case and are typically spread across several of them [54].

### **3.6 Aspect Oriented Requirements Engineering (AORE)**

Modern systems have to run in highly volatile environments where the business rules change rapidly. Therefore, systems must be easy to adapt and evolve. If not handled properly, crosscutting concerns inhibit adaptability. The influence of an aspectual requirement and the constraints it imposes on specific requirements within the viewpoints affected by the aspect cannot be determined.

It involves identifying and specifying both *concerns* and stakeholders' requirements. The order in which the specification of *concerns* and stakeholders' requirements is accomplished is dependant on the dynamics of the interaction between requirements engineers and the stakeholders.

Once the coarse-grained relationships between *concerns* and stakeholders' requirements have been established and the candidate aspects identified, the next step is to define detailed composition rules. These rules operate at the granularity of individual requirements and not just the modules encapsulating them. Consequently, it is possible to specify how an aspectual requirement influences or constrains the behaviour of a set of non-aspectual requirements in various modules [60]. At the same time, if desired, aspectual trade-offs can be observed at a finer granularity. This alleviates the need for unnecessary negotiations among stakeholders for cases where there might be an apparent trade-off between two (or more) aspects but in fact different, isolated requirements are being influenced by them. It also facilitates identification of individual, conflicting aspectual requirements with respect to which negotiations must be carried out and trade-offs established.

After composing the candidate aspects and stakeholders' requirements using the composition rules, identification and resolution of conflicts among the candidate aspects is carried out.

Conflict resolution might lead to a revision of the requirements specification (stakeholders' requirements, aspectual requirements or composition rules). If this happens, then the requirements are recomposed and any further conflicts arising

are resolved. The cycle is repeated until all conflicts have been resolved through effective negotiations. The last activity in the model is identification of the dimensions of an aspect. Aspects at this early stage that can have an impact can be described in terms of two dimensions:

- *Mapping*: an aspect might map onto a system feature / function (e.g. a simple method, object or component), decision (e.g. a decision for architecture choice) and design (and hence implementation) aspect (e.g. response time). Accordingly, the aspects at this stage are called the RE stage *candidate aspects* as, despite their crosscutting nature at this stage, they might not directly map onto an aspect at later stages.
- *Influence*: an aspect might influence different points in a development cycle, e.g. availability influences the system architecture while response time influences both architecture and detailed design.

The generic AORE model and its concrete realization with viewpoints and XML aims as a stepping-stone towards two goals:

1. Providing improved support for separation of crosscutting functional and non-functional properties during requirements engineering hence offering a better means to identify and manage conflicts arising due to tangled representations;
2. Identifying the mapping and influence of requirements level aspects on artifacts at later development stages hence establishing critical trade-offs before the architecture is derived.

With increasing support for aspects at the design and implementation level, the inclusion of aspects as fundamental modeling primitives at the requirements level and identification of their mappings also helps to ensure homogeneity in an aspect oriented software development project.

### **3.7 Software Architecture View Of Aspects**

As in software architectures, which emphasize relationships between components constituting the software, the relationships among aspects of the system need to be made explicit. This is generally difficult because it cannot be assumed that aspects are always orthogonal [61]. For example, an aspect for treating overflow of data values, and another for encoding values to increase security can both involve the same methods or fields of an underlying system, and may even have overlap in the modifications applied. Such overlap between different aspects introduces a

new type of problem, not seen in conventional languages, where it is clear to which module each language segment belongs. It is also a major source of complexity when composing and maintaining the aspects.

To alleviate the above problems, a conceptual model called the *aspect architecture* [61] was proposed to provide an aspect-oriented perspective on software architecture. Being a conceptual model, it also outlines an instantiation for UML, corresponding roughly to *Theme/UML*. However, *Theme/UML*, like other UML-based aspect-oriented design approaches, does not provide architectural support, other than that of standard UML, for aspects and their interactions.

Another, main aspect-oriented development approach with explicitly defined aspects relationships is *Aspect-Oriented Component Engineering (AOCE)* [62, 63] which supports aspect-orientation throughout the life-cycle of specification, design, implementation and deployment in the software component domain.

### **3.8 Aspect Oriented Software Development Design Language**

AspectJ [6, 7, 8] is a popular and well established AOP language that provides support for specifying and composing crosscutting code into a core system. It supports the AOP paradigm by providing a special unit, called “aspect”, which encapsulates crosscutting code. Other compositional implementation languages and mechanisms also exist [9, 10]. At the design level, an AOSD design language with extensions to UML [5, 11, 54 and 61] in its capabilities relating to decomposition and modularization is required that would map to a particular AOSD implementation. Further, a standard AOSD design language must be capable of supporting many of these aspect programming languages. A graphical notation helps developers to design and comprehend aspect-oriented programs. Further, it would facilitate the perception of aspect-orientation. A design notation helps developers to assess the crosscutting effects of aspects on their base classes. Its application carries over the advantages of aspect-orientation to the design level and facilitates adaption and reuse of existing design constructs.

Although a lot has been done to study the aspect oriented design approach in enterprise systems for architecture and its implementation, work on a general-purpose design language for aspect-oriented software development is attracting a lot of attention. The development of aspect oriented requirements gathering approach, design notation and environment for development of enterprise systems needs to be further refined in the context of software applications and industry. The thesis work encompasses developing a standard and general purpose AOSD design language with existing UML features and extensions to map AOSD design notations to AOP languages and AOP to legacy code / OO code and design patterns.

### **3.9 Summary**

This chapter has introduced selected work and developments across the broad spectrum of aspect oriented software development research. It provides a comprehensive overview of the state-of-the-art in the field and exhibits a number of open problems that drive continuous research into aspect design. The various projects described throughout this chapter introduce the relevant features of aspect oriented design languages.

At the core of the chapter is the division of design language architecture into *implementation dependent* and *implementation independent* approaches. Although their fundamental goals are identical and the timescale when they have emerged is related, the underlying architectures are inherently different. While the former is tied to a particular implementation of AOP language, the latter with its graphical notation initially tries to maintain a general interface to implement as many aspect oriented programming languages.

A graphical notation helps developers to design and comprehend aspect-oriented programs. Further, it would facilitate the perception of aspect-orientation. A design notation helps developers to assess the crosscutting effects of aspects on their base classes. Its application carries over the advantages of aspect-orientation to the design level and facilitates adaption and reuse of existing design constructs. The following chapter discusses the whole gamut of the design requirements indicated in this chapter and attempts to classify the variety of design constructs into individual and common functionalities.

## **CHAPTER 4**

### **Design Language Requirements**

#### **4.1 Overview**

This chapter discusses the requirements for aspect oriented design language in general and derives the specific requirements for the AOSDDL (Aspect Oriented Software Development Design Language) design language architecture that is proposed within this work.

Most common requirements of a design language have already been mentioned in the last chapter. It has become apparent that design language research deals largely with trade-offs. For example, many of the aspect oriented design systems introduced in chapter 3 trade-off implementation dependency for wide tool support or limited support with general purpose flexibility.

Research into aspect oriented design languages so far has shown that no single solution will meet all possible requirements of aspect oriented software development, and thus, multiple systems for domains with different demands must be able to co-exist and interoperate. The challenge in designing aspect oriented solutions therefore is to draw the optimal line between trade-offs depending on the requirements at hand. For this, it is crucial to understand fully the requirements of a given domain.

#### **4.2 Requirements**

As with systems in any programming paradigm, aspect-oriented systems need to be designed with good software engineering practices in mind.

The analysis and design of a system are at least as important as the implementation itself, with many considering these phases to be more significant in their contribution to the success of a project as a whole.

In any development effort, it is helpful for a developer to be able to consider the structure of the final implementation at all stages of the software lifecycle, rather than having to make a mental leap to get from a particular way of encoding design,



to another way of coding the software. In other words, developers need to be able to easily map their designs to the code in order for the design to continue to make sense during the development lifecycle.

In addition to seamless traceability between the design and code, another consideration is the benefits of separating aspects in the design for the design's own sake. Aspect-oriented design has similar benefits for design artifacts as aspect-oriented code has for code artifacts. In the infancy of aspect-orientation, developers simply used object-oriented methods and languages (such as UML) for designing their aspects. This proved difficult, as UML was not designed to provide constructs to describe aspects: trying to design aspects using object-oriented modeling techniques proved as problematic as trying to implement aspects using objects.

Without the design constructs to separate crosscutting functionality, similar difficulties in modularizing the designs occur, with similar maintenance and evolution headaches. What is required is special support for designing aspects, as we will then be able to improve the design process, and provide better traceability to aspect-oriented code.

A similar set of problems arises when analyzing requirements documentation to try to arrive at how to design a system. Approaches for decomposing requirements from an object-oriented perspective simply don't go far enough when trying to plan for aspect-orientation. Heuristics and tools to support such an examination will be helpful to the developer.

The thesis work aims to support for how to both identify aspects in a set of requirements, and how to model them in UML style designs. The methodology we use is an aspect based approach to analysis and design.

Aspects may be related to each other, in the same way as requirements or features are related to other parts of the system. Such relationships may cause overlaps in the aspects. We see two kinds of overlap. The first category of overlap is *concept sharing*, where different aspects have design elements that represent the same core concepts in the domain. Each aspect will contain specifications for those same concepts designed from the perspective of the aspect. The second category of overlap is the classic aspect-oriented *crosscutting*, where dynamic behaviour in one aspect will be triggered in tandem with behaviour in other aspects.

### **4.3 Evolving of Concerns in Early Requirements Phase**

The notion of a concern is fundamental to problem solving and separation of concerns is a fundamental principle for organising software development. The sheer size and complexity of many modern software-based systems, however, indicates that the criteria for separation are rarely fixed, and that concerns themselves are often overlapping and interact in ways that may not be easy to pre-determine. The aspect-oriented software development community has emerged in response to the need to address such issues. The focus of this community has been largely on software itself - its structuring and restructuring. Aspects provide a structuring construct that allows program code to be written, or re-written, to facilitate the representation of multiple concerns and to alleviate tangling of overlapping, aka crosscutting concerns.

In the problem world, inhabited by customers and users, is fertile ground for identifying concerns and for exploring their interaction. Indeed, it is a fact that the problem world is often the most appropriate source for early identification of concerns, but not necessarily of aspects. An understanding of the problem is a prerequisite to constructing a suitable solution, we also recognize that the processes of understanding problems and constructing solutions are inevitably intertwined. Producing a robust statement of requirements often needs an exploration of the solution space - the question is: does that solution space need to be populated with aspects while the requirements are still being formulated? Often, during the requirements and specification process, some architectural position will be taken, probably implicitly, and this position may drive what is an aspect and what is not. This may be advantageous, as identifying aspects early may lead to more robust designs and implementations. But how early? During the exploration of a problem space, and associated requirements, is not normally the right time to examine aspect-oriented solutions. However, when we start generating specifications - that map the problem and solution spaces - then identifying aspects becomes useful.

### **4.4 Separation of Concerns**

Separation of concerns (SOC) is a long-established principle in software engineering [19]. It has received widespread attention in modern programming languages, with constructs such as modules, packages, classes, and interfaces,

which support properties such as abstraction, encapsulation, and information hiding. SOC has also received attention in software architecture and design, with techniques such as composition filters [64] and design patterns [65].

While advances in all of these areas have had significant benefits, problems related to inadequate separation of concerns remain. This has led to recent work on "advanced separation of concerns" (ASOC), including subject-oriented programming and design, aspect-oriented programming, and multidimensional separation of concerns. These bring a number of innovative ideas to programming in particular and to software development in general, which are now beginning to mature and coalesce under the heading of aspect-oriented software development (AOSD).

Although ASOC has been emphasized in recent work, concerns themselves have remained something of second-class citizens. Current ASOC tools provide only limited support for explicit concern modeling, representations of concerns tend to be tied to particular tools or artifacts, and concern modeling usually occurs just in the context of a particular type of development activity such as coding or design. A global perspective on concerns, that spans the life cycle and is independent of particular development tools or artifacts, has been lacking.

Of course, concerns do not play a second-class role in software development. They arise at every stage of the life cycle, spanning activities, artifacts, methods, and tools. If aspect-oriented software development is to be fully realized, concerns must be treated as first-class entities throughout the life cycle.

#### **4.5 Multiple Perspectives to Concern Requirements**

Crosscutting requirements serve a dual purpose. On the one hand, they provide a description of the overlap between requirements - the first step to managing any inconsistency that arises at such overlap. On the other hand, they can provide useful input into aspect-oriented design and implementation, as they provide the potential join points upon which an aspect-oriented implementation might be based.

Identifying aspects too early is counter-productive, but than an early understanding of requirements and the concerns they address is crucial. At the stage when requirements need to be mapped onto elements of a software solution, identifying aspects may become much more worthwhile.

Although the notion of *concern* is well understood intuitively, good definitions of it are surprisingly hard to come by [18]. *Aspects* are one category of concern: an aspect is a (program) property that cannot be cleanly encapsulated in a "generalized procedure" (such as an object, method, procedure, or API) [53]. This definition identifies a critical property of some concerns that makes SOC problematic in conventional programming languages. However, it is too relative to program structure (and to code) to make a good general definition of concern. Tart and others [25] define a concern as a predicate over software units. This definition is not particular to code and appropriately spans the whole software life cycle, but it is still based on software units.

To promote concerns to first-class entities ("concerns") in software development, they must be defined independently of any specific type of software artifact and even of software artifacts in general. One dictionary definition of *concern* is "a matter for consideration" [10]. More specifically to software, the IEEE defines the concerns for a system as "... those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders" [18]. We will take *concern* generally to be any matter of interest in a software system.

We define a *concern space* as an organized representation of concerns and their relationships. This is a generalization of the notion of a concern hyperspace in which the concerns are organized according to multiple, overlapping dimensions, where each dimension is partitioned by concerns of the same general type (such as functions, classes, features). Our definition contrasts with the alternative characterization of a concern space as a body of software, as we are concerned first of all with the matters of interest pertaining to a body of software rather than with the software itself (although the relationship of concerns to software is fundamentally important and is indeed reflected in our approach). Based on previous studies, we believe that concern spaces are multidimensional, that is, multiple concerns of multiple types may apply to a particular software unit at any one time. Additionally, we believe that a concern space is highly structured; that is, that concerns can be organized by multiple relationships of multiple types, that these relationships may be independent or dependent, and that they commonly have a hierarchical or lattice like organization.

During software development, concerns arise at all stages of the life cycle, from requirements specification through design, coding and testing to maintenance and evolution. Concerns also span multiple phases of the life cycle, relate to multiple instances and types of artifacts, and crosscut phases and artifacts in different ways. Finally, concerns are dynamic and relative, that is, that the concerns relevant to a particular software unit will change over time and that they also depend on the perspective or purpose of the user or stakeholder who considers the software [11, 53, 54].

Given the above, we believe that a general-purpose concern-space modeling schema should [18]:

- 1) Support the representation of arbitrary concerns
- 2) Support the representation of composite concerns (such as emergent concerns based on interactions of base concerns)
- 3) Support the representation of arbitrary relationships among concerns
- 4) Support the association of concerns to arbitrary software units, work products, or system elements
- 5) Be language independent; that is,
  - a) Not depend on any particular programming language or other development formalism
  - b) Accommodate different development formalisms appropriate to different stages of the life cycle
  - c) Be able to capture information that is not necessarily reflected in particular development formalisms
- 6) Be methodology independent
- 7) Be applicable across the software life cycle
- 8) Support a variety of types of software engineering tasks (as appropriate to the development methods in which it is used)

The first four of these requirements address needs in the modeling of concerns and their interrelationships, the last five address needs in the use of the schema and particular models.

One determinant of when to move from exploring concerns to identifying aspects is evolution. In particular, changing requirements may add further crosscutting concerns that, in turn, necessitate restructuring the problem or solution space.

## 4.6 Architectural Design Enforcement

The construction of complex, evolving software systems requires a high-level design model. This model should be made explicit[33], particularly the part of it that specifies the principles and guidelines that are to govern the structure of the system. In reality, however, implementors tend to overlook the documental design models and guidelines, causing the implemented system to diverge from its model. Reasoning about a system whose models and implementation diverge is error prone - the knowledge we gain from these models is not of the system itself, but of some fictitious system, the system we intended to build. The system's comprehensibility is impeded, and so using software engineering techniques goes against our intended goals - quality, maintainability and cost minimization.

Two major approaches have been suggested to bridge the gap between high-level design models and the system itself: user invoked i.e. the use of codified design principles must be supplemented by checks to ensure that the actual implementation adheres to its design constraints and guidelines versus the environment invoked i.e. the gap between the architectural model and the implemented system can be bridged effectively if the model is not just stated, but is enforced.

These principles cannot be localized in a single module, they must be observed everywhere in the system, which means that they crosscut the system's architecture.

Aspect Oriented Programming (AOP) is a programming technique for modularizing concerns that crosscut the basic functionality of systems. Aspects provide a means to clearly capture design decisions.

### 4.6.1 Enforcing Architectural Regularities

Aspects can be used for:

- *Design by Contract* : AOP can be used as a mean for the implementation of the *design by Contract* design methodology. For example, pre / post conditions are checked using *before and after* (respectively) on a method execution join point.
- *Exception Handling*: The design regularity of "all exceptions of a certain type should be handled the same way".

- *Observer Design Pattern* : The enforcement of the Observer design pattern [8] is illustrated in the example supplied by the AspectJ team. The example of this behavioral pattern uses the *introduction mechanism* as well as method call receptions.

What all of these examples have in common is the fact that the regularities they define are of a dynamic nature and are enforced upon a monolithic system. These examples can be used for enforcing architectural principles, but when attempting to implement design restrictions with AspectJ, one quickly reaches realms which are not covered by the literature.

In other words, we are addressing the possibility of using AOP in general, and AspectJ in particular, in order to solve the problem of design enforcement.

#### **4.7 Avoiding Design Incompatibility**

Early Aspects refer to crosscutting properties at the requirements and architecture level. The term denotes aspect-orientation within the early development stages of requirements engineering and architecture design. The focus is on the separation of crosscutting concerns at the high level architecture and the low level design while offering an approach for aspect-oriented modeling and automated code generation. Typically, design artifacts that crosscut an architecture cannot be encapsulated by single components or packages and are typically spread across several of them and therefore also make design hard to understand and maintain. This work addresses the specification of crosscutting concerns at the architecture level in order to maintain the separation of concerns at an early stage in the software development lifecycle. Crosscutting design artifacts can clearly be encapsulated avoiding tangling and scattering.

The architecture design is an important step within the software development lifecycle. OO design has proved its strength when it comes to modeling common behavior. However, OO design does not adequately address design artifacts that crosscut an architecture. They cannot be encapsulated by single components or packages and are typically spread across several of them and therefore also make design hard to understand and maintain. Crosscutting concerns are present during all phases of a software development lifecycle, leading to code tangling or code scattering during the implementation phase and *graphical tangling* during the design phase. AOSD is still lacking standardized concepts at the design phase that

would foster the specification of crosscutting concerns at the high-level architecture and low-level design. Development of large software systems follows processes that all include activities like requirements engineering, analysis, design and implementations. Following a design methodology like OOD, and focusing on AOP at coding level causes a shift of paradigms between OO design and AO code. This leads to inconsistencies between design and implementation, as the AO paradigm is not seamlessly supported during the early stages of the development lifecycle. To avoid the divergence of design models and code, crosscutting concerns must be identified at the requirements and architecture level and carried forward in the implementation phase. Concepts are needed for a seamless integration of AO design and implementation and will be a first step towards an integrated AO development process. To make AOSD more widely accepted, the different phases of an AOSD lifecycle have to be integrated more smoothly by supporting the AO paradigm in every phase. This work includes a design notation for validation of AO models. Supporting design models and their transition to concrete implementations makes AOSD more usable, more efficient and more accepted among software engineers.

When analyzing OO design, one can see that OO modeling tries to adopt many of the OO programming features for design and analysis. Classes, their structures, and their relationships are identified and generalization and aggregation hierarchies are built. OO design techniques are not sufficient when focusing on the AO paradigm as crosscutting concerns also make design tangled and therefore hard to understand and maintain. When developing an AO modeling approach, the following requirements are obvious:

- A sufficient notation should be simple to understand and straightforward to use for developers who are familiar with common design notations (such as UML).
- Design modeling should be supported by powerful CASE tools to improve developer productivity and to ensure syntactical correctness of the AO model.
- Design notations should support modeling according to the paradigms behind the most common AO approaches and languages.



- Models should be easy to read and offer a clear separation of concerns to avoid crosscutting concerns spanning over many design elements.
- A direct mapping between the notation and supported implementation languages should allow automatic code generation based on the design model.
- The notation should be applicable in real-world development projects and should be part of an integrated AO development process.

This work can be seen as a step towards a standardized way to capture aspects at the design phase of an AO development process. Existing approaches and prototypes are well aware of the fact that aspect-oriented modeling is a critical part of AOSD. Obviously, to obtain an AO development lifecycle, the gap between AO requirements engineering and AOP has to be filled. This work makes a contribution to the problem of bridging this gap.

#### **4.8 Requirements Validation**

In general, modeling is a broad notion that can be involved in various perspectives of software development, such as design specification, code generation, testing, and reverse engineering. Models from different perspectives require different level of details although their structures may appear to be similar. For example, a traditional state model for design specification does not carry sufficient information for test generation. The aspect-oriented extensions to state models and UML are primarily for the purposes of design specification. Under testing, we explore aspect-oriented state models for testable specification and test generation of aspect oriented programs.

While AOP provides a flexible mechanism for modularizing crosscutting concerns, it raises new challenges for testing aspect oriented programs. A fault model has been proposed for aspect-oriented programming, which includes six types of faults: incorrect strength in pointcut patterns, incorrect aspect precedence, failure to establish post-conditions, failure to preserve state invariants, incorrect focus of control flow, and incorrect changes in control dependencies [67]. This fault model has not yet constituted a fully-developed testing approach. Control flow graphs

were constructed at system and module levels, and then test suites were derived from control flow graphs.

While aspects in aspect-oriented programming (AOP) offer an effective way for modularizing separate concerns, the new programming constructs of AOP languages introduce numerous opportunities for programmers to bring various potential faults with respect to aspects. Generally, an aspect-oriented program consists of aspects and their base classes (or components) that can be woven into an executable whole. The base classes in an aspect-oriented program can also be executed independently. From the system architecture perspective, aspects often crosscut multiple base classes. From the base class perspective, however, aspects are essentially incremental modifications to base classes with additional operations and constraints for separate concerns. The incremental modifications of aspects to base classes can impose a significant impact on the object states of base classes. Although aspects in AOP add more code to their base classes, they can not only introduce new object states and transitions, but also remove and update state transitions. As such, aspects may lead to subtle differences in the sequence of messages that can be accepted by the base class objects. In particular, aspect-specific faults likely result in unexpected object states and transitions.

To reveal aspect-specific faults, we need to investigate model-based testing, i.e. testing whether or not aspect-oriented programs and their base classes conform to their respective behavior models. Model-based testing is appealing because of several benefits: (1) the modeling activity helps clarify requirements and enhance communication between developers and testers; (2) design models, if available, can be reused for testing purposes; (3) model-based testing process can also be (partially) automated; and (4) more importantly, model-based testing can improve error detection capability and reduce testing cost by automatically generating and executing many test cases.

#### **4.9 AOSDDL Requirements**

The primary goal of the work presented here is to develop a generic aspect oriented design language that can be used to design and build aspect oriented applications. However, since the prototyping of such a system is impeded by financial resources and time constraints in the context of a PhD project, the objective is rather to use open source tools as a base platform and to focus on the

development of a minimal, but extensible design language that may serve as a flexible platform for future aspect oriented research.

#### **4.10 Summary**

This chapter has examined the requirements for aspect oriented software development and design language in particular.

It discussed the general requirements for aspect oriented systems. These requirements have been derived from related work and acknowledged publications in the field. They summarize the general requirements of today's aspect oriented systems.

The following chapters present the design (chapter 5) and implementation (chapter 6) of the aspect oriented design language. Both chapters show how AOSDDL fulfils the requirements defined in this chapter by design and implementation. Finally, chapter 6 also evaluates to what extent AOSDDL has succeeded in meeting these requirements.

## CHAPTER 5

### The AOSDDL (Aspect Oriented Software Development Design Language)

#### 5.1 Overview

The actual realization of a design language form has been revealed to be non-trivial. Chapter 3 has described a large variety of design language forms. However, most of them are very much tailored towards a specific application or application domain. Moreover, with the exception of very few hardly any of these language notations have been used outside their own research environment. The development of a more generic aspect oriented design language requires a wider and more thorough look at the requirements. Chapter 4 has discussed the multitude of requirements and has defined the specific requirements for the development of the Aspect Oriented Software Development Design Language (AOSDDL).

This central chapter of the work introduces the notations and discusses how it fulfils the requirements that have been defined above in principle and design and consequently would be a general purpose AOSD design language (AOSDDL) that will map AOSD design notations to the existing AOP languages.

#### 5.2 Tools Environment

The Eclipse Platform for Java was used to carry out the implementation and testing of the abstract notations in AspectJ. To implement graphical notations and diagrams the Together CASE tool was used. The CASE tool *Together* from Borland is an enterprise development platform enabling application design, development, and deployment. It is extensible through an open Java API offering the possibility to develop custom software that plugs into the Together platform in the form of modules. The open API is composed of a three-tier interface that enables varying degrees of access to the infrastructure of Together.

### 5.3 AOSDDL Notations

This work specifies an approach for AO modeling to address the specification of crosscutting concerns at the architecture level in order to maintain the separation of concerns at an early stage in the software development lifecycle. A key intention is to offer standard development tool support and interchangeability among various CASE tools, thus an extension to UML was developed without changing its metamodel specification to achieve standard UML conformity. Using UML as a modeling language improves developer productivity and offers high acceptance, as it is the industry-standard modeling language for the software engineering community. When using standard UML for aspect-oriented modeling, developers do modeling by using familiar tools and environments to gain all the benefits they are used to in OO design. UML is an extensible modeling language that enables domain-specific modeling which raises its suitability as a modeling language for supporting aspect-oriented modeling.

Another important goal was to gain the benefits both of code and design reuse of AO software, including the ability to reuse aspect and base elements separately. Thus, aspects and base elements should be completely kept apart and independent of the implementation technology in order to simplify the replacement of the AO language. A clear separation of the language dependent crosscutting parts eases the support of many different AO languages and concepts. This work focuses on adopting AspectJ concepts for the implementation language dependent parts of AOSDDL. For the support of other AO concepts (such as Hyper/J) is considered and part of some future work. AOSDDL considers the fact that crosscutting concerns tend to affect multiple classes in a system. Since a concern itself can consist of several classes and since all of these classes may be associated with the class the concern crosscuts, the module construct for a concern should be higher-level than a class. Otherwise associations modeled on class-level would supersede the logical grouping of the classes belonging to one concern. This would make the design models hard to read and lead to graphical tangling of crosscutting concerns instead of a clear separation.

### 5.3.1 Symbols for AOSDDL Notations



**Package**

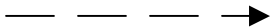
The Package notation is either base package containing business logic or aspect package containing cross cutting concern or a connector that links aspects and base elements.



**Class**

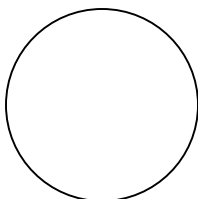
The class is either

- *Introduction* class that defines the rules for Aspect J's introduction mechanism or
- *Pointcut* class that defines execution points in the control flow of the program.
- *Advice* class that defines the code to be executed at the pointcuts defines in the pointcut class.

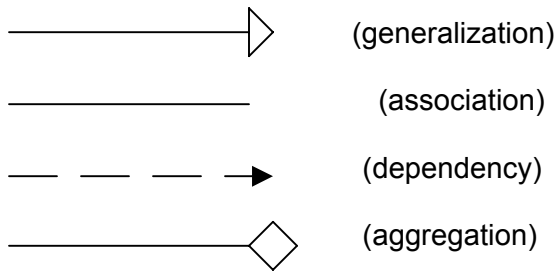


**Joint Point**

Joint points represent points in the dynamic execution of the program.

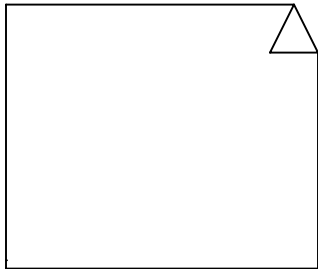


**Connector**



**Relationships**

The relationships is defined to indicate either dependency or generalization or association or aggregation.



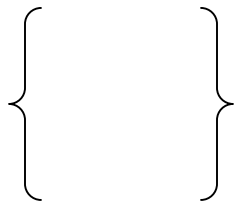
**Note**

Attaching comments to an element or a collection of elements.



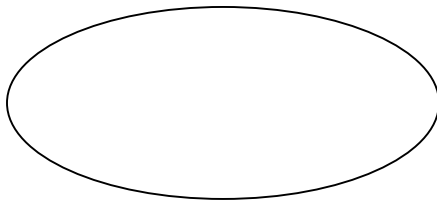
**Stereotype aspect**

Stereotype aspects extends aspects to define new modeling elements like boundary, control or entity aspects during software development.



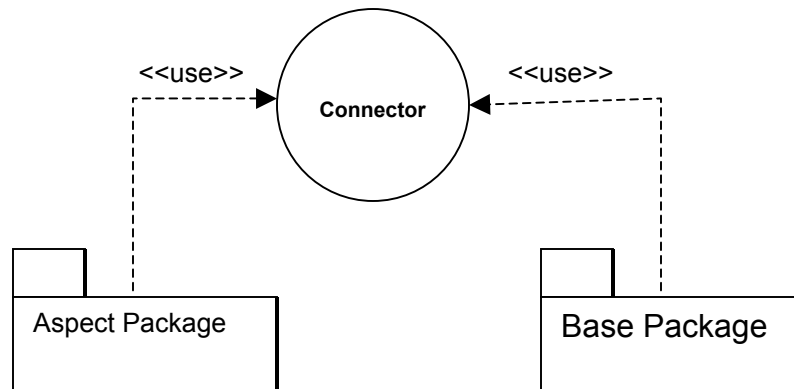
### Constraints

Defining constraints in aspects.



### Weaving advice

To Implement weaving mechanism for advice in Aspect Oriented Programming.



**Figure 5.1 Using Package and Connector**

Figure 5.1 provides an overview of the notation and its focus on using package and a connector. AOSDDL includes a *base package* (having the business logic), an *aspect package* (having the crosscutting concern) and a *connector* to link aspects and base elements. Being one of the most popular Aspect Oriented language, AspectJ has been used to describe and present the AOSDDL notation model. Both the aspect package and the base package are used to express any crosscutting concern that can occur and might affect the system. Further, they can contain any valid UML design construct that might be describing either the complete



system or a part of the system based on Aspect Oriented Design. The aspect can be modeled as an individual entity or independently of any design it may potentially affect or be a part of. The connection between base design and aspect design is specified separately. Support of different AO technologies is therefore rather simple and straightforward, as it is only the connector's syntax that has to be changed. This connector will hide the details of the interaction between components. To model any design construct, the connector can be considered in terms of a client connector that communicates with the aspect packages via the <uses> relationship. The type of connector used to interconnect aspect components also influences the performance of component based systems. This kind of separation enables high degree of reusability of both the aspect and base elements since the connector notation (element) is the only crosscutting element. This way of focusing on UML notations and standard notation of packages as a single unit leads to design models that are easy to read, as they avoid *graphical tangling*. Additionally, the connector encapsulates the underlying implementation technology using AspectJ.

As described above, the AOSDDL notation can contain the following classes that conform to the concepts AspectJ offers for the specification of weaving rules:

1. The *Introduction* class, which defines the rules for AspectJ's introduction mechanism.
2. The *Pointcut* class, which defines execution points in the control flow of the program.
3. The *Advice* class, which defines the code to be executed at the pointcuts defined in the *Pointcut* class.

All classes contain operations with special semantics to specify how aspect and base elements have to be recomposed. The complete syntax of the AspectJ specific connector will not be presented here, however a few examples described here, provide a macroscopic view of how the notation can be used and shows some of the most important constructs.

AOSDDL is a simple and powerful notation for aspect-oriented modeling. In order to reduce errors when mapping models to code and offer low-level architecture design support, the development of code generator is part of a future work.

## 5.4 Other Design Notations

The following issues were considered for modeling a general purpose AOSD design language (AOSDDL) with regard to a programming language namely, AspectJ and a standard Object Oriented design language namely, UML quite widely in use in the software industry:

- Mapping AOP to Aspect Oriented UML Extensions
- Identifying Software Concerns
- Design Language Issues for Component Based software Development
- Mapping UML extensions through composition patterns to Aspects
- AspectJ Extensions for Distributed Computing

The issues basically provide a broad outline that sums up the parameters for various concerns and scenarios prevalent in the software industry and how AOSDDL will address them under various forms.

## 5.5 Mapping AOP to Aspect Oriented UML Extensions

Aspect-oriented programming (AOP) is a new software development paradigm that aims to increase comprehensibility, adaptability, and reusability by introducing a new modular unit called "aspect", for the specification of crosscutting concerns. AspectJ is a programming language that supports the aspect-oriented programming paradigm by providing new language constructs to implement crosscutting code. At present, no design notation is available that appears to be appropriate for the design of aspect-oriented programs in AspectJ. The need of such a design notation is obvious. First, it would ease the development of AspectJ programs. A graphical notation helps developers to design and comprehend AspectJ programs. Further, it would facilitate the perception of aspect-orientation. A design notation helps developers to assess the crosscutting effects of aspects on their base classes. Its application carries over the advantages of aspect orientation to the design level and facilitates adaptation and reuse of existing design constructs.

The approach presented here extends the Unified Modeling Language (UML) with the aspect-oriented design concepts as they are specified in AspectJ (in the following, the approach called the "aspect-oriented software development design language", or AOSDDL for short). The approach reproduces these concepts by extending existing UML concepts using UML's standard extension mechanisms. Doing so assures an immediate understanding of aspect-oriented design models and enables rapid support by a wide variety of CASE tools.

### **5.5.1 Problem Identification**

The work is organized in the following way. An UML implementation of AspectJ's weaving mechanism is described. A new relationship is introduced to represent this weaving mechanism. Next, existing approaches to extend the UML with aspect-oriented design concepts are regarded with respect to their compliance with AspectJ's semantic.

Concise and independent examples were created and the same were implemented in both UML and AspectJ.

### **5.5.2 Basic Notations for AspectJ**

UML representations are presented for AspectJ's basic abstractions, such as connector, join points, pointcuts, pieces of advice, introductions, and aspects.

#### **Connector**

Deploying an aspect within an application is done by making use of connectors. A connector contains three types of constructs: one or more initializations, zero or more behaviour method executions, and finally any number of aspectJ language constructs.

The advantage of permitting the calling of behavior methods in the connector is that it enables advanced users of an aspect to tightly control the execution of the aspect-behavior. The default-method on the other hand, provides an easy way for deploying an aspect within an application, without needing any knowledge about how the aspect-behavior is executed.

```

connector PrinterController {
    AccessManager.AccessControl control =
        new AccessManager.AccessControl(* Printer.*(*));
    contro.replace();
}

```

**Figure 5. 2: Connector Syntax**

## Join Points

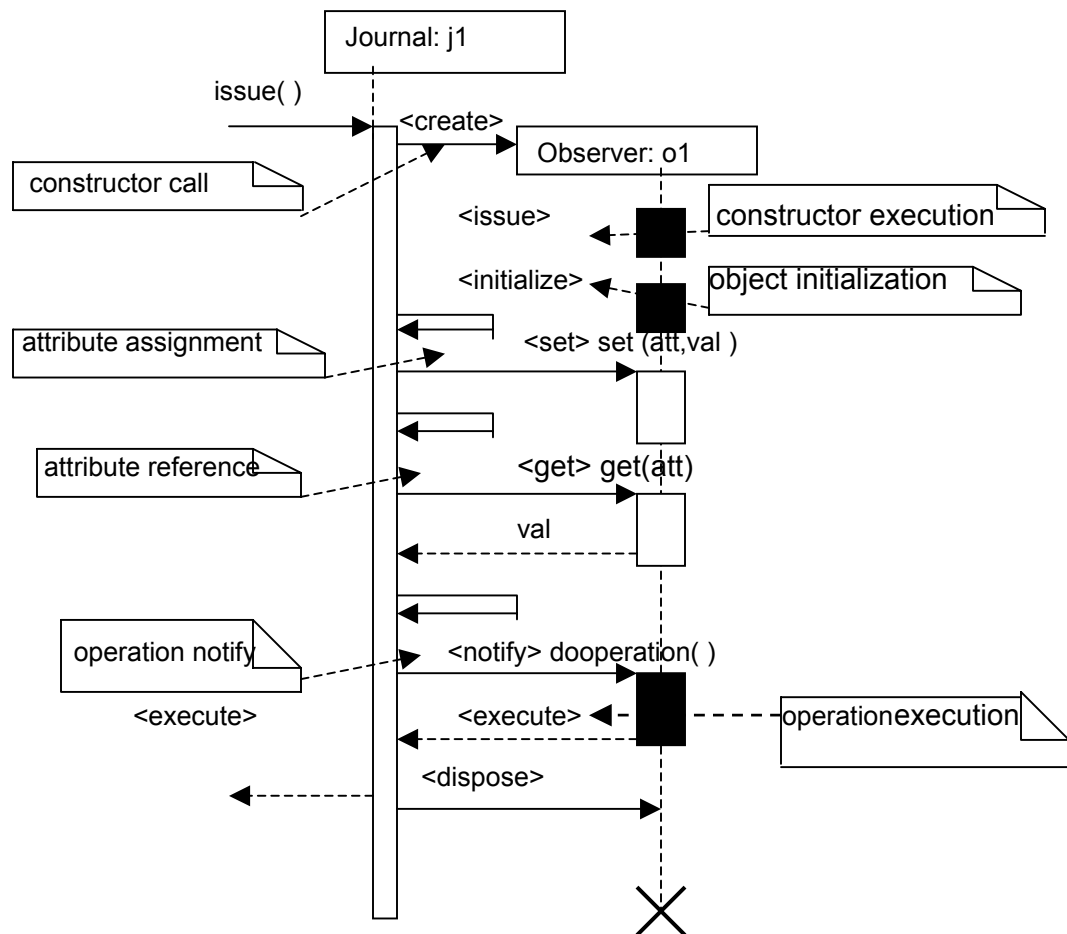
Looking for an appropriate UML representation for join points, links can be identified as the one model element which represents them best.

In the UML, links serve as communication collection for stimuli. A stimulus defines a communication between two instances that is dispatched by an action, such as an invocation of an operation, a request to create or to destroy an instance, or a raise of a (asynchronous) signal. This means that control is passed from one instance to another via communication links. Hence, links in the UML represent "principled points in the dynamic execution of a program" just like join points do in AspectJ. And just like join points in AspectJ, control passes each communication link in the UML two times, once the control is passed down to the called instance, and once control flows back up again.

However, whether a link actually represents a join point depends on the exact communication that is dispatched over the link. A link used to communicate the destruction of an instance, for example, does not represent a join point in the sense of AspectJ. AspectJ's join point model defines precisely which kind of communications promotes an ordinary link to a representation of a join point. In the UML some communications such as field references or field assignments do not dispatch stimuli. This means that control flow passes no link at all, and no link can be assigned to represent the respective join points. To solve this problem, in the AOSDDL, these communications are stereotyped as "pseudo" invocations of "pseudo" operations that have no other purpose than to read or write (respectively) a specific field. Similar, no link can be identified to represent execution and initialization join points. Considering that the execution of an operation or a constructor or the initialization of an object never occurs without a (preceding) operation or constructor call, it is legitimate to use one link (i.e., the one associated with the call or create action) to represent all two or three join points. To represent

the order in which control passes these join points, corresponding call, execute, and initialize actions are organized to an UML action sequence.

Join points may be visualized in UML interaction diagrams by highlighting messages. Considering that messages are associated with communications and require the existence of links, it is proper to highlight messages in collaborations to indicate join points.

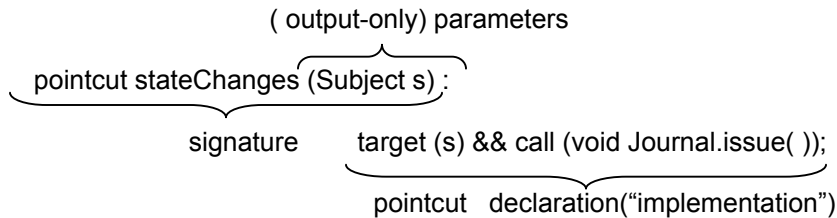


**Figure 5.3: Representing Join Points in Sequence Diagrams**

#### Pointcuts

A pointcut is a set of join points, which are well defined instants in the execution of the program. Abstract pointcuts can be labeled via template parameters. In the AOSDDL, pointcuts are represented as operations of a special stereotype, named **<pointcut>**. This is legitimate due to the strong structural resemblance of pointcuts to standard UML operations. Just like standard UML operations, pointcuts are

features of a particular classifier (i.e., an aspect), they may have an arbitrary number of (output-only) parameters, and their declaration comprises a signature and an "implementation" (see Figure 5.4).



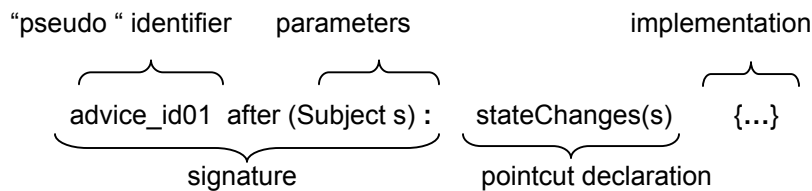
**Figure 5.4: Mapping of Pointcuts to Operations**

The <pointcut> stereotype captures a new semantic and specifies several additional constraints. One of those constraints declares that operations of stereotype <pointcut> must be implemented by methods of a special stereotype that equips the standard UML Method meta-class with an additional property named "base" to hold the "implementation" of the pointcut (i.e., its declaration).

#### Advice

Advice in AspectJ is code similar to an operation that is, code executed when a joinpoint is reached. Standard UML has very high degree of support for depicting interactions. This support is used to depict the behaviour of an advice. The before, after and around constructs of AspectJ are easily mapped using the interaction diagrams. Similar to a pointcut, an advice is represented as an operation of a special stereotype, named, <advice>. A piece of advice is a feature of a particular classifier (i.e., an aspect), it may have an arbitrary number of parameters, and its declaration comprises a signature and an implementation (see Figure 5.5). In contrast to a pointcut, an advice is also semantically comparable to a standard UML operation because it defines some dynamic feature that effects behavior. However, there is a semantic difference between an advice and an operation. One important difference is, for example, that an advice does not have a unique identifier. This circumstance may cause conflicts with existing well-formedness rules of the UML, stating that two operations (i.e., two pieces of advice) in the same classifier (i.e., aspect) must not have the same signature. To avoid such conflicts, the AOSDDL supplies an advice with a "pseudo" identifier. Another difference pertains to inheritance. Since in AspectJ a piece of advice has

no unique identifier in the super-aspect, it cannot be overridden in the sub-aspect. The <advice> stereotype captures this semantic difference by constraining that an advice in the AOSDDL (although having a "pseudo" identifier) cannot be overridden. Then, advice declarations in AspectJ contain pointcut declarations that specify the set of join points at which the advice is to be executed. Therefore, operations of stereotype <advice>, must be implemented by methods of a special stereotype that equips the standard UML Method meta-class with an additional property named "base" to hold the pointcut declaration. Note how this proceeding coincides with the way that pointcuts are implemented in the AOSDDL. In fact, the same method stereotype is used for the implementation of both pieces of advice and pointcuts.



**Figure 5.5: Mapping of Advice to an Operation**

### Pointcut designators

A symbolic name to identify an pointcut. Name based pointcut designators correspond to binding named operations. Further, the standardized UML semantics are used to depict pointcuts designations.

### Introductions

In AspectJ, introductions are used to insert members (such as constructors, methods, and fields) and relationships (such as generalization/specialization and realization relationships) to the base class structure.

Since introductions in AspectJ may insert both members and relationships, the parameterized model element destined to represent introductions in the UML must be able to describe members and relationships, too. After reviewing the UML specification, parameterized *collaborations* can be identified to meet these requirements best. In the UML, collaborations are used to specify a set of

instances together with their members and relationships (i.e., a structural context) and a set of interactions that describes some communication between these instances (i.e., some behavior performed within the structural context). So, collaboration templates prove to be suitable to specify structural and behavioral characteristics of introductions. The AOSDDL specifies an extra stereotype of collaboration templates, named <<introductions>>, to capture the particular semantic of introductions.

The AOSDDL specifies a special binding mechanism for collaboration templates of stereotype, <<introduction>>(see Figure 5.6). Note that introductions in AspectJ are conceptually *always* bound to (a fixed set of) actual base classes, which are specified as type pattern in the introduction declaration.

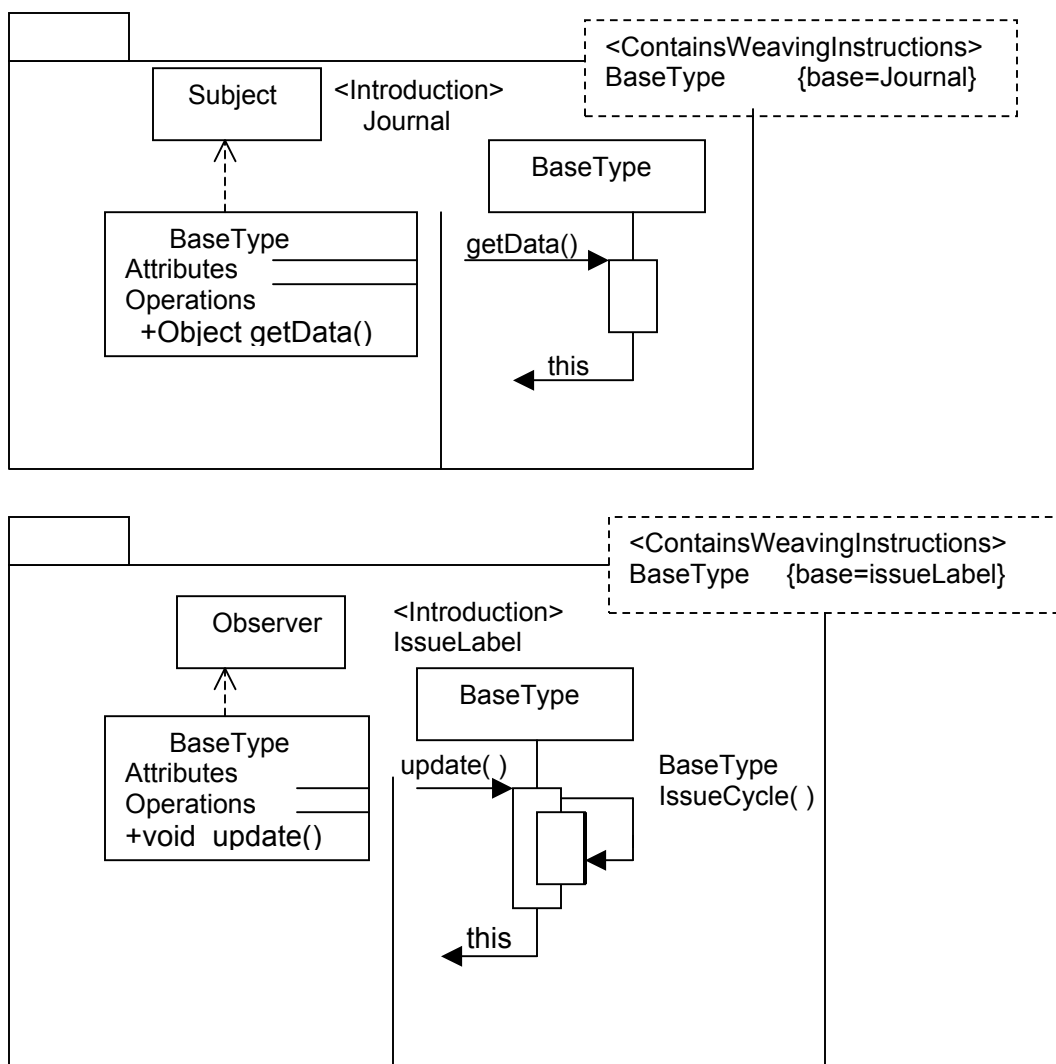


Figure 5.6: Design of Introductions



Accordingly, in the AOSDDL, template parameters of a collaboration template stereotyped with `<<introduction>>` are required to be of a special stereotype, named `<containsWeavingInstructions>`. That stereotype equips the standard UML `TemplateParameter` metaclass with a supplementary meta-attribute, named "base", to hold the type pattern that specifies the set of actual base classes to be crosscut. A collaboration template of stereotype `<<introduction>>` is generally considered to be implicitly bound to the actual arguments specified in that "base" expression. Thus, it is proper to use introduction templates in design models directly.

## Aspects

In the AOSDDL, aspects are represented as classes of a special stereotype, named `<<aspect>>`. This is legitimate due to the strong structural similarity between aspects and standard UML classes. Just like standard UML classes, aspects serve as containers and namespaces for various features, such as attributes, operations, pointcuts, pieces of advice, and introductions. And just like them, they may participate in associations and generalization relationships. However, there are differences between aspects and classes concerning their instantiation and inheritance mechanisms. For instance, aspect declarations in AspectJ contain instantiation clauses that specify the precise way in which an aspect is to be instantiated (e.g., per object, per control flow, or once for the global environment). Further, sub-aspects in AspectJ inherit all features from their super-aspects, yet only ordinary Java operations and abstract pointcuts may be overridden. The new `<aspect>` stereotype captures these semantic differences. Besides that, the stereotype equips the standard UML `Class` meta-class with a couple of additional meta-attributes to hold the instantiation clause, the pointcut declaration contained in that instantiation clause, and a boolean expression specifying whether the aspect (not just its introductions) may access the members of the base classes as a privileged "friend".

In the AOSDDL, the crosscutting effects of aspects and its components are indicated by `<<crosscut>>` relationships.

### **5.5.3 Weaving Mechanism of AspectJ**

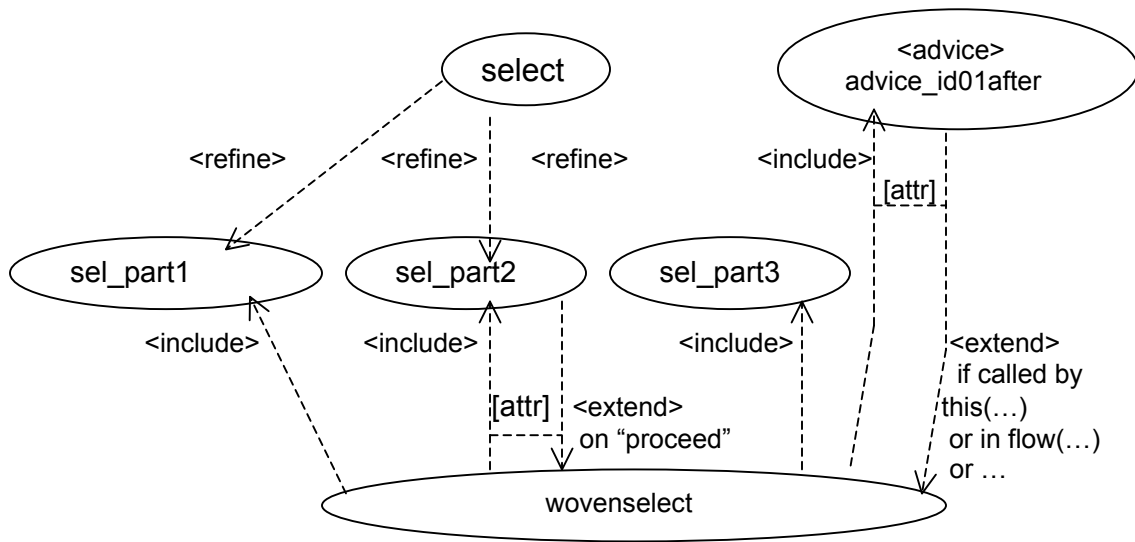
A relationship is introduced denoting the crosscutting effects of aspects on their base classes. Both the weaving mechanism and the relationship are derived from weaving instructions specified in the aspects.

### **5.5.4 Weaving Advice**

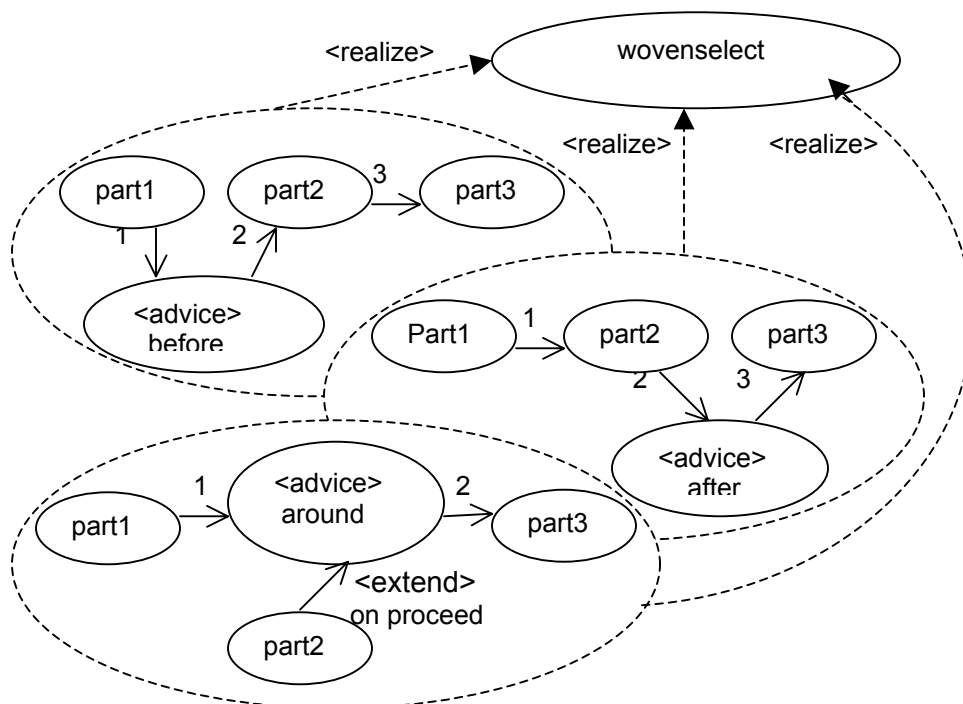
The AOSDDL implements AspectJ's weaving mechanism for advice with help of collaborations. For weaving purposes, the collaboration describing the behavior of the base classes' operations is split at first. Splitting always takes place at a particular join point. Depending on the kind of advice to be inserted, the collaboration is split before, after, or (in the case of around advice) before and after the particular join point. Then, the split fragments are composed with the collaboration describing the advice to form a new collaboration. In the composition of collaborations can be accomplished by identifying and matching instances that participate in each of the collaborations to be composed.

To explicitly state the order of weaving, the AOSDDL utilizes UML use cases. In the UML use cases are used to define a piece of behavior of a semantic entity, e.g., the operation of a class or the advice of an aspect. (Super-ordinate) use cases can be split into a set of smaller (sub-ordinate) use cases using refinement relationships. Further, use cases may (unconditionally) include the behavior defined in other use cases by means of include relationships. At last, a use case may augment the behavior of another use case by means of extend relationships. Extend relationships provide a condition that must be fulfilled for the extension to take place. To represent the weaving order in the UML, the AOSDDL refines the use case describing the base classes' operations (for example, the "select" use case in Figure 5.7) into three sub-ordinate use cases; one describing the behavior at the join point ("select...part 2"), the others describing the behavior before ("select..part 1") and after that join point "select\_part 3"). Then, the AOSDDL composes a new use case ("wovenSelect") that includes the behavior (i.e., the use cases) of both the base classes' operations and the advice. In the UML, collaborations may be specified to explicitly describe how the included use cases

cooperate to perform the behavior of the including use case. Figure 5.8 shows three collaborations specifying



**Fig 5.7: Weaving Advice with UML Use Cases**



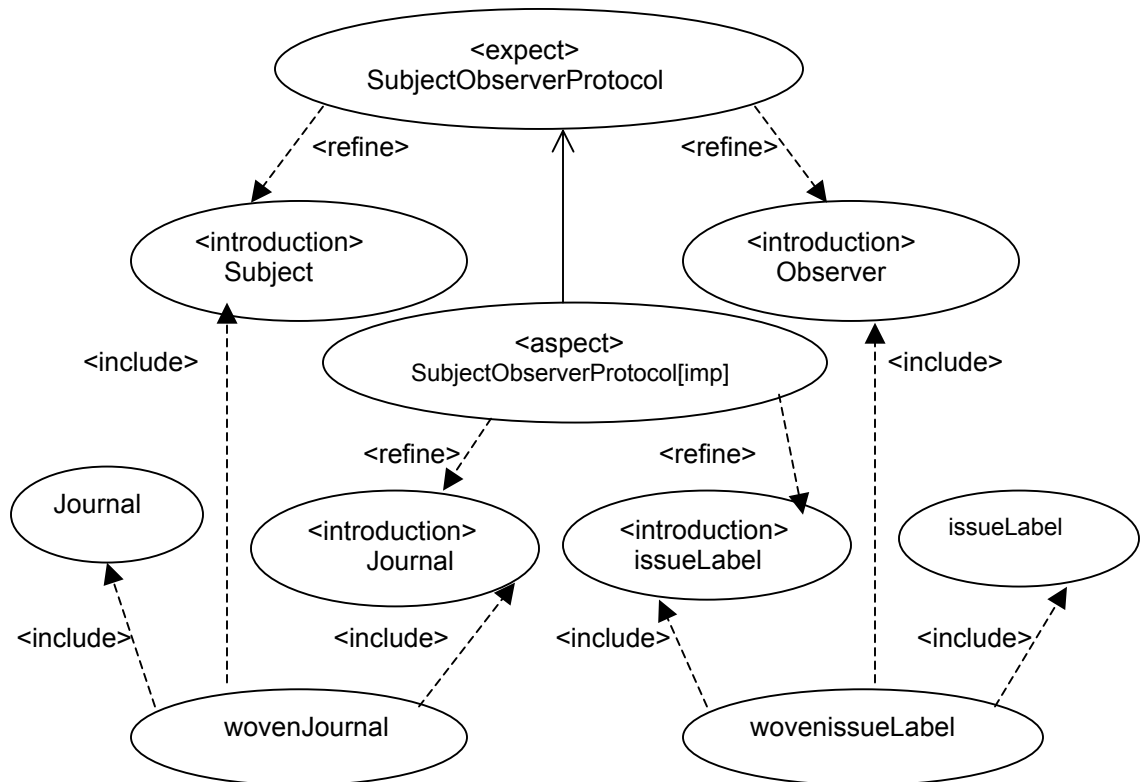
**Fig 5.8: Specifying Weaving Order**

how the included use cases cooperate in case of a before, after, or around advice to perform the behavior of the including use case (i.e., of the crosscut operation of the base classes). Special regards must be given to pieces of around advice and of advice that are attached to context-based pointcuts. In these cases, the woven use case is generated by means of extend relationships that precisely specify under which circumstances the behavior of the extending use case is to be performed. If an advice is attached to a context-based pointcut, for example, the extend relationship's condition reflects on the dynamic context in which extension has to take place. For an around advice, the condition generally states that extension shall be performed only if 'proceed' is called. Figure 5.7 illustrates how these conditions are expressed in UML use case diagrams.

The weaving process may lead to multiple collaborations. This is particularly likely in the case of dynamic crosscutting based on a join point's current execution context (i.e., when a piece of advice is attached to a context-based pointcut). Multiple collaborations may be needed also to describe all possible flows of control through an around advice. This means no conflict with the UML specification, though, as it explicitly allows the existence of multiple collaborations for a single use case.

### **5.5.5 Weaving Introductions**

Just like weaving of advice, the AOSDDL implements weaving of introductions with help of collaborations. Recall that introductions are represented in the AOSDDL as collaboration templates of stereotype <<introduction>>. Thus, weaving of introductions is realized by instantiating of the collaboration template in the base classes namespace. Before the instantiating the base classes (specified in template parameter's "base" tag) are supplemented with the features and relationships specified in the collaboration template so that the design model will not be ill-formed after the weaving process.



**Fig 5.9: Weaving Introductions with UML Use Cases**

Just like the weaving mechanism of advice, the weaving mechanism of introductions is represented in the AOSDDL in a more abstract manner using UML use cases. In Figure 5.9 for example, the use cases describing the aspects are refined into sets of (subordinate) use cases each specifying the behavior of one individual introduction contained in the aspects. These subordinate use cases (together with the use cases describing the base classes) are then included into new (woven) use cases describing the behavior of the woven (i.e., crosscut) base classes.

### 5.5.6 Weaving Relationship

The AOSDDL introduces a new relationship (named "<<crosscut>>") to the UML to signify the crosscutting effects of aspects on their base classes. This relationship is specified in imitation of the extend relationship that is already specified by the UML specification. It is no special stereotype of the extend relationship, though, since extend relationships may only exist between two use cases. Crosscut relationships, however, must connect other kinds of classifiers, as well (such as classes, interfaces, and aspects). Similar to extend relationships, the crosscut relationship is a directed relationship from one classifier (i.e., an aspect) to another classifier (i.e., a base class) stating that the former classifier affects the latter classifier (in the way that the former classifier is woven into the latter classifier). At the same time, though, the latter classifier remains independent from the former classifier (in the way that its implementation or functioning does not require the presence of the former classifier). Instead, the opposite is true. The crosscut relationship signifies that the former classifier (i.e., the aspect) requires the presence of the latter (i.e., the base class). These characteristics make (the extend relationship as well as) the crosscut relationship distinct from other relationships in the UML, such as the various kinds of dependency relationships. The crosscut relationship states further that the former classifier (i.e., the aspect) is woven into the latter classifier (i.e., the base class) according to the weaving mechanism described above. Note that crosscut relationships and weaving instructions are related to each other by a one-to-one mapping. So (provided with appropriate tool support), designers may specify the crosscutting effects of aspects either by drawing crosscut relationships or by specifying weaving instructions.

## **5.6 Identifying Software Concerns**

Separation of concerns (SOC) is a long-established principle in software engineering. It has received widespread attention in modern programming languages, with constructs such as modules, packages, classes, and interfaces, which support properties such as abstraction, encapsulation, and information hiding. SOC has also received attention in software architecture and design, with techniques such as composition filters and design patterns. While advances in all of these areas have had significant benefits, problems related to inadequate separation of concerns remain. This has led to work on "advanced separation of concerns", (ASOC), aspect-oriented programming, and multidimensional

separation of concerns. These bring a number of innovative ideas to programming in particular and to software development in general, which are now beginning to mature and coalesce under the heading of aspect-oriented software development (AOSD).

Although ASOC has been emphasized in recent work, concerns themselves have remained something of an ignored. Current ASOC tools provide only limited support for explicit concern modeling, representations of concerns tend to be tied to particular tools or artifacts, and concern modeling usually occurs just in the context of a particular type of development activity, such as coding or design. A global perspective on concerns, that spans the life cycle and is independent of particular development tools or artifacts, has been lacking.

#### **5.6.1 Concern Modeling Schema Framework**

During software development [18], concerns arise at all stages of the life cycle, from requirements specification, through design, coding, and testing, to maintenance and evolution. Concerns also span multiple phases of the life cycle, relate to multiple instances and types of artifacts, and crosscut phases and artifacts in different ways. Finally, concerns are dynamic and relative, that is, that the concerns relevant to a particular software unit will change over time and that they also depend on the perspective or purpose of the user or stakeholder who considers the software.

This implies that a general-purpose concern modeling framework that may be a part of the design language should support the representation of arbitrary concerns, the representation of composite concerns, support the association of concerns to arbitrary software units, work products, or system elements, language independent, methodology compatible, applicability across the software development life cycle, support the representation of arbitrary relationships among concerns and widely supported by various software engineering phases.

#### **5.6.2 Applications**

Concern modeling framework has many potential applications in software development. It provides a form of documentation for basic information about

concerns and their relationships. This kind of model can afford a global perspective that draws on, combines, and relates concerns from multiple work products and life cycle stages.

This kind of concern modeling framework that contains physical concerns (representing work products) and mapping relationships (that relate logical concerns to physical concerns) can serve as a semantic hyperindex that allows concerns to be traced into work products and development tasks. This supports traceability of concerns into and across work products and stages, and it makes it possible to see how concerns arise, are propagated, and possibly dropped across stages and iterations of the life cycle.

Mapping relationships further allow us to assess the impact on the physical level of changes on the logical level. For example, if we no longer care about robustness and lose that motivation for a concern such as logging, then we may be able to safely drop the software units that implement that concern. However, we may also find that a unit considered for deletion also contributes to other purposes (as logging may also support auditing) and so should be retained.

Another application is in organizing code (or other units) for purposes of concern-driven program composition.

## **5.7 Design Language Issues for Component Based software Development**

Component based software development (CBSD) and more recently, aspect-oriented software development (AOSD) have been proposed to tackle problems experienced during the software engineering process. When applying CBSD, a full-fledged software-system is developed by assembling a set of premanufactured components. Each component is a black-box entity, which can be deployed independently and is able to deliver specific services. The deployment of this paradigm drastically improves the speed of development and the quality of the produced software. AOSD on the other hand, tries to improve the separation of concerns in current software engineering methodologies, by providing an extra separation dimension along which the properties of a software-system can be described.



Currently available AOSD-research mainly focuses on object oriented software development (OOSD). CBSD however, also suffers from the problems that arise with the tyranny of the dominant decomposition. Similar to OOSD, aspects such as synchronization and logging are encountered, which crosscut several components from which the system is composed. Consequently, the ideas behind AOSD should also be integrated into CBSD. The other way around, namely the integration of CBSD within AOSD, is a valuable concept as well. CBSD puts a lot of stress on the plug-and-play characteristic of components; for example, it should be possible to extract a component from a particular composition and replace it with another one. Introducing a similar plug-and-play concept in AOSD, would make aspects reusable and their deployment easy and flexible.

Combining the AOSD and CBSD principles is a valuable contribution to both paradigms. However, currently available AOSD and CBSD research cannot be straightforwardly integrated, this because of several restrictions which are imposed by the existing approaches:

- The deployment of an aspect within a software-system is at this moment rather static. In AspectJ for example, an aspect loses its identity when it is integrated within the base-implementation of a software system. This makes it very difficult to extract an aspect from a particular composition and to replace it afterwards with a totally different aspect. This plug-and-play property is vital in some environments where the dynamic characteristic of components is considered an essential requirement.
- Most AOSD approaches describe their aspects with a specific context in mind. Therefore, it is impossible to reuse aspects. This is not acceptable within CBSD, since every component of a software-system should be independently deployable.
- The communication between the various components from which an application is composed, is in most cases specific to the employed component model. Java Beans for instance, makes use of an event-model.

Currently available AOSD-technologies however, are not suited to deal with these specific kinds of interactions.

To integrate the ideas of AOSD into CBSD, we need a new aspect-oriented implementation language, designed especially for CBSD. This language should enable the development of software along another separation dimension, on top of the Java class hierarchy. It stays as close as possible to the regular Java syntax and introduces two concepts: aspect beans and connectors. An aspect bean is a regular Java bean that is able to declare one or more logically related hooks, as a special kind of inner classes. Hooks are genetic and reusable entities and can be considered as a combination of the AspectJ's pointcut and advice. Since aspect beans are described independent from a specific context, they can be reused and applied upon a variety of components. The initialization of a hook with a specific context is done by making use of connectors.

To make such a language operational, we need a new component model that already incorporates the necessary traps to enable dynamic aspect application and removal. Another advantage of this new component model will be that component developers are still able to guarantee QOS for their components. However, the dynamicity and flexibility gained by using this new component model comes with a price in the form of large performance overhead compared to static languages, like for example AspectJ. As a consequence, this approach can be limited in use where limited resources is an issue.

## **5.8 Mapping UML extensions through composition patterns to Aspects**

Requirements such as distribution or tracing have an impact on multiple classes in a system and are described, in general, as cross-cutting requirements, or aspects. Scattering and tangling make object-oriented software difficult to understand, extend and reuse. Though software design is an important activity within the software lifecycle with well-documented benefits, those benefits are reduced when cross-cutting requirements are present. One approach to mitigate these problems is by separating the design of cross-cutting requirements into *composition patterns*.

Composition patterns require extensions to the UML, and are based on a combination of the subject oriented model for composing separate, overlapping designs, and UML templates. We also show how composition patterns map to one programming model that provides a solution for separation of cross-cutting requirements in code—aspect-oriented programming. This mapping serves to illustrate that separation of aspects may be maintained throughout the software lifecycle.

A composition pattern is a design model that specifies the design of a cross-cutting requirement independently from any design it may potentially cross-cut, and how that design may be re-used wherever it may be required. Composition patterns are based on a combination of the subject-oriented model for decomposing and composing separate, potentially overlapping designs, and UML templates.

### **5.8.1 Mapping To AspectJ**

At the conceptual level, composition pattern design and aspect-oriented programming also have the same goals. Composition patterns provide a means for separating and designing reusable cross-cutting behaviour, and aspect-oriented programming provides a means for separating and programming reusable cross-cutting behaviour. The advantages of this are two-fold. First, from a design perspective, mapping the composition pattern constructs to constructs from a programming environment ensure that the clear separation of cross-cutting behaviour is maintained in the programming phase, making design changes easier to incorporate into code. Secondly, from the programming perspective, the existence of a design approach that supports separation of cross-cutting behaviour makes the design phase more relevant to this kind of programming, lending the standard benefits of software design to the approach.

## **5.9 AspectJ Extensions for Distributed Computing**

Current programming systems do not provide mechanisms for modularizing crosscutting concerns in distributed systems and thus they are major sources of low readability and maintainability of the software [66]. Issues like transactions,

security, and fault tolerance are typical crosscutting concerns in distributed systems.

Many crosscutting concerns also arise during unit testing of distributed systems. The code for unit testing includes typical crosscutting concerns that AspectJ can deal with. AspectJ is a widely used language for aspect-oriented programming (AOP) in Java. Unfortunately, if we use AspectJ to modularize testing code for distributed software, the code ("aspect") can be somewhat modular but it often consists of several sub-components distributed on different hosts. They must be manually deployed on each host and the code of these sub-components must include explicit network processing among the sub-components for exchanging data since they cannot have shared variables or fields. These facts complicate the code of the aspect and degrade the benefits of using aspect oriented programming.

### **5.9.1 Implications on Network Processing**

AspectJ is a useful programming language for developing distributed software. It enables modular implementation even if some crosscutting concerns are included in the implementation. However, the developers of distributed software must consider the deployment of the executable code. Even if some concerns can be implemented as a single component ("aspect") at the code level, it might need to be deployed on different hosts and it would therefore consist of several sub components or sub-processes running on each host. Since Java (or Aspect J) does not provide variables or fields that can be shared among multiple hosts, the implementation of such a concern would include complicated network processing for exchanging data among the sub components.

Programming frameworks such as Java RMI do not solve this problem of complication. Although they make details of network processing implicit and transparent from the programmers' viewpoint, the programmers still must consider distribution and they are forced to implement the concern as a collection of several distributed sub-components exchanging data through remote method calls. The programmers cannot implement such a concern as a simple, non distributed monolithic component without concerns about network processing. This is never

desirable with respect to aspect orientation since it means that the programmers must be concerned about distribution when implementing a different concern.

### **5.10 Summary**

AOSDDL attempts to reproduce the semantic of AspectJ in the UML. It provides suitable representations for all components of an aspect (such as join points, pointcuts, pieces of advice, and introductions) as well as for the aspect itself. These representations are extended from existing UML concepts using the standard UML extension mechanisms. This way, aspects may be fully specified in concise units in an UML design model, thus carrying over the advantages of aspect-oriented modularity (such as higher comprehensibility, adaptability, and reusability) to the design level.

Concerns represent the "matters of interest" in a software system, and they arise and pertain throughout the software life cycle. The concern space models have many applications. Generally they embody knowledge about a software system and its components and in effect provide a semantic hyper-index into work products and other resources. This information can support of many software development tasks, such as rationale capture, impact analysis, change propagation, and software composition and decomposition. These tasks are useful in initial system development but are especially important for "downstream" software processes such as maintenance, extension, adoption, customization, integration, and reuse. Concern-space modeling is already being applied within individual tools; sooner it will eventually provide a framework for integrated software development environments.

To integrate the ideas of AOSD into CBSD, we need a new aspect-oriented implementation language, designed especially for CBSD. To make such a language operational, we need a new component model that already incorporates the necessary traps to enable dynamic aspect application and removal. However, the dynamicity and flexibility gained by using this new component model comes with a price in the form of large performance overhead compared to static languages, like for example AspectJ. As a consequence, this approach can be limited in use where limited resources is an issue.

Software design is an important activity in the development lifecycle but its benefits are often not realized. Scattering and tangling of cross-cutting behaviour with other elements causes problems of comprehensibility, traceability, evolvability, and reusability. Attempts have been made to address this problem in the programming domain but the problem has not been addressed effectively at earlier stages in the lifecycle. Composition patterns presents an approach to addressing this problem at the design stage.

AspectJ extensions for identifying join points in the execution of a program running on a remote host can simplify the description of aspects with respect to network processing if the aspects implement a crosscutting concern spanning over multiple hosts.

Further, this chapter has presented the design of AOSDDL, an aspect oriented design language. The primary focus of this design language is to provide a highly flexible and extensible set of notations suitable for aspect oriented software development in all real world scenarios, suitable as a research platform for aspects that can form the basis for further research into aspect oriented systems and software engineering in general.

The following chapter continues this thesis with an overview of the AOSDDL prototype implementation and its evaluation. Due to the broad scope, the prototype implementations serve mainly as proof of concept for key mechanisms and design decisions of AOSDDL form.

## **CHAPTER 6**

### **Implementation and Evaluation**

#### **6.1 Overview**

This chapter describes the ongoing efforts to engineer a prototypical realization of the AOSSDL design language. The previous chapter has presented the design and notations of this language. Due to the extent of the AOSDDL structure, the prototyping implementations focus primarily on validating the key aspects of the language by implementation.

Also present in this chapter is the evaluation of the AOSSDL structure and the prototype implementation as described in the previous chapter and this chapter.

Since the main objective of this work was to design an aspect oriented software development design language from the ground up, it has not been feasible to fully realize such a language. As a consequence, the evaluation of the AOSDDL structure is to a large extent a theoretical analysis.

#### **6.2 Processing and Test Environments**

The operating systems used is Windows XP edition. The Eclipse Platform for Java [68, 69, 70, 71, 72, 73, 74, 75] was used to carry out the implementation and testing of the abstract notations in AspectJ. To implement graphical notations and diagrams the Together CASE tool [76] was used. Moreover, the implementation tests can also be conducted under Linux environment by the very nature of it being open source.

#### **6.3 Mapping Learning Resource Center (LRC) Design to Aspects**

We now look at an example application problem that demonstrates implementation of aspects for a learning resource center that provides services to its customers in the form of periodicals, books, newsletters and magazines. It shows how cross-cutting requirements may be designed independently of any base design, making aspect design truly reusable.

This learning resource center library has various resources (books) of which all copies are located in the same room and shelf. An Information officer handles the maintenance of the association between these resources and their locations. The Information officer also maintains an up-to-date view of the lending status of copies of books, periodicals and journals.

### **6.3.1 Functional Decomposition**

Based on the problem definition above the following types of services will be required:

**Finding resources on a topic** An search operation that takes partial or incomplete description of the resources and lists accurate matches in the form of books or journals.

**Getting list of Journals** An search service that lists all the journal issues for a particular publishing month (quarterly / bi-monthly etc). The user will be able to filter the selections based on the criteria specified regarding the journal details.

**Upcoming Issues** This will list the future issues for publication and the issue month.

**Authentication** Before providing any kind of services the application needs to verify the credentials (username and password) to it.

**Logging** A logging function requirement to keep track of the calls made to the the webservices. These kind of features are useful to track the preferences of cardholders and resources in demand type of statistics from within the application.

### **6.3.2 Design Diagrams for Learning Resources Center (LRC)**

Figure 6.1 shows the LRC design and figure 6.2 shows the design hierarchy diagram for the learning resource center(LRC). To avoid line cluttering on the hierarchy diagram, we have omitted a few dependencies.



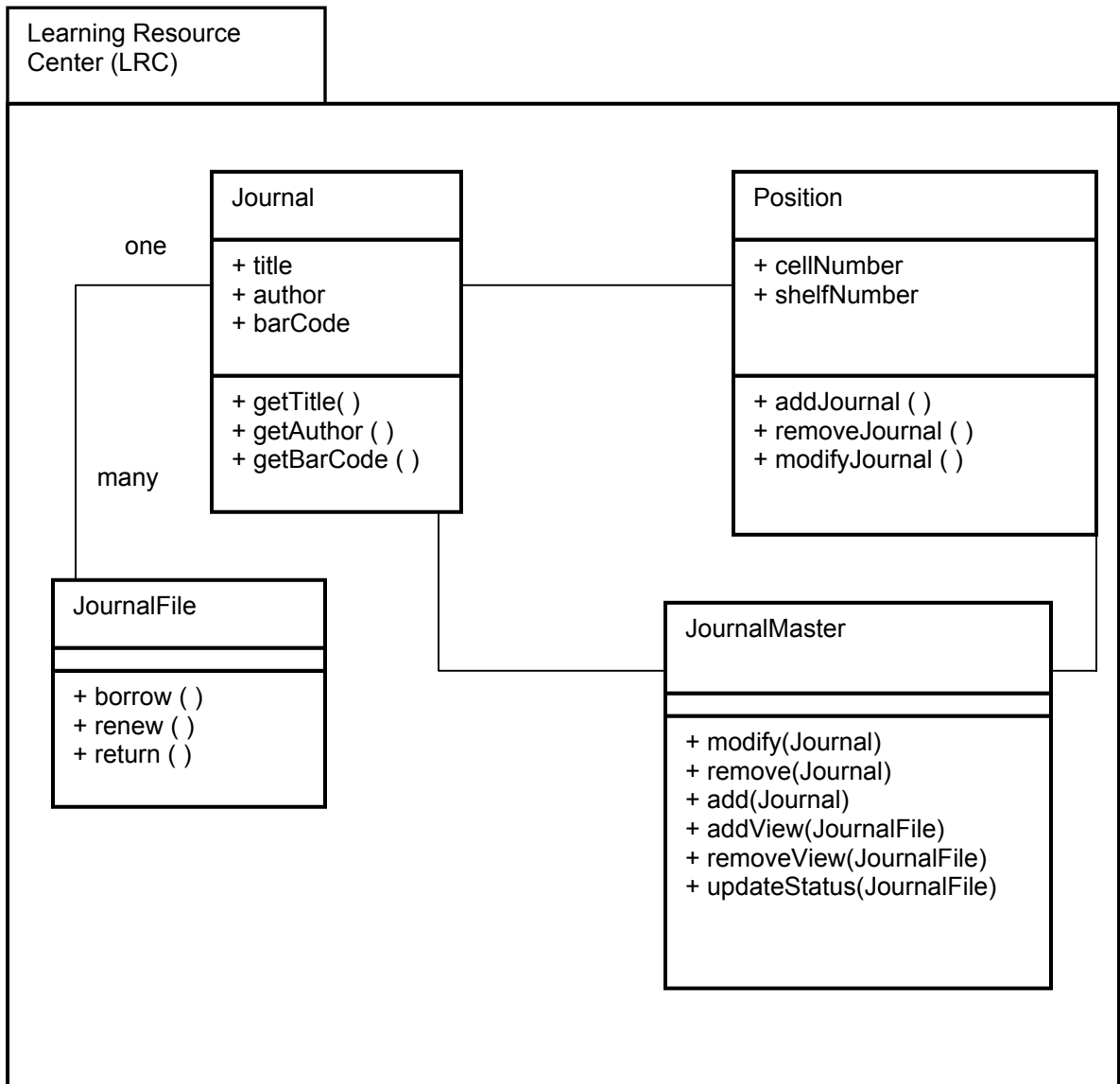


Figure 6.1: Learning Resource Center (LRC) Design

### 6.3.3 Aspects Modularization and Dependency Effects of Aspects

We perform two forms of modularization namely, logging and authentication using aspects to remove dependencies between the modules. Using *pointcut-advise* mechanism we remove the dependencies between the various modules. A *logging aspect* that captures the calls to the webservices directly from the design rules for *JournalFind*. The logging aspect module hooks these calls with the module *WebServicesLogger*. Secondly, for *authentication* we use *introductions* to inject the

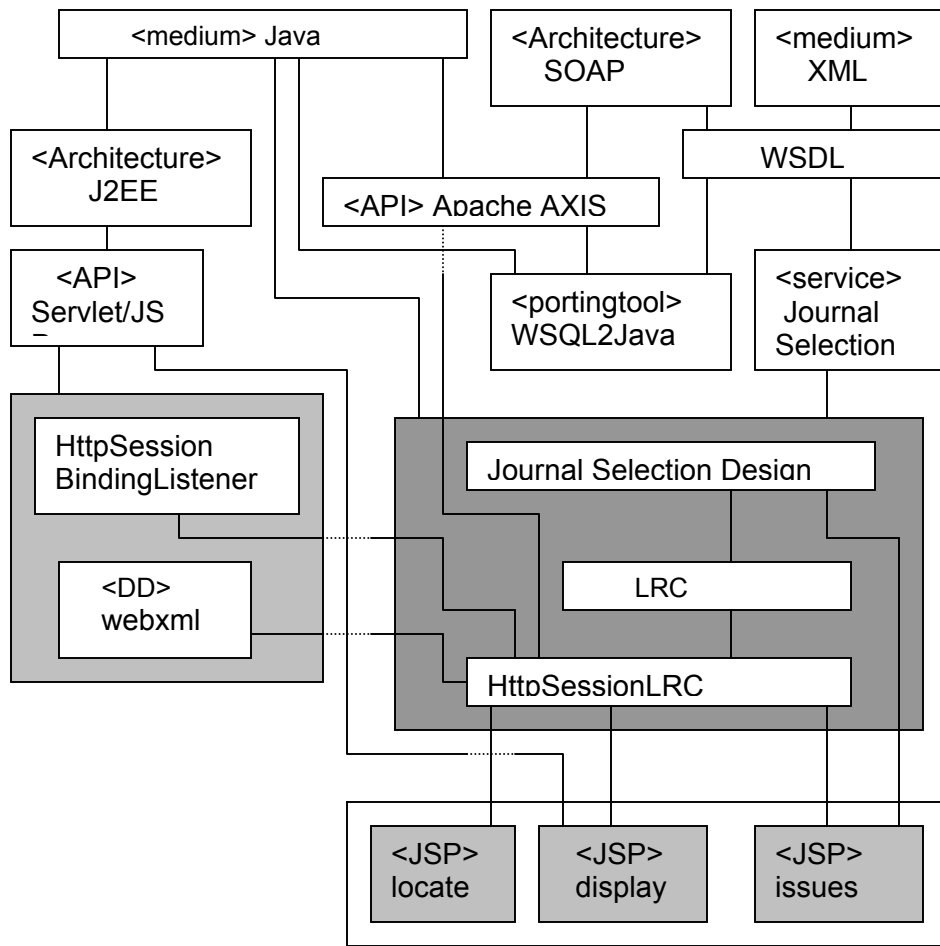


Figure 6.2: Hierarchy Diagram for LRC

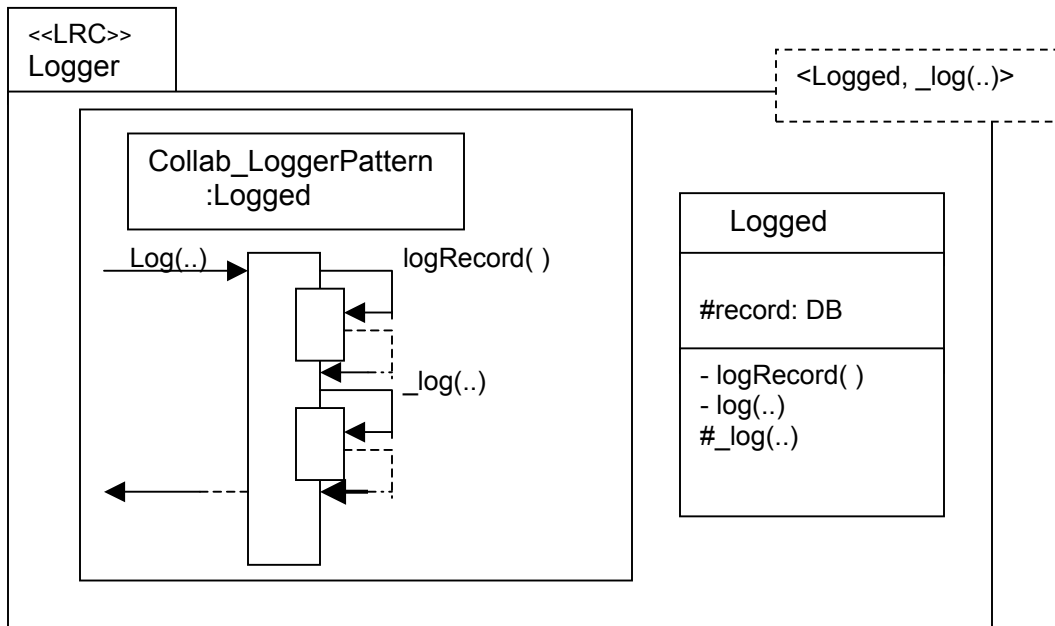
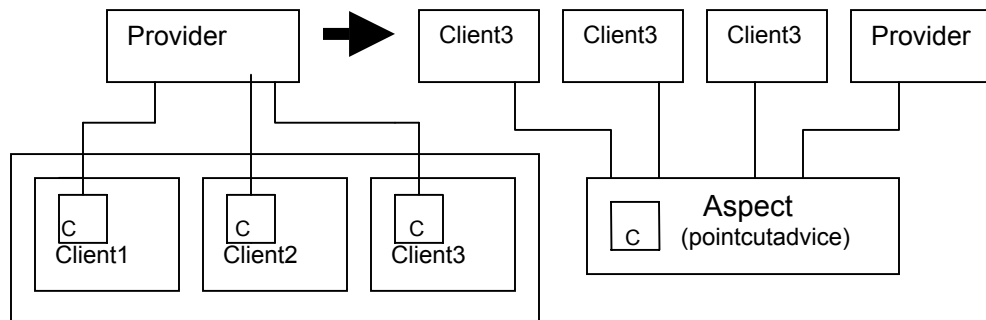


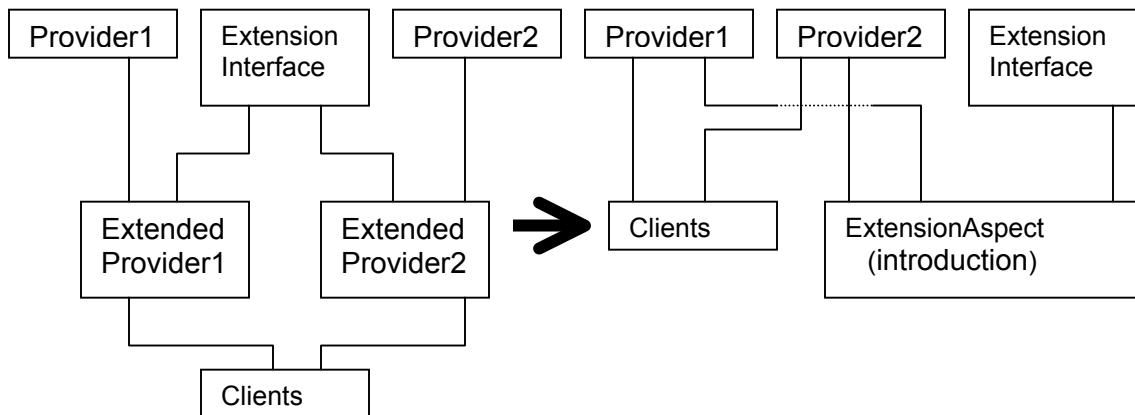
Figure 6.3: AOSDDL's Authentication functionality

authentication specific functionality into the application modules. This will result in another aspect *Authentication*, in the final design shown in figure 6.3.

The hierarchy diagrams in figure 6.4 and figure 6.5 show the effects that aspects have on module dependencies. Figure 6.4 models the design change that was made to perform aspect oriented modularization to logging and Figure 6.5 models the same for authentication. Aspect oriented mechanisms eliminate the dependencies clients have on providers by introducing aspects as new modular structures. Aspects depend on these clients and providers, and are responsible to provide connections between them. Aspect oriented modularization with introductions is similar to the module with injected dependency.



**Figure 6.4: Effect of aspects with pointcut-advise on dependencies**

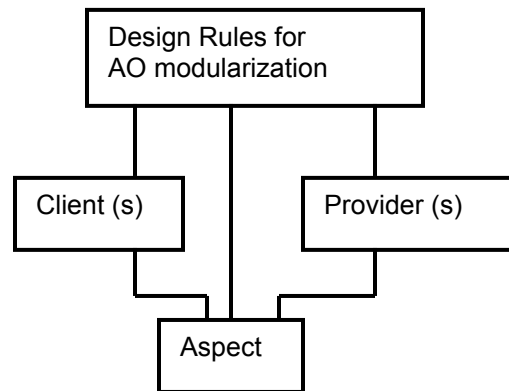


**Figure 6.5: Effect of aspects with introduction on dependencies**

### 6.3.4 Designing Aspects

Figures 6.4 and 6.5 are just one of several variations of pointcut-advise and introduction mechanisms. Particularly, in these figures we do not see what the visible design rules for aspects are. In both cases aspects depend on clients or/and

providers. In Figure 6.4, a small box labeled as C denotes the common points in clients accessing the providers. C is moved into the aspect after aspect oriented modularization and it represents two things: (i) interfaces that a provider provides, and (ii) points in clients that access such interfaces. A typical way to design aspects following this process (as in AspectJ) is to capture these points as joinpoints, (for example the method names a provider provides and the method names of clients that access the provider), that need to be advised. Such joinpoints constitute C, and, in a way, become design rules for the aspect. Defining design rules for aspects implies making such joinpoints explicit. Just as architectural modules emerge after sustaining a considerable design evolution, an aspect oriented design would also result in well defined design rules for aspect oriented modularization, as in the structure shown in Figure 6.6.

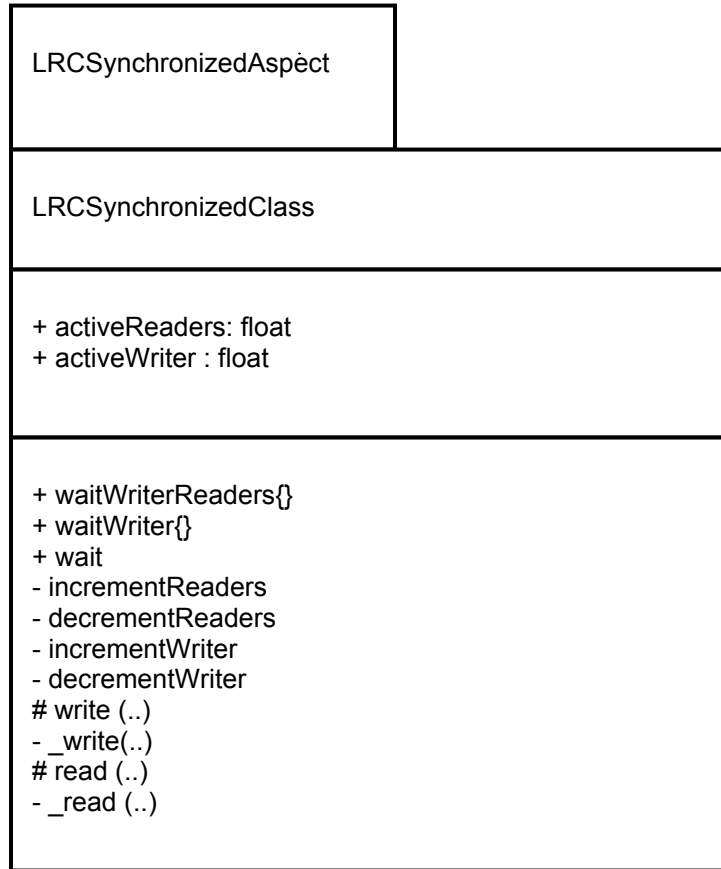


**Figure 6.6: Design rules for Aspect Oriented Modularization**

### 6.3.5 Crosscutting Requirements: Aspects

Synchronizing the aspect is the first cross-cutting requirement that requires that the journal master should handle several requests to manage journals and their locations concurrently. This aspect example, first supports the journal master handling several “read” requests concurrently, while temporarily blocking “write” requests. Individual “write” requests should block all other services.

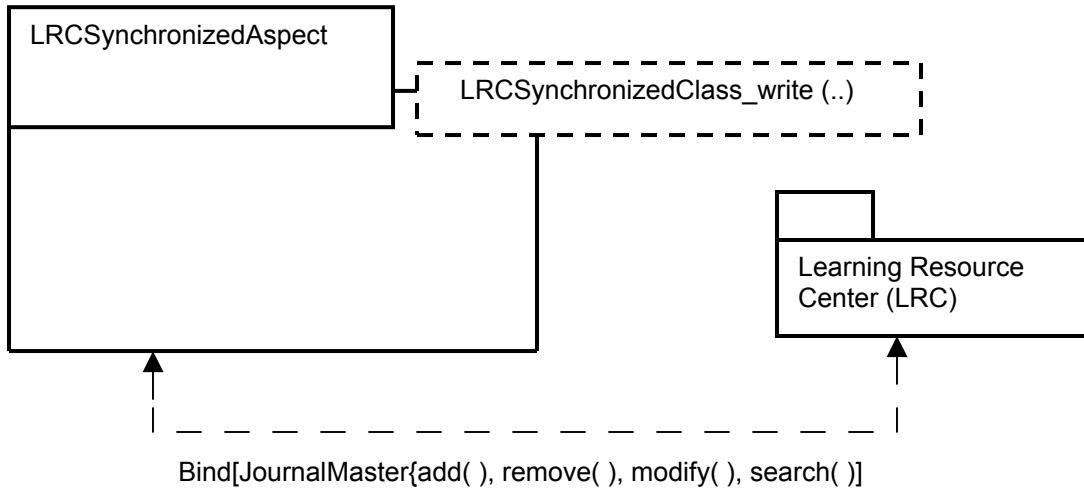
Synchronization of concurrent processes is a common requirement, and therefore it is useful to design this behaviour without any reference to our LRC example. Fig. 6.7 illustrates how this can be achieved. The Synchronize composition pattern has one pattern class, LRCSynchronizedClass, representing any class requiring synchronization behaviour.



**Figure 6.7: LRC Synchronization for Aspect Design**

Specifying how to compose the LRC base design subject with the Synchronize composition pattern is a simple matter of defining a composition relationship between the two, denoting which class(es) are to be supplemented with synchronization behaviour, and which read and write operations are to be synchronized.

In this case, the LRC’s JournalMaster class replaces the pattern class in the output, modify(), add() and remove() operations are defined as write operations, and the search() operation defined as read (see Figure 6.8).



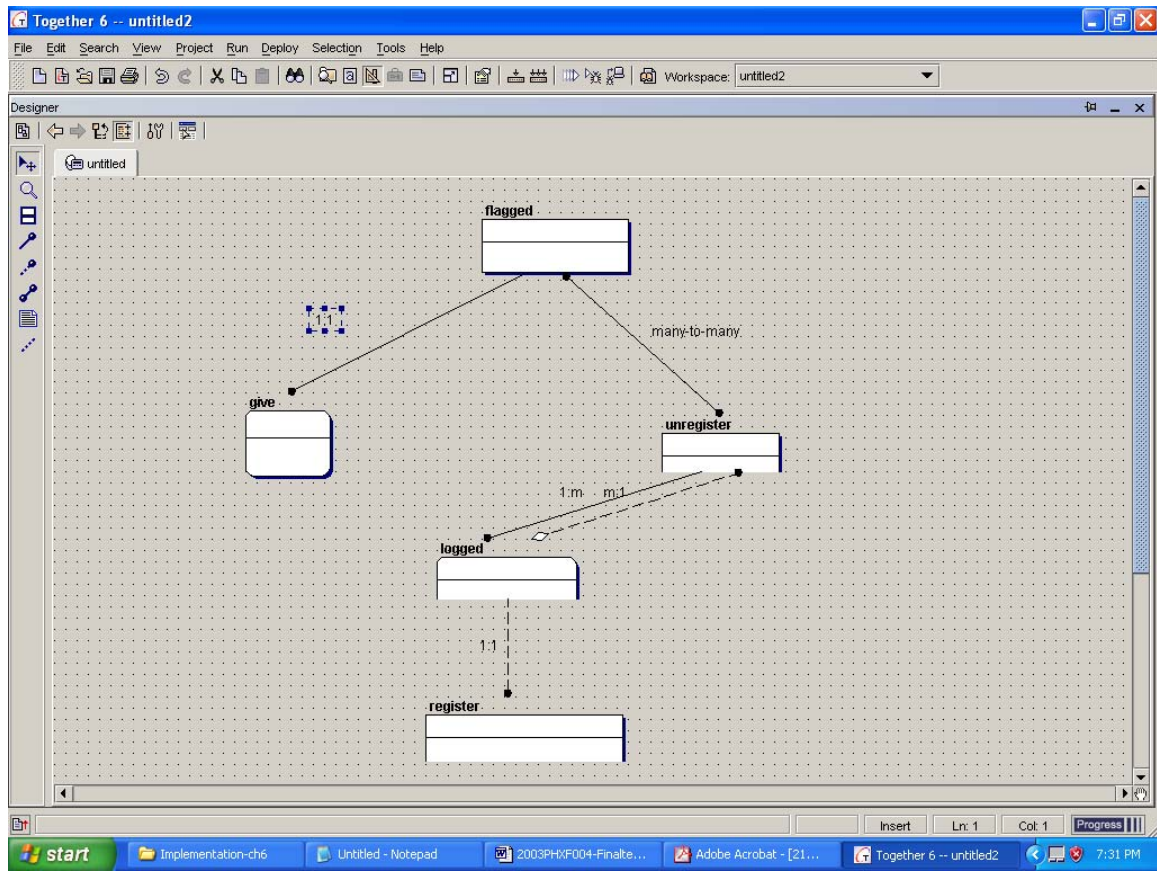
**Figure 6.8: Synchronization of Learning Resource Center (LRC)**

Pattern specification and binding, is all the designer has to do to define truly reusable aspects patterns, and specify how they are to be composed with base designs.

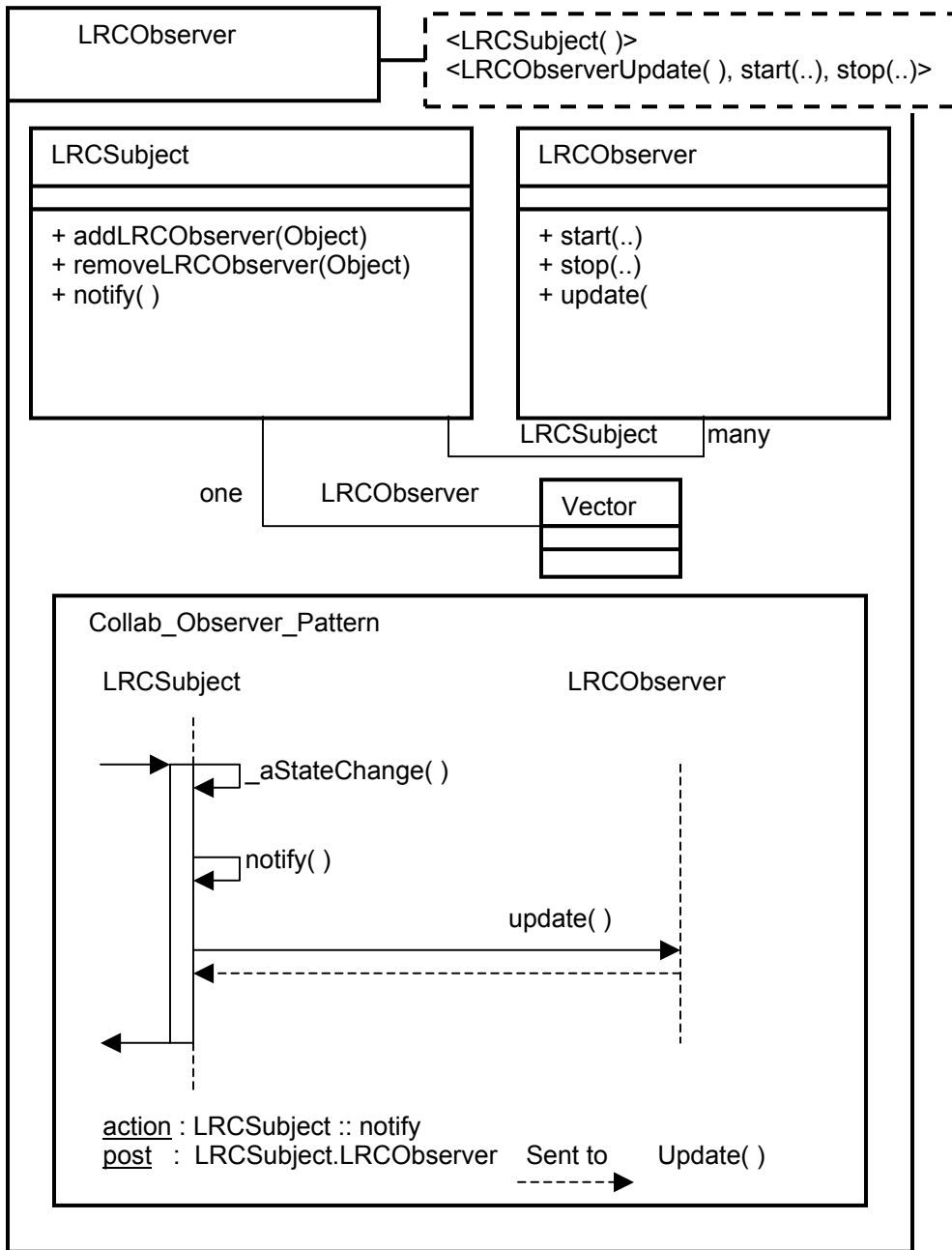
AOSDDL approach reflects on the various aspects of the analysis in terms of effectiveness at support for aspect identification, requirements coverage, traceability and scalability. Effective support exists for determining the binding order for multiple crosscutting aspects. This may be otherwise difficult to locate. The supplementary requirements studied to determine whether any of their minor actions should be enhanced to major, or whether to group those requirements. Further, all actions could be turned into design aspects and hence scalability was efficient. Traceability also followed from and to aspect design.

### 6.3.6 Designing LRCObserver Aspect

In the Observer composition pattern, two pattern classes are defined. LRCSubject is defined as a pattern class representing the class of objects whose changes in state are of interest to other objects, and LRCObserver is defined as a pattern class representing the class of objects interested in a Subject's change in state (see Fig. 6.9).



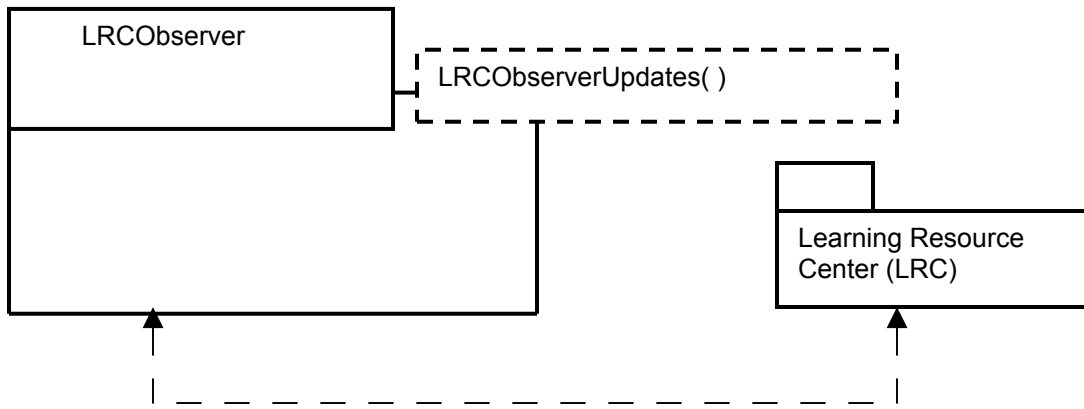
The interaction in Fig. 6.9 illustrates specifying behaviour that cross-cuts templates, with Subject's template parameter `_aStateChange(..)` supplemented with behaviour relating to notifying observers of changes in state. Again, this achieved by referring to the actual replacing operation with a prepended "`_`", i.e., `_aStateChange(..)`. Here also is an example of an operation template parameter that does not require any delegating semantics. The `update()` operation in observers is simply called within the pattern, and is not, itself, supplemented otherwise. It is defined as a template so that replacing observer classes may specify the operation that performs this task. This pattern also supports the addition and removal of observers to a subject's list using `_start(.., Subject, ..)` and `_stop(.., Subject, ..)` template parameters, where each is replaced by operations denoting the start and end, respectively, of an observer's interest in a subject. For space reasons, the interactions are not illustrated here, as they do not illustrate any additional interesting properties of the composition pattern model.



**Figure 6.9: Aspect Design for LRCObserver**

As with the Synchronize pattern, specifying the composition of Library with the LRCObserver pattern is done by specifying a composition relationship between the two, defining the class(es) acting as LRCsubject, and the class(es) acting as LRCobserver. In this example, there is only one of each (see Fig. 6.10), JournalFile and JournalMaster, respectively.



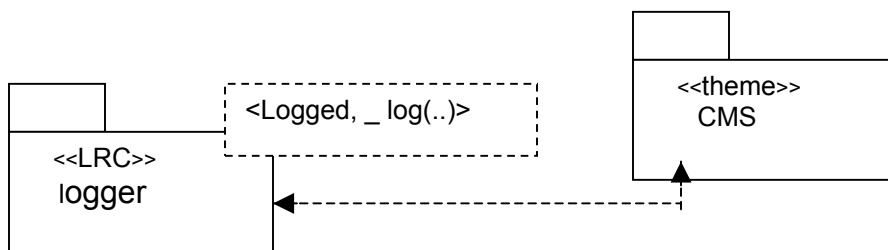


```
bind[<JournalFile(query = false)>,
      <JournalMaster(updateStatus( ), addView( ), removeView( ), modifyView( )>]
```

**Figure 6.10: Design Aspect for LRCObserver**

The output of composing LRCObserver with Learning Resource Center, will show JournalFile demonstrating subject behaviour, with the operations borrow() and return() initiating the notification of observers, as they are the only state-changing operations. JournalMaster, as an observer, has defined updateStatus() as the operation to be called for notification purposes. Operations addView(..), modifyView(..) and removeView(..) initiate a JournalFile adding and removing a JournalMaster from its list of observers.

Similarly, the design Aspect for Authentication (LRCLogger) is shown in figure 6.11.



```
Bind[<{Person,Student,Professor},
      {Student.register( ), Person.unregister( ), Professor.issue( )}]
```

**Figure 6.11: Design Aspect for Authentication (LRCLogger)**

## 6.4 Testing

In our work, we present a state-based approach to the incremental testing of aspect-oriented programs, which addresses the following research issues:

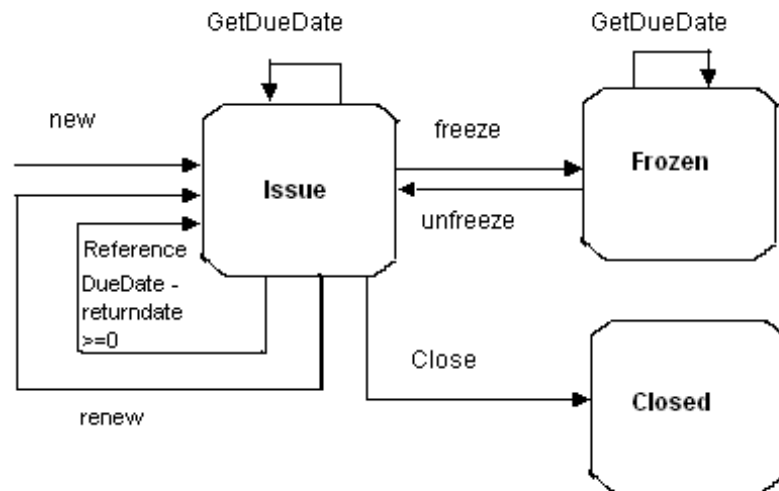
- How to specify the expected impact of aspects on object states for test generation purposes?
- To what extent can base class tests be reused for testing aspects? Base class tests are not necessarily valid for testing aspect-oriented programs as aspects are likely to change transitions of object states.
- How to determine that a programming fault actually has to do with aspects rather than base classes?

To capture the expected impact of aspects on the states of base class objects, we exploit aspect-oriented state models, an aspect oriented extension to state models with testability, for specifying base classes as well as aspects. We compose state models of aspects and base classes by an explicit weaving mechanism and generate abstract test cases from state models for an aspect oriented program and the corresponding base program. Taking aspects as incremental modifications to their base classes, we identify how to reuse the concrete base class tests for testing aspect-oriented programs according to aspect-oriented state models. Such an incremental approach to testing aspect-oriented programs can significantly reduce testing cost for two reasons: (1) it reuses test cases, the development of which is often an expensive investment; and (2) it helps localize programming problems by identifying aspect-specific faults. For instance, if the base classes of an aspect-oriented program pass all of the state based tests but the aspect-oriented program as a whole fail some of the tests, the failure would have to do with aspects.

## **6.5 Defining Validation Parameters for Aspects**

We first formally define extended state models as a basis for class and aspect specification to describe aspect oriented state models for generating test cases. Objects are encapsulated entities of data and operations that can receive messages from and send messages to other objects. Constraints often exist on the sequence of messages that can be accepted by objects. As these constraints are typically related to object states, state models are a common approach for capturing object behaviors, especially intra-class behaviors. In the following, we extend traditional finite state models as a basis for aspect-oriented state models.

For example, fig. 6.12 and the listing shows the state model and public interface of the *Journal* class, respectively. For clarity, we use  $d$  to denote the instance field *duedate* and assume that  $\text{returndate} \leq \text{duedate}$  is a precondition for methods  $\text{return}(\text{returndate})$  and  $\text{payfine}(\text{amt})$ . Transition  $(\text{Issue}, \text{payfine}, \text{duedate} - \text{returndate} \geq 0, \text{Issue})$  means that method call  $\text{payfine}(\text{amt})$  with condition  $\text{duedate} - \text{returndate} \geq 0$  under state *Issue* does not change the state.



**Figure 6.12: The state model of class *Journal***

```

public class Journal {
// constructor, or the new operator
public Journal();
// indicating instance field duedate – d for short
public double getDuedate();
public void renew(date);
public void issue(date);
public void freeze();
public void unfreeze();
public void close();
}
  
```

**Listing for the interface of class *Journal***

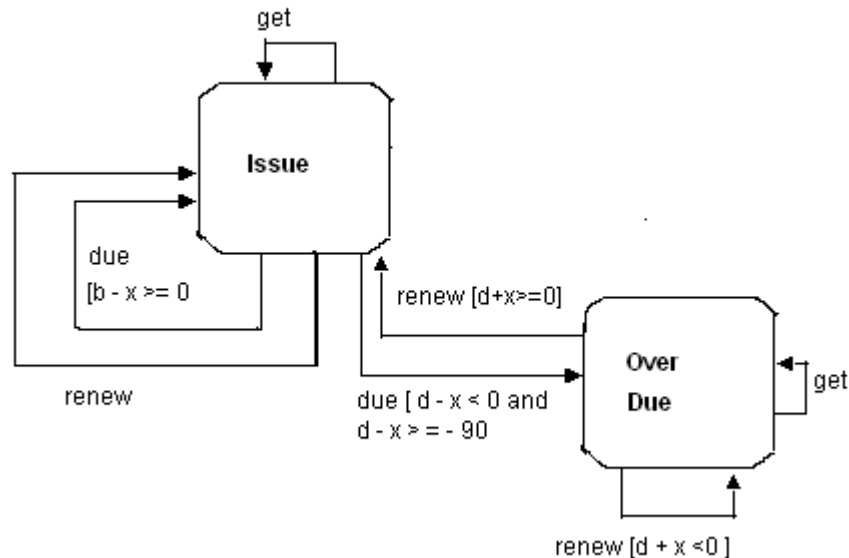
We incorporate aspect-orientation into state models by following the fundamental concepts of AOP, such as aspects, join points, pointcuts, and advices. In our approach, join points can be states, events, or variables in a state model; a pointcut picks out a group of join points; advices are specified as a state model; and an aspect is an encapsulated entity of pointcuts and advice model.

We build a model for each aspect. Fig. 6.13 shows the aspect model. *Overdue* that enforces a new resource center policy for the base class *Journal* in fig.6.12. Although it can crosscut other account (e.g. credit card) classes, for simplicity, we specify it only with respect to *Journal*. The *Overdue* aspect allows *one* overdue periodical as long as the due date is less than three months. In the aspect, the states are *Issue* (a different name can be used, though) and *Overdue*, where *Issue* is corresponding to the *Issue* state in the base model and *Overdue* is a new state. The events are *return*, *due* and *get*, which are corresponding to *return*, *renew* and *getBalance* in the base model, respectively. The variables used to represent guard conditions are *x* and *b*, which are corresponding to *amt* and *d* in the base model, respectively. Note that the aspect is an addition to the base model as all the transitions from *Issue* to *Issue* in the base model remain unchanged.

```

aspect Overdue
state pointcut Issue: Journal.Issue
event pointcut get: Journal.getBalance
event pointcut due(x): Journal.renew(date)
event pointcut return(x): Journal.return(date)
variable pointcut d: Journal.d

```



**Fig 6.13: The Overdue Aspect**

The general process of our approach to incremental testing of an aspect-oriented program is as follows: (1) build the state models of the base classes; (2) generate

abstract test cases from the base models; (3) instantiate the abstract test cases to form concrete test suites for the base classes; (4) test the base classes; (5) build aspect models and weave them into the base models; (6) generate abstract test cases from the woven state models; (7) generate test suites for the aspect-oriented program as a whole by reusing, modifying, and extending concrete base class test cases and instantiate new abstract test cases; and (8) test the aspect-oriented program. Of course, we can combine step (5) into step (1), that is, build complete aspect-oriented models before testing base classes.

## **6.6 Detecting Aspect Faults**

A great variety of aspect-specific faults may exist in aspect oriented programs. Examples include pointcut expressions picking out extra join points, pointcut expressions missing certain join points, incorrect advice types, and incorrect advice implementation. In this section, we discuss how these faults would affect object states and how they can be revealed by the state-based testing approach. The incremental testing approach is similar to traditional regression testing. The essential difference is that, aspects as a structured way to specify modifications make it feasible to investigate systematic reuse and modification of the existing tests. Our approach can be adapted to the UML class diagrams and start charts by using class interfaces, flattening start chart diagrams, and following the convention of guard conditions.

## **6.7 Evaluation**

In general there are two methods used for the evaluation of research contributions namely, qualitative and quantitative evaluation. However, since the contributions here are mostly concerned with the use of the notation and its implementation, a qualitative evaluation of the concepts and design of the design language will be more meaningful.

Since it has been feasible to implement only a subset of the overall design language architecture, a quantitative evaluation of the entire system cannot be provided at this stage.

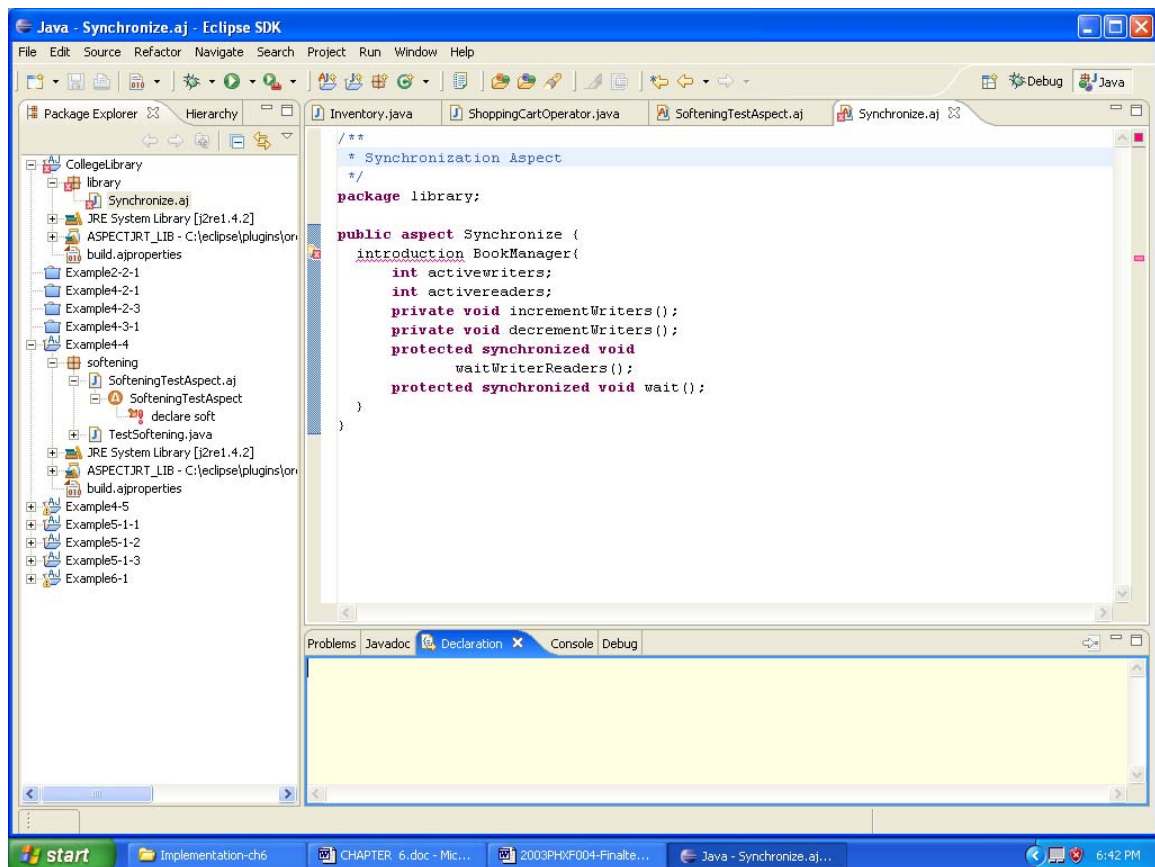
## **6.8 Qualitative Evaluation: Mapping To AspectJ**

This section presents a qualitative evaluation of the AOSDDL design language. Firstly, a case study used to evaluate the features and usability of the design language is examined.

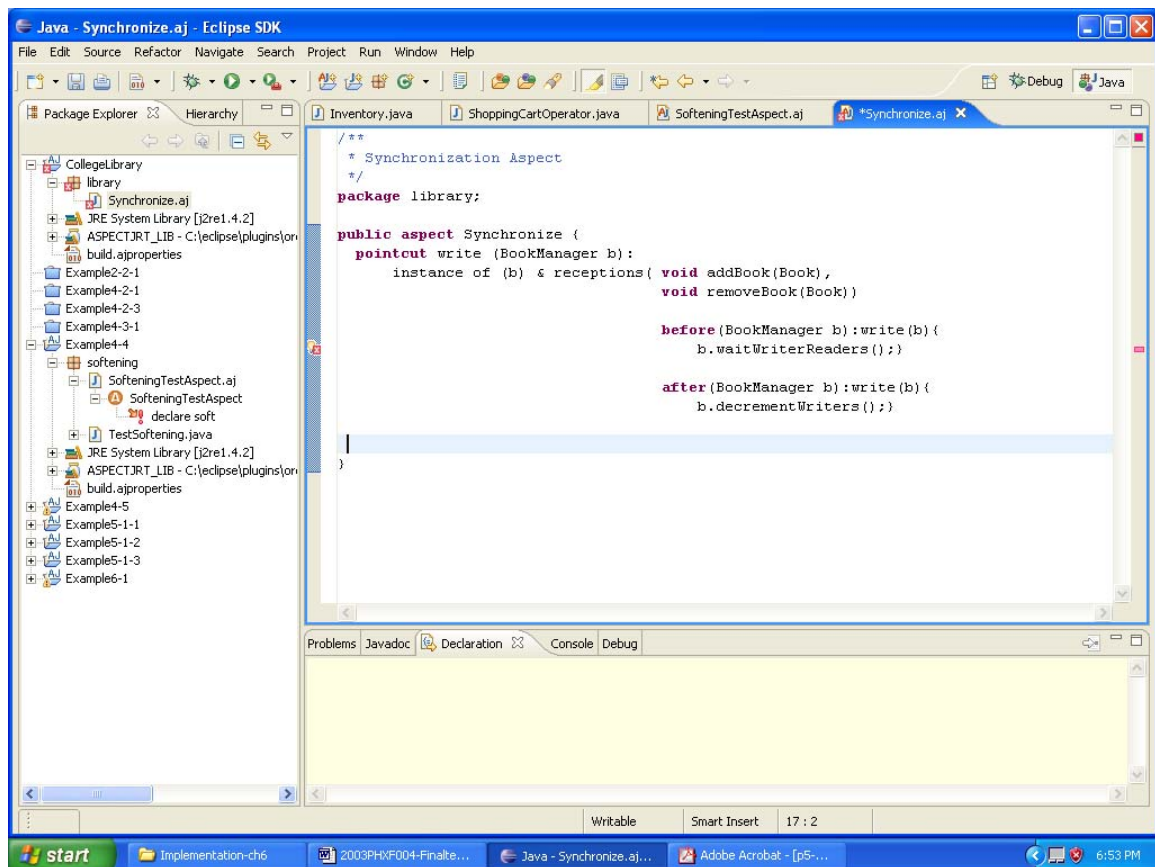
At the conceptual level, composition pattern design and aspect-oriented programming also have the same goals. Composition patterns provide a means for separating and designing reusable cross-cutting behaviour, and aspect oriented programming provides a means for separating and programming reusable cross-cutting behaviour. This section introduces possibilities for mapping composition pattern constructs to current aspect-oriented programming constructs. Research into, and development of, technology support for the aspect-oriented programming paradigm is currently centered around AspectJ, and so, using the synchronization example, we assess how composition patterns map to AspectJ programming constructs.

#### **6.8.1 Synchronize in AspectJ**

The Synchronize composition pattern (Fig. 6.7) with its composition specification to the learning resource center subject (Fig. 6.8) provides the information required for the structure of an aspect program. The composition pattern has one class defined, which is a pattern class, and therefore is replaced with a concrete design class. The composition relationship's binding specification indicates that JournalMaster replaces the LRCSynchronizedClass pattern, and therefore, all non-pattern elements defined within LRCSynchronizedClass are introduced to JournalMaster.



First, the composition pattern's name may be used for the aspect declaration. Also, the operation template parameter defined in Synchronize, write(..), may be seen as a pointcut in replacing classes. The composition relationship between Synchronize and Learning Resource Center (LRC) indicates that the JournalMaster operations add(Journal) and remove(Journal) replace write(..). As regards the advice code, the interaction (sequence) diagrams specified within the Synchronize composition pattern indicate when "advice" operations should be called relative to the template operations. These directly translate to the before and after constructs of the AspectJ advice element. This information maps to the following programming elements of aspects:



This illustrates the possibilities for mapping composition pattern constructs to AspectJ programming elements. The advantages of this are two-fold. First, from a design perspective, mapping the composition pattern constructs to constructs from a programming environment ensure that the clear separation of cross-cutting behaviour is maintained in the programming phase, making design changes easier to incorporate into code. Secondly, from the programming perspective, the existence of a design approach that supports separation of cross-cutting behaviour makes the design phase more relevant to this kind of programming, lending the standard benefits of software design to the approach.

## 6.9 AspectJ Extensions for Distributed Computing

Current programming systems do not provide mechanisms for modularizing crosscutting concerns in distributed systems and thus they are major sources of low readability and maintainability of the software. Issues like transactions, security, and fault tolerance are typical crosscutting concerns in distributed systems.



Many crosscutting concerns also arise during unit testing of distributed systems. The code for unit testing includes typical crosscutting concerns that AspectJ can deal with. AspectJ is a widely used language for aspect-oriented programming (AOP) in Java. Unfortunately, if we use AspectJ to modularize testing code for distributed software, the code ("aspect") can be somewhat modular but it often consists of several sub-components distributed on different hosts. They must be manually deployed on each host and the code of these sub-components must include explicit network processing among the sub-components for exchanging data since they cannot have shared variables or fields. These facts complicate the code of the aspect and degrade the benefits of using aspect oriented programming.

### **6.9.1 Implications on Network Processing**

AspectJ is a useful programming language for developing distributed software. It enables modular implementation even if some crosscutting concerns are included in the implementation. However, the developers of distributed software must consider the deployment of the executable code. Even if some concerns can be implemented as a single component ("aspect") at the code level, it might need to be deployed on different hosts and it would therefore consist of several sub components or sub-processes running on each host. Since Java (or Aspect J) does not provide variables or fields that can be shared among multiple hosts, the implementation of such a concern would include complicated network processing for exchanging data among the sub components.

Programming frameworks such as Java RMI do not solve this problem of complication. Although they make details of network processing implicit and transparent from the programmers' viewpoint, the programmers still must consider distribution and they are forced to implement the concern as a collection of several distributed sub-components exchanging data through remote method calls. The programmers cannot implement such a concern as a simple, non distributed monolithic component without concerns about network processing. This is never desirable with respect to aspect orientation since it means that the programmers must be concerned about distribution when implementing a different concern.

Consider the case of developing unit tests for distributed software, in our case, a distributed authenticated service. The distributed test code includes crosscutting concerns but, if they are modularized in AspectJ, the code develops the complexities mentioned above.

The implementation of this service consists of two components: a front-end server *AuthenticationServer* on a *host W* and a database server *DbServer* on another *host D*. This is a typical architecture for enterprise Web application systems. If a client application needs to register a new user, it remotely calls *registerUser0* on the front-end server using Java RMI. Then the *confirmUser()* method remotely calls *addUser()* on the database server, which will actually access the database system to update the user list. To unit-test the *confirmUser()* method, the test code would first remotely call the *confirmUser()* method and then confirm that the *addUser()* method is actually executed by the database server. Note that since the test code must confirm that remote method invocation is correctly executed, it must confirm not only that *confirmUser()* on the *host W* calls *addUser()* but also that *addUser()* starts running on the *host D* after the call. The test code would be simple and straightforward if the examined program is not distributed. It calls the *confirmUser()* method and then confirms the *tempUser* field is true. This field is set to true by the before advice (lines 10 to 14) when the *addUser()* method is executed.

We below show the test code written in Aspect J on the Eclipse Platform:

```

Java - AuthenticationServer.aj - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
*AuthenticationServer.aj
/aspect AuthenticationServer extends Testbed {
    boolean tempUser;
    void testRegisterUser() {
        tempUser = false;
        String userId = "alpha", password = "****";
        AuthenticationServer au = new AuthenticationServer();
        au.confirmUser(userId, password);
        assertTrue (tempUser);
    }
    before() : execution (void DbServer. addUser (String, String)) {
        tempUser = true;
    }
}

```

The above test code becomes more complicated if the examined program is distributed.

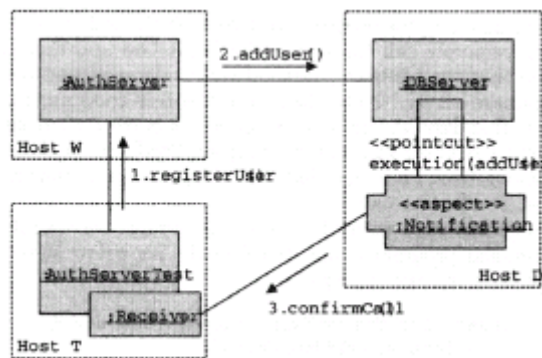


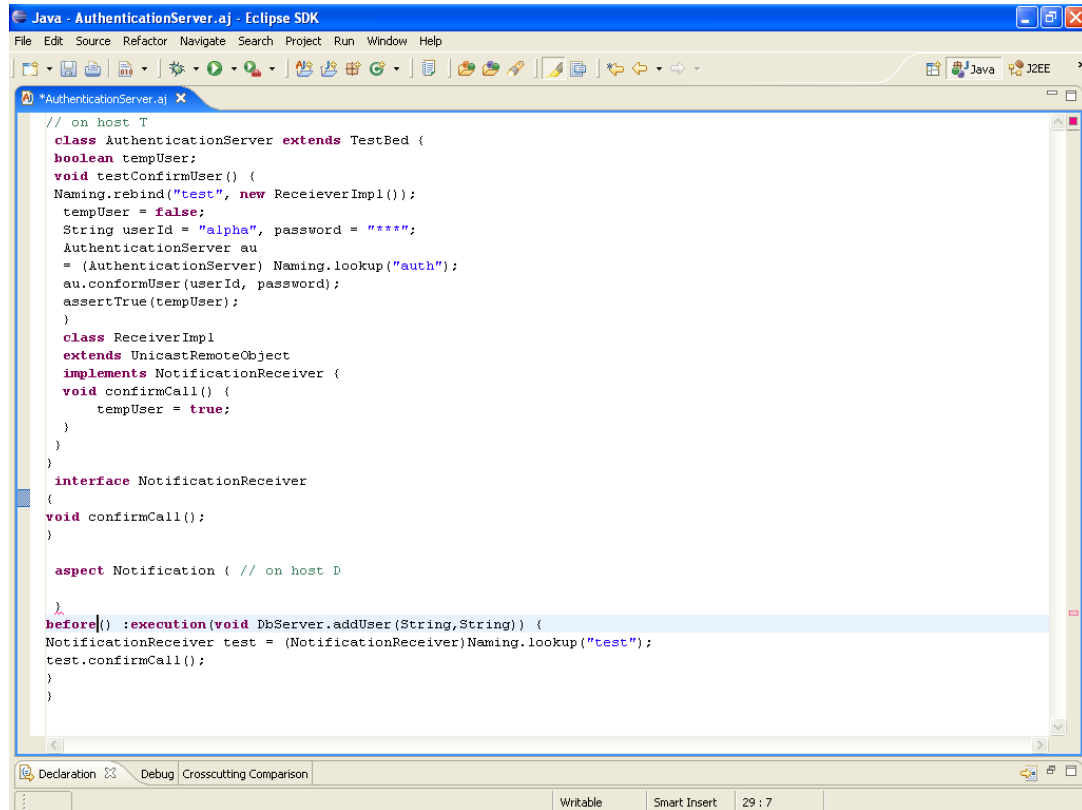
Figure 6.14: The testing code in AspectJ

The test code now consists of three sub-components: *AuthenticationServer*, *ReceiverImpl*, and *Notification* (Figure 6.14). Although the overall structure is the same, the *AuthenticationServer* and *ReceiverImpl* objects run on a testing host *T* but the *Notification* aspect runs on the host *D*, where the *DbServer* is running. The host *T* is different from *W* or *D*.

The *testRegisterUser()* method (lines 4 to 12) on *T* remotely calls *confirmUser()* on *W* and then confirms that the *tempUser* field is true. This field is set to **true** by the

*confirmCall()* method in *ReceiverImpl*, which is remotely called by the before advice (lines 28 to 35) of *Notification* running on *D*. The *confirmCall()* method cannot be defined in *AuthenticationServer* since *AuthenticationServer* must extend the *TestBed* class whereas Java RMI requires that remotely accessible classes extends the *UnicastRemoteObject* class.

The test code shown below is a distributed version:



```
// on host T
class AuthenticationServer extends TestBed {
    boolean tempUser;
    void testConfirmUser() {
        Naming.rebind("test", new ReceiverImpl());
        tempUser = false;
        String userId = "alpha", password = "****";
        AuthenticationServer au
        = (AuthenticationServer) Naming.lookup("auth");
        au.confirmUser(userId, password);
        assertTrue(tempUser);
    }
    class ReceiverImpl
    extends UnicastRemoteObject
    implements NotificationReceiver {
        void confirmCall() {
            tempUser = true;
        }
    }
}

interface NotificationReceiver
{
    void confirmCall();
}

aspect Notification ( // on host D
}

before():execution(void DbServer.addUser(String,String)) {
    NotificationReceiver test = (NotificationReceiver)Naming.lookup("test");
    test.confirmCall();
}
}
```

Even this simple testing concern is implemented by distributed sub-components and hence we had to write complicated network processing code using Java RMI despite that it is not related to the testing concern. In particular, the *Notification* aspect is used only for notifying *confirmCall()* on the host *T* beyond the network that the thread of control on the host *D* reaches *addUser()*. The *Notification* aspect is a sub-component that are necessary only because *confirmCall()* and *addUser()* are deployed on different hosts.

This means that the component design of the unit testing is influenced by concerns about distributed. Furthermore, this notification code is similar to what the AspectJ compiler produces for implementing the pointcut-advice framework. It should not

be hand-coded, but implicit within the language constructs provided by an AOP language.

## **6.10 Summary**

Improvement from using AspectJ in pattern implementations is directly correlated to the presence of crosscutting structure in the patterns. This crosscutting structure arises in patterns that superimpose behavior on their participants. In such patterns the roles can crosscut participant classes, and conceptual operations can crosscut methods (and constructors). Multiple such patterns can also crosscut each other with respect to shared participants.

Software design is an important activity in the development lifecycle but its benefits are often not realized. Scattering and tangling of cross-cutting behaviour with other elements causes problems of comprehensibility, traceability, evolvability, and reusability. Attempts have been made to address this problem in the programming domain but the problem has not been addressed effectively at earlier stages in the lifecycle. Composition patterns presents an approach to addressing this problem at the design stage.

AspectJ extensions for identifying join points in the execution of a program running on a remote host can simplify the description of aspects with respect to network processing if the aspects implement a crosscutting concern spanning over multiple hosts.

Further, the realisation of the prototype implementations provided in this chapter does not attempt to provide a complete implementation of the AOSDDL structure previously described in chapter 5. For example, significant parts of the component and distributed framework, which are both key to the aspect design language, have not been implemented due to the overall complexity of the system and the time constraints. The objective was rather to demonstrate the feasibility of the design language notations through a 'proof-of-concept' implementation of the AOSDDL specific mechanisms such as concern, join point, introduction etc (the major representations).

The implementation of the various mechanisms is described according to the overall structure of the AOSDDL structure, the processing and test environments. This chapter also discussed the evaluation of the design language structure. A qualitative evaluation has been considered best suitable for the design evaluation of AOSDDL. The qualitative evaluation is demonstrated based on a real case study.

## **CHAPTER 7**

### **Conclusion**

#### **7.1 Overview**

This chapter recapitulates the work that has been carried out as part of this research effort. It summarizes the conclusions that could be gained from the design and development of the AOSDDL architecture.

It provides an overview of the thesis structure and a summary of each chapter. A series of conclusion summarizes what has been learnt from this work, and how these experiences contribute to the wider field of research.

#### **7.2 Thesis Summary**

Chapter one of this thesis sets the scene by unfolding the evolution of software programming from the early days of computer science until today. It continued introducing the concepts of aspect oriented software development and describing the problem of today's programming methodologies that have led to the establishment of this new research area. It provided a discussion of the research motivation for the field, highlighting the need for aspect oriented design capabilities and the potential beneficiaries of such a technology. Finally, it presented a summary of the research goals and challenges that are taken on by this work.

Chapter two provides a general background on the field of aspect oriented programming. It looks back to the initial developments of this trend in the 1990's and shows how the field has evolved since. It defined the basic methodology for aspect oriented programming and introduced various approaches towards aspect oriented software development. The chapter then continued with a discussion of several architectural approaches and other key aspects for aspect oriented software development.

Chapter three provided a comprehensive overview of the current state-of-the-art in the field of aspect oriented software development. A large number of different projects were presented, describing their specifications and the distinction of their design. The variety of projects based on the design language architecture were

divided into *implementation dependent* and *implementation independent* approaches. Although their fundamental goals are identical and the timescale when they have emerged is related, the underlying architectures are inherently different.

Chapter four discussed the requirements for aspect oriented design language in general and derived the requirements for the AOSDDL (Aspect Oriented Software Development Design Language) architecture. These requirements have been derived from related work and acknowledged publications in the field. General factors, for example the commercial aspects such as the deployment of aspect oriented technologies are also taken into consideration. A differentiation between the absolutely vital requirements and the more long-term requirements for an aspect oriented software system was made. From this multitude of general requirements a subset of requirements, which were considered important for the design of a flexibly extensible aspect design language, was drawn. These specialized requirements form the basis for the subsequent AOSDDL model and implementation.

Chapter five and six presented the bulk of the contributions made in this thesis, namely the AOSDDL design notation and the prototype implementations of this design language. This chapter has introduced the design of AOSDDL, an aspect oriented design language with the motivation to provide a highly flexible and extensible set of notations suitable for aspect oriented software development in all real world scenarios, can form the basis for further research into aspect oriented systems and software engineering in general.

Chapter six presented the ongoing efforts to engineer a prototypical realization of the AOSDDL design language. Described in this section is the development of the core design constructs and notations of the AOSDDL model. Due to the considerable extent of this model, the development work has focused on validating the core design decisions and key mechanisms (i.e. separation of concerns, design incompatibility, synchronization, etc.) through a 'proof-of-concept' implementation. This chapter continued with the evaluation of the AOSDDL structure and the prototype implementation. A qualitative evaluation was considered best suitable for the design evaluation of AOSDDL.



Chapter seven finally concludes the thesis by bringing together the thread of arguments throughout this work.

### **7.3 Concluding Remarks**

Several conclusions can be drawn from the development of AOSDDL:

#### Enforcing Architectural Regularities

The problems encountered were not as a result of an incorrect AOP design concept or idea in general but a consequence of its particular implementation. AspectJ being the only implementation available that is widely in use and is still undergoing changes. The language was not designed for the purpose of regulating architectural decisions and thus lacks sufficient tools to accommodate this task. The various design considerations regarding distributed architecture are possible with design constructs of AOP but it is their realization that caused difficulties.

#### AOSDDL Features

- An approach for high level architecture design, called AOSDDL, has been developed to enable separation of concerns at the design level of an AO development process. Within this approach it is assumed that the requirements have already been defined and specified during previous development stages.
- Since AOSDDL is UML conform, any CASE tool that supports UML modeling can be used.
- Aspects and base elements are completely kept apart; they are connected via a special language-specific connector element that encapsulates the underlying implementation technology. Any desired AO technology can be supported; it is just the connector's syntax and semantics that have to be specified.

- Both, aspects and base elements, can be reused separately as the connector is the only crosscutting, language-dependent part. This sort of encapsulation offers a logical grouping of all classes belonging to one concern and eases the readability of design models as avoiding graphical tangling.
- To offer low-level architecture design support, a code generator needs to be developed to improve productivity and reduce errors when mapping model to code.

### AOP Testing

The incremental testing approach is similar to traditional regression testing. The essential difference is that, aspects as a structured way to specify modifications make it feasible to investigate systematic reuse and modification of the existing tests. Our approach can be adapted to the UML class diagrams and start charts by using class interfaces, flattening start chart diagrams, and following the convention of guard conditions.

The work can be seen as a first step towards a simple and powerful modeling approach that fosters support from existing CASE tools since it is based on standard UML. AOSDDL in combination with the code generator should make AOSD more usable and more efficient for software development. The assumptions about the usefulness of the notation and the AO code generation have to be proven in the near future when using it in business development projects.

## **CONTRIBUTIONS**

Here we summarize the main contributions and achievements of the research carried out as part of this thesis.

The overall goal of this work, namely to design a aspect oriented design language that enables flexible extensibility of requirements and design functionality, has been successfully fulfilled in the form of AOSDDL structure. The validation of the architectural design with respect to its feasibility and practicality has been accomplished through prototype implementations of the AOSDDL architecture.

### **Natural Extension to UML**

A sufficient notation that is simple to understand and straightforward to use for developers who are familiar with common design notations (such as UML).

### **CASE Tool Support**

Design modeling is supported by powerful CASE tools like Together to improve developer productivity and to ensure syntactical correctness of the AO model.

### **Extension of Architectural framework for design constructs**

An extension to UML is presented, without changing its metamodel specification, to achieve standard UML conformity. This helps developers to become acquainted with AO modeling when they are already familiar with OO modeling and UML. A key intention was to offer standard development tool support and interchangeability between various tools. UML is customized by using standard extension mechanisms only. To gain the benefits of code and design reuse of AO software, the ability to reuse aspect and business logic separately is needed. A notation is presented where aspect and business logic are completely kept apart. Thus, both are reusable and at the same time independent of the implementation technology. Within this approach it is assumed that the requirements have already been

defined and specified during previous development stages.

### **Enforcing Architectural Regularities**

A natural outcome of the research work undertaken during design and prototype implementation was the realization that the problems encountered were not as a result of an incorrect AOP design concept or idea in general but a consequence of its particular implementation. AspectJ being the only implementation available that is widely in use and is still undergoing changes. The language was not designed for the purpose of regulating architectural decisions and thus lacks sufficient tools to accommodate this task. The various design considerations regarding distributed architecture are possible with design constructs of AOP but it is their realization that caused difficulties

### **Implementation Support**

A direct mapping between the notation and supported implementation languages to allow automatic code generation based on the design model is a natural outcome for the next stage of work.

### **Software Development**

The notation fulfils its applicability in real-world development projects because of smooth integration with existing and widely used tools and methodologies.

## **FUTURE SCOPE OF WORK**

Besides the ongoing development efforts to complete the AOSDDL prototype implementation, further work in this area focuses on using and extending the AOSDDL notation architecture and prototype platform in order to build and experiment with design language specifications.

The code generators, tool integration and notation deployment and are few examples of ongoing research that take advantage of the AOSDDL architecture and platform.

### **Code Generators for Aspect modeling**

To ease the transition from design to implementation and to offer low-level architecture design support, a code generator has to be developed to support automatic generation of AO code skeletons from design models. This will help developers to focus on models having the code skeletons generated automatically to gain the benefits they are used to in OOSD. Code generation improves developer productivity, ensures syntactical correctness and reduces errors when mapping a model to code. The presented UML notation in combination with the code generator will make AOSD more usable and more efficient for software development by avoiding inconsistencies among design and implementation. Developers can then concentrate on AO design having the code skeletons generated automatically.

### **Tools Integration**

After evaluating the prototype's features in real world development projects, some concepts may have to be added (e.g. complex relationships between aspects). Another important feature will be a complete CASE tool support including roundtrip engineering for aspect mining. As Together supports the development of modules offering roundtrip engineering features, this will be included in the near future in the code generator. The connector package encapsulates the underlying implementation technology. Currently, the syntax and semantics of an AspectJ specific connector type are defined. This sort of encapsulation eases the

replacement of the AO language, the support of different technologies and language concepts (such as Hyper/J [18] [19] [23]) will be part of some future work. An automated code generation for different languages is rather straightforward, too. It is only the code generator's mapping rules that have to be changed.

### **Notation deployment**

The assumptions about the usefulness of the notation and the AO code generation have to be proven in the near future when using it in business development projects.

### **Support for Hyper/J**

Support of other AO concepts (such as Hyper/J) that are implementation dependent parts of AOSDDL can also be considered as part of future work.

### **Testing of Aspect Oriented Requirements**

Addressing issues like the kind of base class tests that are less likely to be helpful for revealing aspect faults, how to prioritize the test cases to be reused and on how to model and test interference of multiple interacting aspects.

### **Summary**

There are still many issues to be solved until efficient AO development support comparable to current OO support is established. When offering an integrated development process, the gaps between the early phases and AO programming have to be filled as so far the paradigm focuses mainly at the implementation level. There is still a lot of challenging research to be done in the future until the paradigm is widely accepted and developers are aware of the benefits AOSD offers.

## REFERENCES

- [1] Aspect Oriented Programming  
<http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html> , 2007
- [2] Object Management Group (OMG). Unified Modeling Language Specification.  
[http://www.omg.org/technology/documents/modeling\\_spec\\_catalog.htm](http://www.omg.org/technology/documents/modeling_spec_catalog.htm), Feb. 2007
- [3] Rumbaugh, Jacobson Booch. UML Reference Manual. Addison-Wesley International, Boston, Massachusetts, 1998
- [4] Aspect-oriented programming: <http://aosd.net>, 2007
- [5] Siobhan Clarke and Robert J. Walker. “ Towards a Standard Design Language for AOSD,” ACM Proceedings of the 1<sup>st</sup> International Conference on Aspect Oriented Software Development, Enschede, Netherlands, (April 2002), pp. 113- 119.
- [6] G. Kiczales, E. Hilsdale, J. Hugunin, M.Kersten, J. Palm and W. Griswold. “ An overview of AspectJ,” Proceedings of the 15<sup>th</sup> European Conference on Object Oriented Programming, Budapest, Hungary (June 2001), pp. 327-353.
- [7] Palo Alto Research Center. <http://www.parc.com/>, 2007
- [8] The AspectJ Team. The AspectJ programming Guide. <http://www.eclipse.org/>, 2007
- [9] IBM Research. <http://www.research.ibm.com/>, 2006
- [10] IBM alphaWorks. <http://www.alphaworks.ibm.com/tech/hyperj>, 2007.
- [11] Wai-Ming Ho, Jean-Marc Jezequel, Francois Pennaneac’h and Noel Plouzeau. “A Toolkit for Weaving Aspect Oriented UML Designs,” ACM Proceedings of the 1<sup>st</sup> International Conference on Aspect Oriented Software Development, Enschede, Netherlands, (April 2002), pp. 99-105.
- [12] Awais Rashid, Ana Moreira and Joao Araujo. “Modularisation and Composition of Aspectual Requirements,” ACM Proceedings of the 2<sup>nd</sup> International Conference on Aspect Oriented Software Development, Boston, Massachusetts, (March 2003), pp. 11-20.
- [13] Mika Katara, Shmuel Katz. “ Architectural Views of Aspects,” ACM Proceedings of the 2<sup>nd</sup> International Conference on Aspect Oriented Software Development, Boston, Massachusetts, (March 2003), pp. 1- 10.
- [14] Ramnivas Laddad. AspectJ in Action. PDF ebook, Manning Publications, Greenwich, Connecticut, 2003
- [15] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns— Elements of Reusable Object-Oriented Software. Addison-Wesley International, Boston, Massachusetts, 1995
- [16] Fowler, Martin. Refactoring: Improving the Design of Existing Code. Addison-Wesley International, Boston, Massachusetts, 1999.

- [17] Hannemann, Jan, and Gregor Kiczales. "Design Pattern Implementation in Java and AspectJ," Proceedings of the 17<sup>th</sup> Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Seattle, Washington, (Nov 2002), pp. 161–173.
- [18] Stanley M. Sutton and Isabelle Rouvellou. "Modeling of Software Concerns in Cosmos," ACM Proceedings of the 1<sup>st</sup> International Conference on Aspect Oriented Software Development, Enschede, Netherlands, (April 2002), pp. 127 - 133.
- [19] Ossher, H., and P. Tarr. "Multi-Dimensional Separation of Concerns Using Hyperspaces." IBM Research Report 21452, April, 1999. Available online at <http://www.research.ibm.com/hyperspace/Papers/tr21452.ps>
- [20] Natsuko Noda and Tomoji Kishi. "Design Pattern Concerns for Software Evolution," ACM Proceedings of the 4<sup>th</sup> International Workshop on Principles of Software Evolution, Vienna, Austria (October 2001), pp. 158-161.
- [21] Tate, Bruce, Mike Clark, Bob Lee, and Patrick Linskey. Bitter EJB. Manning Publications, Greenwich, Connecticut, 2003.
- [22] Objectdb. <http://www.objectdb.com/>, 2006
- [23] Tate, Bruce. Bitter Java. Manning Publications, Greenwich, Connecticut, 2002.
- [24] The ServerSide.com J2EE Community. <http://www.theserverside.com/>, 2006
- [25] Deepak Alur, John Crupi and Dan Malks. Core J2EE Patterns: Best Practices and Design Strategies, 2<sup>nd</sup> Edition. Prentice Hall, Indianapolis, Indiana, 2003.
- [26] Floyd Marinescu. EJB Design Patterns: Advanced Patterns, Processes, and Idioms. John Wiley & Sons, New Jersey 2002
- [27] Schmidt, Douglas C., Michael Stal, Hans Rohnert, and Frank Buschmann. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. New York, John Wiley & Sons, 2000
- [28] Aksit, M., B. Tekinerdogan, and L. Bergmans. "Achieving Adaptability through Separation and Composition of Concerns." In Special Issues in Object-Oriented Programming, M. Muhlhauser (ed.), dpunkt verlag, pp. 12–23, 1996. Also available online at <http://trese.ewi.utwente.nl/oldhtml/publications/paperinfo/sioop96.pi.ref.htm>
- [29] Ossher, Harold, William Harrison, Frank Budinsky, and Ian Simmonds. "Subject-Oriented Programming: Supporting Decentralized Development of Objects," Proceedings of the 7<sup>th</sup> IBM Conference on Object-Oriented Technology, US. July 1994.
- [30] Walker, Robert J., Elisa L.A. Baniassad, and Gail C. Murphy. "An Initial Assessment of Aspect-oriented Programming," IEEE Proceedings of the 21<sup>st</sup> International Conference on Software Engineering, Los Angeles, California, (May 1999), pp. 120 – 130.
- [31] Janzen, D., and K. De Volder. "Navigating and Querying Code without Getting Lost," ACM Proceedings of the 2<sup>nd</sup> International Conference on Aspect Oriented Software Development, Boston, Massachusetts, (March 2003), pp. 178–87.
- [32] Siobhan Clarke and Robert J. Walker. "Composition Patterns: An Approach to Designing Reusable Aspects," IEEE Proceedings of the 23<sup>rd</sup> International Conference on Software Engineering, Toronto, Ontario, (Oct 2001), pp. 5-14.



- [33] Dominik Stein, Stefan hanenberg, and Rainer Unland. "A UML-based Aspect Orientation Design Notation for AspectJ." ACM Proceedings of the 1<sup>st</sup> International Conference on Aspect Oriented Software Development, Enschede, Netherlands, (April 2002), pp. 106-112.
- [34] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. "An overview of AspectJ," Proceedings of the 15<sup>th</sup> European Conference on Object Oriented Programming, Budapest, Hungary (June 2001), pp. 327-353.
- [35] Palo Alto Research Center. <http://www.parc.com/>, 2007
- [36] The AspectJ Team. The AspectJ programming Guide. <http://www.eclipse.org/>, 2007
- [37] David Mapelsden, John Hosking and John Grundy. "Design Pattern Modeling and Instantiation using DPML," Proceedings of the 40<sup>th</sup> International Conference on Tools Pacific: Objects for internet, mobile and embedded applications, Sydney, Australia, Australian Computer Society, (Feb 2002), pp.3-11.
- [38] Elisa L.A. Baniassad, Gail Murphy, Christa Schwanninger and Michael Kircher. "Managing Crosscutting Concerns During Software Evolution Tasks: An Inquisitive Study," ACM Proceedings of the 1<sup>st</sup> International Conference on Aspect Oriented Software Development, Enschede, Netherlands, (April 2002), pp. 120- 126.
- [39] Hyper/J: <http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm>, 2006
- [40] P. Tarr, H. Ossher, W. Harrison and S. Sutton. "N degrees of separation: Multi-dimensional separation of concerns," IEEE Proceedings of the 21<sup>st</sup> International Conference on Software Engineering, Los Angeles, California, (May 1999), pp. 107-119.
- [41] UMLAUT. <http://www.irisa.fr/UMLAUT/>, 2006
- [42] UMLAUT model transformation. <http://modelware.inria.fr/>, 2006
- [43] Theme/UML. <http://www.dsg.cs.tcd.ie/~sclarke/ThemeUML/>, 2003
- [44] Elisa Banniassad and Siobhàn Clarke. Aspect Oriented Analysis and Design: The Theme Approach. Addison-Wesley, Boston, Massachusetts, 2005.
- [45] Elisa Banniassad and Sio'ghan Clarke. "Theme: An Approach for Aspect Oriented Analysis and Design," IEEE Proceedings of the 26<sup>th</sup> International Conference on Software Engineering, Scotland, UK, (May 2004), pp. 158 – 167.
- [46] Davy Suvee, Wim Vanderperren and Viviane Jonckers. "JAsCo: an Aspect- Oriented approach tailored for Component Based Software Development," ACM Proceedings of the 2<sup>nd</sup> International Conference on Aspect Oriented Software Development, Boston, Massachusetts, (March 2003), pp. 21- 29.
- [47] Ossher, H., and P. Tarr. "Multi-Dimensional Separation of Concerns Using Hyperspaces." IBM Research Report 21452, April, 1999. Available online at <http://www.research.ibm.com/hyperspace/Papers/tr21452.ps>.
- [48] Adaptive Programming. <http://www.ccs.neu.edu/home/lieber/demeter.html>, 2004

- [49] Lieberherr, Karl J. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Co., Boston, Massachusetts, 1996. Also available online at <http://www.ccs.neu.edu/research/demeter/book/book-download.html>.
- [50] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, Viviane Jonckers. "Adaptive Programming in JasCo," ACM Proceedings of the 4<sup>th</sup> International Conference on Aspect Oriented Software Development, Chicago, Illinois, (March 2005), pp. 75-86.
- [51] Richard Cardone, Adam Brown, Sean McDermid and Calvin Lin. "Using Mixins to Build Flexible Widgets," ACM Proceedings of the 1<sup>st</sup> International Conference on Aspect Oriented Software Development, Enschede, Netherlands, (April 2002), pp. 76-85.
- [52] M. Griss. "Implementing Product Line Features by Composing Component Aspects," Proceedings of First Software Product Line Conference, Denver, Colorado, Kluwer (Aug 2000), pp. 271–288.
- [53] Mati Shomrat and Amiram Yehudai. "Obvious or Not? Regulating Architectural Decisions Using Aspect-Oriented Programming," ACM Proceedings of the 1<sup>st</sup> International Conference on Aspect Oriented Software Development, Enschede, Netherlands, (April 2002), pp. 3-9.
- [54] Awais Rashid, Ana Moreira and Joao Araujo. "Modularisation and Composition of Aspectual Requirements," ACM Proceedings of the 2<sup>nd</sup> International Conference on Aspect Oriented Software Development, Boston, Massachusetts, (March 2003), pp. 11-20.
- [55] Rashid, A., P. Sawyer, A. Moreira and J. Araujo. "Early Aspects: A Model for Aspect-Oriented Requirements Engineering," IEEE Joint International Conference on Requirements Engineering, Essen, Germany, (Sep 2002), pp. 199-202.
- [56] Araujo, J., A. Moreira, I. Brito and A. Rashid. "Aspect-Oriented Requirements with UML," UML 2002 – The Unified Modeling Language: 5<sup>th</sup> International Conference, Dresden, Germany, (Sep 2002), Lecture Notes in Computer Science, Springer Berlin, pp. 442 – 447.
- [57] A. Rashid, A. Moreira, B. Tekinerdogan. "Early Aspects: Aspect-oriented Requirements Engineering and Architecture Design," Object Oriented Programming, Systems, Languages and Applications Workshop 2004, Vancouver, British Columbia, IEE – Proceedings Software. IEE, (Oct 2004), Volume 151(4), pp. 153-156.
- [58] R.Chitchyan, A.Rashid, P.Sawyer. "Comparing Requirements Engineering Approaches for Handling Crosscutting Concerns," WER Proceedings of the VIIIth Workshop on Requirements Engineering (held with CaiSE 2005), Porto, Portugal, (June 2005), pp. 1 – 12.
- [59] A. Moreira, J. Araujo, A. Rashid. "A Concern-Oriented Requirements Engineering Model," 17<sup>th</sup> International Conference on Advanced Information Systems Engineering, Porto, Portugal, (June 2005), Lecture Notes in Computer Science, Springer-Verlag, pp. 293-308.
- [60] Frddric Duclos, Jacky Estublier, Philippe Morat. "Describing and Using Non Functional Aspects in Component Based Applications," ACM Proceedings of the 1<sup>st</sup> International Conference on Aspect Oriented Software Development, Enschede, Netherlands, (April 2002), pp. 65 – 75.

- [61] Mika Katara, Shmuel Katz. " Architectural Views of Aspects," ACM Proceedings of the 2<sup>nd</sup> International Conference on Aspect Oriented Software Development, Boston, Massachusetts, (March 2003), pp. 1- 10.
- [62] AOCE. <http://www.cs.auckland.ac.nz/~john-g/aspects.html>, 2000
- [63] AOCE. <http://citeseer.ist.psu.edu/grundy99supporting.html>, 2005
- [64] Composition filters: [http://trese.cs.utwente.nl/composition\\_filters](http://trese.cs.utwente.nl/composition_filters), 2005
- [65] Aspect-oriented design pattern implementations. <http://www.cs.ubc.ca/~jan/AODPs>, 2004
- [66] M. Nishizawa, S.Chiba and Michiaki Tatsubori. " Remote Pointcut: A Language Construct for Distributed AOP," ACM Proceedings of the 3<sup>rd</sup> International Conference on Aspect Oriented Software Development, Lancaster, UK (March 2004), pp. 7 - 15.
- [67] Dianxiang Xu and Weifeng Xu. " State Based Incremental Testing of Aspect Oriented Programs, " ACM Proceedings of the 5<sup>th</sup> International Conference on Aspect Oriented Software Development, Bonn, Germany, (March 2006), pp. 180 - 189.

## BIBLIOGRAPHY

- [68] JDK 1.4 tool. <http://java.sun.com/j2se>, 2005
- [69] AspectJ 1.1 tool. <http://www.eclipse.org/aspectj>, 2006
- [70] log4j 1.2 library. <http://jakarta.apache.org/log4j>, 2006
- [71] J2EE SDK 1.3 tool. <http://java.sun.com/j2ee>, 2005
- [72] Ant tool. <http://ant.apache.org>, 2006
- [73] Eclipse IDE integration with AspectJ. <http://www.eclipse.org/ajdt>, 2005
- [74] AspectJ user mailing list. <https://dev.eclipse.org/mailman/listinfo/aspectj-users>
- [75] AOSD user mailing list. <http://aosd.net/mailman/listinfo/discuss>
- [76] Together CASE tool. <http://www.borland.com/>, 2005

## LIST OF PUBLICATIONS AND PRESENTATIONS

### AOSD Design Language Requirements

- Deepak Dahiya, Rajinder K. Sachdeva. *“Role of Requirements in Aspect Oriented Design Language Architecture,”* Paper published in Proceedings of the 5<sup>th</sup> RoEduNet IEEE International Conference in Networking and Computer Technology, Sibiu, Romania (RoEduNet IEEE 2006), ISBN: 973-739-277-9, pp. 204 – 207.
- Deepak Dahiya, Rajinder K. Sachdeva. *“Understanding Requirements: Aspect Oriented Software Development,”* Paper published in the IEEE Proceedings of the 30<sup>th</sup> IEEE Computer Software and Applications Conference, Chicago, US (IEEE COMPSAC 2006), pp. 303-308.
- *“Role of Requirements in Aspect Oriented Design Language Architecture”*. Paper accepted for publication in the “Acta Universitatis Cibiniensis”, Technical Series in Computer Science and Automatic Control (2006) Journal, University of Sibiu, Romania (to be published).

### Related and Current Work on Design Language

- Deepak Dahiya, Rajinder K. Sachdeva. *“Design Language for Aspect Oriented Software Development and Design Pattern Extensions”*. Paper published in Proceedings of the 4<sup>th</sup> International Conference on Computer Science and its Applications, San Diego, California, USA (ICCSA 2006), ISBN: 9742448-5-6, pp. 127 - 131.
- Deepak Dahiya, Rajinder K. Sachdeva. *“Moving Towards Aspect Oriented Design,”* Paper published in Proceedings of the 1<sup>st</sup> International Conference on Web Engineering and Applications, Bhubaneswar, Orissa, India (ICWA 2006), pp. 156 – 161.

### The AOSDDL Design Language

- *“Design Issues in Aspect Oriented Programming”*. Paper accepted for publication in the Information Technology Journal, (ISSN 1812–5638), ANSI Journals, April 2007 (to be published).

- Deepak Dahiya, R.K. Sachdeva and Sudha. “*Dealing with Software Concerns in Aspects*,” Paper published in the IEEE proceedings of the 1<sup>st</sup> International Conference on Digital Information Management, Bangalore, India (IEEE ICDIM 2006), pp. 29 – 36.
- “*Dealing with Software Concerns in Aspects*”. Paper accepted for publication in the Journal of Digital Information Management, (ISSN 0972–7272), a peer reviewed Journal, Digital Information Research Foundation, Chennai, India (to be published).

### **Implementation and Evaluation**

- Deepak Dahiya, R.K. Sachdeva and Sudha,” *A prototype implementation using Aspect Oriented Software Development*,” Paper published in the IEEE proceedings of the 1<sup>st</sup> International Conference on Digital Information Management, Bangalore, India (IEEE ICDIM 2006), pp. 6– 12.
- Deepak Dahiya, Rajinder K. Sachdeva. “*Moving from AOP to AOSD Design Language*”. Paper published in the IJCSNS International Journal of Computer Science and Network Security, Vol. 6, No. 10 (ISSN 1738–7906), October 2006, pp. 36 - 41.

## APPENDICES

### A. Installation and Configuration of Eclipse Platform

#### A.1 Running Eclipse

In case, *eclipse.exe* does not start eclipse, then do the following:

It's generally a good idea to explicitly specify which Java VM to use when running Eclipse. This is achieved with the "-vm" command line argument as illustrated below:

```
c:\eclipse>eclipse -vm c:\j2sdk1.4.2\jre\bin\javaw -vmargs -Xmx256M
```

OR

Step 1:

create a batch file "autoexec.bat" as follows:

```
/****** autoexec.bat *****/  
  
eclipse -vm c:\j2sdk1.4.2\jre\bin\javaw -vmargs -Xmx256M  
exit  
  
/****** */
```

Step 2:

Double click on the file to start eclipse. (you can also create shortcut on desktop and renaming it as "Eclipse")

If you don't use "-vm", Eclipse will look on the O/S path. When you install other Java-based products, they may change your path and could result in a different Java VM being used when you next launch Eclipse.

## **A.2 Importing an existing AspectJ project**

1. Physically copy all the Eclipse projects (i.e. Example1, Example1-1, etc ) from the specified location ( i.e. from the old workspace directory etc. ) into the destination (i.e. new workspace directory).
2. Start Eclipse.
3. Go to File --> Import...
4. From the "select an import source" options choose "Existing project into workspace"
5. select "Next".
6. Browse to the newly copied (workspace directory) "project location" and choose the project (i.e. Example1, Example1-1 etc).
7. click "Finish".
8. Repeat this procedure for as many projects to be imported.

### Note:

If the eclipse projects are not physically copied to the new location (workspace), then the projects are created using the old workspaces. However, This can be damaging in case, you forget that the project files are actually existing in the old location and not in the new workspace directory.

## **A.3 Running AspectJ project in Eclipse**

1. Start Eclipse Platform.
2. Go to windows --> Preferences --> Java --> Installed JREs
3. See that the JRE home directory is checked and the path is:  
`c:\Program Files\Java\j2re1.4.2_05`
4. Now compile and run your AspectJ project.



## **BRIEF BIOGRAPHY OF THE CANDIDATE**

**DEEPAK DAHIYA**

**(Area of Interest: Software Engineering & IT Management)**

Deepak Dahiya has over 13 years of Experience in Academics and IT Industry in India and Abroad. His work experience includes working for Computech (New Jersey, US), PIPAL (Delhi, India), CASE and ITIL (Delhi, India), PHILCORP (Goa, India), NIC Pune (Ministry of Communications and Information Technology, India). His work involved designing and implementing Client- Server, N-tier J2EE Applications and Software Project Management.

In Academics, Deepak's assignments include working as Associate Professor in Information Technology at IILM (Delhi, India). He has conducted courses for both both postgraduate (MCA / MBA) and undergraduate (BCA / BBA / BSc / BEngg) programmes.

Presented and Published around 15 research papers in International Journals and Conferences including IEEE and ACM.

Reviewed papers for various International Journals.

Deepak has been University Examiner and Paper Setter for various IT courses of Goa University, India and Ch. Charan Singh University, India.

In the IT Industry, Deepak has consulted for Clients like Verilytics (Massachusetts, US), State Department of Treasury (Michigan, US) FIT (London, UK), i-flex solutions limited (mumbai, India and a subsidiary of Citibank).

Deepak has conducted senior executive training programmes for organizations like National Academy for Training & Research in Social Security (EPFO, Ministry of Labour, Govt. of India, New Delhi), Goa Shipyard Limited (a Public Sector Undertaking, Goa, India) and the Indian Navy.

Deepak is also Guest Faculty to the prestigious Indian Institute of Management (IIM) Kozhikode, India for delivering elective courses to PGP students.

Deepak is a member of IEEE Computer Society, ACM Society and is Life Member of Computer Society of India.

## **BRIEF BIOGRAPHY OF THE SUPERVISOR**

**Dr. Rajinder Kumar Sachdeva, Professor of Management, IIPA Delhi**

### **Education:**

Doctor of Philosophy (Ph.D.) from Indian Institute of Technology, Delhi in 1989.  
**Topic** - Distributed Information Systems.

Master of Technology (M.Tech.) from Indian Institute of Technology, Delhi in 1969.

Bachelor of Engineering with Honours (B.E.Hons.) from University of Roorkee  
(Now I.I.T., Roorkee) in 1966.

### **International Assignments:**

National Representative, Committee on Information Systems (TC-8), International Federation for Information Processing (IFIP), 1986-91, nominated by Computer Society of India.

Massachusetts Institute of Technology (MIT), USA as UNDP Fellow on Advanced Study Program, Fall Semester, Sept.-Dec., 1989.

National Computing Centre (NCC), U.K. on one year sabbatical, 1988-89.

Carl Duisberg Gessellschaft (CDG) Fellow on Senior Management Program, West Germany, Oct.-Dec. 1977.

Expert Advisor on Computers to Govt. of Iraq on deputation from Govt. of India, May-Oct., 1978.

### **Publications: (Books / Articles)**

1. Management Handbook of Computer Usage, NCC Blackwell, Oxford, U.K.
2. Management Information Systems : A New Framework, Vikas Publishing House, New Delhi (Co-author with Dr. U.K. Banerjee)
3. Indian Standard USE OF NETWORK ANALYSIS FOR PROJECT MANAGEMENT as Chairman, Expert Panel, Bureau of Indian Standards, New Delhi

Over two dozen articles in leading professional journals and economic dailies.

### **Teaching and Training:**

Conducted MDPs and taught courses at IIPA, Faculty of Management Studies (FMS), University of Delhi, Department of Computer Science, University of Delhi, University of Roorkee ( Now I.I.T ) and School of Planning and Architecture (S.P.A.), New Delhi.

### **Consultancy and Industrial Experience:**

Consultant to Govt. of Gujarat, Transformers and Electricals Kerala Ltd., and Madras Refineries Ltd. for design and implementation of Computer based Information Systems.

Worked with Philips India Ltd. and Warner Hindustan Ltd. as Systems Analyst ; with Sarabai's ORG as Operations Research Analyst ; and Hindustan Motors Ltd. as Planner.