Aspects and Objects: A Unified Software Design Framework

A thesis submitted to the University of Huddersfield
in partial fulfilment of the requirements for
the degree of Doctor of Philosophy

By

SAQIB IQBAL

July 2012

Department of Informatics
The University of Huddersfield

Abstract

Aspect-Oriented Software Development provides a means to modularize concerns of a system which are scattered over multiple system modules. These concerns are known as crosscutting concerns and they cause code to be scattered and tangled in multiple system units. The technique was first proposed at the programming level but evolved up through to the other phases of the software development lifecycle with the passage of time. At the moment, aspect-orientation is addressed in all phases of software development, such as requirements engineering, architecture, design and implementation. This thesis focuses on aspect-oriented software design and provides a design language, Aspect-Oriented Design Language (AODL), to specify, represent and design aspectual constructs. The language has been designed to implement co-designing of aspectual and non-aspectual constructs. The obliviousness between the constructs has been minimized to improve comprehensibility of the models. The language is applied in three phases and for each phase a separate set of design notations has been introduced. The design notations and diagrams are extensions of Unified Modelling Language (UML) and follow UML Meta Object Facility (UML MOF) rules. There is a separate notation for each aspectual construct and a set of design diagrams to represent their structural and behavioural characteristics.

In the first phase, join points are identified and represented in the base program. A distinct design notation has been designated for join points, through which they are located using two diagrams, Join Point Identification Diagram and Join Point Behavioural Diagram. The former diagram identifies join points in a structural depiction of message passing among objects and the later locates them during the behavioural flow of activities of the system.

In the second phase, aspects are designed using an Aspect Design Model that models the structural representation of an aspect. The model contains the aspect’s elements and associations among them. A special diagram, known as the pointcut-advice diagram, is nested in the model to represent relationship between pointcuts and their related advices. The rest of the features, such as attributes, operations and inter-type declarations are statically represented in the model.

In the third and the final phase, composition of aspects is designed. There are three diagrams included in this phase. To design dynamic composition of aspects with base classes, an Aspect-Class Dynamic Model has been introduced. It depicts the weaving of advices into the base program during the execution of the system. The structural representation of this weaving is modelled using Aspect-Class Structural Model, which represents the relationships between aspects and base classes. The third model is the Pointcut Composition Model, which is a fine-grained version of the Aspect-Class Dynamic Model and has been proposed to depict a detailed model of compositions at pointcut-level. Besides these models, a tabular specification of pointcuts has also been introduced that helps in documenting pointcuts along with their parent aspects and interacting classes.

AODL has been evaluated in two stages. In the first stage, two detailed case studies have been modelled using AODL. The first case study is an unimplemented system that is forward designed using AODL notations and diagrams, and the second is an implemented system which is reverse engineered and designed in AODL. A qualitative evaluation has been conducted in the second stage of evaluation to assess the efficacy and maturity of the language. The evaluation also compares the language with peer modelling approaches.

Acknowledgements

First and foremost I would like to extend my sincerest thanks to my supervisor, Dr. Gary Allen, whose expertise, experience and knowledge taught me the meaning of research and the way good research can be carried out. I am extremely grateful for his invaluable guidance, mentorship and more importantly his patience with me. Without his consistent support and encouragement, this thesis would not have become a reality.

Secondly, I would like to appreciate my university and my department for generously funding this research. I deeply commend their efforts to support PhD students for making a strong research base in the university. I also owe them thanks for financially supporting me to attend conferences during the period of this research.

I would also like to thank my friends and research fellows, Mr. Munir Naveed and Mr. Shahin Shah, for always being out there for me whenever I found myself down and depressed due to the burden of this research.

And finally I would thank my parents and would like to dedicate this work to them who stood by me with their persistent encouragement and motivation in toughest of times.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

This chapter describes the problem and motivation behind the conception and initiation of this research. It describes the goals and objectives set for this study and the approaches adopted to achieve these objectives. This chapter also provides a list of contributions which have been made to knowledge through this research. The chapter concludes with a description of the structure of the thesis.

1.1 Introduction

Dijkstra (1982) coined the term *separation of concerns* to divide a system into multiple separately manageable parts to make the system easy to specify, implement and document. This approach helps in identifying, designing, implementing, tracing and managing software in an easy way by providing encapsulation of distinct concerns into independent modules. Many approaches, including object-oriented programming, were invented on this very principle (Booch, 1982). Object-Oriented programming decomposesa system into only one dimension,a class hierarchy, which creates a problem commonly known as *tyranny of the dominant decomposition* (Tarr et. al, 1999). This problem results in some concerns being scatteredover multiple classes with theirlogic distributed over several modules. These concerns are known as*crosscutting concerns*as they affect multiple modules and compositional units.Examples of such concerns include logging (Kiczales et al., 1997), authentication (Vanderperren et al., 2003), security (Win et al., 2002) and business rules (Cibran et al., 2003). The logic of these concerns cannot be captured by independent units, such as classes, and is represented in several classes redundantly, causing “scattered” and “tangled” code problems. The scattered code problem arises when code of a particular concern is found in multiple modules, and the tangled code problem happens when such scattered code causes logic of a concern to be present in a module where it does not belong. These problems make systems hard to understand, modify and maintain.

Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) was invented to address this problem at implementation-level. It provides a way of modularizing crosscutting concerns into independent modules. Such concerns are implemented independently from other system concerns, and are linked with themautomatically during the execution of the system. As a result, redundant implementation of these concerns is removed and the system becomes more modular and comprehensible. The identified concerns are called *aspects*. Their implementation is in the form of a piece of code (called an *advice*) which is run at predefined points (called *join points*) during the execution of the system. These aspects merge with the implementation of the base modules during executionthrough a process called weaving, which is defined in terms of composition techniques specific to the aspect-oriented technology in use.

Since AOP was originally proposed as an implementation solution so a lot ofwork has been dedicated to developing new and improved implementation technologies.As a result implementation tools like AspectJ (Eclipse, 2012), AspectWerkz (Boner, 2004),JBoss AOP (JBoss, 2012), and Spring AOP (Spring, 2011) were created. With the passage of time,demand for strategies to support the analysis and design phases of aspect-orientedsoftware development started arising. As a result, strategies such asAODM (Stein, Hanenberg and Unland, 2002), Theme/UML (Baniassad and Clarke, 2004b), SUP (Aldawudet al., 2002) and AAM (Clarke and Walker, 2002) were introduced to facilitate the designof aspects. They all design aspects in different styles. But most of these existingdesign strategies do not provide a common design framework for both aspectualand non-aspectual concerns of the system. This limitation forces the designerto design these concerns in two different design strategies, which makes thewhole design hard to understand, maintain and extend.

The purpose of this PhD work is to develop a unified design framework for aspectual and non-aspectual concerns (objects in the case of object-oriented paradigm). This study aims to develop techniques to represent and design concerns with the help of design notations and design diagrams. The primary hypothesis of this research can be stated as:

> *A unified design approach for aspectual and non-aspectual concerns of a system improves quality of the design and makes it comprehensible and effective.*

The rest of this chapter provides a description about the objectives of the study, approach applied for this research and contributions of this thesis.

1.2 Research Goals and Objectives

This research aims to achieve following goals:

1. **Comprehensibility:** It has been observed in the existing design methodologies that comprehensibility of the design is not properly addressed (discussed in detail in Chapter 3). The proposed notations and models are not developed to improve readability, which becomes even more cumbersome when the system is complex and distributed. The majority of these methodologies propose different sets of notations to model aspect-oriented and object-oriented constructs, which forces designers to learn and adapt to two entirely different design models for a single system. This study aims to propose design notations for aspect-oriented constructs which are similar in nature to those used for object-oriented constructs in UML.

2. **Co-Designing aspectual and non-aspectual constructs:** A unified design framework, which provides a single platform to model both aspect-oriented and object-oriented constructs, can improve the modelling of associations among both the concepts and can also provide a better platform to design the weaving process. This capability of the design methodology can also help in improving comprehensibility.

3. **Achieving Modularity and Composability by breaking Obliviousness:** Steimann (2005) proposed that obliviousness is a fundamental property that must be implemented in every AOSD approach. His proposition was based on the arguments presented in favour of quantification and obliviousness in aspect-oriented programming in (Filman and Friedman, 2000). The proposal was rejected by Rashid and Moreira (2006), who argued that abstraction, modularity and composability are more fundamental properties than quantification and obliviousness. This research will follow Rashid and Moreira (2006) and will focus on achieving better modelling techniques for modularity and composability.

To prove the hypothesis and achieve abovementioned goals, a number of objectives have been set.The primary objective of this research is to provide a unified design framework for objects and aspects. Due to the lack of design languages for the aspect-oriented paradigm, designers struggle to find a good approach to specify, represent, design and document aspects along with objects in the same design environment. The existing aspect-oriented design techniques either lack comprehensiveness or provide separate techniques to handle objects and aspects thatmakeit difficult for the designers to learn and adopt these techniques.

This study aims to achieve the following research objectives:

- To develop a unified design framework for objects and aspects to co-design both the constructs.
- To design notations for aspects similar to those for objects (UML notations) to represent them during the development life cycle
- To develop design diagrams for aspects similar to those for objects (UML diagrams) to be represented and designed properly
- To evaluate and test the design notations and design diagrams by qualitative methods and by applying them to a range of case studies to verify their suitability, efficacy, scalability and comprehensiveness.

1.3 Approach

This study was initiated to provide a unified design approach for aspects and objects. It has been felt that the existing design approaches force designers to adopt two different design methodologies for aspect-oriented and object-oriented constructs. UML is the widely used design language for object-oriented constructs, and most of the designers use this language while designing the base constructs. Whereas for aspect-oriented constructs, a number of design approaches are available, the majority of which are different from UML. The designers have to use two different design languages to design one system which makes it hard for these new aspect-oriented design languages to be adopted. The purpose of this research was to find new ways to unify design of both objects and aspects together in one design environment and co-design both the constructs using similar design notations and diagrams. This property will not only improve the effectiveness of the design but will also enhance comprehensibility of all the models.

The project started with the exploration of aspect-oriented analysis and design techniques. A number of approaches, such as View Point based approach (Rashid et al., 2003), Goal based approach (Yu et al., 2004), Use Case based approach (Jacobson and Pan-Wei, 2005), Scenario based approach (Whittle et al., 2003) and Multiple Dimension Separation of Concerns approach (Tarr et. al, 1999), were studied and analysed in the first stage of the research.

Design of aspects is one area of research that has attracted less attention since the inception of aspect-oriented programming. Although there are a number of aspect-oriented design approaches such as, AODM (Stein et al., Unland, 2002a), Theme/UML (Baniassad and Clarke, 2004b), SUP (Aldawudet al., 2002) and AAM (Clarke and Walker, 2002), which have been proposed over the last decade, a comprehensive design approach is still required. All of these approaches lack a unified design approach for both aspectual concerns and non-aspectual concerns that forces designers to adopt two different design approaches. Multiple design methodologies do not only misguide the designers but also create problems in documenting and representing all concerns properly.

This study has been carried out to find a solution to this problem by introducing a unified design framework that provides similar types of notations for both aspectual and non-aspectual concerns to be represented and designed in a single design environment. A new design language, AODL, has been developed to accommodate representation of aspects and objects together. This language works along with Unified Modelling Language (UML) and provides design notations and design diagrams for aspects, pointcuts, advices and weaving associations of aspects. The design notations and diagrams follow Meta Object Facility (MOF) rules and resemble UML in syntax and semantics.

1.4 Contribution

This PhD study contributes a design language for aspect-oriented software development to represent and design the aspects. It is called Aspect-Oriented Design Language (AODL). This language is similar to Unified Modelling Language (UML) and works seamlessly with it. The following are the main contributions of this study:

- An Aspect-Oriented Design Language (AODL) (Iqbal and Allen, 2011) has been developed that specifies and designs aspectual components. The language has been designed by keeping the syntax, semantics and design constructs similar to those of UML so that traditional UML designers and novice designers should feel comfortable while using it for designing aspects along with base objects.
- AODL provides notational support to all the aspectual components, such as aspects, pointcuts, join points, advices and weaving associations. Each notation is diagrammatically designed and it contains characteristics and feature of the notation.

- The language provides design models to model pointcuts, aspects and their relationships with other aspectual constructs. These models represent the internal structure of the modelling constructs and their associations and relationships with related constructs.
- An aspect composition technique has been proposed that provides support for aspect-to-aspect and aspect-to-base compositions. The technique contains design notations and design diagrams that can be applied to capture these compositions and demonstrate the weaving process in a notational and diagrammatic way. The inner-aspect composition has also been supported in the approach. A new set of notations and diagrams have been developed to compose pointcuts with advices and pointcuts with each other. There are separate diagrams for structural and dynamic compositions for all the constructs.
- This thesis provides a detailed evaluation of the proposed design notations and design diagrams of AODL. The evaluation has been carried out in two stages. In the first stage, a qualitative evaluation of AODL has been performed. A comparison has been made between AODL and existing design and modelling approaches. In the second stage, AODL has been applied to two case studies. One case study system is designed using forward engineering technique and the other is designed in a reverse engineering method.
- A tool is under development for AODL which is aimed to provide an automated environment to use AODL. The tool will also be capable of generating code for visualized models.
- One refereed journal paper and 5 refereed conference papers have been published so far on the work carried out during this research.
- Two more Journal papers have been submitted to refereed journals.

The following are the lists of the publications during this research:

Journal Papers:

1. Iqbal, S. and Allen, G. (2011) ‘Designing Aspects with AODL’ International Journal of Software Engineering. ISSN 1687-6954
2. Iqbal, S. and Allen, G. (2012) ‘Application of AODL: A Case Study’, Software: Practice and Experience, (Submitted).
3. Iqbal, S. and Allen, G. (2012) ‘Composition of Aspects in AODL’, Journal of Systems and Software, (Submitted).

Conference Papers:

1. Iqbal, S. and Allen, G. (2012) ‘Pointcut Design with AODL’. In: The Twenty-Fourth International Conference on Software Engineering and Knowledge Engineering (SEKE 2012), July 1-3, 2012. Redwood City, California, USA. (In Press)
2. Iqbal, S. and Allen, G. (2010) ‘Aspect-Oriented Modelling: Issues and Misconceptions’. In: Proceedings of Software Engineering Advances (ICSEA), 2010 Fifth International Conference. : IEEE. Nice, France. pp. 337-340. ISBN 978-1-4244-7788-3
3. Iqbal, S. and Allen, G. (2010) ‘A notational Design of Join Points’. In: Future Technologies in Computing and Engineering: Proceedings of Computing and Engineering Annual Researchers' Conference 2010: CEARC’10. Huddersfield: University of Huddersfield. pp. 27-30. ISBN 9781862180932
4. Iqbal, S. and Allen, G. (2009) ‘On identifying and representing aspects’. In: SERP'09 - The 2009 International Conference on Software Engineering Research and Practice, July 13-16, Las Vegas, USA
5. Iqbal, S. and Allen, G. (2009) ‘Representing Aspects in Design’. In: Theoretical Aspects of Software Engineering, 2009 TASE 2009, theThird IEEE International Symposium on. : IEEE. China, pp. 313-314. ISBN 978-0-7695-3757-3
6. Iqbal, S. and Allen, G. (2009) ‘Aspect-oriented design model.’ In: Proceedings of Computing and Engineering Annual Researchers' Conference 2009: CEARC’09. Huddersfield: University of Huddersfield. pp. 137-141. ISBN 9781862180857

1.5 Structure of theThesis

The rest of the thesis is structured as follows:

Chapter 2 presents a detailed survey of aspect-oriented software development techniques. It provides a comprehensive survey of the past, existing and on-going research in the field of AOSD.

Chapter 3 contains a detailed description of aspect-oriented design methodologies. It describes in detail the different methodologies proposed so far for designing aspects. A comprehensive survey of the past, contemporary and on-goingresearch in this field has been provided in this chapter.

Chapter 4 describes aspect-oriented design language (AODL), which was developed during this research. A full description about the inception, design and development of the language has been provided. Description about each design notation is provided along with explanation about its usage. A simple case study has been designed using AODL to illustrate the language.

Chapter 5provides a description of the evaluation of AODL by application. The selected case studies have been explained. The process of application of AODL on these case studies has been explained in detail. The results gathered during this evaluation have been presented and explained.

Chapter 6contains a description of the evaluation strategies and their implementation on AODL. A detailed description is provided about the selection of evaluation strategies and their implementation on AODL. Evaluation has been carried out by defining evaluation factors which were implemented on existing peer strategies and AODL to test the efficacy of AODL against its competitive methodologies. A comprehensive description has been provided on the selection of case studies for the evaluation.

Chapter 7 concludes the research by providing a comprehensive discussion on the achievements and learning from the research. This chapter also discusses the possible areas of application of the proposed language along with the limitations and weaknesses of the language. It also provides a description of the possible future research that can be carried out to improve and extend AODL.

Chapter 2: Aspect-Orientation and Requirements Engineering

This chapter introduces the Aspect-Oriented Programming (AOP) paradigm, the reasoning behind its inception, the parameters on which it was invented and all the concepts, terms and terminologies of AOP. The chapter covers details about the separation of concerns, concepts and progress in this area. The chapter also provides a brief survey of aspect-oriented requirements engineering approaches. Aspect-oriented design is discussed separately in Chapter 3.

2.1 Introduction

Aspect-Oriented Programming (AOP) (Kiczales et al, 1997) is used to implement a system in a modular way by separating crosscutting concerns into independent modules. AOP is a programming language which implements concerns that crosscut the system due to their scattered logic in multiple modules. These concerns are known as crosscutting concerns and they affect several modules or units of the system. Crosscutting concerns are not captured in other peer technologies such as functional programming and object-oriented programming and as a result they create problems like redundancy and replication of code and tangling of code. AOP was introduced to address these problems by introducing a new construct, called an aspect, to capture and modularize a crosscutting concern into a distinct construct and to implement it separately from other concerns of the system.

AOP was proposed based on the 1997 PhD thesis by Christina Lopes, titled "D: A Language Framework for Distributed Programming" (Lopes, 1997). Later, George Kiczales and his team formalized the paradigm and introduced this concept to the world in their paper, "Aspect-Oriented Programming" published in 1997 (Kiczales et al, 1997). They argued that although other programming paradigms implement separation of concerns by implementing concerns in distinct units, for example, as objects in object-oriented programming and as procedures in procedural programming, they overlook implementing crosscutting concerns into distinct modules which violates the encapsulation principle of programming and decreases modularity of the system. AOP, on the other hand, implements crosscutting concerns in distinct constructs called aspects and weaves them with other system concerns through composition rules. Consequently, the system becomes modular and reusable.

2.2 Separation of Concerns

The term *Separation of Concerns* was coined by Dijkstra in his paper "On the role of scientific thought" (Dijkstra, 1982). His proposition was to identify and implement all the concerns of the system separately by making each unit do one and only one thing. This idea is a basic founding pillar of most of the implementation paradigms, for instance, object-oriented paradigm separates concerns of the system in the form of objects, service-oriented design separates them in form of services, functional programming separates them as functions and procedural programming as procedures. Similarly, aspect-oriented programming has been designed to separately design and implement these concerns in the form of aspects. The biggest challenge in this kind of development is to pin down what a concern is. Some suggest a concern is a functionality of the system and some consider any piece of interest as a concern. Following are some of the examples of concerns which have frequently been pointed out in the literature (Chitchyan et al., 2005):

- Functional/Application-Dependent Concerns: They are the core functionalities of the system. The examples may include transaction management in a banking system or calculation of toll in a toll system.
- Quality Concerns: These are the concerns responsible for the quality management of the system. These include performance, ease-of-use, reliability, etc.
- Policy Concerns: These concerns are related to policy implementation of the system such as security, user management, access rights, etc.
- System Concerns: These concerns implement efficiency of the system. They include performance management, memory management, efficiency, fault-tolerance, etc.

In the perspective of aspect-oriented software development, there are two types of concerns, core concerns and crosscutting concerns. Core concerns are functional requirements of the system and crosscutting concerns are such functional or non-functional requirements whose implementation is scattered over multiple core concerns.

2.3 Crosscutting Concerns

Some concerns are normally linked with each other or are dependent on each other, which makes their implementation complicated. For example, in a banking system, every transaction has to be logged in the logger and it has to be checked for security. Tracing is also performed on every transaction. If we want to implement this system in an object-oriented technology we might have to implement logging, security and tracing concerns along with the implementation of all transactions. This way, we are implementing logic where it does not belong which clearly violates principles of separation of concerns and encapsulation (as shown in Figure 1). These concerns are known as crosscutting concerns because they crosscut the system by overlapping on multiple implementations (Kiczales et al, 1997).

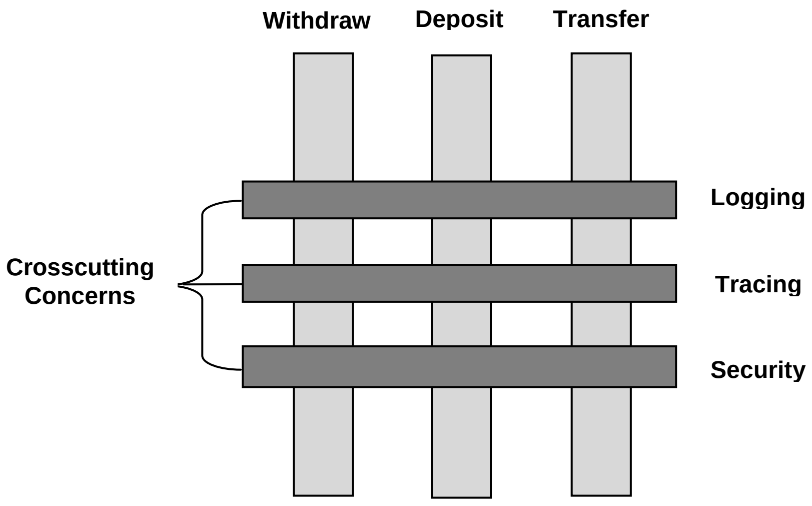


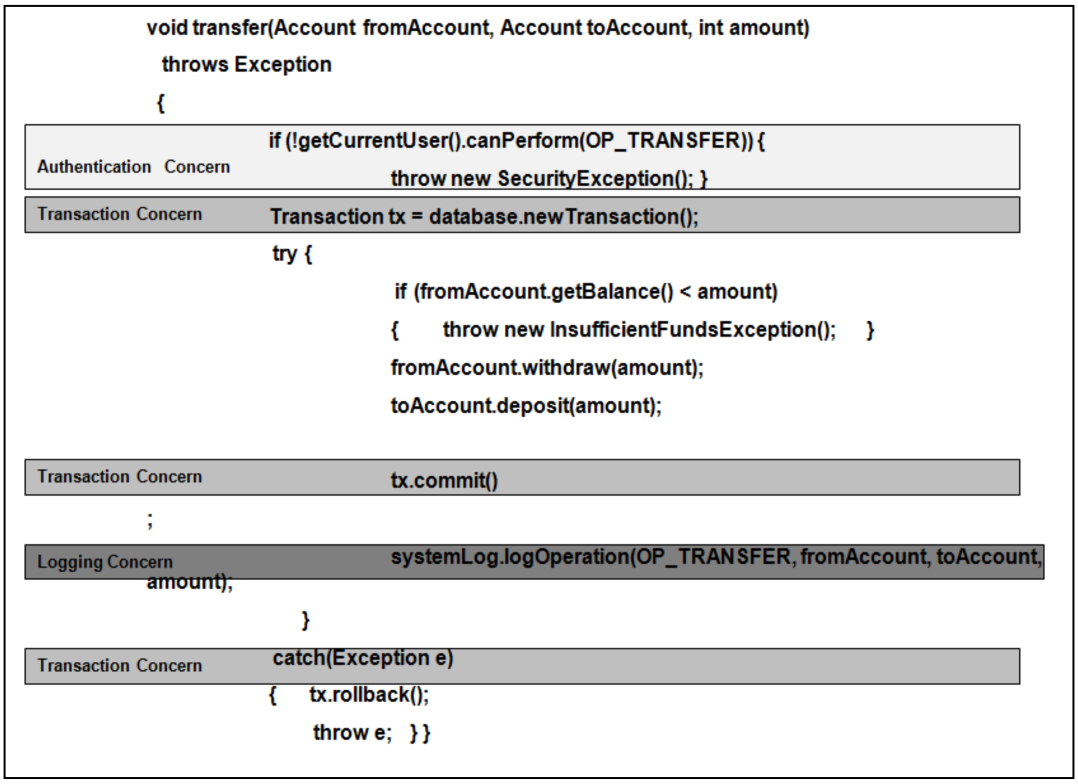
Figure 2.1: Example of Crosscutting Concerns

Several studies have proved that handling crosscutting concerns properly and separating them from other concerns improves quality of the system (Kulesza et al., 2006; Lippert and Lopes, 2000; Lopes and Bajracharya, 2005, Tsang et al., 2004). But separating crosscutting concerns is never easy. Most of the concerns are abstract and they are not properly defined in the requirements or design of the system (Eisenbarth, 2003; Sutton and Rouvellou, 2005) which makes it hard to identify and separate them from other concerns. The crosscutting concerns problem creates two more problems in the system, scattered code problem and tangled code problem.

2.3.1 Scattered Code and Tangled Code Problems

Crosscutting concerns reside at multiple places and they also reside in places where they do not belong as specified in the business logic of the system. Their presence in multiple modules, components and functions causes the scattered code problem (Figueiredo et al, 2005). Scattered code is an anomaly that results in inconsistencies and maintenance problems (Lopes and Bajracharya, 2005). It also makes code hard to test and document because of the replicated code.

The other problem crosscutting concerns generate is the tangled code problem. The code is tangled if it is present in functions, modules or components where it does not belong according to the specified business logic (Kiczales et al., 1997).



```
void transfer(Account fromAccount, Account toAccount, int amount)
  throws Exception
 {
        if (!getCurrentUser().canPerform(OP_TRANSFER)) {
                throw new SecurityException(); }
        Transaction tx = database.newTransaction();
        try {
                if (fromAccount.getBalance() < amount)
                {     throw new InsufficientFundsException();    }
                fromAccount.withdraw(amount);
                toAccount.deposit(amount);

                tx.commit()
 ;
                systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount,
 amount);
            }
        catch(Exception e)
        {   tx.rollback();
            throw e;   } }
```

Figure 2.2: An example of a tangled code (Source: Brito, 2008)

Figure 2 shows an example of tangled code. There are three concerns, Authentication, Transaction and Logging, which do not belong to this particular method according to the business logic of the method but they are yet present in the code.

2.4 Aspect-Orientation

Software Engineering is an evolving field and it keeps on improving with the innovation and new ideas to improve modularity, reusability and extensibility of the systems. Kiczales (1997) suggested that contemporary approaches follow a dominant decomposition criterion which cannot capture all the existing concerns. The problem was named as *tyranny of the dominant decomposition* by Tarr et al (1999) who described that once a decomposition criterion is decided, all the concerns are captured according to that particular criterion leaving other concerns scattered if they do not meet the criterion. History tells us that the majority of development techniques are introduced at the programming level and later are extended to the other phases of the development life cycle. For example, the most renowned technique, object-oriented programming, was introduced in a language named SIMULA-67 but today requirement analysis and design notations have also been developed for the object-oriented paradigm, such as Unified Modelling Language.

Aspect-Orientation was also introduced at the programming level. It started with the introduction of an extended C language named as Composition Filters by researchers at the University of Twente, Netherlands in 1992 (Bosch and Aksit, 1992), followed by Adaptive Programming (Lieberherr, 1996) and Aspect-Oriented Programming (Kiczales, 1997). With the passage of time, Requirements analysis and design strategies were also introduced for all these programming paradigms.

There are two different approaches to separation of concerns, Symmetric and Asymmetric (Harrison et al., 2002). Symmetric approaches employ a single type of construct for both crosscutting and non-crosscutting concerns, thus maintaining symmetry in the representation of both types of concerns, whereas Asymmetric approaches employ two (or more) different kinds of constructs (more detail is given in section 2.6).

2.5 Aspect-Oriented Programming

Object-Oriented programming (Meyer, 1988) is probably the most popular programming paradigm today. The reason behind this paradigm's popularity is its ability of encapsulation and separation of concerns in the form of objects to promote reusability. However, after enjoying two decades of popularity, object-oriented programming started to be questioned as well, just like functional programming and structured programming, on its inability to capture non-functional concerns in separate implementation units (in OO case, objects). As discussed in the earlier section, there are some concerns, such as security and persistence, which cannot be contained in single objects. Their scattered nature compels object-oriented programmers to write them redundantly in multiple places in the program.

Kiczales and his team raised these questions on object-oriented programming in their paper (Kiczales et al, 1997) and displayed the shortcomings of object-oriented programming in handling such concerns properly. They presented a solution in the form of a programming technique, named Aspect-oriented programming (AOP), to address these problems. AOP separates scattering concerns from the system and implements them in distinct system constructs called aspects. This way these concerns are easily identified, implemented, documented and maintained and they are also easily reused and extended.

The following sections provide a detailed description of aspects and their key constituent elements, such as join points, pointcuts and advices. For the sake of consistency, the banking example will be followed to learn the concepts of an aspect and its key elements.

2.5.1 Aspects and Key Elements

As described earlier, a crosscutting concern is a functional or non-functional concern of the system whose implementation is spread over multiple system modules. Such concerns have a scattered nature and cause code tangling problems. Aspect-Oriented programming separates crosscutting concerns from the system and implements them as distinct modular constructs called *aspects*.

Aspects contain the implementation of the crosscutting concern in the form of *advices.* Advices are just like methods and are executed at *join points* in the base system. A join point is a point where a particular aspect has to run its behaviour. Sometimes there are multiple join points where a particular advice of an aspect has to run so these join points are gathered in a set called a *pointcut*. Definitions of key terms of aspect are given in Table 1.

Table 2.1: Explanation of aspectual terms

Term	**Explanation**
Aspect	An abstraction of a crosscutting concern in a program. It contains pointcuts to indicate execution points in the base program and advices to run on those execution points
Advice	The behaviour of a crosscutting concern
Join Point	An execution point where an advice is supposed to execute.
Pointcut	A set of predicates to define related join points
Weaving	A process of incorporating aspect's behaviour (an advice) into base program at a specified execution point (join point).

2.6 Symmetric and Asymmetric Aspect-Orientation

Symmetric approaches treat all concerns of the system equally without dealing with any construct differently because of its nature. The asymmetric approaches, on the other hand, provide different techniques for specifying, designing and implementing aspect-oriented constructs (aspects) and non-aspect-oriented constructs (base elements). Such approaches additionally provide composition rules to compose both types of constructs together at the implementation level.

Symmetry is usually implemented on composable entities, join points and composition relationships (Harrison et al., 2002). The symmetry in designing composable entities entails component-component composition, where each entity is represented as a component. Each component is similar in nature, behaviour and associations. The examples of such components include subjects (Clarke et al., 1999) and Themes (Clarke and Baniassad, 2005). The asymmetric design, on the other hand, implements aspect-component composition, where aspects and components are designed using different methods. The composition of both the constructs is then modelled using composition rules. Aspect technology is a prime example of implementation of asymmetric entities. Join Point symmetry is only defined on the static composition of aspect-oriented constructs (when the composition is performed on lexical basis) (Bálik and Vranić, 2012), so existing AO approaches hardly apply it. The relationship asymmetry is implemented by an element if it defines within its body other elements that it is supposed to be composed with (as AspectJ has aspects that define composable elements in form of pointcuts). The symmetry in relationship is achieved when the information about relationship is kept outside the body of elements (Bálik and Vranić, 2012).

In the following subsections, we will discuss AO approaches that implement symmetric and asymmetric aspect-orientations.

2.6.1 Symmetric Aspect-Oriented Approaches

As stated above, symmetric approaches implement all elements equally by declaring composition rules separately from the bodies of these elements. HyperJ is a symmetric language, which ended at the prototype level and was never used at the industrial level (Ossher and Tarr, 2002). CaesarJ is another language that was based on aspect-oriented symmetry but just like HyperJ could not be adopted on a large scale in industry, except for one controlled experiment mentioned by Rashid et al., (2010). Subject-Oriented programming is also a symmetric language.

Symmetry in aspect-orientation starts from the requirements engineering phase. Some of the aspect-oriented requirements engineering approaches that implement symmetric aspect-orientation are discussed below:

Concern-Oriented AORE Approach

Concern Oriented approach was proposed by Moreira et al (2005a, 2005b) to address the so called tyranny of dominant decomposition. This approach views a system as a set of various concerns which are subsets of abstract concerns.

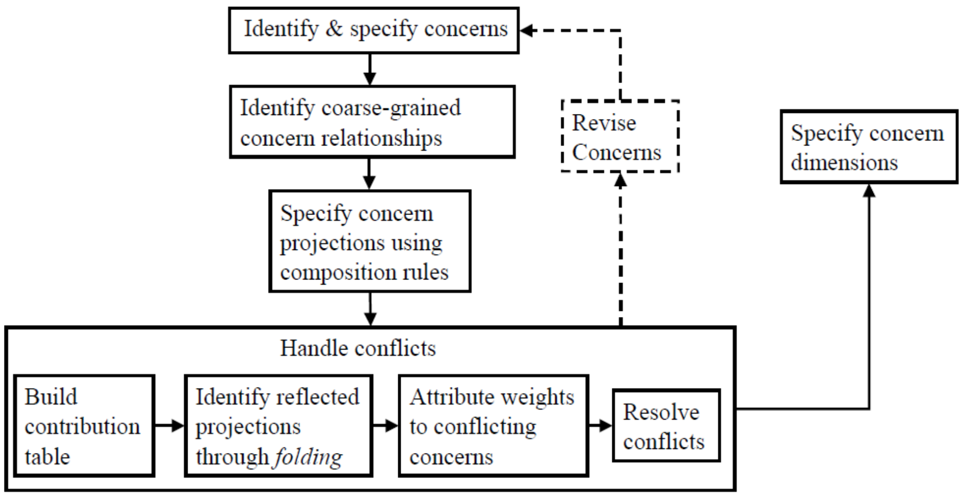

Identify & specify concerns
Identify coarse-grained concern relationships
Revise Concerns
Specify concern dimensions
Specify concern projections using composition rules
Handle conflicts
Build contribution table
Identify reflected projections through *folding*
Attribute weights to conflicting concerns
Resolve conflicts

Figure 2.3: Concern-Oriented Requirements Engineering Process (Source: Moreira, 2005a)

As shown in Figure 5, the process of this approach starts with the identification of concerns using any existing requirements capturing approach. The identified concerns are represented in templates and their relationships are identified by representation in a matrix. The crosscutting behaviour is represented using composition rules. The conflicts are identified using a contribution matrix where each concern makes either negative (-) or positive (+) contribution to other concerns. Conflicts are removed by revising requirement specifications until all conflicts are resolved. At the end, dimensions of each concern are identified using mapping and influence techniques which have also been used in Early Aspect approach (Rashid et al., 2003).

2.6.2 Asymmetric Aspect-Oriented Approaches

PARC AOP ((Kiczales et al., 1997) and AspectJ are the prime examples of tools implementing asymmetry of constructs. AspectJ defines aspect-oriented constructs separately and composes them with base classes during the weaving process.

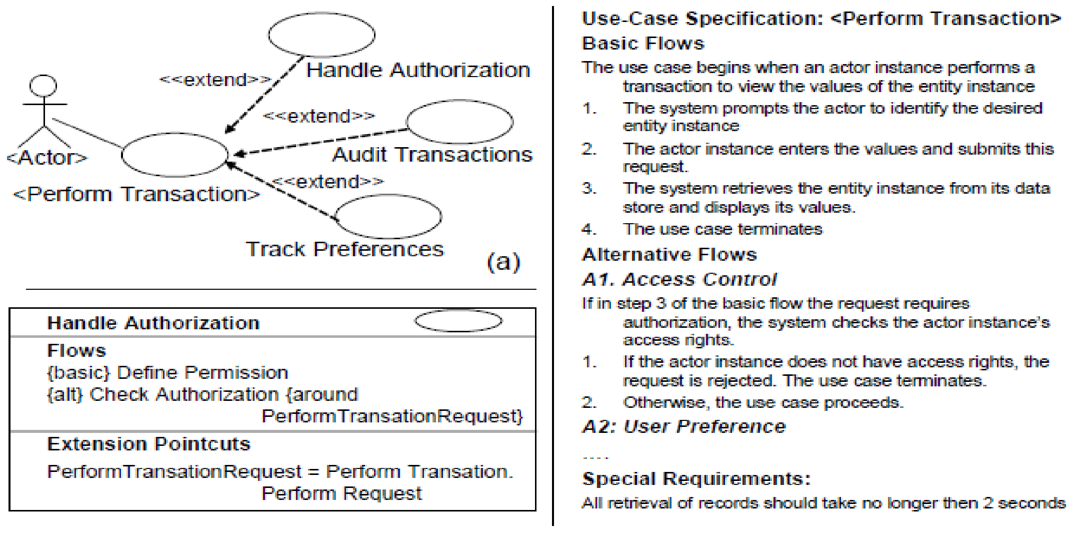
Grundy (1999) and Rashid et al. (2002, 2003) are considered to be some of the earliest approaches that introduction aspect-orientation in requirements engineering. Some of the other renowned requirements engineering approaches that implemented asymmetric aspect-orientation are discussed below:

Use Cases Based AORE Approach

Jacobson (2003) proposed that systems should be designed by breaking down use case diagrams into use case slices and use case modules as overlay on top of classes. These overlays can then be composed using any suitable aspect-oriented technology to form a complete system. Jacobson (2003) suggested that use cases are crosscutting concerns as their realization spans over multiple classes.

In (Jacobson and Ng, 2005) the authors have also presented a method for aspect-oriented software development with use cases. They have extended traditional use cases with two more elements, pointcuts and artefacts for use case slices and use case modules:

- Pointcuts have been represented as sets of related join points which are represented by extension points (Jacobson, 2003)
- A use case slice contains information about a particular use case at a given phase of development and a use case module contains all types of information about the use case throughout the development cycle.



Handle Authorization

Flows
{basic} Define Permission
{alt} Check Authorization {around PerformTransationRequest}

Extension Pointcuts
PerformTransationRequest = Perform Transation. Perform Request

Use-Case Specification: <Perform Transaction>

Basic Flows

The use case begins when an actor instance performs a transaction to view the values of the entity instance

1. The system prompts the actor to identify the desired entity instance
2. The actor instance enters the values and submits this request.
3. The system retrieves the entity instance from its data store and displays its values.
4. The use case terminates

Alternative Flows

A1. Access Control

If in step 3 of the basic flow the request requires authorization, the system checks the actor instance's access rights.

1. If the actor instance does not have access rights, the request is rejected. The use case terminates.
2. Otherwise, the use case proceeds.

A2: User Preference

....

Special Requirements:

All retrieval of records should take no longer then 2 seconds

Figure 2.4: A typical use case in AOSD with Use Cases Approach

(Source: Jacobson and Pan-Wei, 2005)

Figure 4 shows a typical description of a use case in this approach. The template <*Perform Transaction*> represents capturing of a non-functional requirement. A non-functional requirement is represented as an extension of a use case. The advantage of representing non-functional requirement in a template is that it helps in visualizing the context of a requirement and it also aids in identification of extension points.

AORE Using Theme/Doc

Theme/Doc (Baniassad and Clarke, 2004a; Clarke and Baniassad 2005) proposed a requirement engineering approach for aspect-oriented systems. In this approach *Theme* is the core concept which represents a distinct and meaningful unit of the system. Themes are similar to functionalities of the system. They are represented as Theme/Doc at the requirement level and Theme/UML at the design level. Theme/Doc is supported by a tool which captures four views of the requirements to identify themes (shown in Figure 6). These views are (i) action, (ii) clipped, (iii) theme and (iv) theme augmentation (Chitchyan et al, 2005).

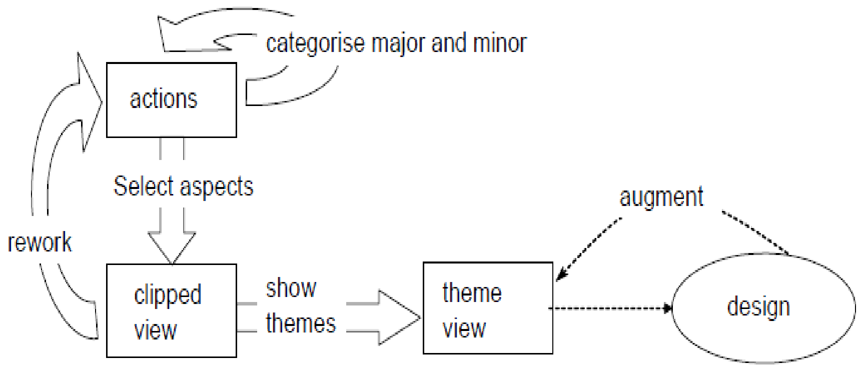


Figure 2.5: Theme/Doc Process (Adopted from: Chitchyan et al, 2005)

The Theme/Doc approach supports identification of aspects by capturing concerns with shared requirements at action view. It then verifies the design decision at the augmentation view. The approach provides good traceability as one can map requirements from Theme/Doc views to Theme/UML models.

AORE Component-Based Approach

In this approach, components are identified using any component based requirements analysis approach and then aspects are identified either by separating crosscutting features in the components or by using component specification and design information (Filman, 2005). The approach was introduced to identify aspects in reusable components. This is an asymmetric aspect-oriented approach as it only handles crosscutting concerns (Brito, 2008).

AORE with Arcade

This is a view-based approach that extends the traditional viewpoint method along with design notations for crosscutting concerns and their composition. This approach introduced the renowned "Early Aspect" (Rashid et al., 2002) term to denote identification of aspects at the requirements engineering level. The approach uses viewpoints and provides a multi-dimensional separation of concerns through the software development life cycle (Rashid et al., 2002, 2003). An XML-based composition mechanism complements the technique to separate and compose aspectual requirements. The process model of AORE with Arcade approach is shown in Figure 3. Concerns are modularized and composed by producing a requirements specification document which ensures consistency by detecting conflicts through requirements composition (Rashid et al., 2003).

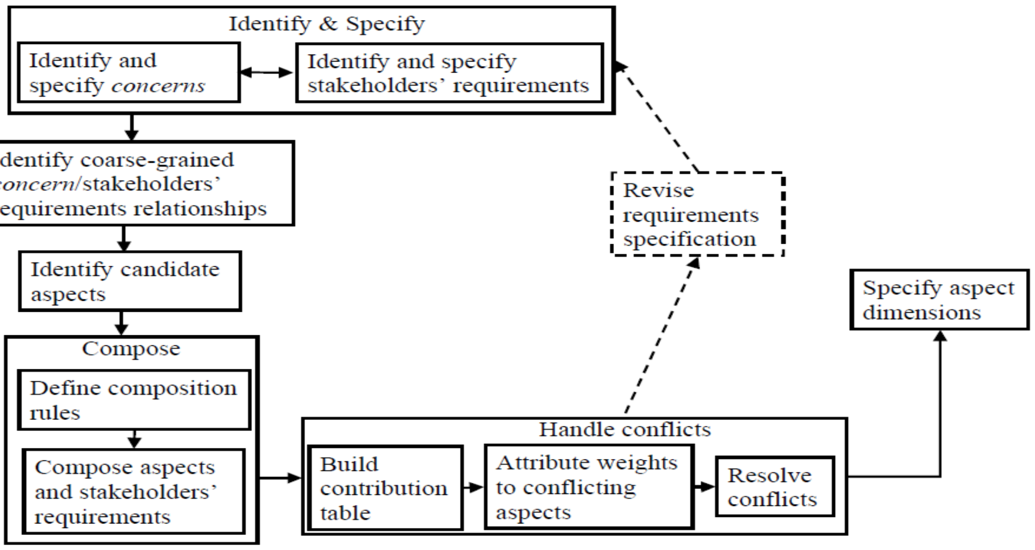

Identify & Specify
Identify and specify *concerns*
Identify and specify stakeholders' requirements
Identify coarse-grained *concern*/stakeholders' requirements relationships
Identify candidate aspects
Compose
Define composition rules
Compose aspects and stakeholders' requirements
Revise requirements specification
Specify aspect dimensions
Handle conflicts
Build contribution table
Attribute weights to conflicting aspects
Resolve conflicts

Figure 2.6: The process model for AORE with Arcade approach

(Source: Rashid et al., 2003)

2.7 Comparison of Aspect-Oriented Requirements Engineering Approaches

There are weaknesses in almost all of the AORE approaches mentioned above. An overview of problems with each approach is given below:

The Aspect-Oriented Component Requirements Engineering Approach is only specific to component-based development and does not support other development paradigms. In addition, the approach does not help in identifying aspects from every component and it also lacks tool support.

AORE with Arcade is the most cited aspect-orientedrequirements engineering approach. It is a simple and straight forward approach that provides a multi-dimensional separation of concerns through the development lifecycle, which makes it more traceable compared to other AORE approaches.

AORE with Use Cases approach is similar to UML. It does not provide any mechanism to handle conflicts. Since this approach forms use case slices and use case modules for all the concerns of the system so it can be regarded as a symmetric approach.

Concern-Oriented AORE approach provides a multi-dimensional mechanism to separate concerns from requirements. As this technique is applied on all concerns so it can be regarded as a symmetric approach. It provides support to effectively manage early tradeoffs and negotiations among stakeholders. This approach is also equipped with a tool support.

Theme/Doc AORE approach is although a mature approach, it still lacks in providing support to specify and compose concerns in a systematic way despite being useful at the requirements analysis level. It also does not support traceability scalability as the technique becomes so complicated and cumbersome due to the size of *Theme Views* required for large systems.

2.8 Discussion

It is hard to say which approach is better, symmetric or asymmetric. If we look at the record of adoption of both the approaches, asymmetric approaches have received more recognition and have been adopted in industry more than symmetric approaches. The reason probably is the traditional *Separation of Concerns* concept. It is always more convenient and rather comprehensible to keep crosscutting and non-crosscutting concerns separate during the entire development life cycle until they are composed with each other dynamically. In one of the current research projects by Bálik and Vranić (2012), it has been argued that there are always concepts in the proclaimed asymmetric approaches that can be considered as symmetric. For example, peer uses case and features in the analysis and design techniques and traits (Scala), open classes (Ruby), or prototypes (JavaScript) in the programming languages. Similarly, inter-type declarations and advices can also be considered as symmetric concepts if everything is modelled using aspects and the base code is kept as thin as possible. In this case, intertype declarations can be used to define structure and initial method bodies. Advices can then implement the behaviour. This correlation suggests that asymmetric approaches can always be evolved to be symmetric if it is desired. It is also suggested that neither approach can be hailed to be better than the other; rather it is their functionality and efficacy that matters.

2.9 Chapter Summary

This chapter starts with the introduction of the term Separation of Concerns. It describes in detail how concerns are handled in different programming paradigms. Core concerns and crosscutting concerns have been described and the differences between the two have been shown with the help of examples. Definition of an aspect and how it is implemented in an aspect-oriented system has been described in detail with the help of an example. Key elements of an aspect, such as join points, pointcuts and advices have also been described with examples. A banking example has been selected to demonstrate implementation of an aspect and its constituent elements.

The chapter then explains the symmetry of aspect-orientation and the languages and requirements engineering techniques that follow either symmetric or asymmetric approaches. The chapter then provides an in-depth analysis of aspect-oriented requirements engineering approaches. Each approach has been described and compared to show the strengths and weaknesses of the approaches. The next chapter will discuss contemporary Aspect-Oriented Design (AOD) approaches in detail.

Chapter 3:
Aspect-Oriented Design Approaches

Aspect-Oriented design has been perceived differently by different researchers. Some put modularity of the system as their first objective and some consider composition of aspects with base constructs as the most important factor. This chapter discusses some of the well-known aspect-oriented design approaches in the light of these different characteristics, and provides a detailed literature survey of this area of the research.

3.1 Software Design

One of the pioneering software design methodologists, J. Christopher Jones, commented in his book, Design Methods: Seeds of Human Futures (Jones, 1970), that design methodologies have been moving away from 'drawings and patterns' in the notion of design. The same applies to contemporary design methodologies. Software Design methodologies started appearing in 1950s and 1960s and with time it became an established scientific field. Many researchers have described design in their own way. Lawson (1980) and Dasgupta (1989) described design projects as a combination of real or perceived needs where a need acts as a motivational starting factor for initiating a design project. Similar description has been provided by Willem (1990) who says that integral feature of a design is devising of a plan or prototype for the development of something new. Some design methodologists believe that software design is a symbolic representation of an artefact for implementation (Zhu, 2005) and some consider design as a simulation of a work that we want to do for a number of times until we develop the final product (Freeman, 1980). Simon (1973) explained design as the restructuring of a current product to develop a preferred product and Page (1966) described design as an 'imaginative jump from present facts to future possibilities'.

3.2 Aspect-Oriented Design

Software design is the structural and behavioural representation of the requirements specification. Requirements are shaped into implementable elements, entities or functions in a software design. Due to the complexity of contemporary systems, software design must provide support for the abstraction of system elements and separation of concerns. Object-oriented programs are usually designed in the Unified Modelling Language (UML). UML provides both behavioural and structural representation of the system. For example, for behavioural representation interaction diagrams and state diagrams are used and for structural representation object and class diagrams. UML also allows designers to show the abstraction level of the classes of the system. If we evaluate the ability of UML for separation of concerns, we will have to evaluate object-oriented programming first since UML is an object-oriented modelling language. Object-orientation encapsulates the business logic of concerns within objects. Objects are the units of development in Object-Oriented Programming (OOP). There are some concerns of the system which are not fully handled in OOP, such as performance, persistence, fault-tolerance, etc. Such concerns affect or have connection with more than one object, thus, their representation and code is scattered over the system. Such scattered nature of code causes tangling problem of code. To counter these problems, aspect-oriented programming (AOP) has been proposed that implements such tangled and scattered elements as aspects.

To design aspects, a number of aspect-oriented design approaches have been introduced over the years. Aspect-oriented design (AOD) approaches allow designers to design aspects, their constituent elements, features and associations. They also provide mechanisms to connect aspects' behaviour (advices) to their corresponding join points in the base program. Some of the well-known AOD approaches are discussed in detail in the following sections.

3.3 Aspect-Oriented Design and Modelling Approaches

There are a number of aspect-oriented design approaches currently available. Each approach sees crosscutting concerns in its own way and proposes a methodology to design them. There are two types of recognized AO design methodologies, symmetric and asymmetric. The design languages that propose modelling techniques for both crosscutting and non-crosscutting concerns and designs both of them in a same design framework are called symmetric approaches. Whereas, those design languages that only support design of crosscutting concerns are known as asymmetric approaches.

The other common property among existing modelling approaches is extension of UML notations. Some of them have used UML profiles while some extended UML metamodels. There are only a few which are not based on UML, such as Sutton and Rouvellou, (2005) and Suvee´ et al. (2005). To the best of our knowledge, there are thirty UML-based approaches, namely; Grundy (2000), Ho et al. (2002), Zakaria et al. (2002), Stein et al. (2002a), Kande et al. (2003), Rausch et al. (2003), Von (2004), Ivers et al. (2004), Clarke and Baniassad (2005), Elrad et al. (2005), Pawlak et al. (2005), Reddy et al. (2006a), Coelho and Murphy (2006), Cottenier et al. (2007a), Fuentes et al. (2007), Jacobson and Ng (2005), Krechetov et al. (2006), Katara and Katz (2007), Klein et al. (2007), Lau et al. (2007), Paula and Batista (2007), Bustos and Eterovic (2007), Fuentes et al. (2007), Whittle et al. (2007), Albunni and Petridis (2008), Cui and Xu (2009), Li et al. (2010), Guessi et al. (2011), Gupta et al. (2011) and Evermann et al. (2011). We will only be discussing eight out of these thirty approaches. The reason behind the selection of these approaches is their maturity (number of publications and total citations), and similarity with AODL with respect to proposed notations. A brief summary of notational dependency and number of publications of these eight approaches has been provided in Table 3.1.

Table 3.1 - Graphic Nodes included in Join Point Identification Diagrams

Approach	Notational Dependency	Publications	Notable Publications	Most Cited Publication, citations
AODM	AspectJ, UML	4	Stein et al., 2002a, 2002b, 2003, 2006	Stein et al., 2002a, 194
Theme/UML	AspectJ, UML	>15	Clarke et al., 1999, Baniassad & Clarke, 2004a, 2004b, Clarke & Baniassad, 2005	Clarke & Baniassad, 2005, 338
Motorola Weavr	Motorola Weaver	9	Cottenier et al., 2007a, 2007b, 2007c	Cottenier et al., 2007b, 74
AAM	UML	>10	France et al., 2004, Reddy et al., 2006, Kim et al., 2004, Solberg et al., 2005, Muller et al. (2005)	France et al., 2004, 197
AOSD/UC	UML	3	Jacobson and Ng, 2005	Jacobson and Ng, 2005, 369
JAC Design Notations	UML	3	Pawlak et al., 2002, 2005	Pawlak et al., 2002, 76

Klein's Approach	UML	9	Klein et al., 2005, 2006, 2007	Klein et al., 2006, 90
SUP	MSC	5	Aldawud et al., 2003; Elrad et al., 2005	Aldawud et al., 2003, 132

There are a number of properties that an AOD approach must possess to be considered as an effective approach. A number of evaluation criteria for aspect-oriented modelling approaches have been proposed based on these properties, some of the noted ones are by Wimmer et al. (2011), Blair et al. (2005), Chitchyan et al. (2005), Op de beeck et al. (2006), and Reina et al. (2004). We have chosen some of the most important properties and have assessed the selected eight approaches against each property. AODL has also been evaluated against these properties (along with some additional general software design properties) in Chapter 6. These properties are as follows:

- **Language:** There are some means adopted by AOD approaches to specify and design concerns. Some approaches adapt or extend a modelling language, such as UML, and some propose their own language or design methodology. The language contains artefacts, notations and diagrams to specify and model concerns, their behaviour and associations. The languages that adapt UML either utilize UML notations or extend some of the notations to specify constructs.

- **Design process:** A design approach must follow a defined design process, containing a set of activities to design concerns from specification to composition. Some approaches offer a well-defined design process and some suggest an implicit way of designing concerns in the form of manuals and guidelines. This parameter has also been adopted by Op de beeck et al. (2006) and Wimmer et al (2011) to evaluate AOD approaches.

- **Concern Specification:** The language must also provide support for specification and representation of crosscutting concerns and their associations. The specification can be in the form of design notations, diagrams or textual narrations. In any case, properties and relationships of the concerns must be well-represented.

- **Modelling of Structural and Behavioural Crosscuttings:** A design methodology offers support for both structural and behavioural modelling of concerns and their constituent elements. For instance, in UML, class diagram, component diagram and object diagram are used to depict structural associations and state machines, activity diagrams and interaction diagrams are used to show behavioural properties of the system. Similarly, an AOD language must also offer both types of representation for system concerns.

- **Concern Composition:** Once crosscutting concerns are modelled as aspects, they are composed with base classes through predicates defined in their pointcuts. The composition is required to be modelled before implementation so that aspect interference and conflicts are identified and resolved.According to Kojarski and Lorenz (2006) there are two types of asymmetric compositions, pointcut-advice composition and static crosscutting composition and one type of symmetric composition usually known as compositor. The pointcut-advice composition provides a representation of internal compositions of an aspects and static crosscutting depicts the relationships between aspects and base classes. The compositor mechanism, on the other hand, contains identification of composable element, specification of match method and development of integration strategy describing how the matched elements will proceed after composition (Wimmer et al., 2011).
- **Conflict Resolution:** Conflicts arise as a result of aspect composition. There could be a number of general, domain-related or application-related conflicts that can be encountered during aspect compositions but we will only talk about two of the most common conflicts. One is the shared join point problem, which occurs when two aspects try to impose their behaviour at a join point simultaneously. The second is aspect interference, which arises when an aspect changes or disturbs the definition of a join point or an aspect. The modelling approaches which propose composition strategies must also support conflict resolutions.

The following sections discuss the selected aspect-oriented design approaches in light of these properties:

3.3.1 Aspect-Oriented Design Modelling (AODM)

The Aspect-Oriented Design Modelling (AODM) approach proposed by Stein at al., (2002a and 2002b) is an asymmetric design approach which was developed initially for the AspectJ programming language. It later evolved into a more generic approach by providing support for composition filters and adaptive programming besides AspectJ (Stein et al., 2002c, 2003, 2006).

Language

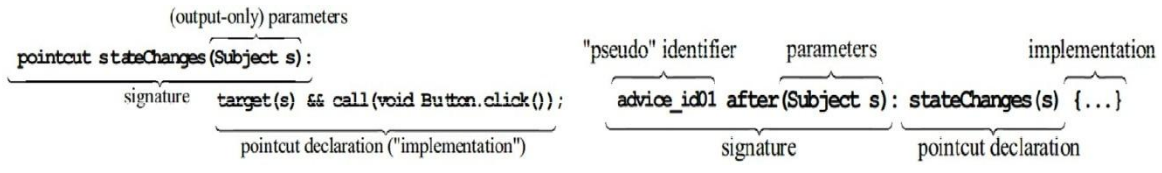
UML has been adopted as the basis for representation and specification of aspects. New notations and diagrams have been introduced which are extended versions of UML artefacts. Though there have been some efforts for turning the approach into a generic one, it still relies heavily on AspectJ platform.

Design Process

The design process is missing in AODM. There are no guidelines provided about the order of usage of the diagrams. The approach does address the majority of the design issues of aspect-oriented design, such as static crosscutting, dynamic crosscutting and composition, but it does not provide a step-wise set of activities to design these issues in order.

Concern Specification

AODM argues that an aspect is similar in structure to a UML class. It also considers pointcuts and advices similar to UML operations. Figure 3.1 shows these claimed similarities.



```
pointcut stateChanges(Subject s):
    target(s) && call(void Button.click());
```

```
advice_id01 after(Subject s): stateChanges(s) {...}
```

(a) Similarity between a pointcut and an operation (b). Similarity between ad advice and an operation

Figure 3.1: Structural Similarities between a pointcut, an advice and an operation (Stein et al., 2002a)

Aspects are contained in a class-like container and are represented with a stereotype<<aspect>> to distinguish them from classes (as shown in Figure 3.2).

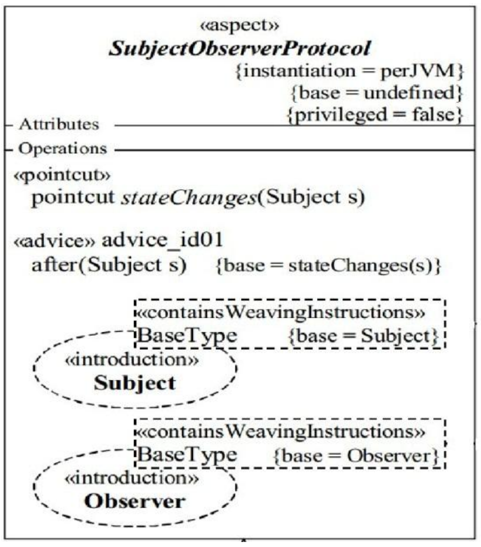


Figure 3.2: Representation of an aspect in AODM (Stein et al., 2002a)

Modelling of Structural and Behavioural Crosscuttings

AODM supports both structural and behavioural types of crosscuttings. The structural crosscutting, which is also known as static crosscutting, involves the introduction of new data types or members of the base class. These additional base characteristics used to be known as “Introductions” in the earlier versions of AspectJ, but now they are known as Inter-type declarations. AODM uses UML parameterized template collaboration diagrams as containers to hold the depiction of structural crosscuttings, as shown in Figure 3.3. UML class diagrams and sequence diagrams are exploited to represent the crosscutting, and templates of the collaboration are used to hold information about the base types.

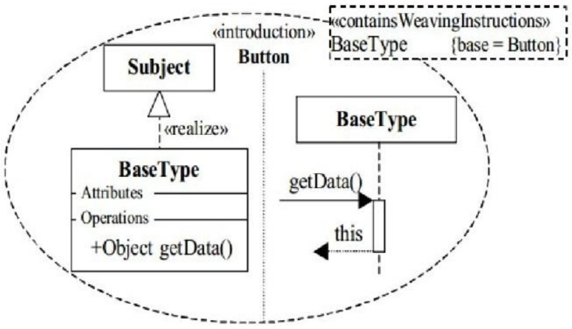


Figure 3.3: Representation of structural crosscutting in AODM (Stein et al., 2002a)

AODL represents behavioural crosscutting by specifying advices with a stereotype <<advice>>, as shown in Figure 3.2. As stated before, AODM considers advices similar to UML operations so they are represented like them. The problem is that they do not have a distinct identifier; they are represented with the signature of the pointcut they are related with. To counter this problem, AODM introduces pseudo identifiers.

Concern Composition

In AODM, composition is captured in a package that contains two types of diagrams (shown in Figure 3.4). The first diagram, which is shown at the left side of Figure 3.4, represents the structure of join points with the help of class diagrams where aspectual behaviour will be woven in. The right side of diagram depicts the behaviour of the join points with the help of sequence diagrams. The crosscutting is depicted in a class diagram with a "crosscutBy" property shown against the operation that contains the join point. The template of the package diagram contains the information about the aspect that is crosscutting the base classes. The join point is depicted with the help of sequence diagrams where the actual location of a join point is indicated with the help of stereotypes, for instance, in Figure 3.4, <<call>> of op1() has been depicted.

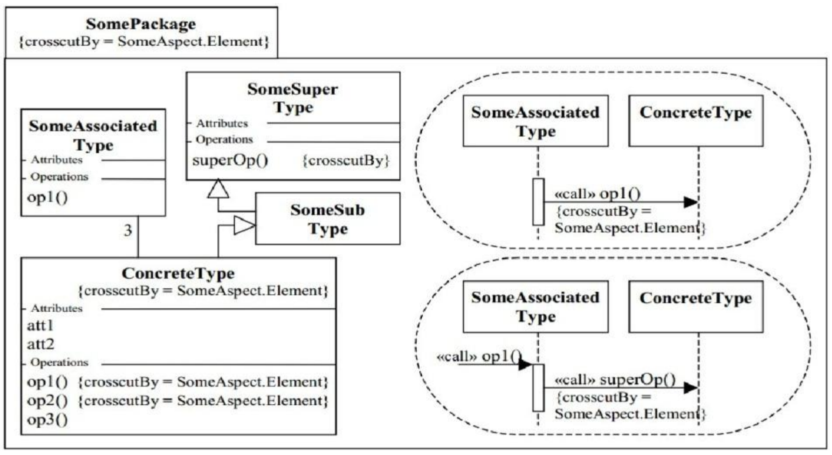


Figure 3.4: Join Point Indication Diagram (Stein et al., 2002c)

Figure 3.5 shows how a collaboration containing a join point is split into three sub-collaborations to show the insertion of advices at *before*, *around* and *after* a method call.

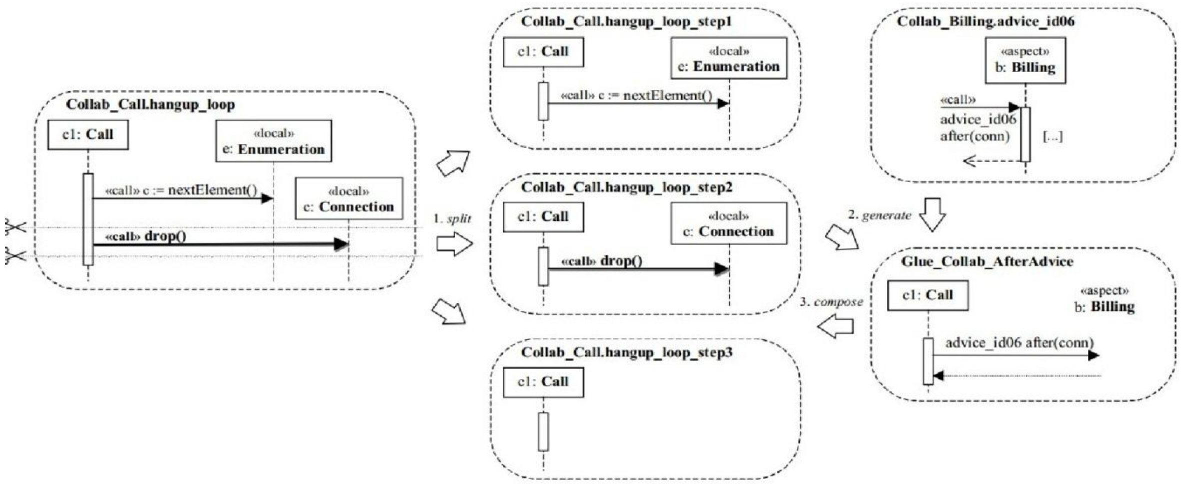


Figure 3.5: Weaving Collaborations (Stein et al., 2002b)

Conflict Resolution

There is no comprehensive conflict resolving mechanism provided by AODM. There is only support for resolving conflicts regarding priority of execution of aspects with the help of a stereotype <<dominates>> (Stein et al., 2002a). This stereotype points from an aspect whose priority is greater to the one with lesser priority.

Limitation and Weakness of AODM Approach

AODM treats aspects as UML classes which is a problematic comparison. The issue has been discussed in detail in (Iqbal and Allen, 2009). As discussed in this paper, classes are object-oriented elements. They are fundamentally encapsulating, inheritable and instantiable constructs. If we assess aspects based on these properties, we first of all find aspects contradicting the basic principle of encapsulation (or data-hiding). Aspects do have their own data but they also access other classes' private data to perform their functionality. For example, Security and Logging aspects need to access the private data of the interacting base classes, which is a clear violation of object-oriented encapsulation. Secondly, Inheritance can partially be implemented in aspects. Aspects can have child aspects but child aspects cannot override advices of the parent aspect because parent aspect's advices do not have unique signatures or identifiers. Finally, instantiation of aspects is not similar to that of classes either. Aspects are instantiated on need, not on demand like classes and objects. Their instantiation cannot be coded within the program; rather their instantiation depends on defined control points (join points) during the execution of the program. This dynamic nature of aspects' instantiation again contradicts the behaviour of classes and objects.

Another similar problem is relating pointcuts and advices with UML operations. Pointcuts cannot return values like operations. They have parameters passed by the base classes to establish a join point, but there is no need of returning any type which is contrary to operations (a problem also pointed by AODM team in (Stein et al., 2002a)). Secondly, pointcuts cannot have local data variables; the reason behind this is that they do not process anything. They are merely used to represent join points as predicates in the program. Now looking at advices, they also have some remarkable behavioural differences to the class's operations. First, they do not have unique and identifiable signatures. This is the reason that aspects do not allow overridden advices in the child aspects. Second, they are dependent on the declaration of a corresponding pointcut. AODM does not provide design notations to specify pointcuts and advices. There is no diagrammatic support either for associations among these constructs. Finally, the approach lacks a design process and there are no guidelines to define the order in which models should be designed.

3.3.2 Theme/UML Design Methodology

Theme/UML (Baniassad and Clarke, 2004b, Clarke and Baniassad, 2005) is a design approach which is implemented on identified Themes in the system with the help of a Theme/Doc (Baniassad and Clarke, 2004a) approach. Themes are concerns of the system which include both crosscutting and non-crosscutting concerns. Theme approach, being a symmetric design approach, designs all concerns of the system as Themes. There are two separate techniques for capturing and designing Themes. The Theme/Doc approach finds Themes from the requirements specification document and the Theme/UML approach designs them.

Language

The approach was developed for the first time for subject-oriented programming paradigm (Clarke et al., 1999). Later, it evolved to accommodate composition filters (Clarke, 2002), AspectJ and HyperJ technologies as well. The current Theme/UML approach is a heavy weight extension of the UML metamodel 3.1 (Clarke and Baniassad, 2005).

Design Process

A three-phased design process has been proposed for Theme approach (Clarke and Baniassad, 2005) that provides step-wise processes to capture and design Themes from analysis to implementation phase. The first phase is the analysis phase in which themes are identified. The second phase is the design phase where identified themes are specified and technically represented. The third and final phase is the composition phase where composition of themes is specified and designed.

Concern Specification

Since Theme/UML is a symmetric approach so it designs both aspectual and base concerns as Themes and represents them in the UML package diagram. The diagram contains the <<theme>> stereotype to identify Themes.

There is a slight difference in aspect and base themes representations. The aspect Theme is represented in a parameterized template package whereas base theme is represented in a simple package diagram. Figure 3.6 depicts representation of an aspect Theme.

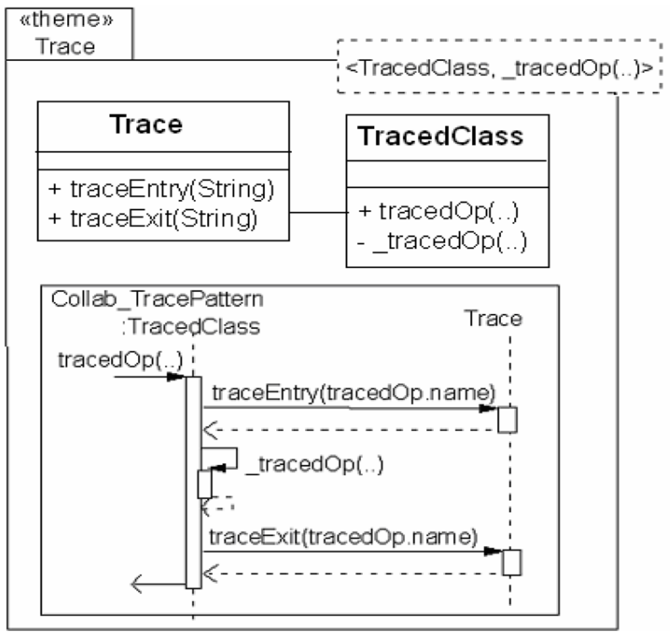


Figure 3.6: An Example of a Theme depicted in Theme/UML (Clarke and Baniassad, 2005)

Parameter contains crosscutting information. A join point can be declared in the parameter as shown in Figure 3.6 where a **tracedop()** operation of **TracedClass** shows that the join point is defined on this particular operation.

Modelling of Structural and Behavioural Crosscuttings

Both structural and behavioural types of crosscutting are depicted within the Theme package. As shown in Figure 3.6, structural crosscutting is represented with the help of a UML class diagram. The aspect Trace is related with **TracedClass** which is a base class through an operation **tracedOp()**. The behavioural representation of this relationship is depicted in a UML sequence diagram, as is shown in Figure 3.6.

Concern Composition

The semantics of theme composition have been described in detail in Clarke (2001). A composed *<<theme>>* is generated by composing related themes. An example of an *ObserverLibrary* theme is shown in Figure 3.7 in which the Observer theme is composed with the Library theme. Besides theme-level compositions, the approach also offers composition at more fine-grained level, for instance, compositions at attributes and class levels.

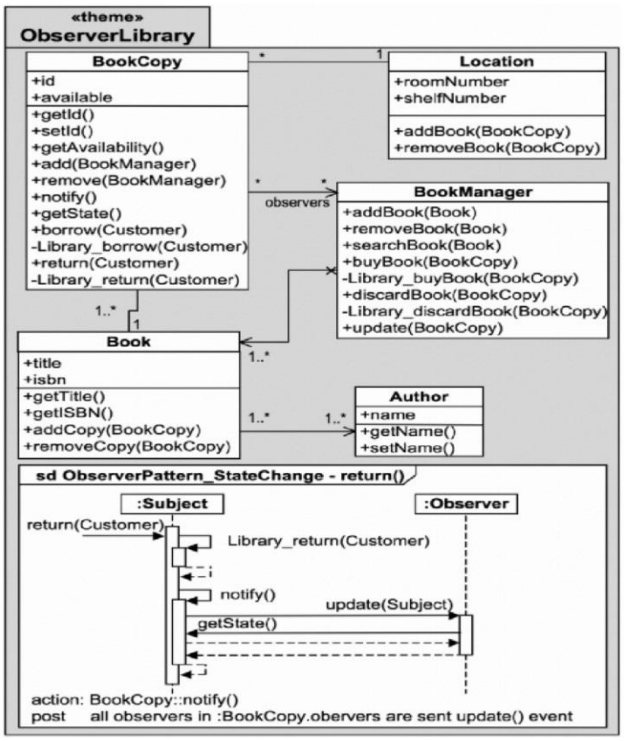


Figure 3.7: The Theme/UML Composed Model (Clarke and Baniassad, 2005)

Conflict Resolution

The conflict resolution is performed by tagging the theme templates (Clarke and Baniassad, 2005). For example, "prec" tag defines the precedence of themes and resolves the ordering clashes, and in the case of theme-level conflicts, "resolve" tag allows users to add more elements to themes (such as visibility attributes) in order to resolve them. The "resolve" tag also allows defining some special elements in themes to resolve any type of theme-level conflicts.

Limitations and Weaknesses of Theme/UML Approach

The approach is well-defined in the available literature but it is too complex to be adopted by UML designers. The diagrams in Theme/UML become even more complicated when the system is complex and distributed. One of the major reasons is the design notations of Theme/UML, which are different from those of UML. Although parameterized templates are used as the primary notation to represent themes, the extensions to the template make it different from UML and reduce its adoptability. Another reason adding to the complexity of Theme/UML models is a lack of a proper technique to model interactions between concern modules. There are composition relationships, borrowed from UML metaclass relationships, which are used for fine grained interactions but notational support for representing association among abstract constructs would have been a better solution to reduce the complexity.

Theme/UML does not provide support for gradual refinement of design models. It is probably because Theme/UML is currently not supporting architectural design. Another weakness of Theme/UML is limited support for modelling all types of join points. At the moment, only execution join points are being supported. The biggest problem of all isabsence of a detailed resultant model after composition of concerns. The composition process is well-defined but if concerns are separated, it becomes hard to picture the overall system. A resultant composition model would have sorted this problem out. The approach also lacks design representation for aspectual elements such as join points, pointcuts, advices and inter-type declarations.

3.3.3 Motorola Weavr Approach

The Motorola Weavr approach (Cottenier et al., 2007a, 2007b, and 2007c) has been developed in a telecom software industry and is aided with a tool that implements all the semantics and design techniques of the approach. It is an asymmetric design approach which means it only supports specification and design of crosscutting concerns (aspects).

Language

The approach is based on the Specification and Description Language (SDL). Composite structure diagrams and transition-oriented state machines of UML 2.0 are used to design aspects. Since SDL has some other design constructs as well besides the ones used in UML, so a UML profile has also been proposed to support design of such constructs. The design approach was initially designed for telecommunication industry, but with the passage of time it has evolved into a platform-independent approach.

Design Process

The approach is comprehensive providing support for representation of all constructs; however, the only problem is that no design process or guidelines are provided to support a procedural way of designing.

Concern Specification

Aspects are represented in transition-oriented state machines. An example representation of BookCopy is shown in Figure 3.8(e) which uses UML notations and the same representation is modelled using transition-oriented state machines in 8(f). The basic design representation is captured using UML class diagrams but the approach also uses composite structure diagrams to refine models designed in class diagrams (Wimmer et al., 2011).

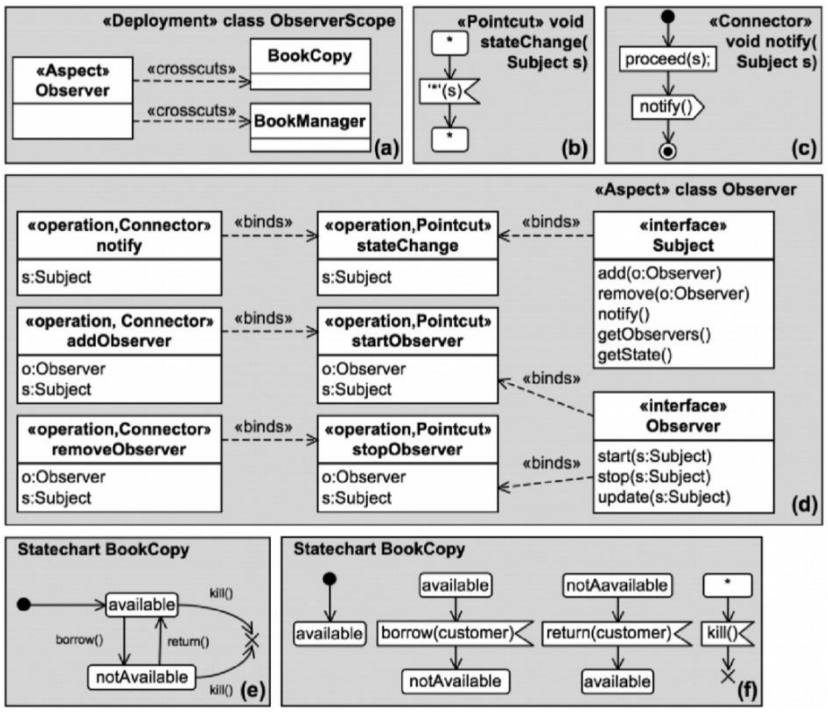


Figure 3.8: The Observer aspect represented in Motorola Weavr approach (Adopted from Wimmer et al., 2011)

Modelling of Structural and Behavioural Crosscuttings

The behavioural crosscutting is modelled using transition-oriented state machines (as shown in Figure 3.8(c) and 8(f)) and the SDL action language. The UML sequence diagrams are also used to define test cases. The composition of the concern modules can be represented in an extended version of the UML deployment diagram. The structural crosscutting is modelled using transition-oriented state machines and class diagrams.

Concerns Composition

Aspects are represented with a stereotype <<*Aspect*>>. The aspects are composed with each other and with base classes. The composition of pointcuts with advices is also supported which is represented along with aspect compositions. The approach follows composition asymmetry which means aspects can be composed with base classes but not the other way around (Cottenier et al, 2007c). The aspect-class association is represented with a stereotype <<*crosscuts*>> in the composition model. Only the static weaving of aspects into base models is supported. The concern composition semantics, however, are clearly defined.

Conflict Resolution

A conflict resolving technique has also been proposed in (Cottenier et al, 2007c) in which a keyword <<*follows*>> has been introduced in order to resolve ordering issues among aspects. The approach has claimed that the shared join point problem can also be resolved using this technique (Zhang et al., 2007d).

Limitations and Weaknesses of Motorola Weavr Approach

One of the modelling weaknesses of Motroala Weavr, also pointed out by Zhang et al., (2007d), is the loosely decoupled nature of pointcuts and advices. The advices are named and are not tightly coupled with only one pointcut as is the case in AspectJ. This approach has certain advantages and makes the design model more comprehensible but the problem arises in the modelling views when Motorola Weavr allows joining of multiple pointcuts with one advice as long as their interfaces are compatible. The model allows dragging and dropping of an advice onto multiple pointcuts inducing them to create direct reference to one advice.

Another related problem is the limited advice type. There is support for only one advice type in the modelling of pointcuts and that is around, which is also used for before and after types (Zhang et al., 2007d). This limitation increases complexity in the modelling of pointcuts with advices.

There is also no support provided by the approach for intra-aspect compositions. The approach is also missing a design process.

3.3.4 Aspect-Oriented Architecture Modelling (AAM)

Aspect-oriented Architecture Modelling (AAM) approach (France et al., 2004; Reddy et al., 2006) was proposed to specify concerns from middle to high design levels. This approach follows role-based metamodelling and is defined on UML 2.0.

Language

An extension to UML, known as UML-based pattern language, is used to design role-based constructs in AAM. Aspects are defined into two types, context free and context-specific. Context free aspects are represented at high design level and are reusable types of aspects. Context specific aspects are instances of context free aspects and are specified according to their role during the design process. The language used in AAM approach is platform-independent.

Design Process

The approach was primarily proposed for architectural solutions for aspect-oriented systems. It lacks detailed design support for concern representation and composition. The composition strategies proposed by the approach focus on architectural composition of concerns only.

Concern Specification

Parameterized template package diagrams are used to specify high-level aspect. The approach is very much similar to Theme/UML with respect to use of templates. The template model elements are marked with a special element '|' to distinguish them from general templates. This notation has been borrowed from Role-Based Metamodelling Language (France et al., 2004; Kim et al., 2004).

Modelling of Structural and Behavioural Crosscuttings

Aspect models and base models are designed differently from each other. Aspect models are designed using template diagrams which are described by parameterized packages. The models are explained with the help of class diagram as shown in Figure 3.9(a), communication diagram as shown in Figure 3.9(b) and sequence diagram templates as shown in Figure 3.10. The structural crosscutting is represented with class diagrams while behavioural crosscutting is depicted in communication diagrams.

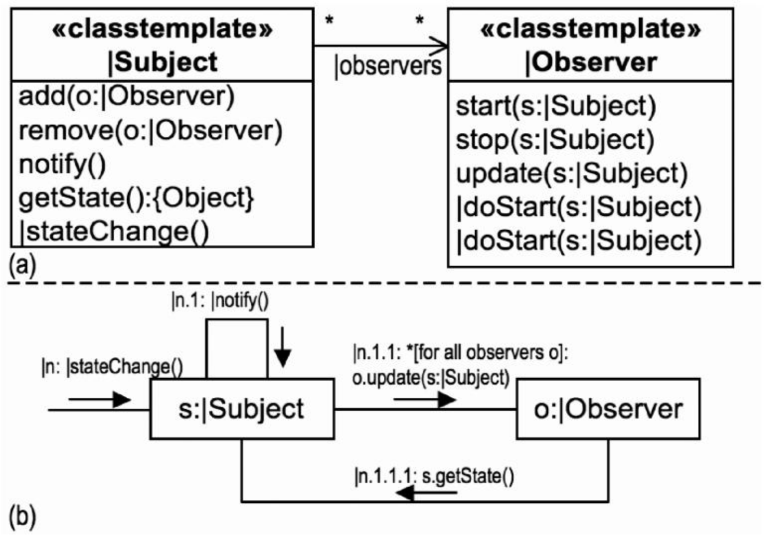


Figure 3.9: The Observer Aspect Model depicted in AAM (Adopted from Wimmer et al., 2011)

Concern Composition

Initially, a compositor composition strategy similar to the Theme/UML approach was adopted. Recently, however, new diagrams based on UML sequence diagrams have been introduced (Reddy et al., 2006; Solberg et al., 2005). Both aspect and base models are specified in UML packages, these packages are then composed together based on textual binding that composes context-related template packages together. Figure 3.10 and 11show static and dynamic composition of the models respectively.

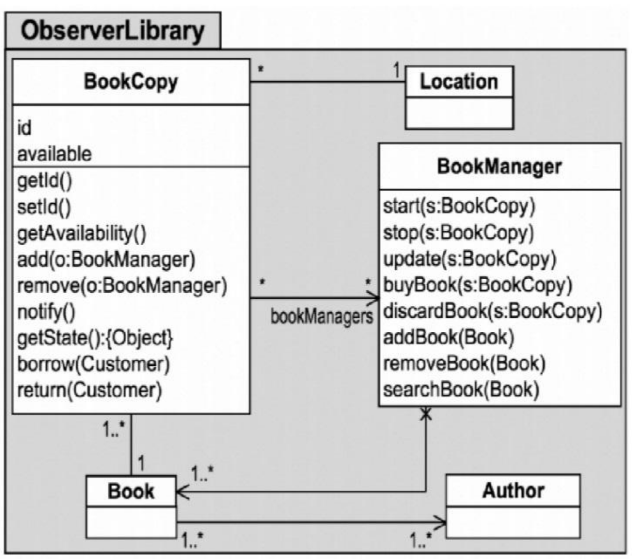


Figure 3.10: A Composed Model in AAM Adopted from Wimmer et al., (2011)

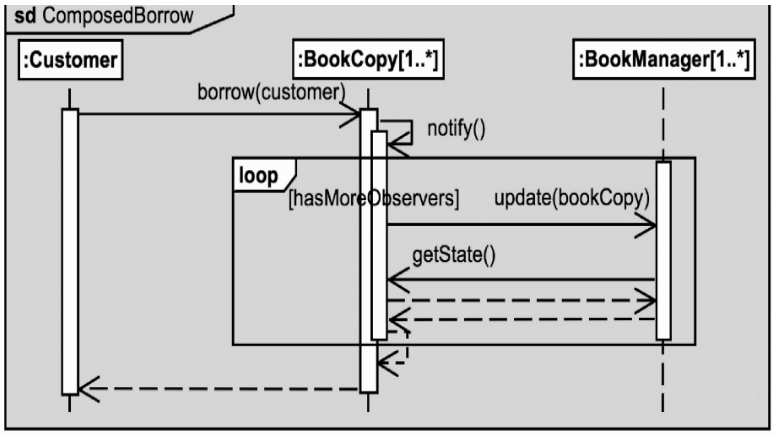


Figure 3.11: Weaving Aspectual Behaviour in AAM Adopted from Wimmer et al., (2011)

Conflict Resolution

Syntactical conflicts can be detected using operationalized techniques proposed by Muller et al. (2005). The paper has also introduced composition semantics and directives to help with composition and conflict detection. Dependencies among aspects are resolved with the help of two stereotypes <<*hidden_by*>> and <<*dependent_on*>>.

Limitations and Weaknesses of AAM Approach

The approach was primarily proposed for architectural solutions for aspect-oriented systems, which is why it lacks detailed design support for concern representation and composition. The composition strategies proposed by the approach focus on architectural composition of concerns only. There is no formal design process available either which makes it hard to model and document concerns properly. No diagrammatic or notational support has been provided for specifying and modelling inner-aspect components such as pointcuts, advices and inter-type declarations. To make the approach more comprehensible, France et al., (2004) and Kim et al. (2004) has proposed notations based on Role-Based Metamodelling language with an additional symbol '|' to distinguish the constructs from those of the language. This approach hampers the comprehensibility even further rather than improving it as the exploited language is less-known and all the aspectual constructs are not well-represented by the proposed notations. The approach primarily focuses on architectural representation; hence traceability from analysis to implementation phase is not supported. As far as internal traceability is concerned, it is only limited to tracing concerns from requirements engineering stage to architecture modelling stage. The scalability of the approach has also not been addressed in the available literature. The approach is yet to be tested on complex systems involving several concerns. The approach lacks tool support as well.

3.3.5 Aspect-Oriented Software Development with Use Cases (AOSD/UC)

AOSD/UC (Jacobson and Ng, 2005) is a software development method based on use cases which is an extension of UML 2.0 metamodel. It is a symmetric approach which means it provides design support for all the concerns of the system. Use cases represent concerns of the system. This method identifies crosscutting concerns and non-crosscutting concerns from the use case diagrams and provides a systematic approach to specify and design them throughout the software development cycle.

Language

An extension to UML 2.0 metamodel has been proposed to represent aspectual constructs. The approach is influenced by AspectJ and HyperJ technologies. The notations and semantics of both the technologies have been mentioned in the literature and used in the development of the proposed techniques.

Design Process

AOSD/UC follows a design process which separates concerns from the analysis phase down to implementation phase in form of use case slices.

Concern Specification

All concerns are modelled with a stereotype <<*use case slice*>>. The crosscutting concerns are represented from analysis phase down to implementation phase. There are a number of UML diagrams that are utilized to identify, specify and design concerns. An Aspect is considered as a classifier and is represented with a stereotype <<*aspect*>>. A graphical notation, in the form of a box with two internal compartments, has been designated to represent an aspect which also contains pointcut declarations and class extensions. Utility and reusable aspects are represented with parameterized template packages.

Modelling of Structural and Behavioural Crosscuttings

Class diagrams are used to represent structural crosscutting and sequence diagrams are utilized to depict behavioural crosscuttings. Figure 3.12 shows a depiction of an Observer aspect that shows the structural and behavioural representation of crosscutting.

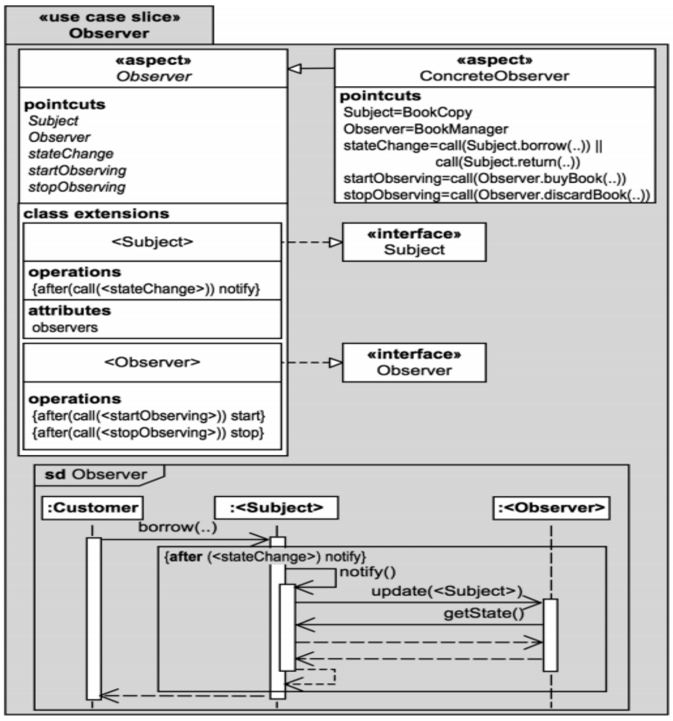

«use case slice» Observer
«aspect» Observer
pointcuts
Subject
Observer
stateChange
startObserving
stopObserving
class extensions
<Subject>
operations
{after(call(<stateChange>)) notify}
attributes
observers
<Observer>
operations
{after(call(<startObserving>)) start}
{after(call(<stopObserving>)) stop}
«aspect» ConcreteObserver
pointcuts
Subject=BookCopy
Observer=BookManager
stateChange=call(Subject.borrow(..)) || call(Subject.return(..))
startObserving=call(Observer.buyBook(..))
stopObserving=call(Observer.discardBook(..))
«interface» Subject
«interface» Observer
sd Observer
:Customer
:<Subject>
:<Observer>
borrow(..)
{after (<stateChange>) notify}
notify()
update(<Subject>)
getState()

Figure 3.12: The Observer aspect modelling using AOSD/UC notations (Jacobson and Ng, 2005)

Concern Composition

There is no composed model provided by the approach. AspectJ's rules of composition are followed. No strategies have been provided to compose aspects with base classes. Aspect to aspect composition is supported but there is no support available for pointcut to pointcut compositions.

Conflict Resolution

Although a clear approach for resolving conflicts has not been presented in the literature, some refactoring methods have been suggested to remove conflicts from the design models.

Limitations and Weakness of AOSD/UC Approach

The approach provides support to design aspects comprehensively but aspectual elements are not separately represented and designed. As far as composition of concerns is concerned, there is no formal method to do it and regarding inner-aspect compositions, only pointcut-advice composition is supported. There is no mechanism available for pointcut-pointcut composition. The approach is comprehensible in a sense that it utilizes UML notations and diagrams but there are some relationships, such as crosscutting and execution precedence among aspects, which cannot be captured by traditional UML semantics.

Similarly, aspects, pointcuts, advices and inter-type declarations require new notations to be represented because of their different nature from object-oriented constructs. There is no tool-support available for the approach either.

3.3.6 The JAC Design Notation

The JAC Design Notation method (Pawlak et al., 2002, 2005) has been designed for the JAC Framework, which has a complete IDE and supports modelling of aspectual components.

Language

The approach presents a light-weight extension of UML. The authors do not claim full compliance with UML rules but assert the simplicity and intuitive nature of the notations. The approach uses UML 1.0 metamodels to extend the diagrams.

Design Process

There is no defined design process proposed by the approach.

Concern Specification

Aspects are specified just like UML classes. A stereotype <<*aspect*>> is used to distinguish them from classes. Just like classes, they contain methods and attributes with additional information about crosscutting.

Modelling of Structural and Behavioural Crosscuttings

The approach only uses class diagrams. Both structural and behavioural crosscuttings are represented by class diagrams with additional stereotypes. Stereotypes <<*role*>> and <<*replace*>> are used for representation of structural crosscutting and <<*before*>>,<<*after*>> and <<*around*>> are used for behavioural crosscutting. Figure 3.13 demonstrates both types of crosscutting for an observer aspect.

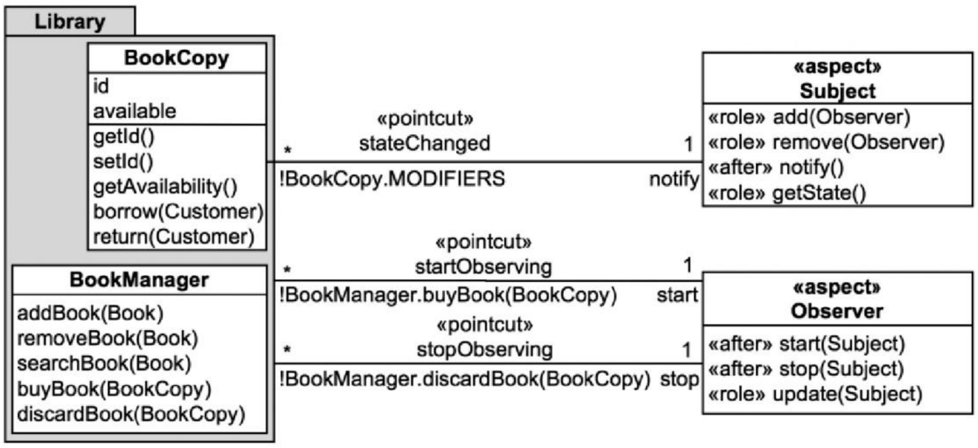


Figure 3.13: The observer aspect modelled using the JAC design notation (Adopted from Wimmer et al., 2011)

Concern Composition

Aspects are composed with each other and with base classes on the structural level using class diagrams. However, there is no support for intra-aspect compositions provided by the approach.

Conflict Resolution

There is no support provided by the approach for resolving any type of conflicts.

Limitations and Weaknesses of JAC Design Notation Approach

Aspectual elements (such as pointcuts, advices and inter-type declarations) are statically represented in this approach. There is no notational support for modelling these elements, which reduces the comprehensibility a great deal. This limitation also increases coupling of the elements with base classes and with other aspectual elements.

The approach implements a light extension of UML profiles. Just like AODM, aspects are modelled in a similar fashion as classes of the base program are designed in UML. It has been discussed before in the limitation of AODM that aspects and classes are altogether different constructs. The former is a non-object construct whereas later is a pure object-oriented. Modelling them in a similar way raises a number of problems and confusions.

There is no also support for designing aspectual elements. The approach only uses class diagrams to design structural representation of concerns. There is no support for developing detailed design models. It does not offer a design process either. Although structural representation of crosscutting is represented with class diagrams, no support is available for behavioural representation. The approach also does not provide diagrammatic or notational support for representing aspectual elements. Regarding composition of concerns, there is no support for inner-aspect compositions, such as pointcut-advice and pointcut-pointcut compositions. Moreover, no rules have been proposed by the approach for resolving aspectual conflicts.

3.3.7 Klein's Approach for Behavioural Aspect Weaving

Klein's Approach (Klein et al., 2006, 2007) only provides technique for behavioural modelling of weaving process of aspects. It does not address specification of aspects and crosscutting at all.

Language

The approach is based on Message Sequence Charts (MSC) which is a scenario based language. The UML 2.0 sequence diagram has been largely used. Scenarios are represented using sequence diagrams. A simplified UML metamodel for sequence diagrams have been provided by Klein et al., (2007). The approach is also platform-independent.

Design Process

There is no formal design process provided for the approach.

Concern Specification

The approach does not provide any support for specifying aspectual components. It only deals with representing behavioural modelling of weaving in sequence charts which are then modelled using sequence diagrams.

Modelling of Structural and Behavioural Crosscuttings

There is no support available for modelling of structural or behavioural crosscutting. The approach is still immature and only deals with weaving process.

Concern Composition

In Klein's approach, each aspect contains two distinct scenarios. One defines the behaviour of the aspect (represented by pointcuts) which is then completed or replaced by advices (an example is shown in Figure 3.14).

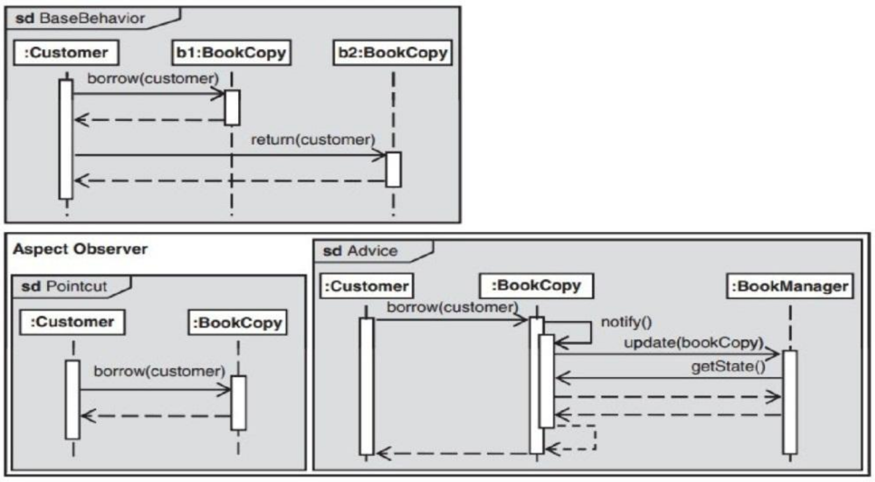


Figure 3.14: A modelling of Observer aspect in Klein's Approach

The same process is repeated for every pointcut of the aspect. Composition is performed in two phases. In the first phase, join points defined in pointcuts are detected in the base program. While in the second phase, advices are composed with the base behaviour as specified in the pointcut and advices. An example of a composed model is shown in Figure 3.15 below.

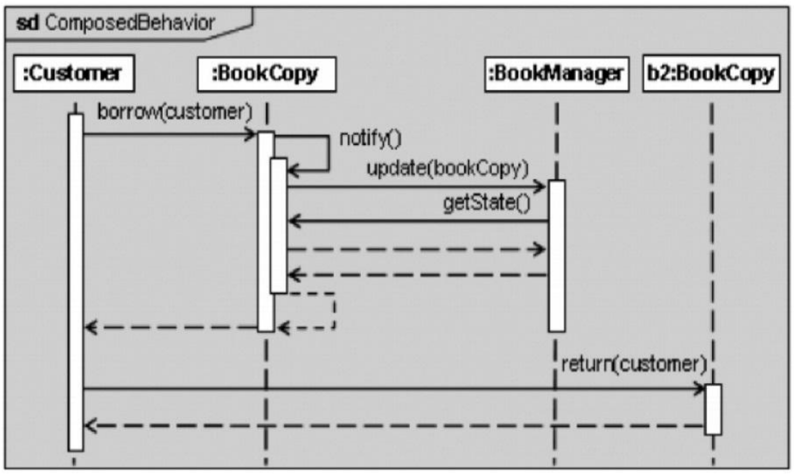


Figure 3.15: An Example of a Composed Model in Klein's Approach (Adopted from Wimmer et al., 2011)

Conflict Resolution

The approach does not provide any method to resolve conflicts and issues arising as a result of aspect compositions.

Limitations and Weaknesses of Klein's Approach

The approach does not suggest any method to specify concerns and no diagrammatic and notational support is available to represent structural or behavioural characteristics of concerns. The approach only offers modelling solutions for composition of aspects. There is no method proposed to design structural or behavioural crosscutting and there is no design process or guidelines available to formalize modelling. Inner-aspect compositions are also not supported by the approach. No conflict resolving techniques are available for aspect compositions and there is no tool-support provided for the approach.

3.3.8 State Chart and UML Profile (SUP) Approach

The SUP approach (Aldawud et al., 2003; Elrad et al., 2005) is an analysis and design approach for capturing and designing aspects. It is complemented with an aspect-oriented design language as well which is based on a UML profile. It is a platform-independent approach.

Language

The language of the approach is based on UML 1.x. A UML profile has also been proposed to introduce new aspectual constructs in UML modelling. The approach uses class diagrams extensively along with state machines. Each class diagram is refined step-by-step to a state machine representation.

Design Process

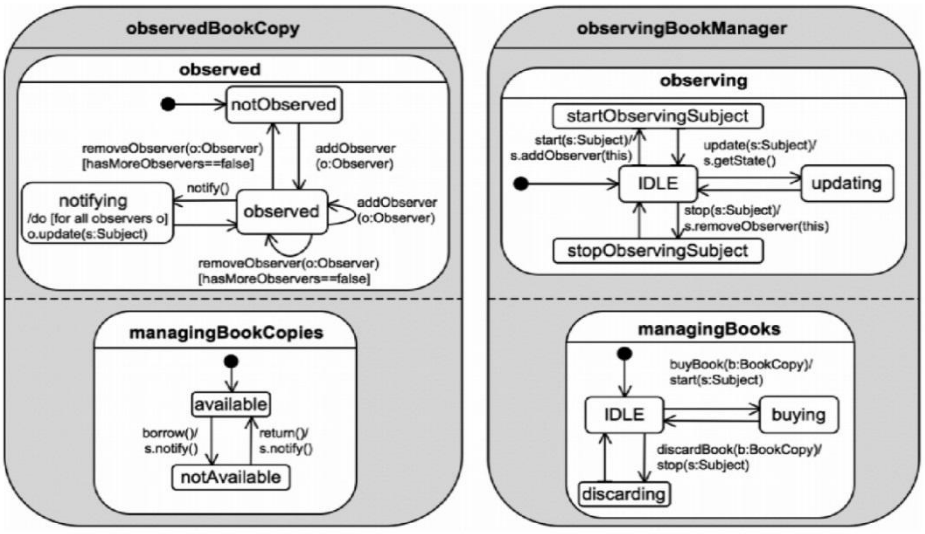
A set of guidelines has been provided on modelling the aspectual behaviour. There is a step-wise design process to refine class diagrams to state machines.

Concern Specification

Aspects are specified in class diagrams. A specific stereotype <<*aspect*>> is allocated to represent aspects. There are two types of aspects, synchronous which can alter the control flow and asynchronous, which cannot alter the control flow. Both types of aspects bear a specific tag *(<synchronous>* or *<asynchronous>*) to represent their nature. Once the class diagram representation is refined into state diagrams, aspects are represented in state machines.

Modelling of Structural and Behavioural Crosscuttings

Structural crosscutting is represented in class diagrams and behavioural crosscutting is modelled using state charts, use cases, state machines and collaboration diagrams. Figure 3.16 provides an illustration of modelling using SUP approach.



observedBookCopy
observed
notObserved
removeObserver(o:Observer) [hasMoreObservers==false]
addObserver (o:Observer)
notify()
notifying /do [for all observers o] o.update(s:Subject)
observed
addObserver (o:Observer)
removeObserver(o:Observer) [hasMoreObservers==false]
managingBookCopies
available
borrow()/ s.notify()
return()/ s.notify()
notAvailable
observingBookManager
observing
startObservingSubject
start(s:Subject)/ s.addObserver(this)
update(s:Subject)/ s.getState()
IDLE
updating
stop(s:Subject)/ s.removeObserver(this)
stopObservingSubject
managingBooks
buyBook(b:BookCopy)/ start(s:Subject)
IDLE
buying
discardBook(b:BookCopy)/ stop(s:Subject)
discarding

Figure 3.16: A representation of Observer aspect using SUP approach (Adopted from Wimmer et al., 2011)

Concern Composition

Concerns are composed through linking events in the state diagrams. The process happens on the flow of events from one state to another. Links are established among events when related states interact with each other. There are no formal guidelines on the process, rather informal composition semantics have been described in Elrad et al. (2005).

Conflict Resolution

The state charts provide sequence of events which can be considered as a solution to the ordering problem so one can say that an implicit conflict-resolving mechanism is provided.

Limitations and Weaknesses of SUP Approach

The approach does not provide high-level abstractions and has not yet been tested on complex systems to suggest scalability. No support has been provided for diagrammatic or notational representation of aspectual elements, such as pointcuts, advices and inter-type declarations. Similarly, no technique is provided to compose inner-aspect elements either. The light-weight extension is possible as UML profiling does allow introducing new features, attributes and relationships other than what are already defined but no support for heavy-weight extension is provided. The approach uses UML profiling, which does not allow introduction of new non object- oriented constructs. External traceability is supported from the requirements to design phase but there is no support for internal traceability available. The approach also does not have tool-support available yet.

3.4Discussion

As gathered from the findings in the Limitation and Weaknesses section of each aspect-oriented modelling approach, no approach is mature enough to be adopted comprehensively yet. There are limitations attached to every approach. One property that is missing in almost all of the discussed approaches is the notational support for inner-aspect components and intra-aspect compositions. The other property that is lacked by the majority of the approaches is provision of a design process. AODL possesses both of these properties. A detailed evaluation of AODL against these properties and some additional software design properties is provided in Chapter 6.

3.5 Chapter Summary

This chapter surveys the available literature on contemporary aspect-oriented design and modelling approaches. Only those approaches have been discussed which are similar to AODL. Each approach has been discussed in light of vital characteristics which should be possessed by an effective aspect-oriented design methodology. The limitations and weaknesses of each approach have also been summarized.

Chapter 4: Aspect-Oriented Design Language

This chapter describes the main contribution of this research, Aspect-Oriented Design Language (ADOL), which has been developed to specify, represent, design and document aspects, aspectual elements, associations and relationships between aspects and base elements and compositions of aspects with the base design. The chapter starts with the objectives and motives behind the language followed by the explanation about each design notation and related design diagrams.

4.1. Introduction

Aspects are identified and captured during the requirements engineering and analysis phase. A number of requirements engineering approaches have been proposed for identification of aspects over the years (details in chapter 2). This thesis does not follow a specific aspect capturing technique rather aspects are assumed to have been identified using a suitable methodology. This thesis only discusses a design language for aspects, which provides design notations and design diagrams to specify, represent, design and document the identified aspects. The language is called Aspect-Oriented Design Language (AODL) and it primarily focuses on providing a design technique, a method and a set of notations and diagrams to effectively design aspects along with base constructs. Following are the primary objectives which have been achieved during the development of the language:

1. To unify design of aspects and objects in a single framework.
2. To develop design notations for aspects and constituent elements.
3. To represent structural and behavioural characteristics of aspects diagrammatically.
4. To develop a language that provides comprehensive design solutions for aspects and their relationships with objects.
5. To design a diagrammatic approach to model both intra-aspect and aspect-base compositions.

4.2. Motivation

Object-oriented systems can be effectively designed in the Unified Modelling Language (UML). UML (OMG, 2012) provides design notations and diagrams to form design models to identify, represent, design and implement objects and data entities. Aspects are implemented along with the object-oriented base system using implementation tools like AspectJ. When implementing such a system, which involves objects and aspects together, we need to have a design technology that can represent and design both artefacts in the same environment. UML is an object-oriented modelling language which does not allow representation of non-OO concepts. One way of designing aspects with objects is to extend UML and another way is to come up with a new language which can accommodate the representation of both objects and aspects and their mutual relationships. Either option must maintain the fundamental software design principles such as:

Separation of Concerns: Parnas (1972) and Dijkstra (1976) regarded *separation of concerns* a vital design principle to manage the complexity of ever-growing systems. The idea is to divide a complicated system into small designable independent units. These units are designed and implemented separately without having a knowledge about each other and then combined together to form a single system. A new design language for aspect-oriented development would be assumed to follow this approach not only because *separation of concerns* is a basic design principle for all software languages but also because AOP was conceived and proposed based on this very principle.

Comprehensibility: As described by Parnas (1972), comprehensibility of a software design is "the ability to understand one part of the system at a time". Aspects are tangled in nature with other modules of the system so understanding aspects and their behaviour without having knowledge of other units of the system is not easy. We suggest that any new design paradigm for aspect-oriented systems must have the ability to represent aspects in their entirety, as separately as possible, while their relationships with the system modules must also be designed independently.

Loose coupling: Aspects are tightly coupled with other system modules because of their direct in-line implementation. As aspect-oriented programming (Kiczales et al., 1997) provides a way of representing aspects as separate modules to reduce such coupling, so their design strategies must also follow the same rule. Any new design language must have the capability of designing aspects and objects separately with minimal dependency on each other. Some of the problems caused by the tightly coupled nature of aspects are outlined in (Iqbal and Allen, 2010).

Maintainability: A comprehensible design of aspects should be easy to maintain. If aspects are tangled in multiple units of the system, their modification, addition and deletion can result in inconsistencies and high regressive overhead. A good software design strategy will represent and design all the units as separate and easily manageable units which will improve their maintainability and reusability.

Reusability: One of the main objectives of aspect-oriented programming is to modularize aspects so that they can be used in other systems as reusable modules. However, this ability of aspects is hard to attain because of their cohesive nature and high coupling with other units of the system (Elrad et al., 2001). An ideal design paradigm will design aspects as separate modules with minimal direct referencing to the rest of the system.

4.3. Aspect-Oriented Design Language (AODL)

The importance of a standard design language and specialized designed notations for aspect-oriented software development has been emphasized by a number of researchers (Clarke and Walker, 2002; Stein et al., 2002a; Dahiya and Dahiya, 2008). In the presence of Unified Modelling Language (UML) for object-oriented design, it becomes imperative to have a de-facto language for properly designing aspects along with the base objects. A number of design approaches have been proposed since the advent of AOP, which have been discussed in detail in chapter 3. Every approach has strived to fill gaps in the earlier proposed design approaches to provide a comprehensive design solution for aspects. However, one aspect of design has been left unaddressed in almost all of these approaches and that is the unification of aspects and objects in one design framework. Aspects cannot be separated completely from base objects due to the tightly coupled nature of pointcuts (Koppen and Stoerzer, 2004; Shonle et al., 2005) to the base program’s structure and behaviour. Aspects are thus required to be designed along with their interacting base objects. Most of the existing design approaches propose separate design techniques for both of the constructs which makes the design susceptible to inconsistencies.

A language similar to UML is required to represent and design aspects and their elements. As mentioned in the motivation above, UML has been chosen to be extended to accommodate aspects for many reasons. One important reason is its popularity as a modelling language. It is used as a standard object-oriented modelling language and since aspects are implemented along with objects (in AspectJ) so an extended version of UML becomes the first choice to design aspects. An altogether new design language will not only make it hard for the designers to adopt but will also force designers to work in two different design languages for objects and aspects. Another important reason is UML's extensibility which makes it easy to introduce new notations (provided Meta-Object-Facility (MOF) rules are followed, for details see (MOF, 2012) and use them with its core notations. Therefore, Aspect-Oriented Design Language (AODL) takes the liberty of introducing some new notations for aspects and their elements. AODL is based on AspectJ technology. It introduces design notations for the main constructs of AspectJ such as aspects, join points, pointcuts and advices. Design notations are used in the AODL models to describe structure and behaviour of an aspect and its elements. Metamodels for AODL have been provided in Appendix A.

Figure 4.1 shows AODL diagrammatic model which depicts the three phase implementation of AODL constructs. There are two diagrams for modelling join points, called Join Point Identification Diagram and Join Point Behavioural Diagram, there is one diagram for designing aspects, called Aspect Design Diagram, and there are two diagrams to design the weaving process of aspects and base classes, called Aspect-Class Static Diagram and Aspect-Class Dynamic Diagram.

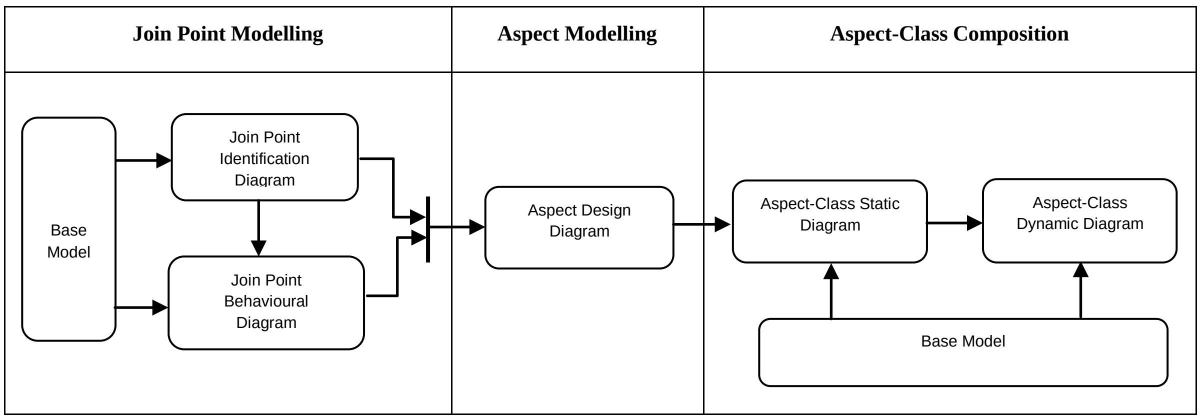

Join Point Modelling
Aspect Modelling
Aspect-Class Composition
Base Model
Join Point Identification Diagram
Join Point Behavioural Diagram
Aspect Design Diagram
Aspect-Class Static Diagram
Aspect-Class Dynamic Diagram
Base Model

Figure 4.1: AODL Diagrammatic Model

Join Point Modelling

The specification document of aspects, which is generated during the requirements engineering phase, provides a list of join points where a particular aspect will superimpose its behaviour. The Join Point Identification Diagram (discussed in 4.3.1.2) is used to identify the locations of these join points in sequence diagrams. A join point is represented with a designated design notation at the exact location within the control flow of objects. This diagram helps in representing the interactions of aspects with the base system at early design stage.

The Join Point Behavioural Diagram (discussed in 4.3.1.3) is another diagram that can be used to represent the location of join points and the corresponding aspects that interact on those locations. This diagram, however, is used when representation is required to be shown in the behavioural model of activities of the system.

Aspect Modelling

The aspect modelling phase starts with the modelling of pointcuts and advices. Both the constructs are represented with distinct notations. The inner structure is modelled using special associations that are distinguished from each other with the help of labelled stereotypes. Each pointcut is modelled using a Pointcut Composition Model that designs each predicate using nested a Collaboration Diagram, details can be found in 4.3.2.

The second phase of modelling designs aspects along with their constituent elements. There is a diagram, the Aspect Design Diagram (discussed in 4.3.3.2), that helps in designing aspects and their associations with the base classes. The diagram contains a designated structural container that represents the internal structure of the aspects and their constituent elements, such as pointcuts and advices. Each construct is represented with a distinct notation and associations among them are denoted by specialized stereotypes.

Composition Modelling

The composition is partially designed during the pointcut composition stage (discussed in 4.3.2.4), which is performed while designing pointcuts in the Aspect Modelling phase. The Pointcut Composition Diagram models inner composition of pointcuts where each predicate of a pointcut is modelled using a UML collaboration diagram. Interacting pointcuts are then composed with each other and with their related advices using composition associations.

The second phase of composition addresses aspect-class associations. The dynamic composition of both the constructs are modelled with the help of an extended Collaboration Diagram that contains specialized notations and associations to represent dynamic weaving of aspects' advices on specified join points in the base program. The diagram is discussed in detail in 4.3.4.2. Along with dynamic composition, a structural model has been proposed that captures crosscutting association between aspects and classes on an abstract level. The model is designed using Aspect-Class Structure Diagram (discussed in 4.3.4.3). The diagram is an extended version of the UML Class diagram and shows the relationships among interacting aspects and classes with the help of specialized crosscutting associations.

Every AODL design diagram serves a particular specialized purpose. The selection of diagrams depends on the nature of the system and requirements of the design model.

4.3.1. How to use AODL

The following guidelines have been set in the light of application of AODL (discussed in Chapter 5) for designers who wish to adopt AODL.

AODL provides structural and behavioural modelling support for all aspectual components. There are diagrams to help in modelling different perspective of these components. It depends on the designer to use the most suitable diagram for the desired model. Behavioural diagrams can ideally be used to design internal flow of the components and their associations with base constructs (objects or classes) at behavioral level. These diagrams are based on behavioural UML diagrams, such as activity diagram and collaboration diagram. Similarly, structural model of the system can be designed using the diagrams that capture structural representation of the components and their associations with base constructs on the structural level. For instance, Aspect Design Diagram presents a structural model where all the features and associations of an aspect are represented in a structural notation. Another example is Aspect-Class Structural Diagram that provides a black box view of relationships between aspects and base classes.

Some critical and safety systems might need more behavioural representation of the system to have better test case generation, and some systems might have emphasis on structural design to understand the relationships between aspectual and base components. It is up to the designer to choose the most suitable diagram to model a system.

The following section provides a detailed description of AODL design notations and AODL diagrams.

The following sections describe aspectual constructs, concepts, associations and elements in detail. The description style has been borrowed from UML's specification provided in (OMG, 2012).

4.3.2. Join Point Design

AODL defines join points with a design notation and provides two diagrams, Join Point Identification Diagram and Join Point Behavioural Diagram, to identify, specify, design and document join points. The join point and related diagrams have been described in detail in the following sub-sections.

4.3.2.1.Join Point

A join point is a point in the program where aspects execute their behaviour and perform a specified task.

Description

A join point is a point in the control flow of the base program. It could be defined on initialization, setting or getting of an attribute. It could also be defined on throwing or handling of an exception or it could be defined on the entire span of life of an object. A set of predicates defined on join points is called a *pointcut*.

AODL designates a notational symbol to represent a join point. The majority of the contemporary languages do not provide modelling support or a designated design notation for a join point. The reason is that they consider a join point a base program element and do not consider its modelling representation along with aspectual components. AODL, on the other hand, advocates design of join point as the key aspectual component, a pointcut, is made up of join points and if a join point is not modelled properly, the related pointcut may have some overlooked design issues.

Constraints

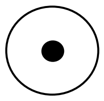
No constraints

Semantics

A join point is considered as a design element in AODL. It represents a direct relationship between an aspect(s) and an object(s). AODL denotes a design notation for join points which can be shown along with the description of interacting aspect in a stereotype convention.

Notation

A join point is represented by a Circle and a dot within the circle. The dot represents a point which connects associations from multiple aspects to multiple base classes in the system.



Naming Convention

Join points appear in the Join Point Identification Diagram with a label explaining the point in the base program. Some of these labels are <<call>>, <<execution>>, <<initialization>>, <<constructor_call>>, etc. A join point may appear in design diagrams in the form of a stereotype along with the related aspect's name.

<<JP_AspectName>>

Example

Figure 4.2 shows a general example of two join points defined on two methods of an object of Class A, one on the call of method1() and the second on the execution of method2(). AspectX runs its behaviour on these two points in this particular example.

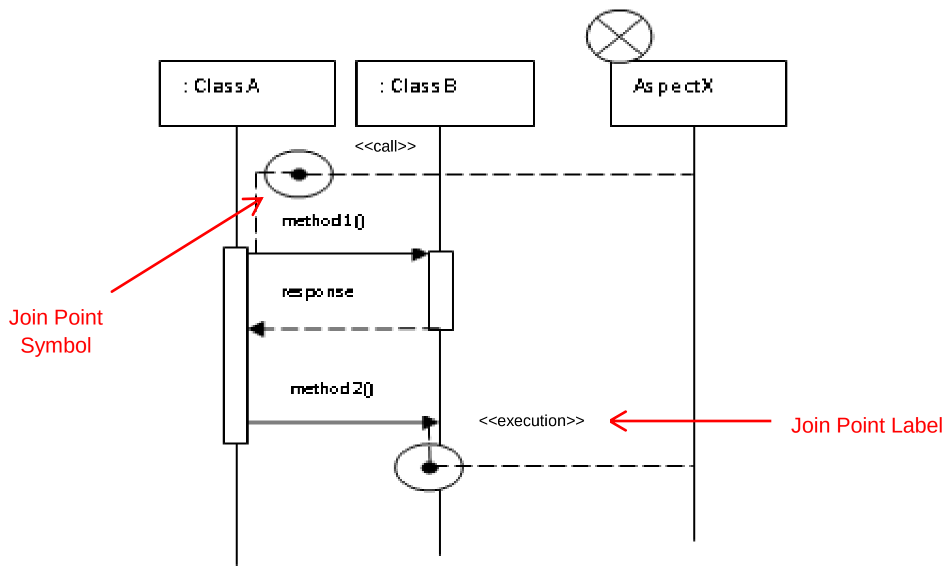


Figure 4.2: Join Point Representation

Rationale

The notational representation of a join point is very important to indicate the exact location(s) of join points in a design model. The notational representation also helps in understanding the weaving mechanism of aspects with objects by indicating merging points.

Purpose of the Notation

AODM (Stein et al., 2002a; 2006) represent join points as links. They don't offer a notational support rather represent them with stereotypes, such as <<*call*>>, <<*execution*>>. AODL on the other hand provides a design notation for join points so that they are distinctly represented along with other aspectual constructs. The notation for pointcut also carries this notation to show that pointcuts are predicates defined on join points. This way, join points and pointcuts are co-designed and make the design more comprehensible.

4.3.2.2.Join Point Identification Diagram

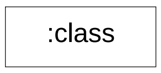
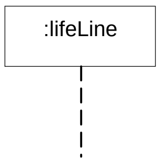
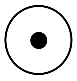
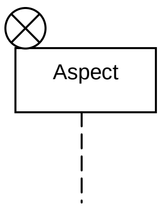
Description

The Join Point Identification Diagram has been developed to identify join points and to locate them at their exact locations in the system design. This diagram is based on UML's sequence diagram where join points are represented with the help of design notations along with the message passing among system objects.

Graphic Nodes

The graphic nodes included in Join Point Identification Diagrams are shown in Table 4.1. Besides these nodes, the diagrams may also have other nodes which are permissible in UML 2.4.1 for a sequence diagram.

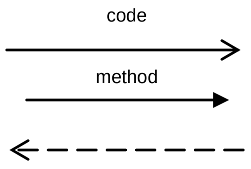
Table 4.1 - Graphic Nodes included in Join Point Identification Diagrams

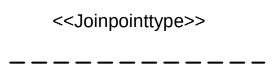
Node Type	Notation	Explanation
Object		Object is an instance of a class which is represented in this diagram to show the message passing between number of lifelines. (Borrowed from (OMG, 2012)
LifeLine		In UML 2.4.1 (OMG, 2012) ExecutionOccurence represents moments in time when a particular message is passed between two objects. Borrowed from (OMG, 2012).
Join Point		A join point indicates the location where an aspect executes its behaviour.
Aspect		Aspects are denoted with a design notation discussed later in the chapter. Aspects are shown along with objects whose join points are identified in the diagram.

Graphic Paths

Graphic paths between the graphic nodes have been shown in Table 4.2

Table 4.2 - Graphic Paths included in Join Point Identification Diagrams

Node Type	Notation	Explanation
Message		These message notations are for call, method and reply taken from UML's sequence diagram. (Borrowed from (OMG, 2012)

Aspect Indication Link	 <<Joinpointtype>>	Aspects are indicated with the help of join point notation and aspect indication link in the Join Point Identification Diagram. The links contains stereotypes to declare the type of join point.

Example

Two Join Points have been represented in the ATM example shown in Figure 4.3. One is defined on the call of *checkBalance()* method and the other is on the call of *withdraw()*. With every join point link there is a stereotype to declare the nature of the join point. For instance, both join points in the given example have <<*call*>>stereotypes. Corresponding aspects of both the join points have also been shown along with the base objects.

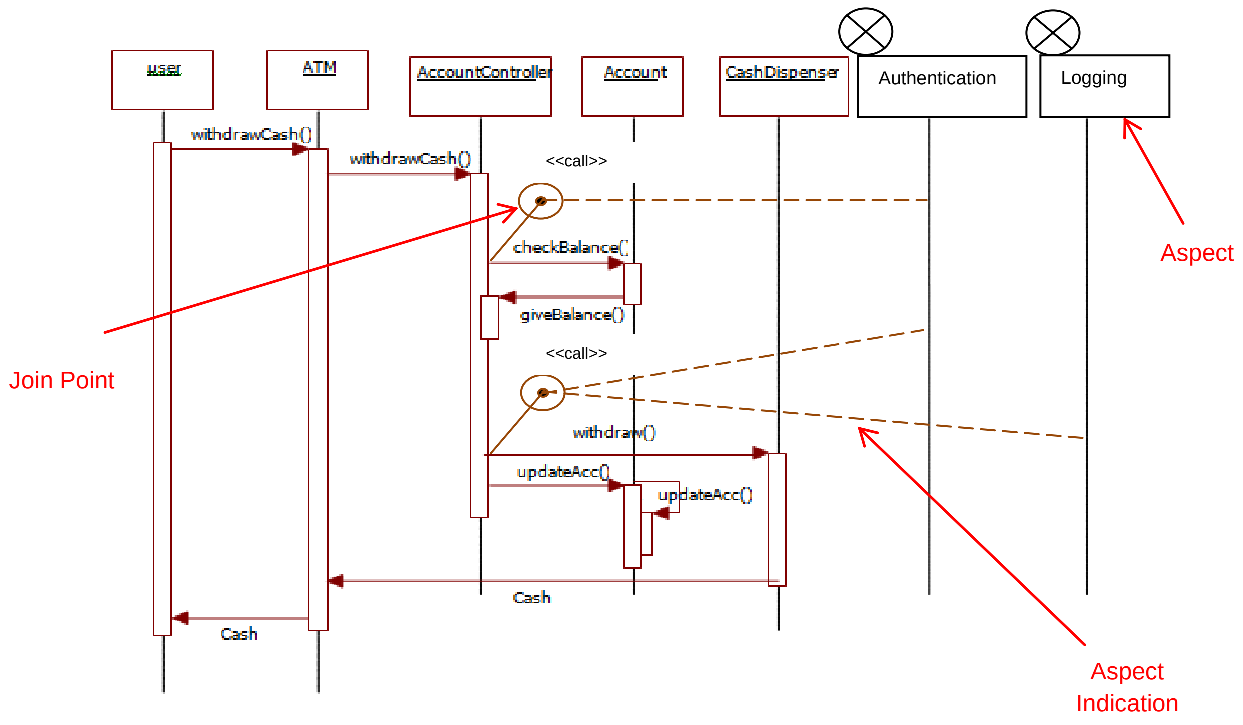


Figure 4.3: Join Point Identification Diagram

Rationale

AODL does not explicitly support the identification of aspects from the requirements of the system. It assumes that aspects have been identified in the requirement analysis phase using any suitable crosscutting concerns capturing approach. It also assumes that the base system is being modelled in the UML technology. In UML, Sequence diagrams show the communication among the objects in the form of methods and control flows. These diagrams can provide a base environment to locate the join points where aspects will insert their behaviour.

Purpose of the Diagram

This diagram has been proposed to identify join points within the message passing among objects. The purpose is to locate join points exactly where they are and represent them with a notation so that they are designed in detail in the low-level design of aspects.

4.3.2.3.Join Point Behavioural Diagram

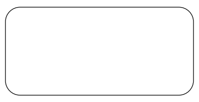
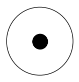
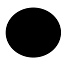
Description

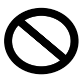
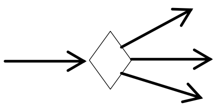
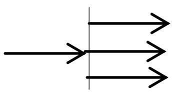
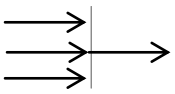
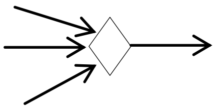
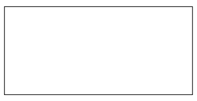
This diagram helps in identifying and representing join points during the flow of activities in the system. For the purpose, UML's activity diagram has been modified to accommodate join points along with the activity's actions and control flows.

Graphic Nodes

The graphic nodes included in Join Point Behavioural Diagrams are shown in Table 4.3. The table also includes all other UML 2.4.1's notations for activity diagrams which have not been mentioned here.

Table 4.3 - Graphic Nodes included in Join Point Behavioural Diagrams

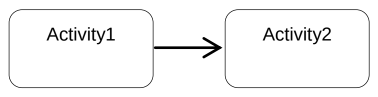
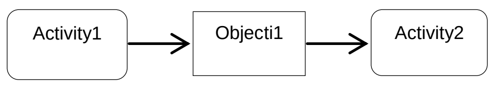
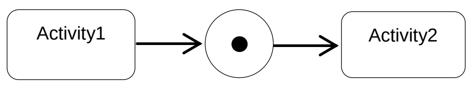
Node Type	**Notation**	**Explanation**
Action		Activities are made up of actions. This box represents an action which is the same as used in UML (OMG, 2012).
Join Point		Join points indicate the location where an aspect executes its behaviour.
InitialNode		Initial node represents the start of actions in an activity diagram. It is the same as in UML 2.4.1. (OMG, 2012).

ActivityFinal		This notation represents the end of an activity. It has been kept different from UML's notation ◉ to avoid confusion with join point.
DecisionNode		A decision node chooses the outgoing flow. It is same as the UML's decision node (OMG, 2012).
ForkNode		A fork node splits a control flow into multiple flows. It is same as the UML's fork node notation (OMG, 2012).
JoinNode		A joinNode synchronizes multiple flows into one control flow. It is same as the UML's notation for join node (OMG, 2012).
MergeNode		Merge node chooses one flow from multiple incoming control flows. It is same as the UML's notation for the Merge node (OMG, 2012).
ObjectNode		ObjectNode is used to define object flow within an activity. It is same as the UML's Object node notation (OMG, 2012).

Graphic Paths

Graphic paths between the graphic nodes in a Join Point Behavioural Diagram have been shown in Table 4.4

Table 4.4 - Graphic Paths included in Join Point Behavioural Diagrams

Node Type	**Notation**	**Explanation**
ControlFlow		A Control Flow starts an activity node after previous node is finished. (Borrowed from (OMG, 2012).
ObjectFlow		An Object Flow starts an object node after an activity node. (Borrowed from (OMG, 2012).
JoinPointFlow		A JoinPointFlow is an edge which shows the location of a join point during the activities.

Example

A general example of a Join Point Behavioural Diagram is shown in Figure 4.4. The join points are represented along with system activities.

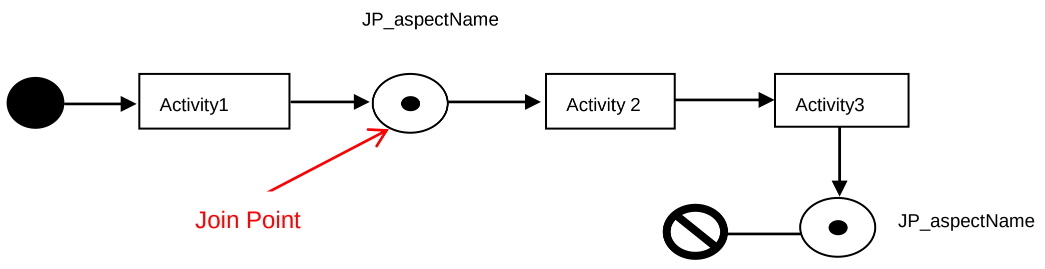


Figure 4.4: Join Point Behavioural Diagram

Rationale

The existing design approaches do not represent join points in the behavioural design of the base system. If a system is requiredto show the control flow among activities and join points are required to be identified within this flow, this diagram can help in locating join points.

Purpose of the Diagram

Once join points have been identified, they are required to be shown within the system flow. For this purpose, a behavioural diagram is proposed which extends UML's Activity diagram. This diagram assists in identifying the location of join points along with system's activities so that join points identified in the Identification diagram can be verified and their exact occurrences can be confirmed with the help of their representation within system flow. We show join points with the help of their join point design notation along with the name of interacting aspect(s). This diagram helps in understanding the weaving process of advices within the flow of system activities.

4.3.3. Pointcut Design

AODL defines Pointcuts with a design notation. It also provides a design notation for the pointcut's corresponding advice. The relationship between pointcut and advice is represented in a diagram, called Pointcut-Advice Diagram. The specification and definition of pointcuts are represented in a table, called Pointcut Table.

All these constructs and related diagrams have been described in detail in the following sub-sections.

4.3.3.1.Pointcut

A pointcut is a set of predicates defined on join points in the base program. It is used to expose data of the base program on particular join points to help run advices.

Description

A pointcut can have multiple predicates joined together through logical functions, such as AND, OR, NOT, etc. Multiple advices can execute their behaviour on a particular pointcut as defined in the aspect.

Constraints

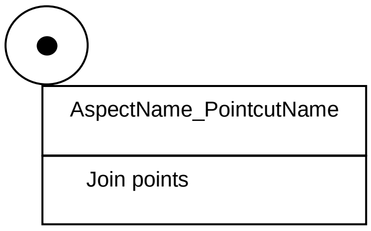
(1) A Pointcut must have a name.
(2) A Pointcut must have at least one related Advice.

Semantics

The pointcut is considered as a key aspectual element and a design construct in AODL. It is represented along with its constituent join points. A pointcut is designed along with its associations with the related advice and the parent aspect. This design is usually shown within an Aspect-Design Diagram (discussed in the following section). The reason is that they constitute key elements of an aspect and their associations with the base constructs are always through their parent aspects.

Notation

The pointcut is represented with a rectangle which contains its name and a list of join points. To distinguish it from other constructs, the rectangle has got a join point symbol on top of it. The rectangular box in the notation symbolizes a container that contains a pointcut's specification and the join point symbol reflects the association of pointcut with join points.



Presentation Option

The notational box for a pointcut has two compartments. The top compartment contains the name of the pointcut and the second compartment holds a list of join points.

Naming Convention

The pointcut's name is preceded by its parent aspect's name.

AspectName_PoincutName

Example

Figure 4.5 shows a general example of a pointcut along with representation of its related advices and join points.

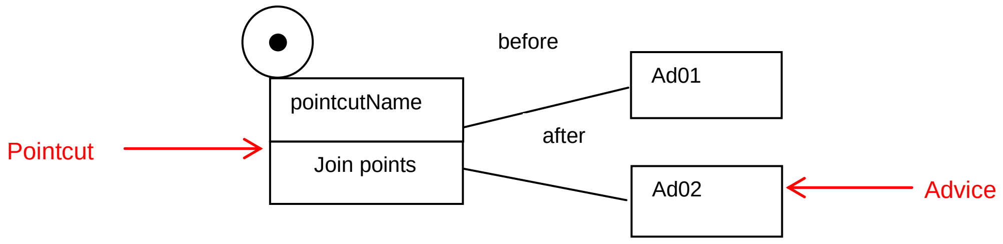


Figure 4.5: A Pointcut Example

Rationale

A pointcut is a vital element in aspect-oriented design. Pointcuts decide how aspects execute and how they interact with the base program. AODL considers pointcut as a distinct design construct which has its characteristics and associations with the base constructs. That is why separate notations and diagrams have been developed to design pointcuts along with their related advices.

Purpose of the Notation

Pointcuts are represented in several AODL diagrams. That is the reason that a distinct notation has been designated to them.

4.3.3.2.Advice

An advice is a piece of behaviour of an aspect which is inserted into the base program at specified locations (join points).

Description

An advice contains the implementation of an aspect. Advice can run before, after or around the locations defined by join points in a pointcut. An advice is tightly connected to its related pointcut which contains the set of join points where the advice is required to run.

Constraints

(1) An advice must have an id.
(2) An advice must have a related pointcut.

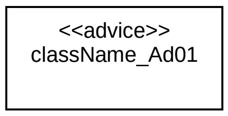
Semantics

An advice is initiated when a pointcut's predicates are satisfied. In other words, an advice is executed when join points of the related pointcut are reached during the execution of the program.

AODL designs an advice along with its pointcut and assigns a design notation to it. The Advice is considered as a combined construct along with its pointcut and occurrence type (before, after and around).

Notation

The design notation for an advice is a rectangular box. It contains keyword <<*advice*>> to distinguish from UML notations used for objects and classes. It contains an advice's id along with the name of the parent aspect. The functionality of advices may also be shown in textual narration in some of AODL models. The details are provided in the explanation of individual models.



Naming Convention

Advice is represented with a unique ID. The id could be a numbered one, such as Ad01, Ad02 or it could have a number along with the aspect's name, such as AdLog_01, AdLog_02 (For example, when the Aspect is *Logging*).

Example

An example of the representation of an advice is shown in Figure 4.5.

Rationale

The advice construct has to be represented in the design to show the behaviour of the parent aspect. AODL, therefore, assigns a distinct design notation to it and represents it along with the related pointcut.

Purpose of the Notation

Advices are represented in multiple AODL diagrams. That is the reason that they have been assigned a distinct design notation.

4.3.3.3.Pointcut-Advice Diagram

The association between a pointcut and its related advices are represented in a pointcut-advice diagram.

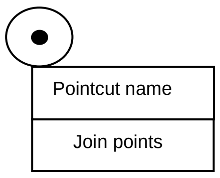
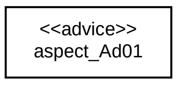
Description

This diagram helps in representing and designing relationships between a pointcut and its related advices. The diagram has been designed to represent an aspect's behaviour, which is implemented by advices, and to show aspect's interacting points with the base program, which are represented by pointcuts.

Graphic Nodes

The graphic nodes included in Pointcut-Advice Diagrams are shown in Table 4.5.

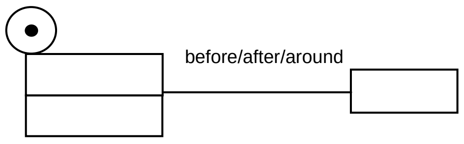
Table 4.5- Graphic Nodes included in Join Point Behavioural Diagrams

Node Type	**Notation**	**Explanation**
Pointcut		This is a pointcut notation which has been described in detail in 4.3.3.2
Advice		This is the design notation for advice, which has been discussed in detail in 4.3.3.2.

Graphic Paths

Graphic paths between the graphic nodes in a Pointcut-Advice Diagram have been shown in Table 4.6

Table 4.6 - Graphic Paths included in Pointcut-Advice Diagrams

Node Type	**Notation**	**Explanation**
Pointcut-Advice Association		A pointcut is associated with an advice through a simple line with occurrence type defined on top of it.

Example

Figure 4.6 shows an example of an *Authentication* aspect in an ATM system. It contains a pointcut-advice diagram which shows a pointcut *authenticateUser* associated with an advice Ad01.

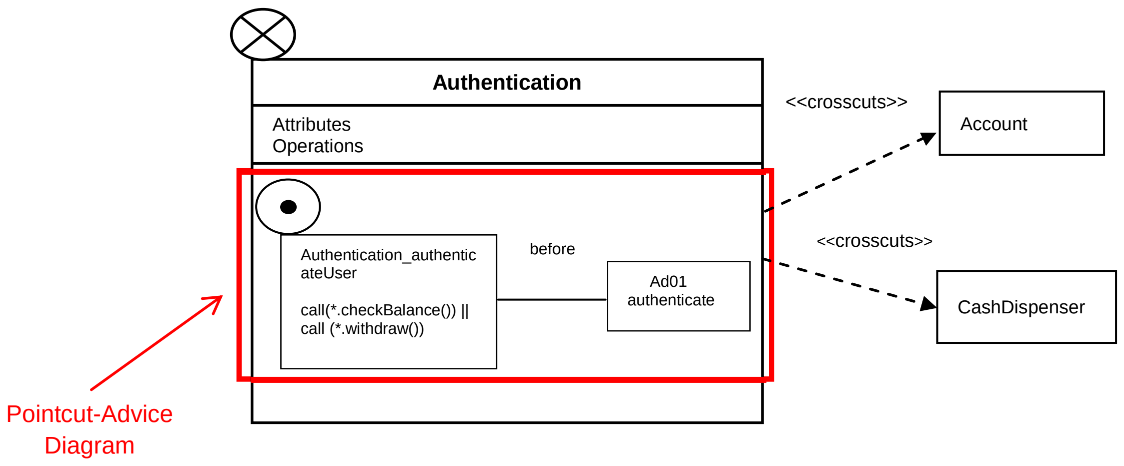


Figure 4.6: Pointcut-Advice Diagram in Authentication Aspect

Rationale

As surveyed in Chapter 3 and evaluated in Chapter 6, there is not a single existing design approach that provides modelling support for depicting relationships between pointcuts and their related advices. AODL offers distinct design notations for these constructs and provides this Pointcut-Advice Diagram to model the relationships between them.

Purpose of the Diagram

The Pointcut-Advice Diagram helps in representing and designing a complete structure of pointcuts and their related advices in one diagram. It helps in understanding the relationship between a pointcut and an advice. The diagram also helps in designing an aspect along with its pointcuts and advices.

4.3.3.4.Pointcut Composition Model

A detailed model of the composition process is captured in this diagram. Pointcuts are designed and represented with their related advices and parent aspects.

Description

Pointcuts are composed dynamically when aspects are woven into the system. Advices are executed on the defined join points in the pointcut, and pointcuts combine together to identify the exact locations where advices are supposed to run. AODL designs each join point with the help of a behavioural diagram. The diagrams are based on the UML communication diagram. Communication diagrams (previously known as collaboration diagrams) help in designing the dynamic collaboration of objects with each other in UML.

The interaction is shown in the form of message passing among objects. AODL exploits this diagram for the join points' selection during the composition of aspects.

Types of Pointcuts

Before explaining the pointcut composition, we introduce categories of pointcuts. We have categorized Pointcuts used in AspectJ into four types.

Scope Pointcuts: The pointcuts that define a scope of selection of join points in the base system are included in this category. For example, pointcuts defined with *within* and *cflow* keywords are meant to define a scope in the base system for selecting join points. Some other examples include w*ithincode(), cflowbelow(), this(), target()* and *args().*

Method Pointcuts: The pointcuts that are defined on methods and constructors of classes of the base system are part of this category. Some of the pointcuts defined in this category are *call(), execution(), get(), set(), call(const), execution(const), initialization(), preinitialization(), staticinitialization(), and handler().*

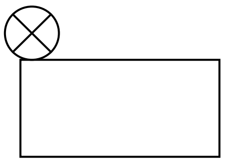
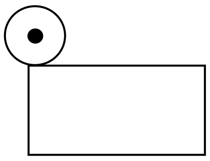
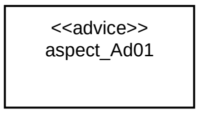
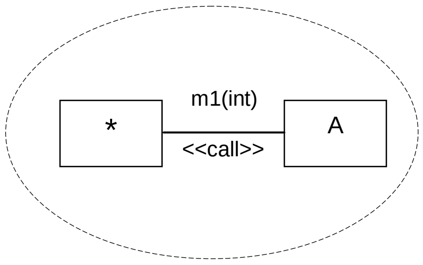
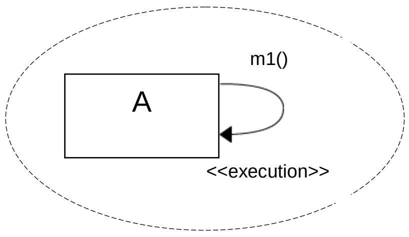
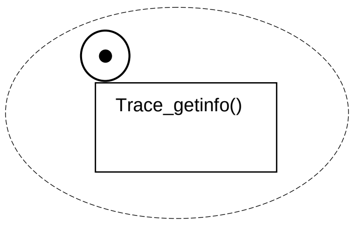
Peer Pointcuts: Peer pointcuts select other pointcuts defined in the same aspect or a related aspect. These pointcuts are defined on already defined pointcuts. Some of the examples in this category are *pointcutID(), !pointcut(), pointcut 0 && pointcut1, pointcut0 || pointcut1* and *(pointcut).*

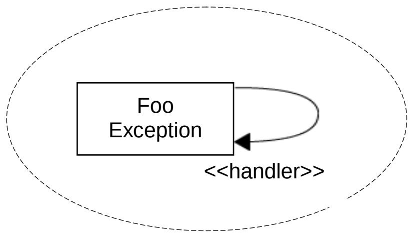
Conditional Pointcuts: Conditional pointcuts are defined on join points satisfying a Boolean condition. These pointcuts may define all kinds of Boolean operators such as AND, OR, NOT etc. The if(Boolean) expression is also part of this category.

Graphic Nodes

The graphic nodes included in Pointcut Composition Model are shown in Table 4.7.

Table 4.7 - Graphic Nodes included in Pointcut Composition Model

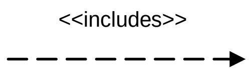
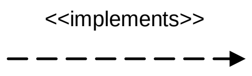
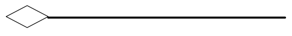
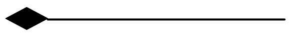
Node Type	Notation	Explanation
Aspect		Details about aspect's notation are provided in 4.3.3.3.
Pointcut		This is a pointcut notation which has been described in detail in 4.3.3.2
Advice		This is the design notation for advice, which has been discussed in detail in 4.3.3.2.
Join Point Collaboration for Method Call		This type of collaboration contains a join point defined on the call of a method. In this example collaboration, join point is defined on the call of method m1(int) of class **A**.
Join Point Collaboration for Method Execution		This type of collaboration contains a join point design which is defined on the execution of a method. In this example, a join point is defined on the execution of method m1() of class **A**.
Join Point Collaboration for Pointcut Reference		This type of collaboration contains a pointcut which is used as a predicate in the main pointcut. In this example collaboration, pointcutgetinfo() of **Trace** aspect is depicted.

Join Point Collaboration for Exception Handler Call		This type of collaboration contains a call to an exception handler. In this example collaboration, a call is made to **Foo Exception**.
Class		Base classes are represented with their conventional UML notations (OMG, 2012).

Graphic Paths

Graphic paths between the graphic nodes in a Pointcut Composition Model have been shown in Table 4.8.

Table 4.8 - Graphic Paths included in Pointcut Composition Model

Node Type	**Notation**	**Explanation**
Pointcut-Pointcut Referencing Association		This association links a pointcut with a related pointcut. The relationship can be because of direct referencing to each other in the pointcut definition.
Pointcut-Pointcut Overriding Association		This association links an implemented pointcut to its abstract pointcut in the parent aspect.
Aggregation		Aggregation is an enumeration type used in UML 2.4.1 to specify literals for defining aggregation property between objects (OMG, 2012).
Composition		Composition association is the same as used in UML 2.4.1 (OMG, 2012).

Generalization	───▷	Generalization association is the same as used in UML 2.4.1 (OMG, 2012). In addition, this association is also used for describing generalization in aspects.

Example

Figure 4.7 shows a Pointcut Composition Model for **Tracing** system taken from the Eclipse AspectJ Programming Guide (2012).The example implements a tracing system. There are two aspects, **Trace** and **TraceMyClasses**. **TraceMyClasses** is a child aspect of Trace. It contains one pointcut myClass(Obj) which implements abstract pointcut Trace_myClass(Obj j) of Trace aspect. Trace aspect contains two pointcuts myMethod and myConstructor(Obj j). Each pointcut has two related advices.

The Pointcut Composition Model in Figure 4.7 designs these pointcuts and shows their compositions with each other and with their related advices and parent aspects.

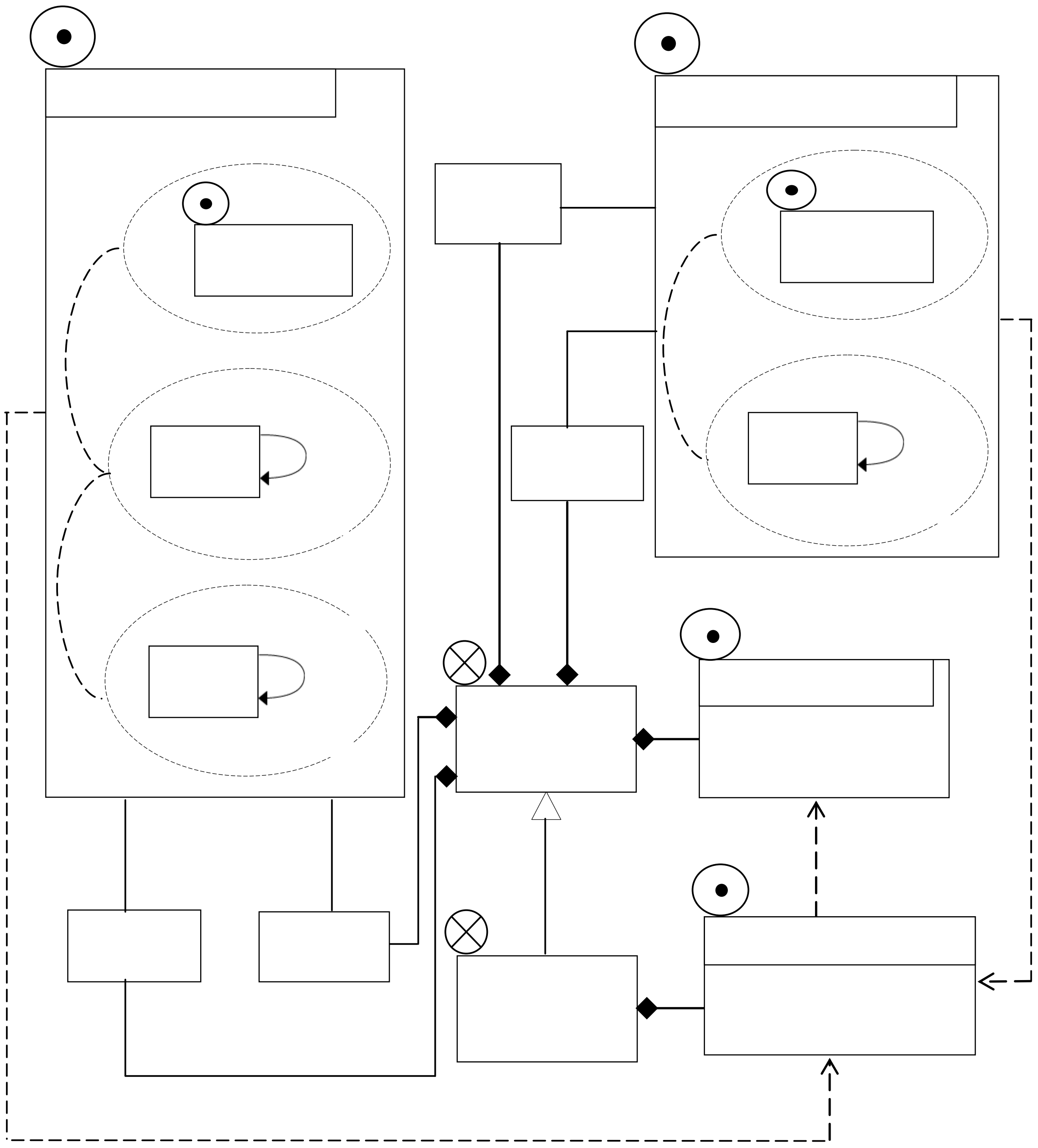


Figure 4.7: Pointcut Composition Model

The contemporary aspect-oriented design methodologies overlook intra-aspect compositions. These compositions include joinpoint-joinpoint compositions, pointcut-pointcut compositions and pointcut-advice compositions. The modelling of these compositions help in designing pointcuts and overall composition of aspects with each other and with base constructs.

4.3.4. Aspect Design

This section introduces AODL's design notations and design diagrams for aspects and their relationships with the base design constructs.

4.3.4.1.Aspect

Description

An aspect is a feature of the system which is designed separately from other features. Its implementation is scattered and crosscuts multiple modules which makes its design hard to implement and understand. That is the reason that it is separated from base modules of the system and is designed as a separate design unit. It is woven back into the system during execution.

Constraints

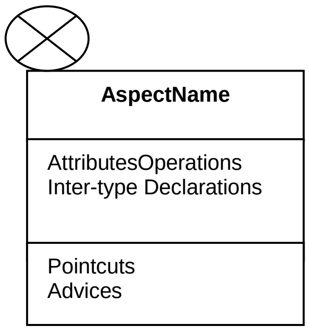
No constraints

Semantics

An aspect is designed separately in AODL. It is represented with a design notation and a design diagram. Aspectual elements are also represented along with the aspect. The relationship of an aspect with base objects is shown through crosscutting association. The design diagram provides all related information and specification of an aspect in a structural fashion.

Notation

Aspect is represented in a rectangular box which is similar to the symbol used for a classifier in UML. Each aspect must be assigned a name. The rectangular box is topped with a crosscutting circular symbol to distinguish it from other design constructs and UML's classifiers.



Presentation Options

The rectangular box used for an aspect's notation is divided into three compartments. The first compartment holds the name of the aspect. The second compartment contains a list of attributes, operations and inter-type declarations. And the third compartment contains a list of pointcuts and advices or a pointcut-advice diagram (in Aspect-Design Models).

Naming conventions

Aspect's name is centred and bold faced. Each word in the name starts with capital letter and has no space in between. Naming conventions for attributes and operations is the same as is used in UML (OMG, 2012). Pointcuts and advices follow the same naming style as is applicable for operations in UML 2.4.1.

Example

An example representing an aspect is shown in Figure 4.8 in the following section

Rationale

An aspect is a primary construct in aspect-oriented software development. It is required to be represented with a distinct design notation. AODL, therefore, assigns a design notation to an aspect which contains all the features of an aspect.

Purpose of the Notation

Aspect is represented in all AODL diagrams. That is the reason that a distinct design notation has been designated to this construct.

4.3.4.2.Aspect-Design Diagram

This diagram represents features of an aspect and its associations with base classes.

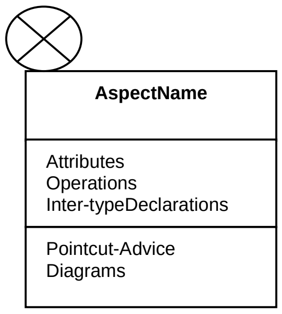
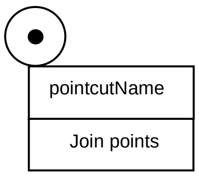
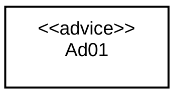
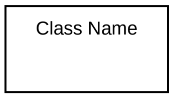
Description

The aspect-design diagram helps in representing complete information of an aspect. An aspect is represented along with its primary features such as attributes, operations, pointcuts and advices. The relationship between pointcuts and advices is represented with the help of a pointcut-advice diagram. The associated base classes are also represented with the aspect through <<*crosscuts*>> stereotypes.

Graphic Nodes

The graphic nodes included in Aspect-Design Diagrams are shown in Table 4.9.

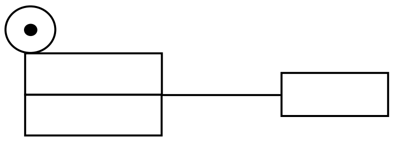
Table 4.9 - Graphic Nodes included in Aspect-Design Diagrams

Node Type	**Notation**	**Explanation**
Aspect	 AspectName Attributes Operations Inter-typeDeclarations Pointcut-Advice Diagrams	Details about aspect's notation are provided in 4.3.3.3.
Pointcut		This is a pointcut notation which has been described in detail in 4.3.3.2
Advice		This is the design notation for advice, which has been discussed in detail in 4.3.3.2.
Class		Base classes are represented with their conventional UML notations (OMG, 2012).

Graphic Paths

Graphic paths between the graphic nodes in a Pointcut-Advice Diagram have been shown in Table 4.10.

Table 4.10 - Graphic Paths included in Pointcut-Advice Diagrams

Node Type	**Notation**	**Explanation**
Pointcut-Advice Association		A pointcut is associated with an advice through a simple line with occurrence type defined on top of it.

Crosscutting Association	<<crosscuts>> – – – – – – – – ►	This association shows relationship between an aspect and its interacting base classes.

Example

Figure 4.8 shows an example of an *Authentication* aspect in an ATM system. It contains a pointcut-advice diagram which shows a pointcut *authenticateUser* associated with an advice Ad01.

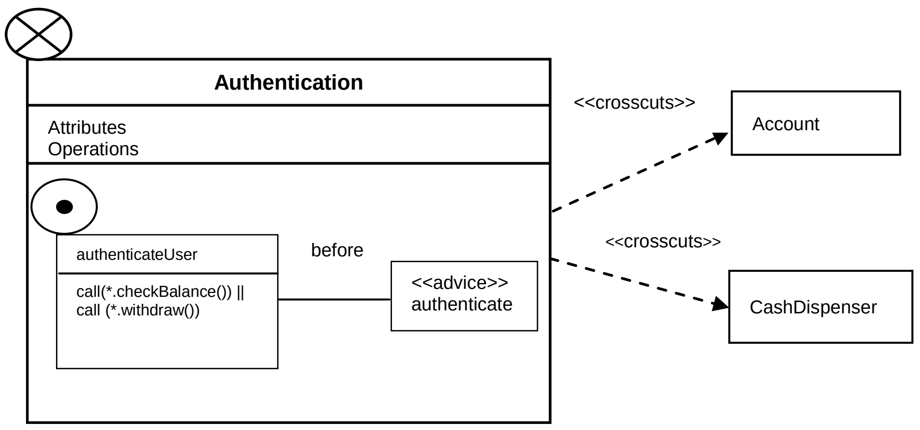


Figure 4.8: Aspect-Design Diagram for Authentication Aspect

Rationale

A structural design diagram is required to represent the internal structure of an aspect. Although a simple aspectual notation is used in all of the AODL diagrams and models, a detailed structural diagram is still required to design complete structure of an aspect.

Purpose of the Diagram

The Aspect-Design Diagram has been developed to represent an aspect along with its features and associations. This diagram helps in understanding the structure of an aspect and its structural relationships with base classes.

4.3.5. Weaving Process Design

This section introduces the design method for weaving of aspects into the base program adopted in AODL. There is a weaving association and two design diagrams, Aspect-Class Dynamic Diagram and Aspect-Class Structure Diagram which are used to design a complete weaving process in AODL.

4.3.5.1.Weaving Association

Description

Weaving is a process where aspects' behaviours are woven into the base program during its execution. The location of weaving is decided by pointcuts of the aspect which contains join points of the base program. Weaving association indicates the locations in the dynamic diagram where an aspect's behaviour is inserted.

Constraints

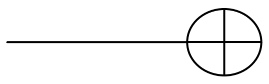
This association can only be used to depict a weaving association between an aspect and a base object.

Semantics

The association between an aspect and objects is shown with the help of this notation which has been designed to reflect the type of association both constructs have. The association also contains information regarding the behaviour of the aspect which is to be inserted into the base program and the location where this weaving process happens.

Notation

The association is represented by a line with a head made up of a circle with + sign. The circle resembles the aspect's circular symbol and the + sign shows the appending process of the aspect's behaviour.



Presentation Options

The association may contain the advice's name and information about the method in which the advice is supposed to be inserted.

Example

The usage of weaving association notation has been shown in Figure 4.9.

Rationale

The weaving association has been designed to distinguish it from other UML associations which are used between objects. An aspect's association with an object is required to be shown as a special relationship where an advice's implementation is to be inserted.

4.3.5.2.Aspect-Class Dynamic Diagram

This diagram shows the weaving process at a dynamic level.

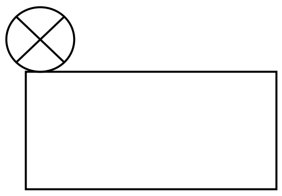
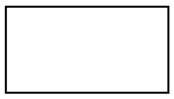
Description

This diagram has been developed to represent the weaving process of aspects with objects during the execution of the program. The communication diagram of UML 2.4.1 (OMG, 2012) has been selected as a base for this diagram. Some extensions have been introduced to the communication diagram to accommodate representation of aspects and aspectual elements and to represent the weaving process. The reason behind selecting the communication diagram is its ability to provide a dynamic picture of the system. Since weaving is a dynamic process which happens during the execution of the program so this diagram is an ideal choice to represent the weaving process.

Graphic Nodes

The graphic nodes included in an Aspect-Class Dynamic Diagram are shown in Table 4.11.

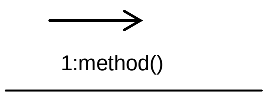
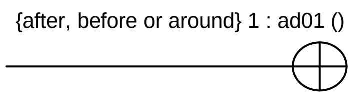
Table 4.11 - Graphic Nodes included in Aspect-Class Dynamic Diagram

Node Type	**Notation**	**Explanation**
Aspect		Details about aspect's notation are provided in 4.3.3.3.
Object		Base objects are represented with their conventional UML notations (OMG, 2012).

Graphic Paths

Graphic paths between the graphic nodes in an Aspect-Class Dynamic Diagram have been shown in Table 4.12.

Table 4.12 - Graphic Paths included in Aspect-Class Dynamic Diagram

Node Type	**Notation**	**Explanation**
Message		This path indicates the method's information along with method's name and order of occurrence. The arrow head indicates the direction of the flow. The representation is similar to the one used in conventional UML (OMG, 2012).
Weaving Association		This association shows that a piece of code (advice) is being appended to the object. The association has occurrence type (before, after or around) followed by number of method and advice id.

Example

Figure 4.9 shows a general Aspect-Class Dynamic Diagram which shows the aspects' weaving with the base objects during the execution of the system.

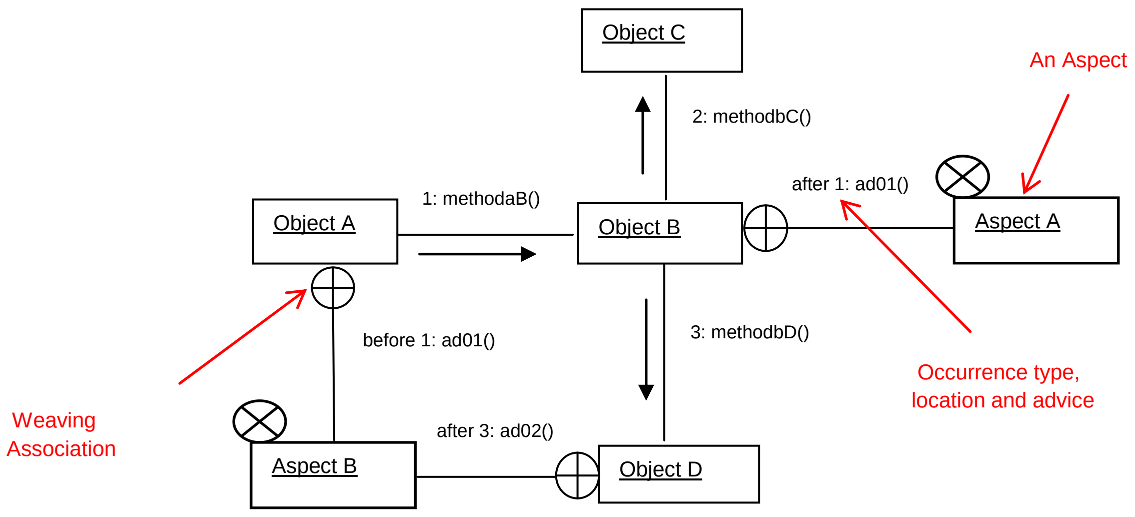


Figure 4.9: Aspect-Class Dynamic Diagram

Rationale

Weaving is a dynamic process that happens at the run time. A dynamic model is required to capture this run-time weaving of advices into the objects' internal control flow.

Purpose of the Diagram

The weaving process is a crucial design document in aspect-oriented software development. The Aspect-Class Dynamic Diagram shows how we can show the appending of advices with the base objects during the dynamic flow of the system. The diagram provides a way of simulating the weaving process using UML's communication diagram. A communication diagram is used to show the dynamic flow of the system in the unified modelling language. Since the weaving process is also dynamic so representation of the aspects' superimposed behaviour can be captured by representing insertion of advices at the specified join points along with the class' method execution.

4.3.5.3.Aspect-Class Structure Diagram

This diagram shows the structure of an aspect-oriented system. It presents a static model of aspects and system classes in one diagram.

Description

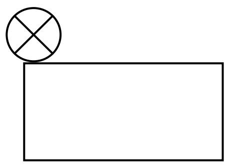
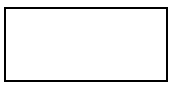
This diagram has been developed to represent structural representation of aspects along with system classes. The diagram extends UML's class diagram which is used in UML to show structure of classes (OMG, 2012).

The diagram helps in presenting a structural picture of the system where aspects and their relationships with classes are shown at a static level.

Graphic Nodes

The graphic nodes included in Aspect-Class Structure Diagram are shown in Table 4.13.

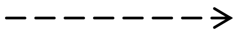
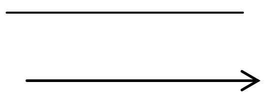
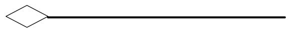
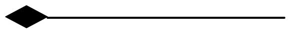
Table 4.13 - Graphic Nodes included in Aspect-Class Structure Diagram

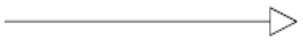
Node Type	Notation	Explanation
Aspect		Details about aspect's notation are provided in 4.3.2.3.1.
Class		Base classes are represented with their conventional UML notations (OMG, 2012).

Graphic Paths

Graphic paths between the graphic nodes in an Aspect-Class Dynamic Diagram have been shown in Table 4.14. The UML 2.4.1 class paths which have not been provided here in this table are also applicable to the Aspect-Class Dynamic Diagrams.

Table 4.14 - Graphic Paths included in Aspect-Class Dynamic Diagram

Node Type	Notation	Explanation
Aspect-Class Association		This association links an aspect with a class in the diagram.
Association		Association defines links between two instances of the same kind in UML 2.4.1 (OMG, 2012).
Aggregation		Aggregation is an enumeration type used in UML 2.4.1 to specify literals for defining aggregation property between objects (OMG, 2012).
Composition		Composition association is the same as used in UML 2.4.1 (OMG, 2012).

Generalization		Generalization association is the same as used in UML 2.4.1 (OMG, 2012). In addition, this association is also used for describing generalization in aspects.

Example

Figure 4.10 shows a general Aspect-Class Structure Diagram which shows static relationshipsbetween aspects and base classes.

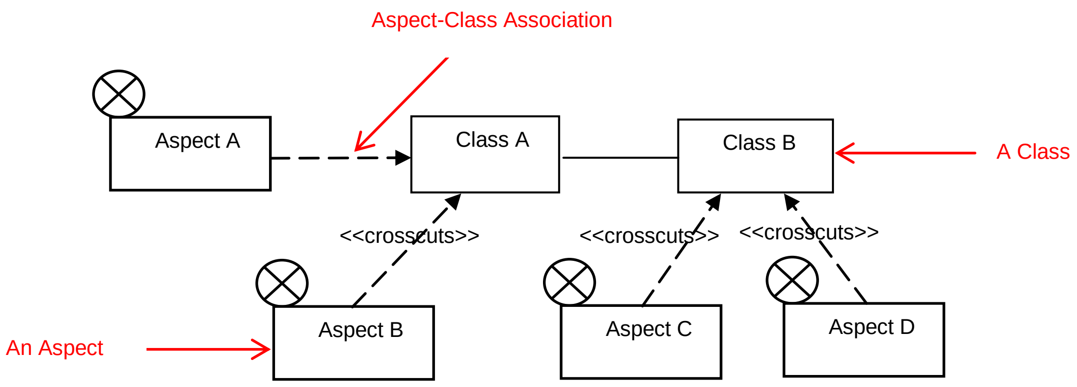


Figure 4.10: Aspect-Class Structure Diagram

Purpose of the Diagram

The aspect-class relationships are designed in a static model in AODL. Aspect-Class Static Model shows the interacting aspects and base classes in one diagram which helps in identifying the entities participating in the weaving process. The crosscutting relationship is denoted by a *<<crosscuts>>* stereotype, which shows class-directional association between an aspect and a base class.

Aspect-Class structure diagram also helps in developing a blue print of the structure of the system depicting the main constructs of the system (aspects and classes) and relationships among them. The diagram also helps in translating the system design into implementable code.

4.3.6. Pointcut Table

Defining and documenting pointcuts properly ensures consistency of the program. AODL proposes a pointcut table to document pointcuts along with their related advices, aspects and base classes. The table defines pointcuts in vertical columns by indicating the join points of the base system horizontally. The columns of the table provide list of aspects and complete definition of their pointcuts along with their related advices. The rows, on the other hand, show the base system attributes, methods and execution points where join points have been identified. The execution order of advices on a single join point is declared in the last column, named Order.

An example pointcut table shown in Table 4.15 specifies following pointcuts:

AspectA:

P1:this(X) && (execution(mX1) || call (mY1)) &&(P2)

P2:exception(type)

Aspect B:

P3:execution(mX1) || call(mY1) && !P4

P4: call(mX2) || call(mY2)

{ mX1 = Method 1 of Class X, mY1 = method1 of Class Y}

Table 4.15 - Example Pointcut Table

	<<aspect>> Aspect A		**<<aspect>> Aspect B**		**Precedence**
	<<advice>> AdA1 (Before)	<<advice>> AdA2 (After)	<<advice>> AdB1 (Before)	<<advice>> AdB2 (around)	
Class X	this				
constructor					
method1	execution		execution		AdA1,AdB1
method2				call	
getX()					
Class Y					
method1	call		call		AdB1,AdA1
method2		exception(type)		call	
Pointcut Definition	this(X) && (exec(mX1) || call (mY1))	exception(type)	execution(mX1) || call(mY1)	call(mX2) || call(mY2)	
Pointcut	P1	P2	P3	P4	
Pointcut Trigger	(P2)		!(P4)		
Complete Definition	this(X) && (exec(mX1) || call (mY1)) &&(P2)	exception(type)	execution(mX1) || call(mY1) && !P4	call(mX2) || call(mY2)	

The above tale provides a simple example to explain the pointcut table. Chapter 5 provides an in-depth application of the table to two case studies to explain it in more detail with examples. The table has been tested and verified to represent all types of legitimate AspectJ pointcuts, as defined in (Iqbal and Allen, 2012).If the system is highly complex and contains a number of aspects, the table can be broken into multiple smaller tables to improve readability.

Purpose of Pointcut Table

The table provides a means to specify pointcuts in a detailed manner along with their related advices, pointcuts and base constructs. The table also helps in identifying and resolving conflicts. It explicitly overcomes the shared join point problem by prioritizing order of execution of advices.

4.5. Chapter Summary

This chapter has discussed Aspect-Oriented Design Language in detail which has been proposed to define, specify, represent and design aspects and their constituent elements along with base program’s constructs. AODL has been proposed on the primary motivation of providing a unified design framework to design both aspects and objects together in one environment. For this purpose a unified language has been proposed which extends UML with some new design notations for aspects and their key elements. This chapter has discussed the motivation behind the language in detail. It has provided description of language formalism which has been adopted for all the design notations and diagrams included in AODL. UML’s specification templates were modified and used to describe each notation and diagram in detail.

Chapter 5: Application of AODL

This chapter evaluates AODL in terms of application of the language to real-world case studies. The language has been applied to two case studies which have been selected on the basis of adequate level of complexity to cover all the proposed notations and design models. The first case study is a Car Crash Crisis Management system, which is a standard case study for the evaluation of aspect-oriented design approaches. The second case study is an implemented game, SpaceWar, borrowed from the AspectJ Tutorials, which, has been selected to apply AODL by reverse engineering an implemented system. The chapter discusses and assesses the efficacy of the language in light of its application to both these case studies.

5.1. Introduction

The evaluation of AODL has been divided into two phases, through qualitative analysis, which is discussed in Chapter 6, and through applying the notations to case studies, which will be the topic of this chapter. It has been demonstrated in detail in Chapter 6 that AODL covers all the basic quality criteria of an aspect-oriented design language. The basic requirements for an effective AO design methodology, such as support for static and dynamic crosscutting, traceability, extensibility and reusability have been assessed in depth in that chapter. However, a design language cannot be deemed effective unless it is demonstrated to design a complex system adequately.

This chapter demonstrates the application of AODL to two case studies. The first case study is a *Car Crash Crisis Management* system, which was a theme case study for aspect-oriented modelling approaches for a special edition of Transactions on Aspect-Oriented Software Development VII (Kienzle et al., 2010). The case study is a detailed implementation of a crisis management system which has enough complexity to exploit all the proposed notations and design models of AODL. The second case study is an example game, called *SpaceWar,* implemented by the AspectJ Team and is available on AspectJ's eclipse plugin (AspectJ, 2012). This case study provides a way to assess AODL by reverse engineering the design of an implemented system using AODL notations and design models.

The rest of the chapter is structured as follows: section 5.2 explains the *Car Crash Crisis Management system* and describes the design of the system using AODL. Section 5.3 explains the *SpaceWar* game and the reverse engineered design of the game using AODL. Section 5.4 discusses the application of AODL to both case studies and provides the results and findings. The last section, section 5.5 concludes the chapter with a chapter summary.

5.2. Case Study: Car Crash Crisis Management System

The Crisis Management System (CMS) case study was the theme of a special edition of Transactions on Aspect-Oriented Software Development VII (Kienzle et al., 2010). The purpose of a common case study was to have a comparative research repository of the existing aspect-oriented software development techniques. CMS is software that facilitates and brings together all the related parties and stakeholders who are involved in handling a crisis. CMS is required to handle many types of crises, such as accidents, attacks, natural disasters, etc. by interacting with external services like hospitals, emergency services, military and police services. More details on the case study can be obtained from (Shmuel and Mezini, 2010; Kienzle et al., 2010). In this chapter, the focus will be on designing a specialized form of CMS that is Car Crash Crisis Management (CCCM).

5.4.2. Crisis Scenario of a Car Crash Crisis Management System

This section will provide only a brief introduction to the system, for more details consult (Kienzle et al., 2010).

A crisis management task is initiated by a coordinator on a crisis report made by a witness at the scene. A coordinator oversees the crisis management system and is responsible for utilizing all the required resources to resolve the crisis. The surveillance system is an external system placed on highways and other busy locations in the form of cameras. Video feeds from the surveillance system may be acquired on the request of crisis management system. A super observer is assigned by the system to observe the crisis scene and make a report on the crisis and to identify the need for internal and external resources depending on the nature and severity of the crisis. The tasks are identified by the super observer and deployed in the form of crisis missions. Crisis missions may include internal and external resources depending on the demands of the crisis.

5.4.3. Identified Aspects

The following functional and non-functional aspects from the car crash crisis management system have been identified:

5.2.2.1. Functional Aspects

1. **Witness Validation:** This aspect validates authenticity of the witness who is reporting a crisis. The aspect may contact external resources for the validity check.
2. **Mission Status:** This aspect is responsible for updating the status of the mission and to inform whether the mission is active, finished, failed, terminated or interrupted at a certain point of time.
3. **Resource Monitor:** This aspect is responsible for setting off an alarm when a minimum threshold value of resources is reached.
4. **National Crisis Center Informer:** This aspect is responsible for informing NCC when a) no internal or external resource is available, b) mission is interrupted or terminated without being completed and no replacement is available c) mission fails d) mission needs assistant from NCC.
5. **Employee Authentication:** This aspect authenticates every employee who is part of the reporting, deployment or handling of the crisis.
6. **Witness Report Observer:** If the witness call gets disconnected in the middle of the report being made, this aspect is responsible for gathering as much information as has been provided by the witness and for collecting more information from the surveillance system in the form of a video feed. It is also responsible for initiating the emergency aid service on the basis of collected data from the witness report and surveillance system.

5.2.2.2. Non-Functional Aspects

1. **Fault-Tolerance:** This aspect starts a back-up system if the current system shuts down or hangs for over 30 seconds.

2. **Persistence:** This aspect is responsible for storing critical information about the crisis such as witness report, crisis type, location, available resources, deployed resources and start and finish time of crisis.

3. **Security:** This aspect prompts employees to re-authenticate if they are idle for more than 30 minutes.

4. **Logging:** This aspect is responsible for keeping logs about all types of activities.

5.4.4. Use Case Diagram of Car Crash Case Study

A detailed use case diagram of the summary-level goal *Resolve Crisis* has been shown in Figure 5.1. For details of all the use cases that are related to the *Resolve Crisis* use case, consult (Kienzle et al., 2010).

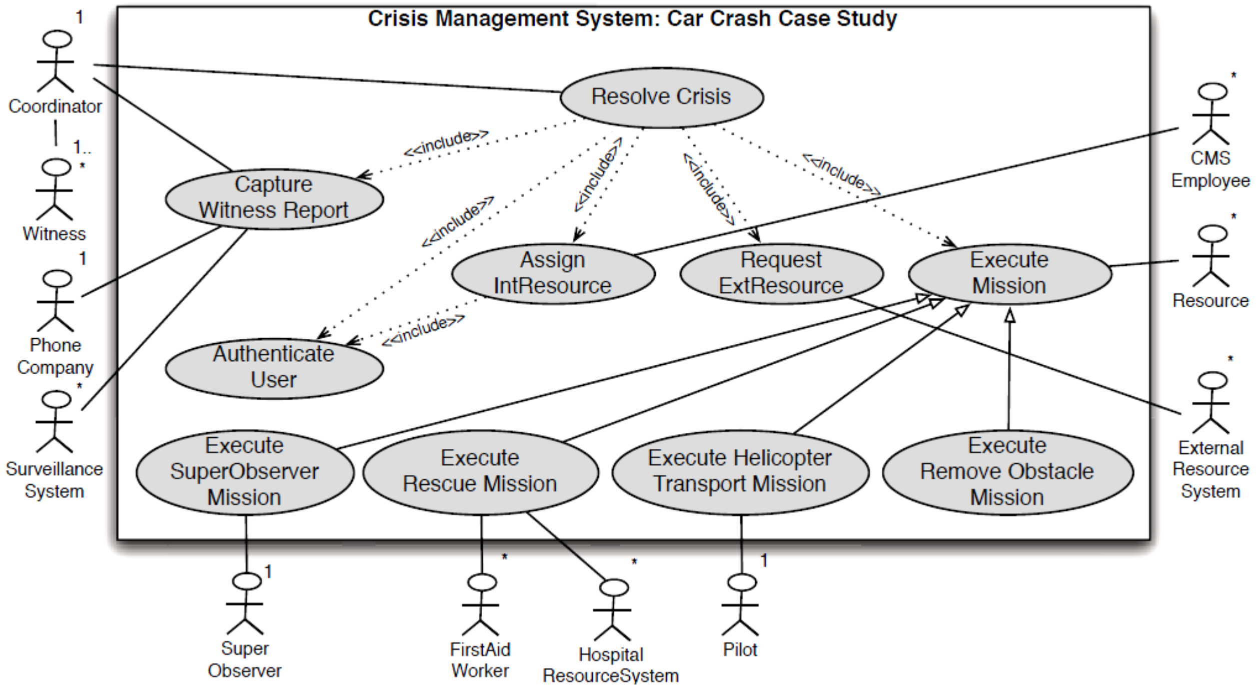


Figure 5.1: CCCM System: A Standard Use Case Diagram (Source: Kienzle et al., 2010)

5.4.5. Application of AODL to Car Crash Crisis Management

As described in the overview of AODL, CCCM system will be designed in three phases. In the first phase, aspects will be represented in the join point identification diagram and their behaviour will be represented along with the behaviour of other objects of the related module in the join point behavioural model. In the second phase, aspects will be designed along with their pointcuts and advices. The relationship between pointcuts and advices will also be captured using an aspect-advice relationship diagram. In the third phase, the weaving process of aspects with their related classes will be designed using an aspect-class composition model that contains an aspect-class dynamic model to represent the weaving process of aspects with base objects, and an aspect-class static diagram to capture the structural relationships between aspects and classes.

5.2.4.1. AODL Structural Model

This section provides structural diagrams of Car Crash Crisis Management system designed in AODL.

5.2.4.1.1. Join Point Identification Diagrams

A Join Joint Identification Diagram is an extension to the UML's sequence diagram. It helps in identifying points or locations where an aspect superimposes its behaviour. There are a few technologies (Stein et al., 2002a; Stein et al., 2004)] which consider join points as links and do not provide design support for them. AODL, on the other hand, considers join points as execution points that define the location for aspects to interact with the base system, so it is imperative to define and represent them while designing an aspect's interaction with the base system.

UML's sequence diagram shows the message passing among the objects representing the execution flow of the system. That is why AODL extends sequence diagrams to define and represent join points along with the corresponding objects and aspects which meet at that particular point (more details in (Iqbal and Allen, 2011).

The following sections contain join point identification diagrams for those use cases of CCCM system which contain interaction with the identified aspects. A full list of use cases can be found in (Shmuel and Mezini, 2010).

Use Case: Capture Witness Report

This use case is related to the reporting done by a car crash's Witness and the receiving and recording of that report by the Coordinator in the reporting office. Figure 5.2 shows the join point identification diagram for this scenario which shows the message passing between the "Coordinator" and "Crisis Manager" objects.

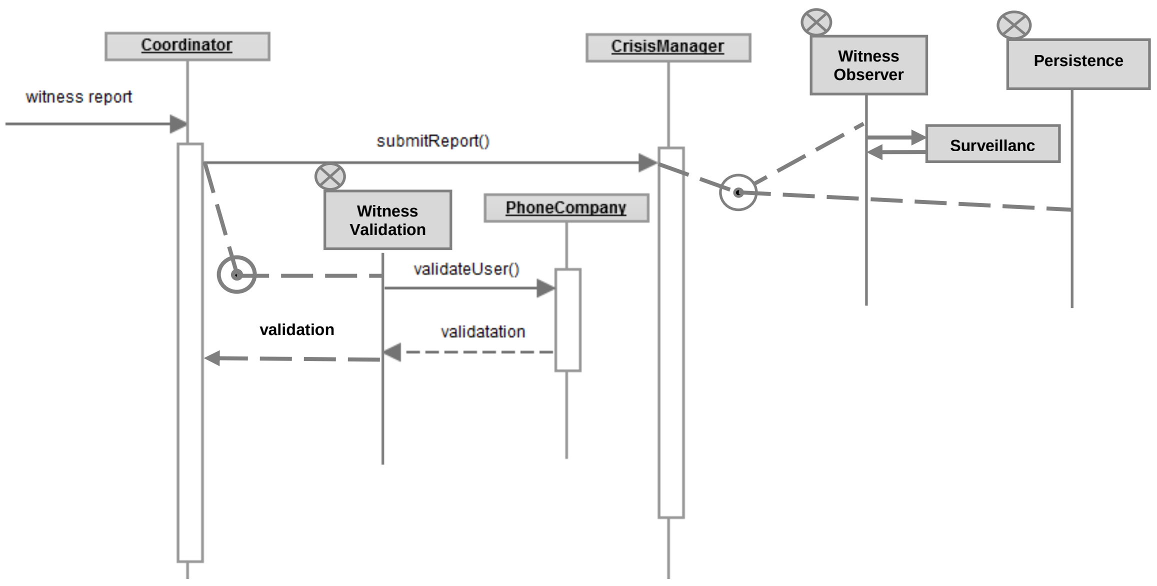


Figure 5.2: Join Point Identification Diagram for "Capture Witness Report"

When the witness report is provided to the coordinator, as an extension to the success scenario, witness might not be a credible source and the report could be a hoax. To avoid this situation, the witness must be validated. The aspect *WitnessValidation* inserts its behaviour at this point and validates the witness's credibility by verifying the phone number from the phone company.

In another extension, the witness report can be incomplete if the call is dropped while the report is being made by the witness. In this scenario, *WitnessObserver* aspect provides video recordings from the surveillance cameras. Once the report is successfully recorded, a persistence record must be maintained which is performed by the *Persistence* aspect.

Use Case: Assign Internal Resource

This use case is responsible for finding and assigning a mission to the most appropriate and available resource.

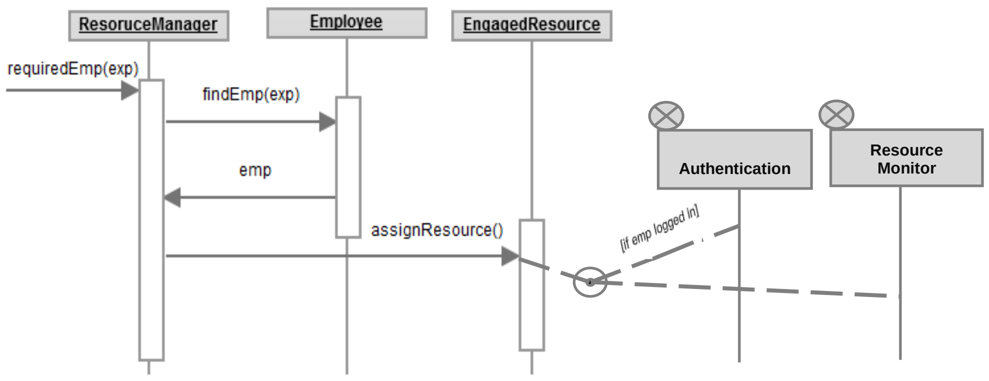


Figure 5.3: Join Point Identification Diagram for "Assign Internal Resource"

In this scenario, a join point has been identified at execution point of the assignResource() method, which assigns a mission to the employee (as shown in Figure 5.3). This join point is used by the *Authentication* aspect to check if the employee is authorized and logged in to the system. At the same join point, another aspect *ResourceMonitor* updates its record about the number of assigned resources because this aspect sets off an alarm when a threshold value of resources have been assigned to the missions to avoid shortage of resources.

Use Case: Execute Super Observer Mission

This use case is related to the SuperObserver who observes the situation at the crisis site and requests a suitable mission. The join point identification diagram (Figure 5.4) shows two identified join points on two methods where aspects *NCCInformer* and *MissionStatus* insert their behaviours.

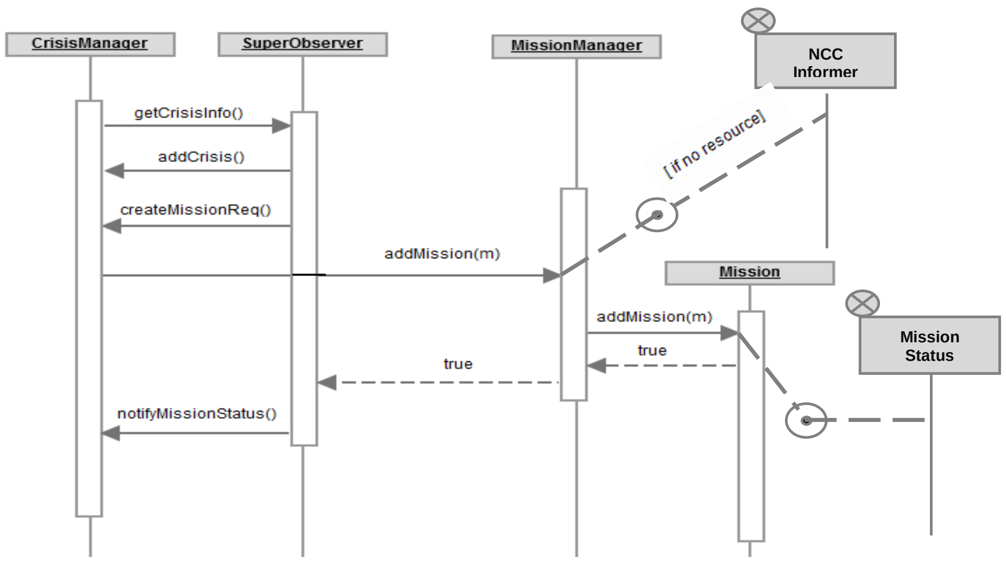


Figure 5.4: Join Point Identification Diagram for “Execute Super Observer Mission” use case

The *NCCInformer* aspect is responsible to inform the National Crisis Cell in case of non-availability of a required resource, and the *MissionStatus* aspect is responsible for assigning an appropriate flag to the mission, which is invoked once the mission is added to the system.

Use Case: Authenticate User

This use case is responsible for authenticating and authorizing employees who access the system. The join point identification diagram for this scenario (Figure 5.5) shows that aspect *Authentication* can handle this job whenever a login attempt is made by an employee.

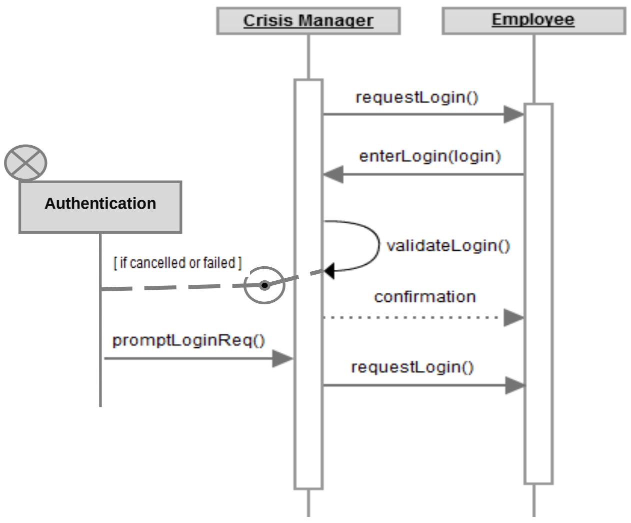


Figure 5.5: Join Point Identification Diagram for "Authenticate User"

The aspect *Authentication* superimposes its behaviour on a join point which is at the execution of the validateLogin() method.

5.2. 4.1.2. Aspect Design Diagrams

Aspect Design Diagrams are used to represent the structure of an aspect. An aspect may contain pointcuts, advices, attributes and operation. Pointcuts and advices are tightly coupled with each other (Iqbal and Allen, 2011) and their cohesive nature is represented with a pointcut-advice diagram in AODL. Pointcut-advice diagrams represent the structure in which the two are related to each other and it also shows the occurrence attribute (before, after and around) along with the advice to represent the point where advice is supposed to execute.

Aspect Design Diagrams of Car Crash Crisis Management System are explained in the following section.

Aspect Design Diagram for *MissionStatus* aspect

The *MissionStatus* aspect is responsible for updating the status of the mission. As shown in Figure 5.6, the Aspect Design Diagram of the *MissionStatus* aspect contains a pointcut-advice diagram which shows the relationship between pointcut setMissionStatus and advice updateSatus. This aspect inserts its behaviour in the CrisisManager class and the SuperObser class, which are also shown in the diagram to represent the crosscutting behaviour of the aspect.

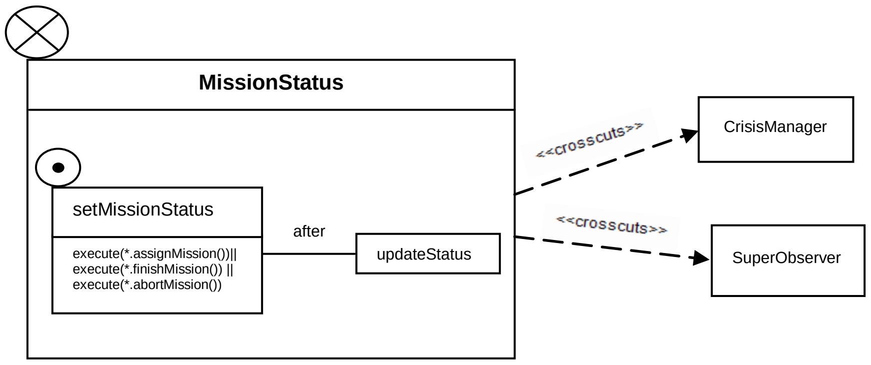


Figure 5.6: Aspect Design Diagram for *Mission Status* aspect

Aspect Design Diagram for *WitnessObserver* aspect

The *WitnessObserver* aspect is responsible for validating a witness report. If the report is incomplete or contains inadequate information, it takes feeds from surveillance cameras installed at the location of the crisis. The Aspect Design Diagram for the *WitnessObserver* aspect (Figure 5.7) shows that it contains a pointcut verifyReport which defines join points in the system where the advice updateReport will insert its behaviour. Two classes, Coordianator and Survelliance, are also shown as they will be crosscut by the *WitnessObserver* aspect.

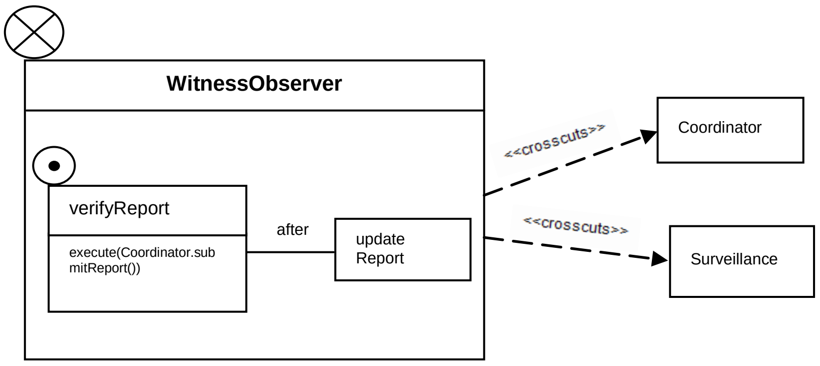


Figure 5.7: Aspect Design Diagram for *WitnessObserver* aspect

Aspect Design Diagram for *Persistence* aspect

The *Persistence* aspectmaintains a persistent record of events and saves information about the important transactions. The Aspect Design Diagram for the *Persistence* aspect (Figure 5.8) shows that it has a saveReport advice which is connected to the saveReport pointcut with an "after" occurrence type that indicates that the saveReport advice will run after join points in the saveReport pointcut successfully execute.

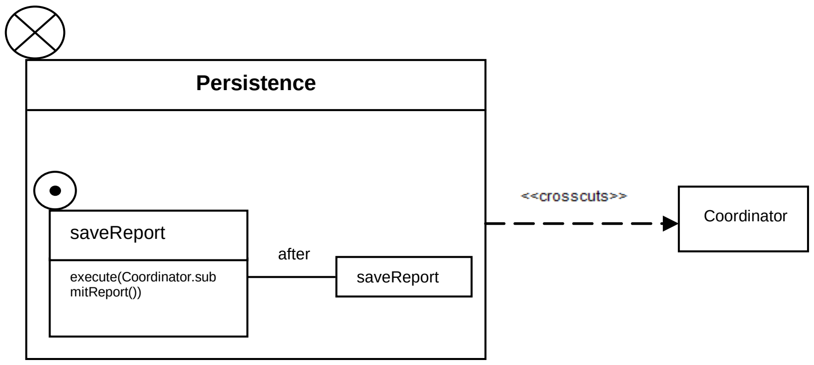


Figure 5.8: Aspect Design Diagram for *Persistence* aspect

Aspect Design Diagram for Authentication aspect

The *Authentication* aspect helps in authenticating all the users who interact with the system. The Aspect Design Diagram for Authentication (Figure 5.9) shows that whenever an object of CrisisManager class or ResourceManager class makes a transaction, *Authentication* aspect executes to verify the users. There is a pointcut checkLogin consisting of join points which identify the locations for the *Authentication* aspect to run the authenticateEmp advice.

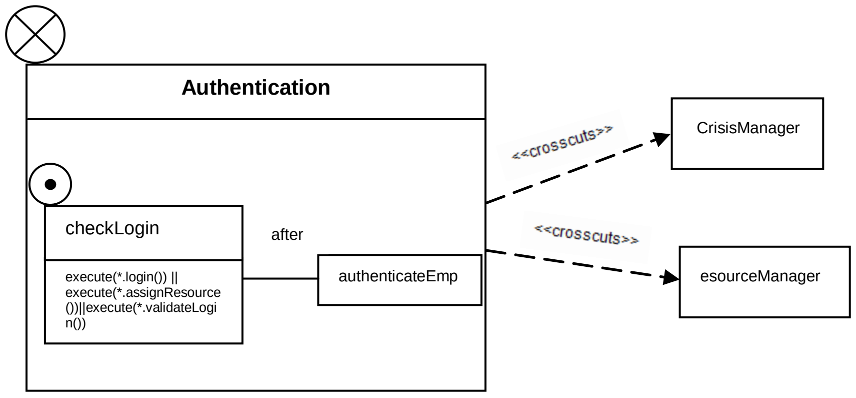


Figure 5.9: Aspect Design Diagram for *Authentication* aspect

5.2. 4.1.3. Aspect-Class Structure Diagram

This diagram helps in organizing all the entities (aspects and objects) which are involved in an executing process. This diagram shows the structure of the module with a representation of aspects interacting with classes. Figure 5.10 shows the Aspect-Class Structure diagram for CCCM system.

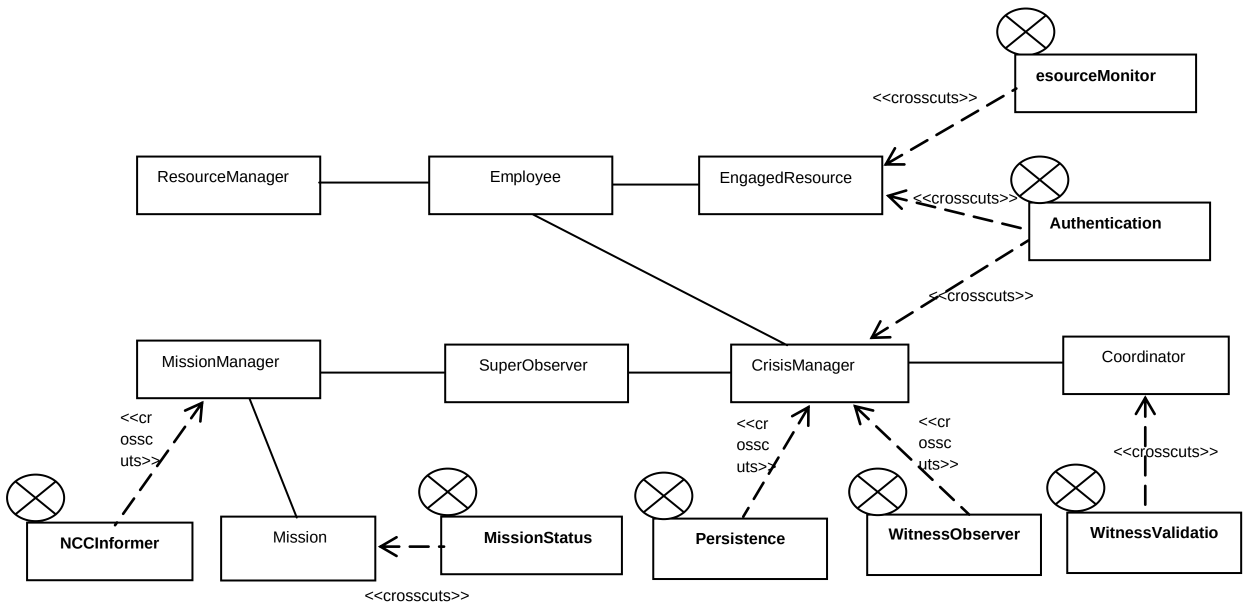


Figure 5.10: Aspect-Class Structure Diagram for Car Crash Crisis Management System

The diagram shows how aspects of CCCM system are connected with the base classes. The diagram helps in representing the structure of the aspectual and base constructs and also helps in depicting crosscutting structure of the system.

5.2.4.2. AODL Behavioural Model

AODL's behavioural model is responsible for designing the behaviour of an aspect, its elements and its weaving process. This model has two diagrams, Join Point Behavioural Diagram and Aspect-Class Dynamic Model. These diagrams show how the behaviour of an aspect and aspectual elements can be represented along with the behaviour of objects and their respective classes.

The following section will provide a complete AODL behavioural model for CCCM system.

5.2.4.2.1. Join Point Behavioural Diagrams

Join point behavioural diagrams help in identifying and representing the location of a join point, where aspects of the system insert their behaviour, within execution flow of the system. The activity diagram of UML is extended to show join points along with the activities of the system.

Join Point Behavioural diagrams for CCCM system are explained in the following sections.

Capturing Witness Report

In CCCM system, when a report is made by the witness from the site of a crash, a report is collected and recorded. At this point, the witness is validated by an aspect *WitnessValidation.* During the recording process of the report, if the report is not fully gathered due to either the witness's call being dropped or the incompletion of the report, the *WitnessObserver* aspect provides extra information from the surveillance cameras installed at the site of the crash. The join point for Persistence is also shown in the Figure 5.11, which records all the data at the end of the process.

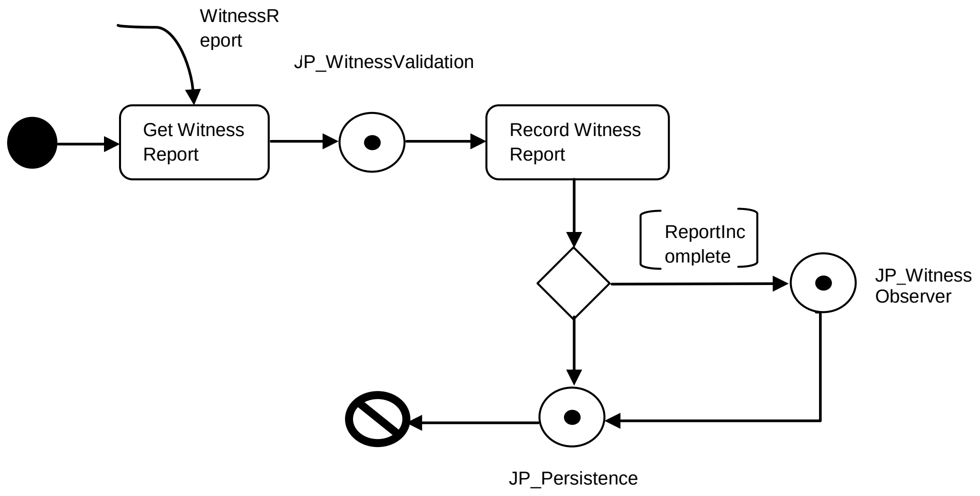


Figure 5.11: Join Point Behavioural Diagram for "Capture Witness Report" module

Assign Internal Resource

Figure 5.12 shows the Joint Point Behavioural Diagram for the "Assign Internal Resource" module, which finds a suitable resource for the required job. The diagram helps in locating the join points for the *ResourceMonitor* aspect, which checks if there are enough resources remaining after a resource is engaged, and the *Authentication* aspect, which asks users to reenter authentication details if they are not logged in.

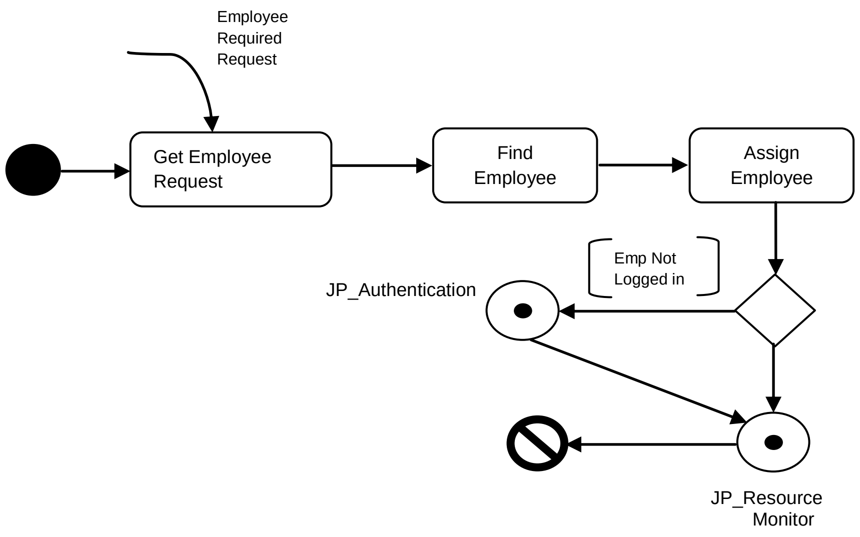


Figure 5.12: Join Point Behavioural Diagram for "Assign Internal Resource" module

Super Observer Mission

The Join Point behavioural diagram shown in Figure 5.13 is for "Super Observer's Mission". It indicates two join points for *MissionStatus* and *NCCInformer* aspects. The MissionStatus aspect assigns a status to a mission. It happens once a mission is successfully added. There is another aspect, *NCCInformer* which is responsible for contacting NCC (National Crisis Center) in case of unavailability of a resource.

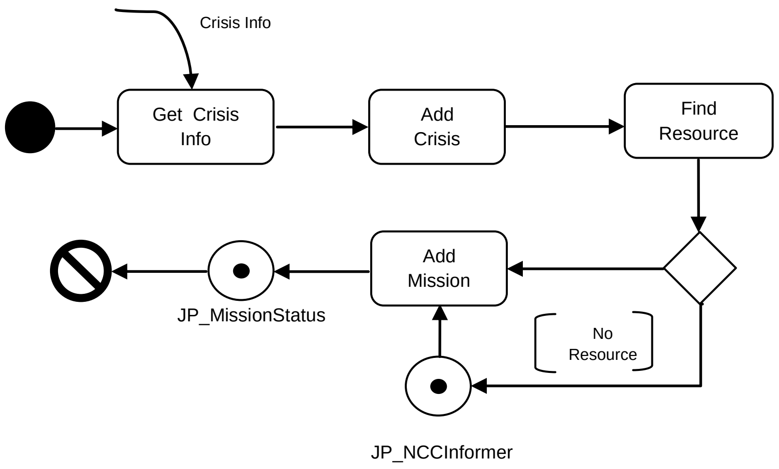


Figure 5.13: Join Point Behavioural Diagram for "Super Observer's Mission" module

Authentication

The Join Point Behavioural model for the Authentication module shows how the *Authentication* aspect interacts during the execution flow of the Authentication process.

Figure 5.14 shows that once the validate login starts the *Authentication* aspect interacts with the base system and checks the login details of the user. If the details match with saved details, the process completes successfully, but where the details do not match, new details are asked and the whole process is repeated.

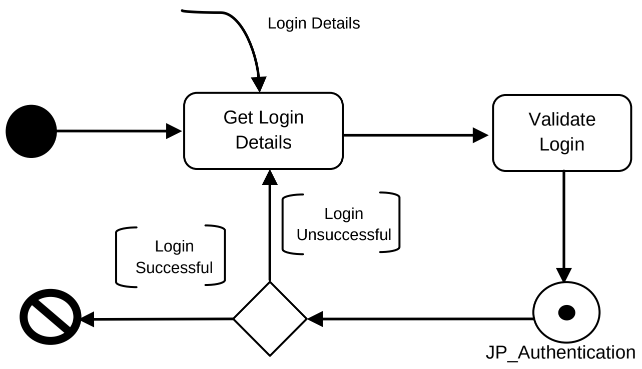


Figure 5.14: Join Point Behavioural Diagram for "Authentication" module

5.2.4.2.2. Aspect-Class Dynamic Model

The weaving process starts at the execution of the system. The behaviour (advices) of aspects run at predefined points (join points). It is imperative to depict this process in the system design to understand the weaving process and locations in the system flow where aspects interact with the base system. The Aspect-Class Dynamic Model is used to capture this information. Details about the model can be found in Iqbal and Allen (2011). The following section shows the Aspect-Class dynamic diagrams for the CCCM system.

Capturing Witness Report

Capturing of witness report involves collecting information from a witness and initiating the process of assigning a mission for the incident. As Figure 5.15 shows the coordinator receives the report from a witness and passes that information to the CrisisManager to start the process of assigning a suitable mission. During this flow, multiple aspects execute their behaviours, such as *WitnessValidation* which verifies the authenticity of the witness, *WitnessObserver* which provides surveillance feeds in case the report is incomplete, *MissionStatus* which updates the status of the mission and *Persistence* ensures all the important data is kept persistent.

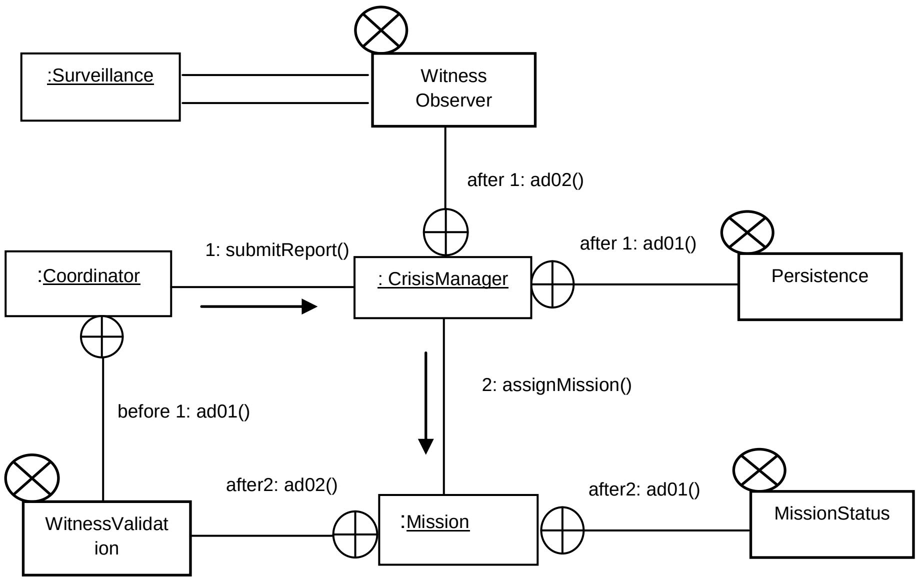


Figure 5.15: Aspect-Class Dynamic Diagram for "Capturing Witness Report"

Figure 5.15 shows how aspects insert their behaviour at join points. For instance, the *Persistence* aspect and the *WitnessObserver* aspect interact with the CrisisManager object after submitReport() method finishes execution. The *MissionStatus* aspect runs its advice ad01() after assignMission() method of the Mission object finishes execution. And, the *WitnessValidation* aspect runs two of its advices ad01() and ad02() before execution of submitReport() method of the CrisisManager object, and assignMission() of the Mission object respectively.

Assigning Internal Resource

The ResourceManager starts a process to find suitable resources for the mission. As Figure 5.19 shows the Employee class finds a suitable employee who is assigned to the mission and is updated as an engaged resource in the system. During the execution of this process the *Authentication* aspect verifies the authentication of all assigned employees and ensures all the users are properly logged in. The *MissionStatus* aspect updates the status of the mission once a mission has been assigned.

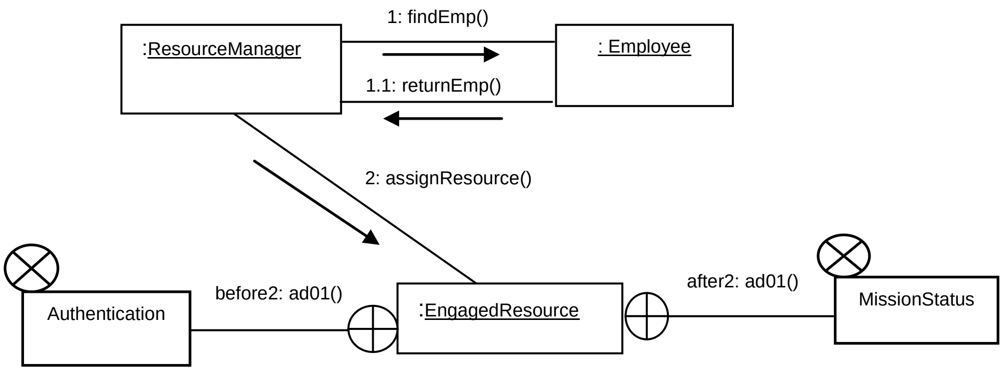


Figure 5.16: Aspect-Class Dynamic Diagram for "Assigning Internal Resource"

The model also shows the join points where these aspects interact with the base objects. For instance, the *Authentication* aspect runs its advice ad01() before assignResource() method of the EngagedResource object is called and the *MissionStatus* aspect runs its advice ad01() after this method finishes its execution.

Execute Super Observer Mission

Figure 5.17 shows a flow of execution of super observer's mission. Super observer collects information from the site of the incident which is used to identify a suitable mission. The CrisisManager adds a new mission on the basis of the information provided by super observer. There are two aspects, *NCCInformer* and *MissionStatus*, which interact with the system at this point. The *NCCInformer* is responsible for contacting national crisis center in case a suitable resource is not found and the *MissionStatus* keeps the status of the mission updated. The following Figure 5. 17 shows the weaving process of both the aspects into the base system.

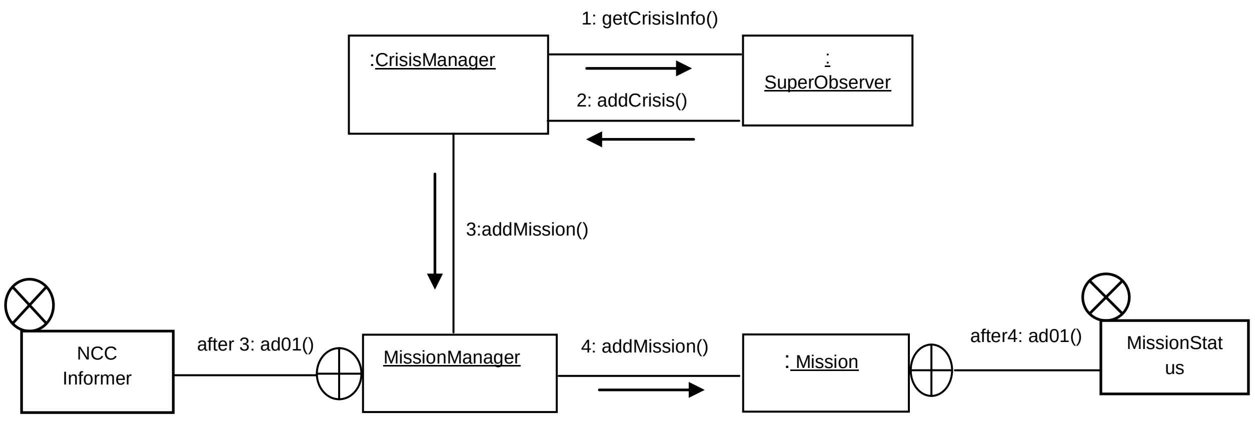


Figure 5.17: Aspect-Class Dynamic Diagram for "Execute Super Observer Mission"

To demonstrate the exact locations of aspect interactions, a notation is associated with weaving association. For instance, to show the join point where the *NCCInformer* aspect inserts its behaviour (advice ad01()) after addMission() method of MissionManager object finishes its exectution, *after 3: ad01()* is labeled on the weaving association. Similarly, ad01() of the *MissionStatus* aspect has also been shown being inserted after addMission() method of Mission object finishes its execution.

Authentication

Authentication aspect weaves its behaviour when the employees are verified to be properly logged in. The aspect checks the login session and details and prompts a reentry message if a validation process fails.

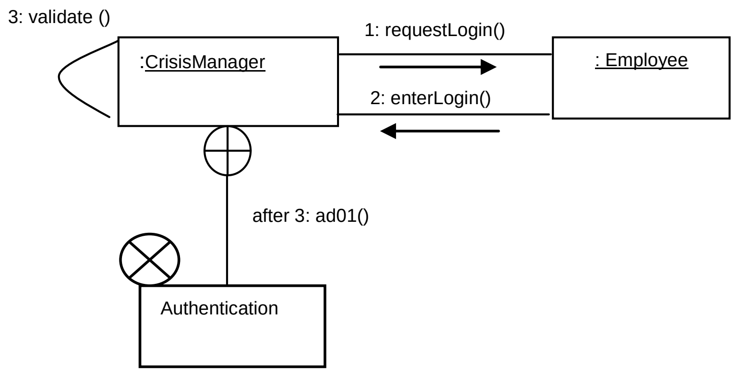


Figure 5.18: Aspect-Class Dynamic Diagram for “Authentication”

The model in Figure 5.18 shows that, after 3: ad01() is labeled on weaving association between the *Authentication* aspect and the CrisisManager object. It can be translated as: the advice ad01() of the *Authentication* aspect will be executed once validate() method of the CrisisManager object finishes its execution.

5.2.4.3. Pointcut Composition Model

The pointcut composition model shown in Figure 5.19 provides a design of pointcuts of *Persistence* aspect and *WitnessObserver* aspect. Each aspect has one pointcut which has one related advice. As shown in the figure, these aspects have a shared join point conflict. Both of them run their advice after the execution of submitReport() method of **Coordinator** class. This model resolves this conflict by associating both aspects with a <<*precedence*>> stereotype which demonstrates that *WitnessObserver* aspect will have priority over *Persistence* aspect.

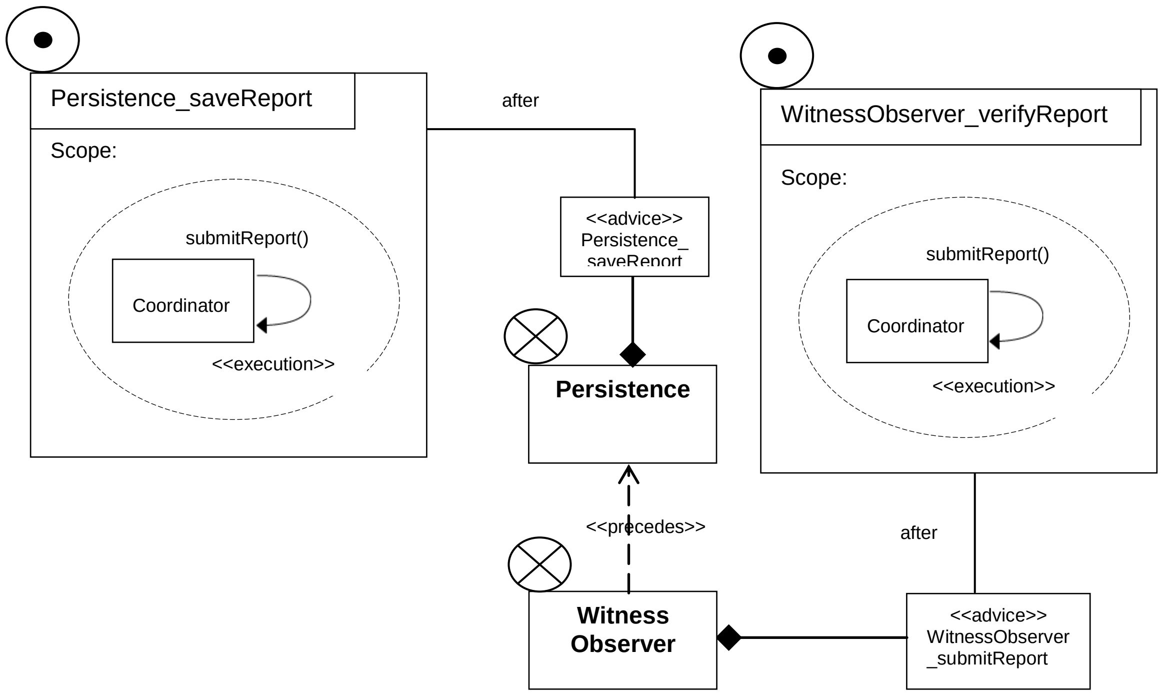


Figure 5.19: Pointcut Composition Model of *Persistence* and *WitnessObserver* aspects

Similarly, a pointcut composition model for MissionStatus aspect and Authentication aspect has been shown in Figure 5.20. The model shows that MissionStatus aspect has one pointcut that contains three join point predicates. Similarly, Authentication aspect has one pointcut and that pointcut contains three join point predicates. Each join point predicate has been designed separately and the composition among them has been modelled using the notations of pointcut composition model.

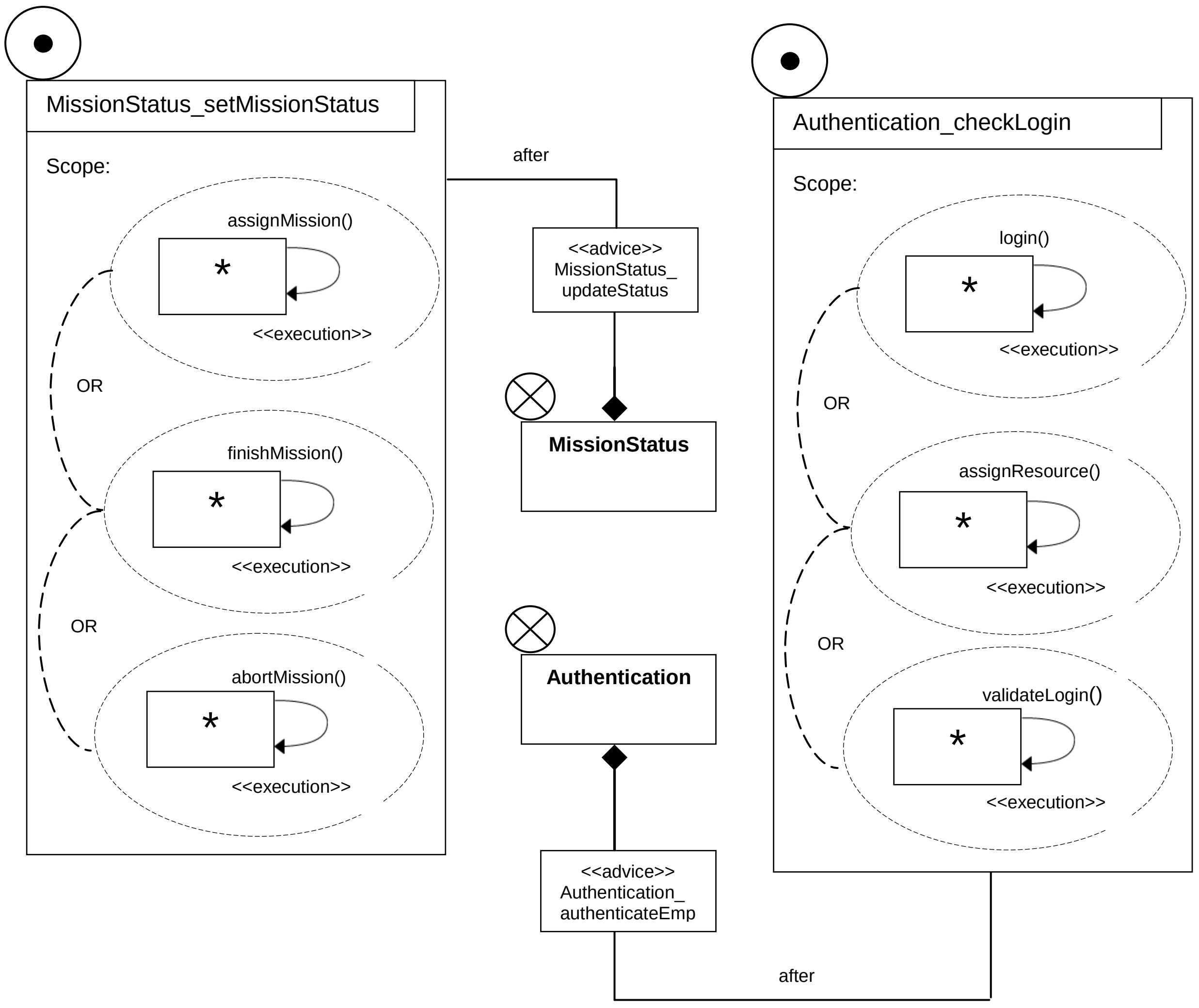


Figure 5.20: Pointcut Composition Model of *MissionStatus* and *Authentication* aspects

5.2.4.4. Pointcut Table

Table 5.1 shows specification of pointcuts of all of the aspects. There are no identified clashes among aspects so Precedence column remains empty.

Table 5.1 - Pointcut Table for Car Crash Crisis Management System

	<<aspect>> **MissionStatus**	<<aspect>> **Persistence**	<<aspect>> **Authentication**	<<aspect>> **WitnessObserver**	**Precedence**
	<<advice>> **updateStatus**	<<advice>> **saveReport**	<<advice>> **authenticateEmp**	<<advice>> **updateReport**	
Class CrisisManager					
login()			execute		
assignMission()	execute				
AbortMission	execute				
Class SuperObserver					

finishMission()	execute				
abortMission()	execute				
Class Coordinator					
submitReport()		execute			
Class ResourceManager					
login()			execute		
assignResource()			execute		
Class Surveillance					
submitReport()				execute	
Pointcut Definition	execute(*.assignMission())|| execute(*.finishMission())|| execute(*.abortMission())	execute(Coordinator.submitReport())	execute(*.login())|| execute(*.assignResource())||execute(*.validateLogin())	execute(Coordinator.submitReport())	
Pointcut Name	**setMissionStatus()**	**saveReport()**	**checkLogin()**	**verifyReport()**	

5.4.6. Discussion

This section has demonstrated application of AODL to the Car Crisis Management System case study. All the identified aspects have been designed using AODL design notations and design diagrams. Each model depicts different perspective of the design of aspectual constructs and their relationships with the base constructs. The application provides sound evidence that AODL can be applied to a complex system involving multiple aspects. The case study also provided a demonstration of usage of all the notations and diagrams of AODL.

5.3. SpaceWar Game Case Study

The game has been chosen from the example projects provided by AspectJ plugin for eclipse (AspectJ, 2012). The traditional *SpaceWar* game has been implemented using aspect-oriented AspectJ programming.

5.5.2. Identified Aspects in the System

There are two types of aspects identified in the system, functional aspects and non-functional aspects.

Functional Aspects:

The following are the functional aspects of the game:

1. **DisplayAspect:** This aspect provides the look and display of the game. This aspect is also responsible for displaying messages, modifications, updates, exceptions and the game itself.
2. ***EnsureShipIsAlive*****:** This aspect ensures the ship is alive after every change and progress in the game.
3. **GameSynchronization:** This aspect is responsible for synchronizing the access to the methods of the game. The aspect executes with every movement of the ship or any change in the game concerning the ship.

Non-Functional Aspects:

The following are non-functional aspects of the game:

1. **RegistrySynchronization:** This aspect is responsible for synchronized access to the registry methods during the game.
2. **RegistryProtection:** This aspect keeps track of every space object in the game.
3. **Debug:** This aspect is responsible for displaying all information related to the debugging process on the main display screen.

5.5.3. Application of AODL toSpaceWar Game

The system is designed in two phases, structural design and behavioural design. The following sections present a complete design of the system.

5.3.2.1. AODL Structural Model

This section provides structural diagrams of the *SpaceWar* game designed in AODL.

5.3.2.1.1. Join Point Identification Diagrams

Join Point Identification Diagrams help in identifying join points early in the design of the system. They are defined on the sequence diagrams of UML. The following sections show JPIDs for *SpaceWar* game.

Use Case: Start Game

In the Start Game use case, the player enters the command to start the game. The scenario is shown in Figure 5.21, where there are three aspects that interact with the base program's objects. The *DisplayAspect* starts the initial energy scores of the player and timer of the game. This has been shown in the figure in the form of two join points identified at the execution of the producePacket() and runTimer() method.

The *DisplayAspect* also displays the game Robot once it starts as a result of execution of the startRobot() method of the Robot object, which has also been shown in the figure below.

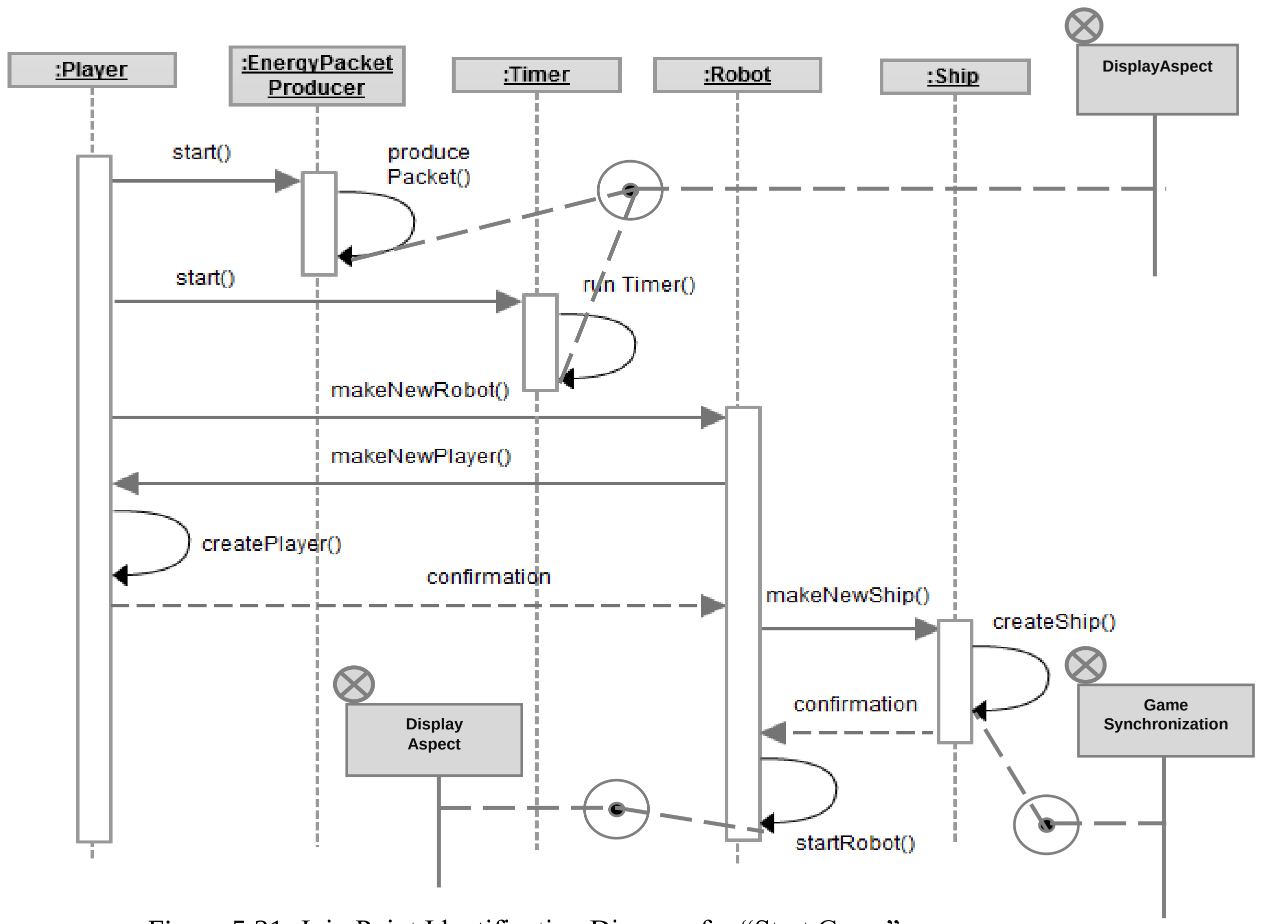


Figure 5.21: Join Point Identification Diagram for “Start Game” use case

The *GameSynchronization* aspect executes its behaviour at the execution of createShip() method which has also been depicted in the JPID shown in Figure 5.21.

Use Case: Move Ship

This use case captures the Move Ship command by the user. AODL identifies a join point for the *EnsureShipIsAlive* aspect around the rotate(string dir) method. The JPID model shown in Figure 5.22 depicts the join point before call of the method and after the execution of the method. As this join point is set around the method so the identification diagram captures both the points around the method.

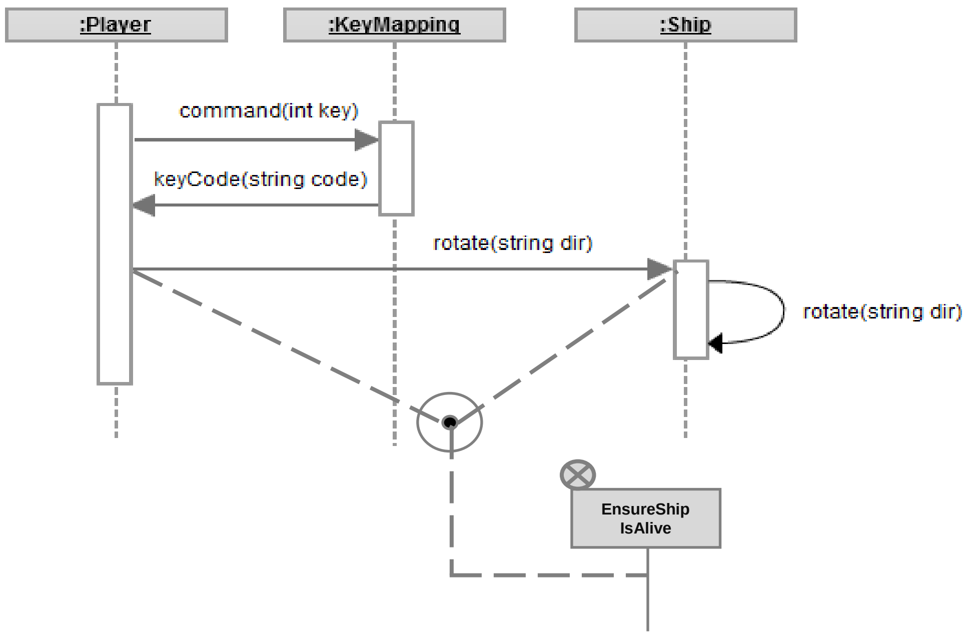


Figure 5.22: Join Point Identification Diagram for "Move Ship" use case

Use Case: Thrust

The use case Thrust captures the scenario where the player commands the ship to thrust. To ensure the ship is alive before the command is made and after the thrust is performed, the aspect *EnsureShipIsAlive* executes its advice to find out whether the ship is alive. An around join point is identified on the thrust(true) method where this aspect executes its advice. The JPID shown in Figure 5.23 captures the location of the join point at the call of the method and after its execution.

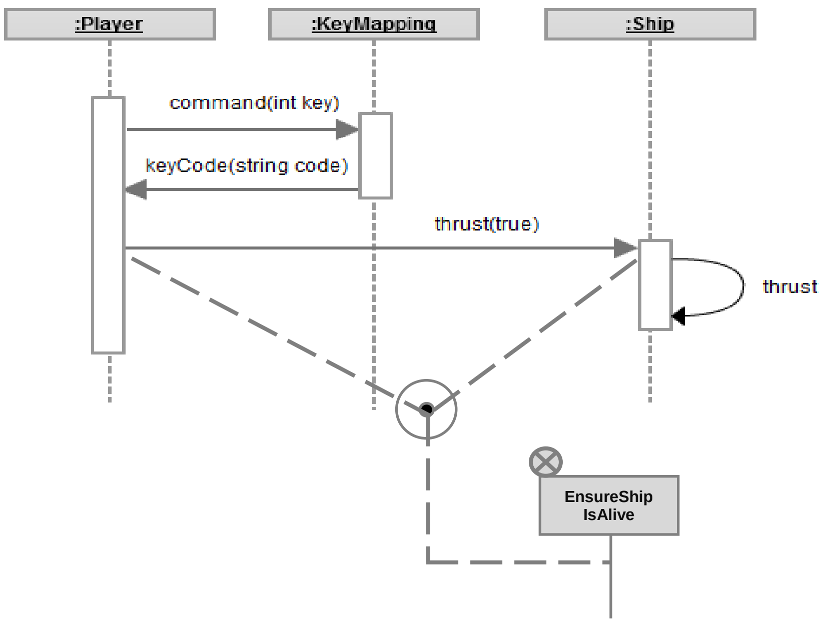


Figure 5.23: Join Point Identification Diagram for "Thrust" use case

Use Case: Fire

This use cases captures the scenario of Fire command by the player. Before fire() message is executed, *EnsureShipIsAlive* aspect checks whether the ship is still alive. The join point for the aspect is around the call of the method. The JPID shown in Figure 5.24 depicts the join point before the call of fire() method and after the execution of the method.

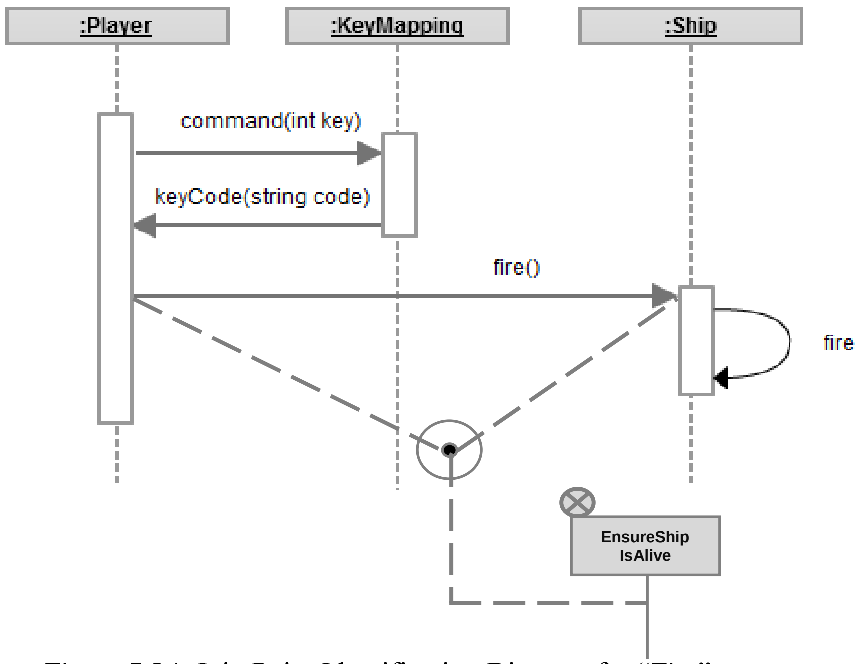


Figure 5.24: Join Point Identification Diagram for "Fire" use case

Use Case: Handle Collision

Once the ship collides with a space object or a space object collides with other space objects, the system handles the collision as shown in the sequence diagram of Handle Collision use case in Figure 5.25. When the System object calls the handleCollision() method of the Game object, *GameSynchrnoization* aspect synchronizes the call in the presence of all thread calls of the game. The aspect's join point has been shown in the figure at the execution of the handleCollision() method.

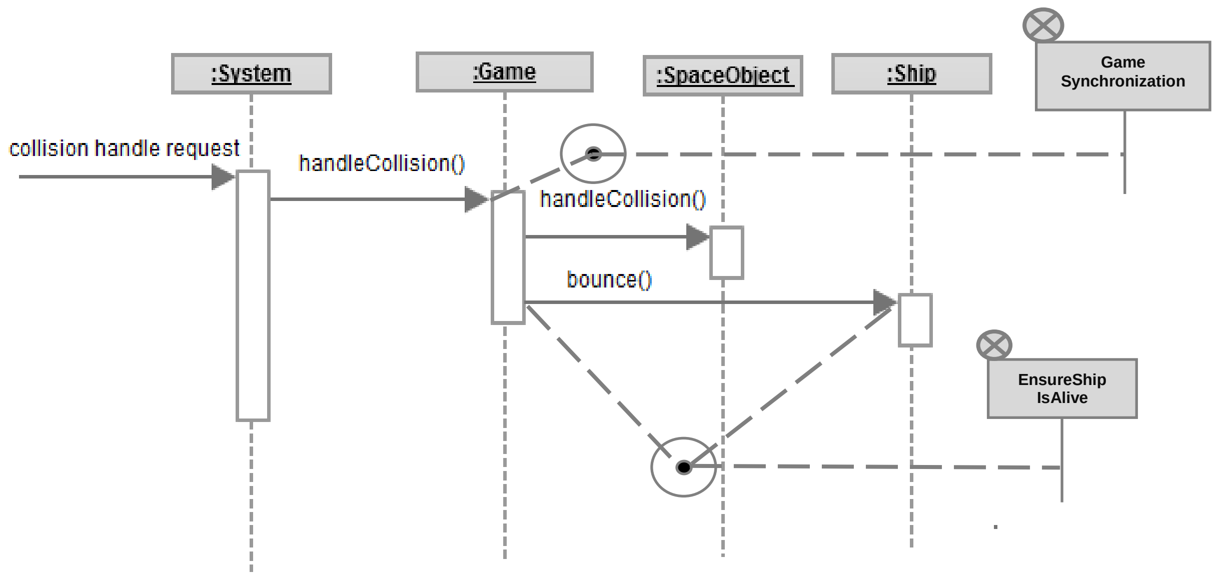


Figure 5.25: Join Point Identification Diagram for "Handle Collision" use case

If one of the colliding objects is the ship, the bounce() method of the Ship object is called. At this point, *EnsureShipIsAlive* aspect makes sure that the ship is still alive. The aspect executes around the bounce() method which has been shown in JPID in Figure 5.25 in form of two join points. One is at the call of bounce() method and one is on the execution of the method.

5.3.2.1.2. Aspect Design Diagrams

Aspect Design Diagrams are used to represent the structure of an aspect. An aspect may contain pointcuts, advices, attributes and operation. The diagrams help in capturing structural properties of the aspects and structural crosscutting of the system.

Coordinator, RegistrySynchronization and *GameSynchronization* Aspects

The Coordinator aspect has two child aspects, *RegistryShnchronization* aspect and *GameSynchronization* aspect, as shown in Figure 5.26. The aspect design model for all the aspects are shown in the figure. The relationships between pointcuts and their advices are depicted with the help of pointcut-advice diagrams.

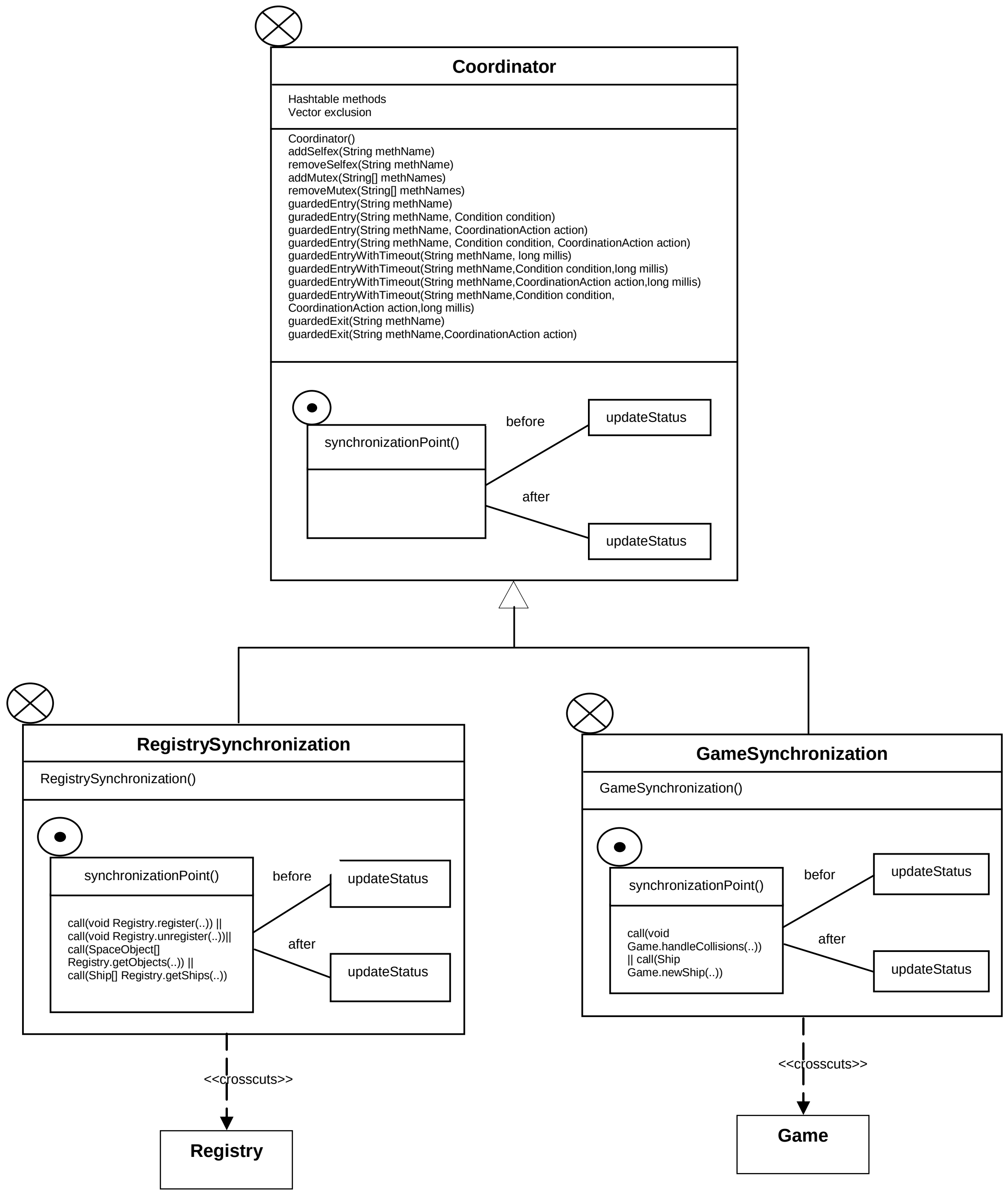

Coordinator
Hashtable methods
Vector exclusion
Coordinator()
addSelfex(String methName)
removeSelfex(String methName)
addMutex(String[] methNames)
removeMutex(String[] methNames)
guardedEntry(String methName)
guradedEntry(String methName, Condition condition)
guardedEntry(String methName, CoordinationAction action)
guardedEntry(String methName, Condition condition, CoordinationAction action)
guardedEntryWithTimeout(String methName, long millis)
guardedEntryWithTimeout(String methName,Condition condition,long millis)
guardedEntryWithTimeout(String methName,CoordinationAction action,long millis)
guardedEntryWithTimeout(String methName,Condition condition, CoordinationAction action,long millis)
guardedExit(String methName)
guardedExit(String methName,CoordinationAction action)
synchronizationPoint()
before
updateStatus
after
updateStatus
RegistrySynchronization
RegistrySynchronization()
synchronizationPoint()
call(void Registry.register(..)) || call(void Registry.unregister(..))|| call(SpaceObject[] Registry.getObjects(..)) || call(Ship[] Registry.getShips(..))
before
updateStatus
after
updateStatus
GameSynchronization
GameSynchronization()
synchronizationPoint()
call(void Game.handleCollisions(..)) || call(Ship Game.newShip(..))
befor
updateStatus
after
updateStatus
<<crosscuts>>
Registry
<<crosscuts>>
Game

Figure 5.26: Aspect Design Diagram for *Coordinator* and its child aspects

The structural crosscutting is depicted by <<*crosscuts*>> stereotype with Game and Registry classes.

DisplayAspect

The *DisplayAspect* is responsible for printing messages on the display screen about the change in the game and calculations of energy and time of the game. The aspect design model for this aspect is shown in Figure 5.27, which contains pointcut-advice diagrams to show the relationships between pointcuts and advices and a crosscuts relationship that shows the structural relationships of the aspect with Game, Player and Display classes.

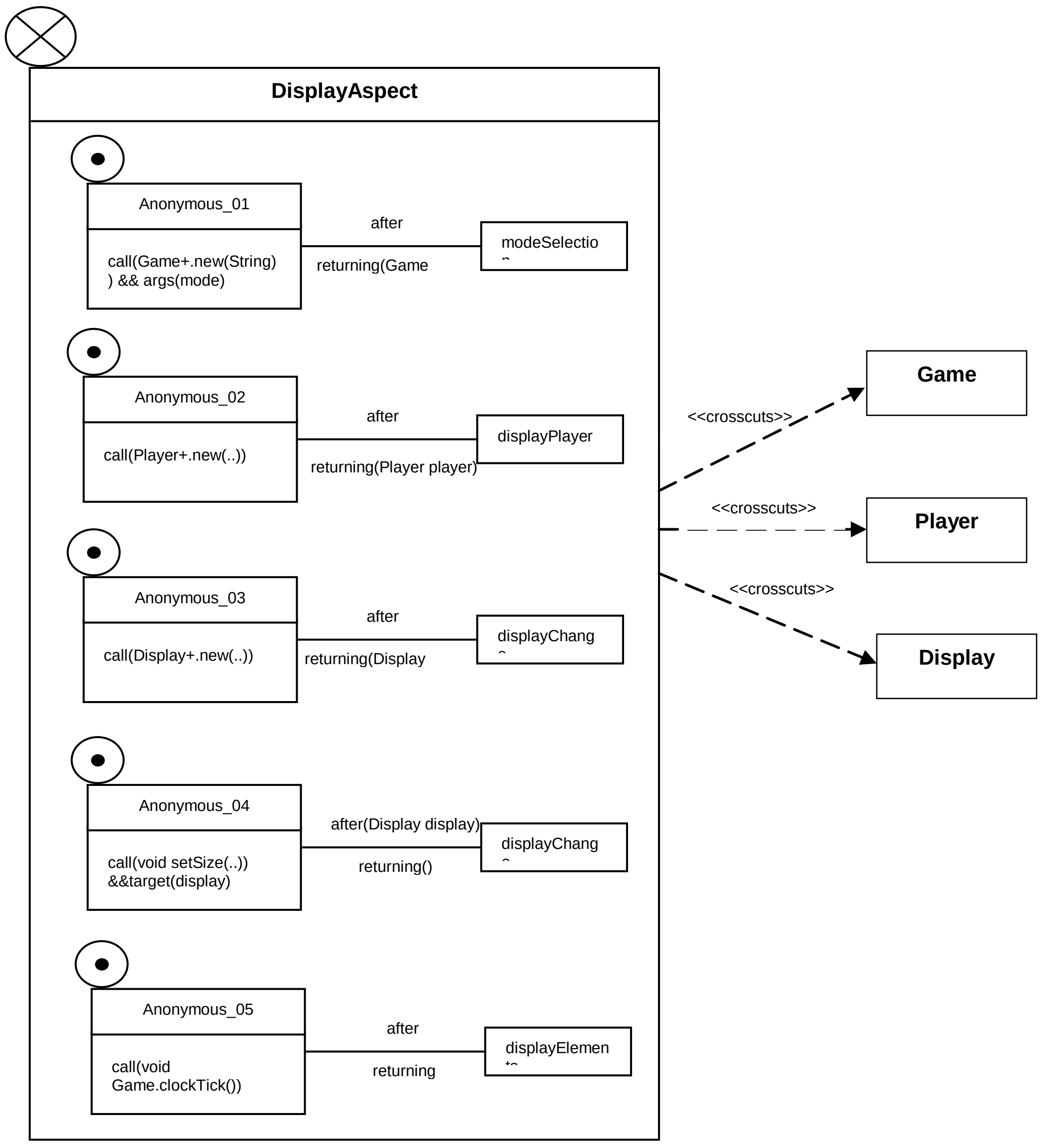


Figure 5.27: Aspect Design Diagram for *DisplayAspect* aspect

EnsureShipIsAlive Aspect

This aspect is responsible for ensuring whether the ship is alive before a new command is passed to the Ship class. The aspect design model for the aspect is shown in Figure 5.28. It shows pointcuts and advices of the aspect in a pointcut-advice diagram. The structural relationship of the aspect with the Ship class is shown with the help of <<crosscuts>> relationship stereotype.

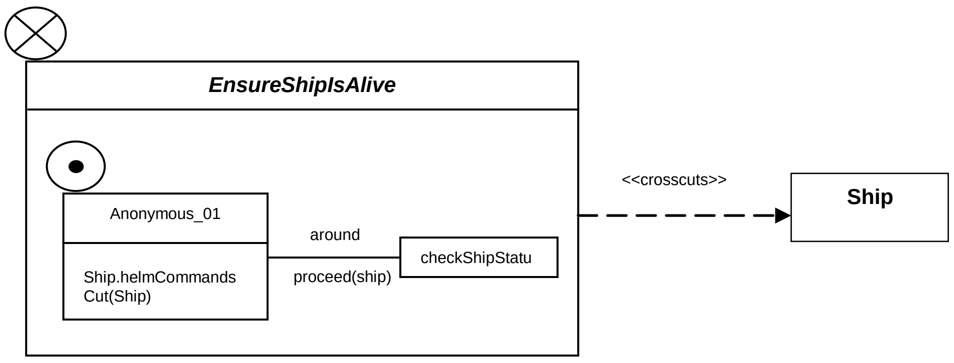


Figure 5.28: Aspect Design Diagram for *EnsureShipIsALive* aspect

RegistrationProtection Aspect

The aspect *RegistrationProtection* handles exceptions thrown by the register() and unregister() methods of the Registry class. The aspect design model depicted in Figure 5.29 shows the pointcuts and advices of the aspect and its relationship with Registry class.

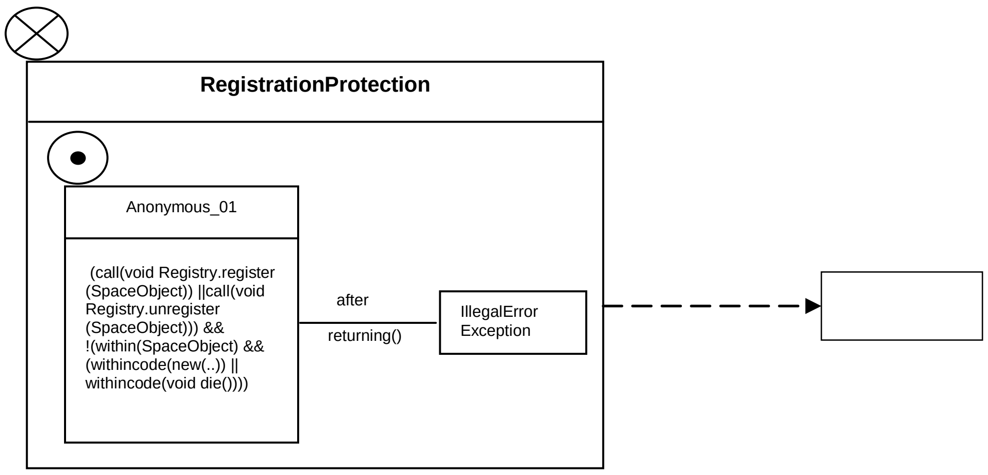

RegistrationProtection
Anonymous_01
(call(void Registry.register (SpaceObject)) ||call(void Registry.unregister (SpaceObject))) && !(within(SpaceObject) && (withincode(new(..)) || withincode(void die())))
after
returning()
IllegalError Exception

Figure 5.29: Aspect Design Diagram for *RegistrationProtection* aspect

Aspect Design Model for Debug Aspect

The aspect Debug is responsible for displaying messages about the control flow during the debug process. The aspect design model for the Debug aspect, depicted in Figure 5.30, shows the relationship between pointcuts and advices with the help of pointcut-advice diagrams. The structural relationships with the base classes have been shown with the help of associations with <<*crosscuts*>> stereotypes.

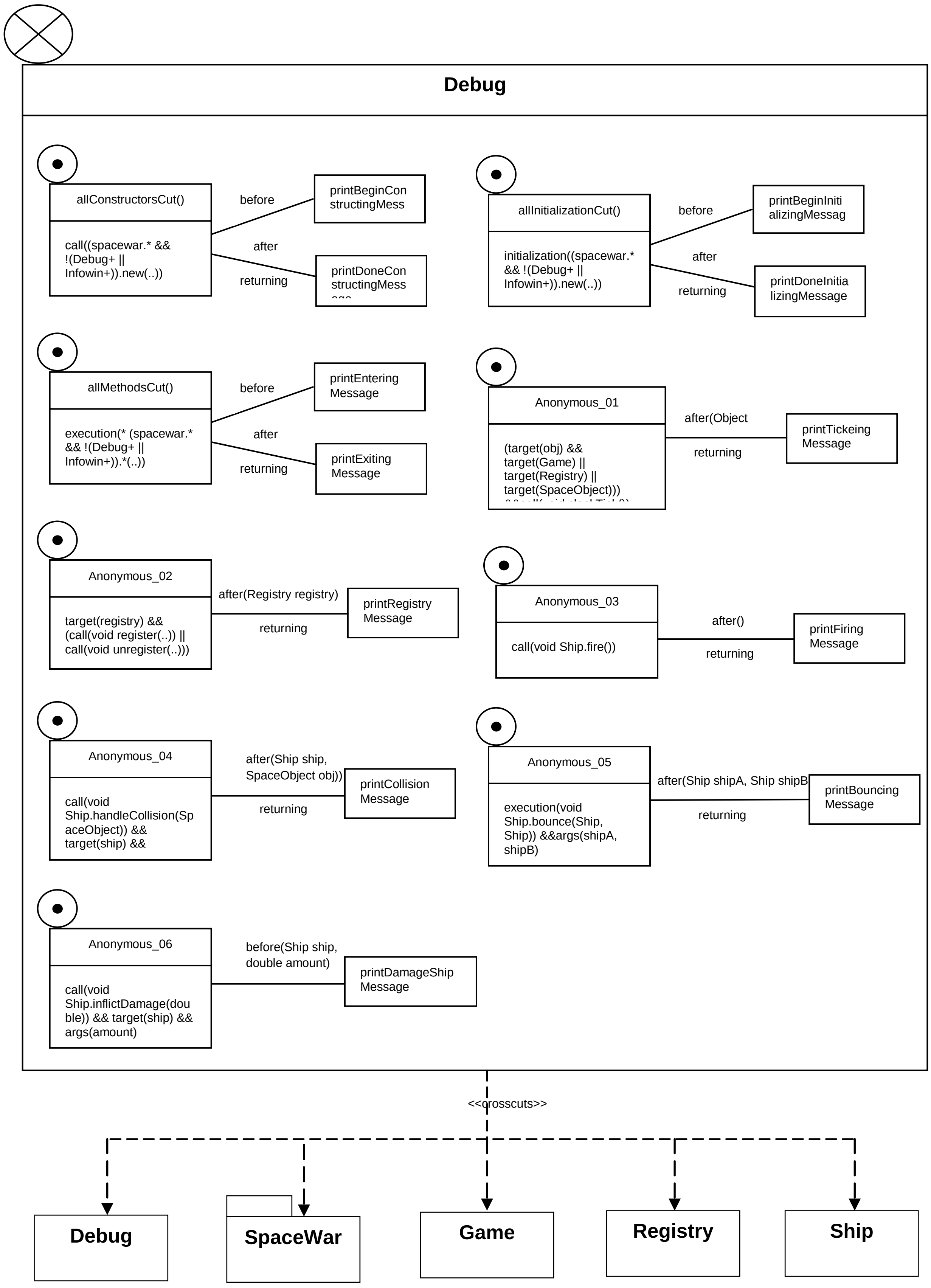

Debug
allConstructorsCut()
call((spacewar.* && !(Debug+ || Infowin+)).new(..))
before
printBeginConstructingMess
after
returning
printDoneConstructingMess
allInitializationCut()
initialization((spacewar.* && !(Debug+ || Infowin+)).new(..))
before
printBeginInitializingMessag
after
returning
printDoneInitializingMessage
allMethodsCut()
execution(* (spacewar.* && !(Debug+ || Infowin+)).*(..))
before
printEntering Message
after
returning
printExiting Message
Anonymous_01
(target(obj) && target(Game) || target(Registry) || target(SpaceObject)))
after(Object
returning
printTickeing Message
Anonymous_02
target(registry) && (call(void register(..)) || call(void unregister(..)))
after(Registry registry)
returning
printRegistry Message
Anonymous_03
call(void Ship.fire())
after()
returning
printFiring Message
Anonymous_04
call(void Ship.handleCollision(SpaceObject)) && target(ship) &&
after(Ship ship, SpaceObject obj))
returning
printCollision Message
Anonymous_05
execution(void Ship.bounce(Ship, Ship)) &&args(shipA, shipB)
after(Ship shipA, Ship shipB
returning
printBouncing Message
Anonymous_06
call(void Ship.inflictDamage(double)) && target(ship) && args(amount)
before(Ship ship, double amount)
printDamageShip Message
<<crosscuts>>
Debug
SpaceWar
Game
Registry
Ship

Figure 5.30: Aspect Design Diagram for *Debug* aspect

5.3.2.1.3. Aspect-Class Structure Diagram

AODL shows structural relationships among all the aspects of the system and base classes in an aspect-class structure diagram. The diagram depicted in Figure 5.31 shows the base classes along with their interacting aspects. The diagram helps in understanding the structural characteristics of the system and aids in establishing the overall relationships among classes and between aspects and classes.

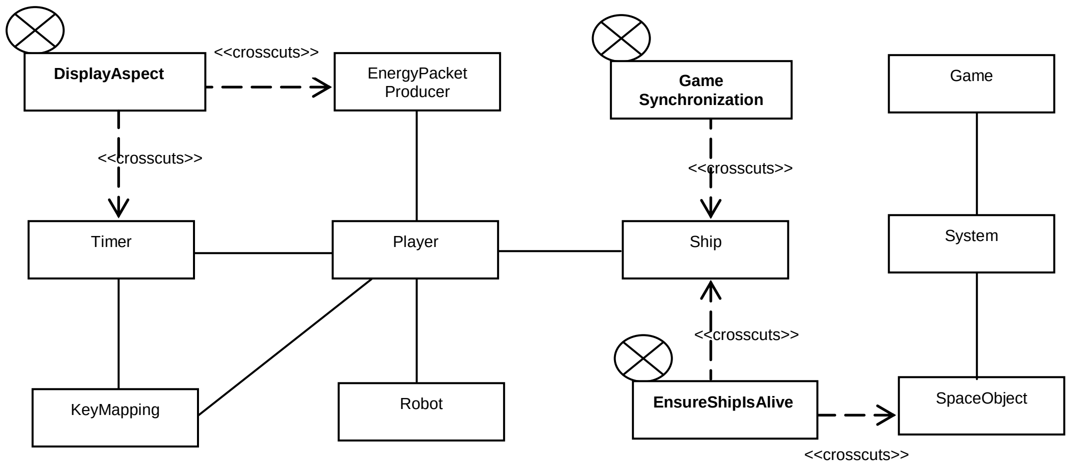


Figure 5.31: Aspect-Class Structure Diagram for *SpaceWar* game

5.3.2.2. AODL Behavioural Diagrams

The behavioural diagrams of AODL help in capturing and designing aspect-oriented elements within the behavioural models of the base program. There are two types of diagrams included, Join Point Behavioural Diagrams and Aspect-Class Dynamic Models.

5.3.2.2.1. Join Point Behavioural Diagrams

These diagrams help in locating join points within the activity diagrams of the base system. The following are the join point behavioural diagrams for ***SpaceWar***game.

Start Game

There are three join points identified in this module. The first join point is located at the completion of “run Energy Packet Producer” and “run Timer” activities and is used by *DisplayAspect*. The second join point is located at the end of “make ship” activity and is used by *GameSynchronization* aspect and the third and the last join point is located at the end of “make player” activity and is used by both *DisplayAspect* and *GameSyncrhnoization*. The Figure 5.32 shows the join point behavioural diagram that captures these join points with the flow of activities of this module.

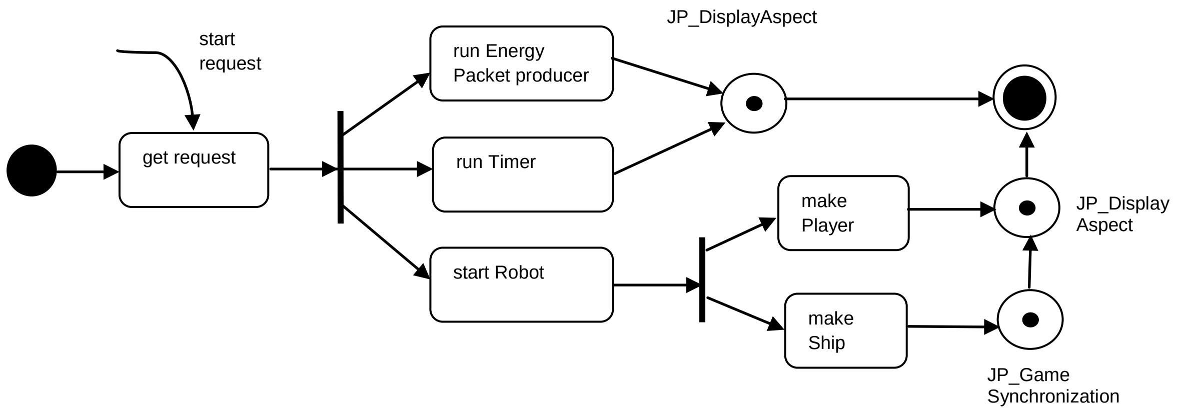


Figure 5.32: Join Point Behavioural Diagram for *Start Game*

Move Ship

The join point behavioural diagram shown in Figure 5.33 depicts a joint point within the flow of activities of Move Ship module. The join point is located around "decode the key" activity and is used by *EnsureShipIsAlive* aspect.

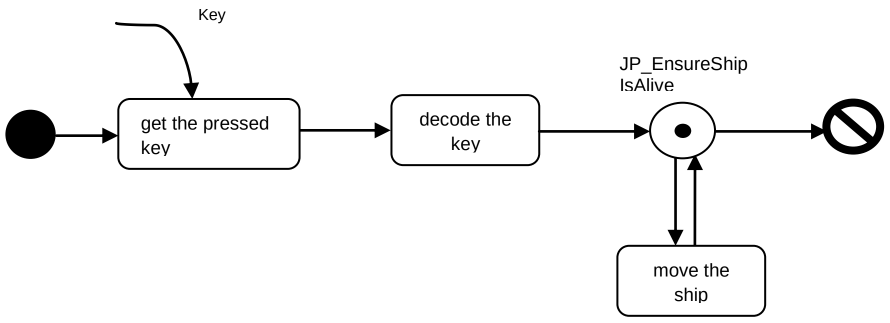


Figure 5.33: Join Point Behavioural Diagram for *Move Ship*

Fire

In this module, a join point is located around "decode the key" activity, which is used by *EnsureShipIsAlive* aspect. This join point has been shown along with the activities of the module in Figure 5.34 below.

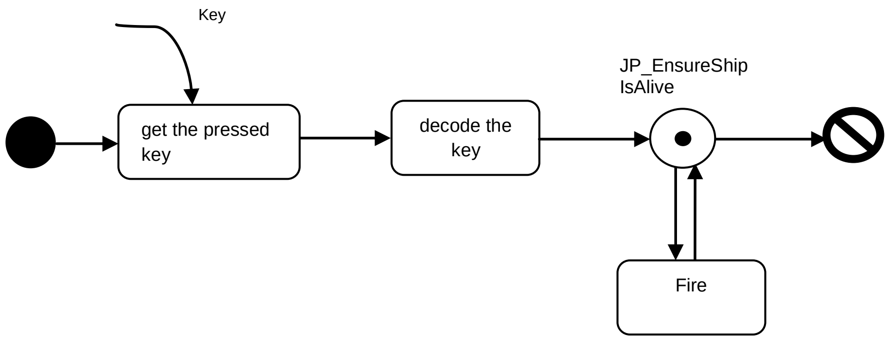


Figure 5.34: Join Point Behavioural Diagram for *Fire*

Handle Collision

In this module, there are two aspects that interact with the base program and insert their behaviour. The join point behavioural diagram shown in Figure 5.35 depicts the locations of join points of both the aspects within the flow of activities. The join point for *GameSynchronization* aspects is located at the completion point of activity "Access colliding objects" and join point for *EnsureShipIsAlive* is located around "bounce the ship" activity.

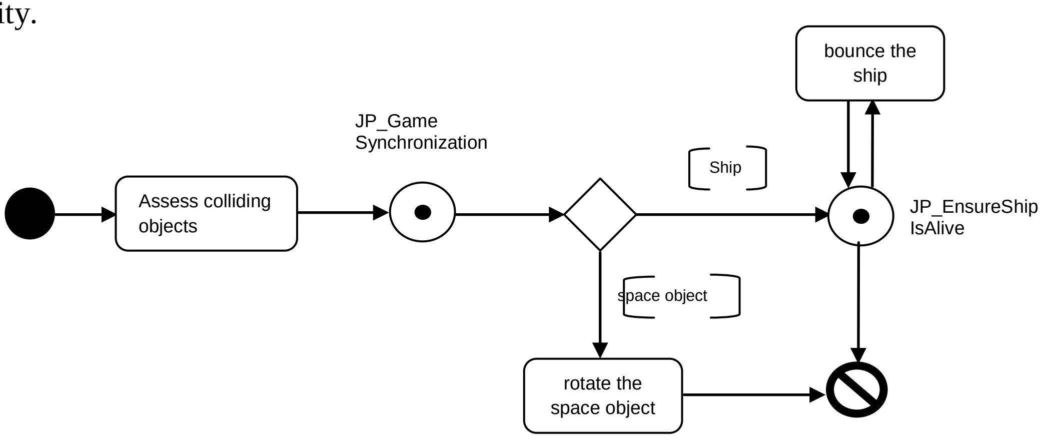


Figure 5.35: Join Point Behavioural Diagram for *Handle Collision*

Thrust

The join point behavioural diagram shown in Figure 5.36 depicts a join point, identified around "decode the key" activity in the flow of activities of thrust module. The join point is used by *EnsureShipIsAlive* aspect.

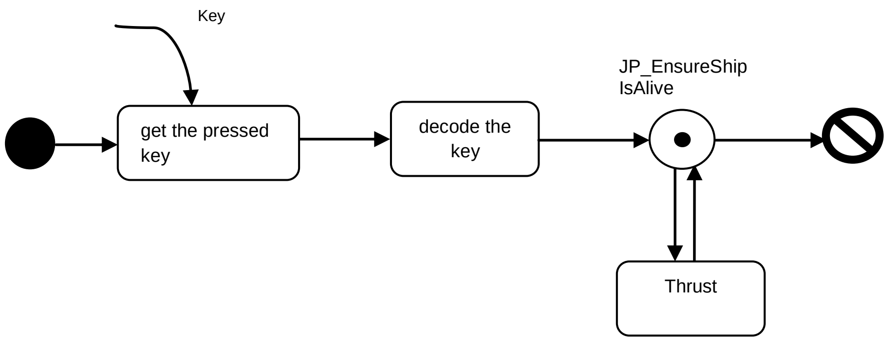


Figure 5.36: Join Point Behavioural Diagram for *Thrust*

5.3.2.2.2. Aspect-Class Dynamic Model

This model captures the weaving process of aspects with the base program dynamically. The UML communication diagrams are used to capture dynamic behaviour of objects while showing message passing at run time. These diagrams have been extended to include aspectual behavioural as well. The details about the model are provided in chapter 4.

The following sections present aspect-class dynamic models for the ***SpaceWar***game.

Start Game

The Figure 5.37 shows an aspect-class dynamic model for the start game module. The model captures weaving of *EnsureShipIsAlive*, *DisplayAspect* and Game Synchronization aspects with the base objects at run time. The game starts by the Player object calling start() methods of EnergyPacketProducer and Timer classes. Both the classes run their internal methods to create energy packets and timer for the game. At this point, the *DisplayAspect* weaves its behaviour after the internal methods of both classes complete their execution. The Player object then starts the robot by calling makeNewRobot() method of the class. At this point again, an aspect *EnsureShipIsAlive* weaves its behaviour after internal method, startRobot() of Robot class, completes its execution. The Robot class in return calls methods of Player and Ship classes to create a new player and a new ship respectively. An aspect *GameSynchrnization* weaves its behaviour after the createShip() method of Ship class at this point.

The diagram shown in Figure 5.37 captures this message passing at the dynamic level to demonstrate the weaving of all three aspects during message passing among the objects.

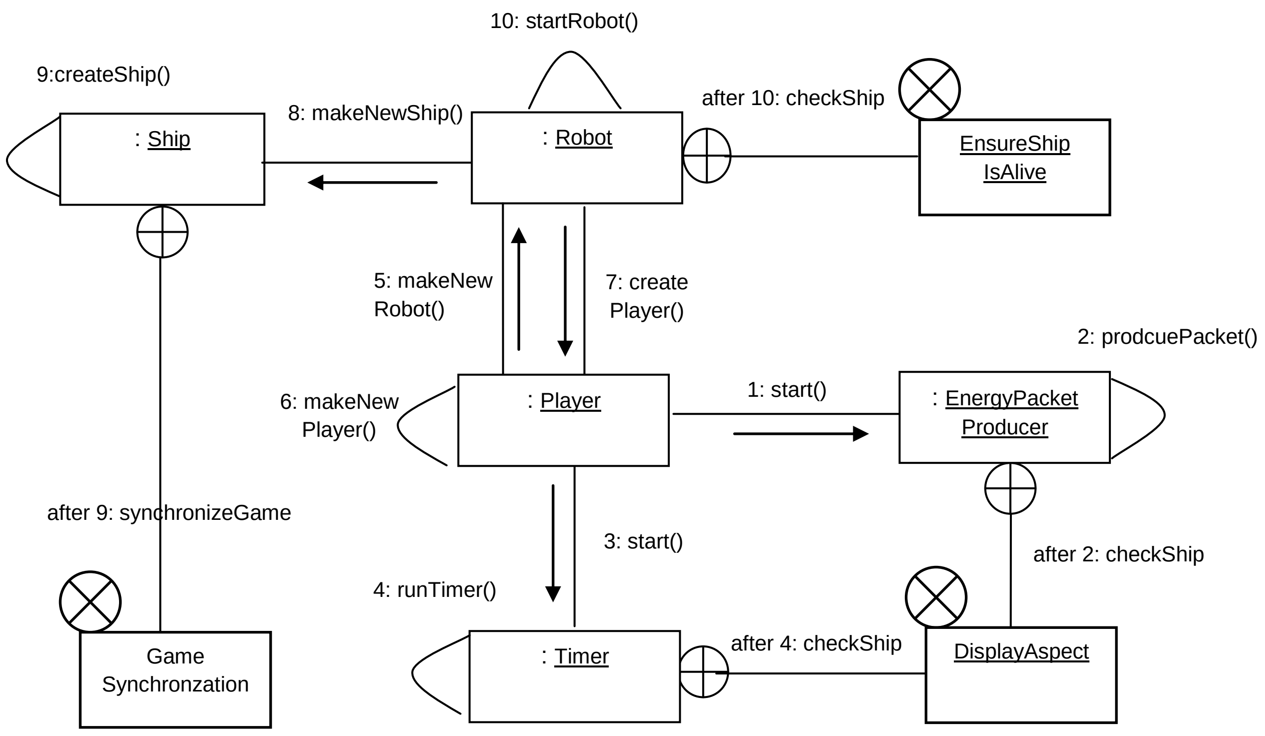


Figure 5.37: Aspect-Class Dynamic Diagram for “Start Game”

Rotate the Ship

The command to rotate the ship is started by the Player object which is mapped by KeyMapping class. The Player class, upon receiving the code of the key, calls for the rotate (string dir) method of Ship class. Figure 5.38 shows an aspect-class dynamic model which captures this message passing at the execution level. The model captures the weaving of the *EnsureShipIsAlive* aspect which superimposes its behaviour around rotate(string dir) method.

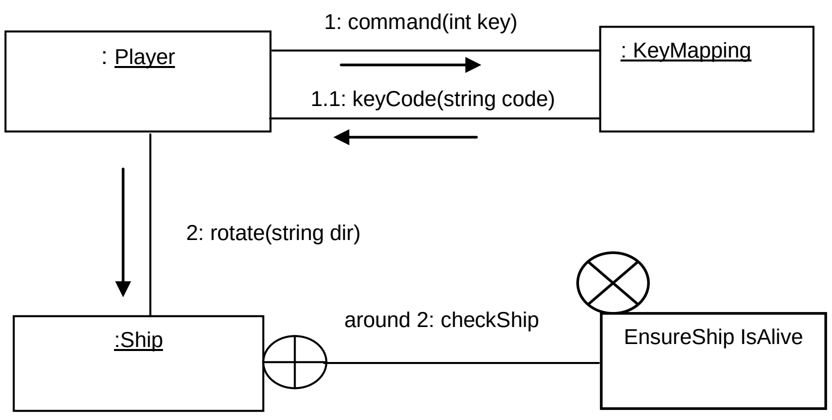


Figure 5.38: Aspect-Class Dynamic Diagram for “Rotate the Ship”

Fire

The Player object receives the command to fire from the player and sends the key to KeyMapping class to decode it. The KeyMapping class returns the code of the command back to Player. The Player object then calls fire() method of Ship class to do the firing from the ship. Figure 5.39 captures this scenario at the dynamic level in an aspect-class dynamic model. The weaving of aspect *EnsureShipIsAlive* is depicted in the model around the fire() method.

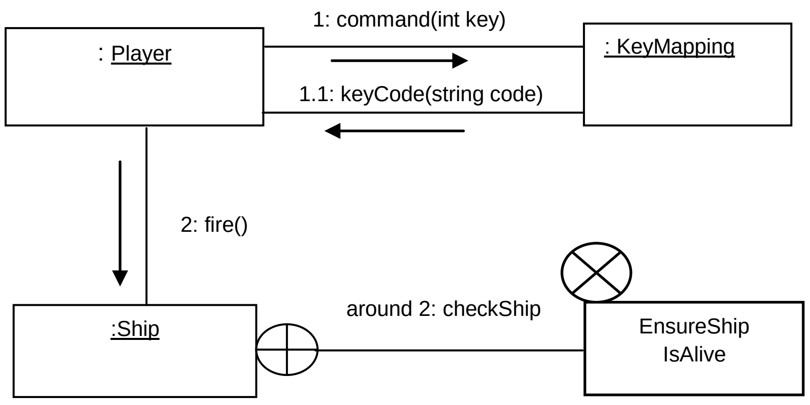


Figure 5.39: Aspect-Class Dynamic Diagram for "Fire"

Handle Collision

Figure 5.40 shows an aspect-class dynamic model for the Handle Collision module. The model captures the dynamic flow of messages between objects and demonstrates dynamic weaving of *GameSynchronization* and *EnsureShipIslAlive* aspects. The model shows that the System class calls the handleCollision() method of Game class to start the process. At this point, *GameSynchronization* aspect inserts its behaviour after the completion of handleCollision() method. The System class also calls handleCollision() method of SpaceObject class. The Game class in return calls bounce() method of Ship class which is covered by an around advice of *EnsureShipIsAlive* aspect to check whether ship remains alive during the process.

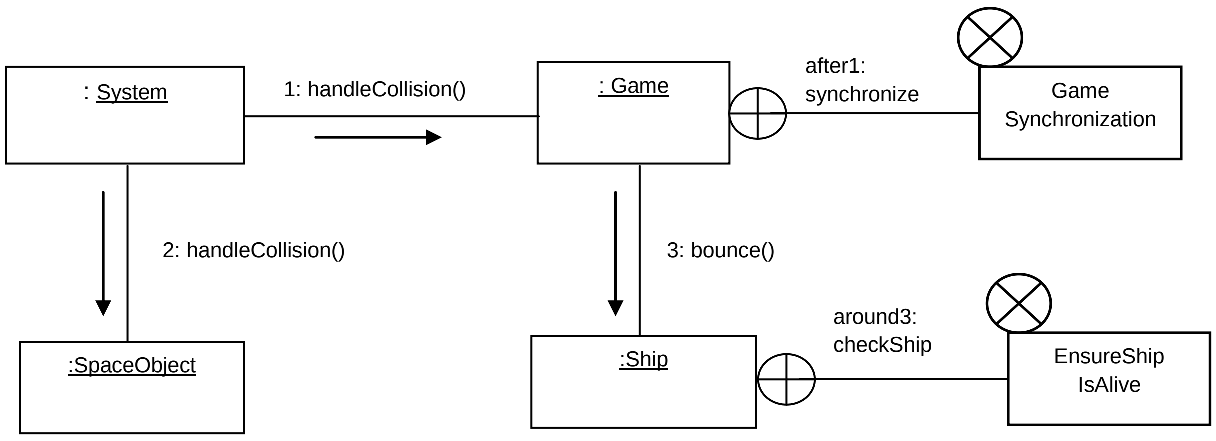


Figure 5.40: Aspect-Class Dynamic Diagram for "Handle Collision"

Thrust

In this model (shown in Figure 5.41), the dynamic flow of messages has been shown in which weaving of aspect *EnsureShipIsAlive* has been depicted around the thrust(true) method of the Ship class. The process is initiated from the Player object, which sends the pressed key to the KeyMapping class to get the code of the key. The Player object then calls thrust(true) method of the Ship class to perform thrust by the ship.

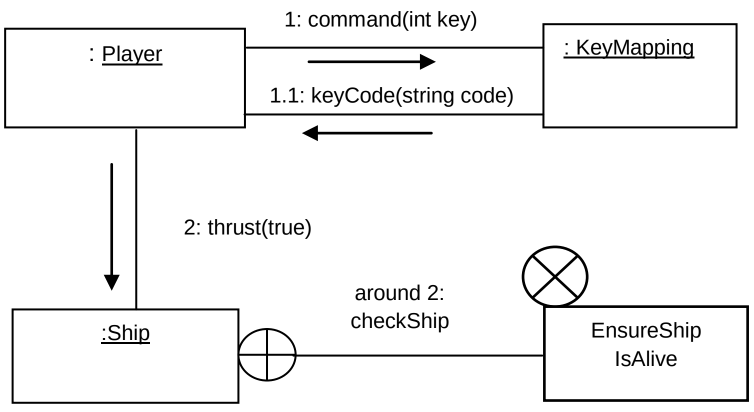


Figure 5.41: Aspect-Class Dynamic Diagram for "Thrust"

5.3.2.3. Pointcut Composition Model

A composition model for *Coordinator*, *RegistrySynchronization*, *GameSynchronization* and *RegistryProtection* aspects has been depicted in Figure 5.42. There are three other aspects of the system as well, *EnsureShipIsAlive*, *DisplayAspect* and *Debug*, but their models have not be included in this section as the purpose of this chapter is to demonstrate application of AODL on complex associations and conflicts in the system which are demonstrated in the model shown in Figure 5.42.

The figure shows that *RegistrationSychnronization* is a child aspect of *Coordinator* and it overrides synchronizationPoint pointcut of its parent aspect. Similarly *GameSyncronization* is also a child aspect of *Coordinator* aspect and it overrides the same pointcut as well. The overriding association is depicted by a stereotype <<*implements*>>, which shows that a pointcut implements an abstract pointcut of its parent aspect.

RegistrationProtection aspect has also been depicted here because it has higher precedence over *RegistrationSynchronization*. Both the aspects have an execution conflict as they run their advices after same join points, Register.register() and Register.unregister(), simultaneously. The Pointcut Composition Model resolves this conflict by introducing <<*precedence*>> stereotype that indicates which aspect will execute its advice first.

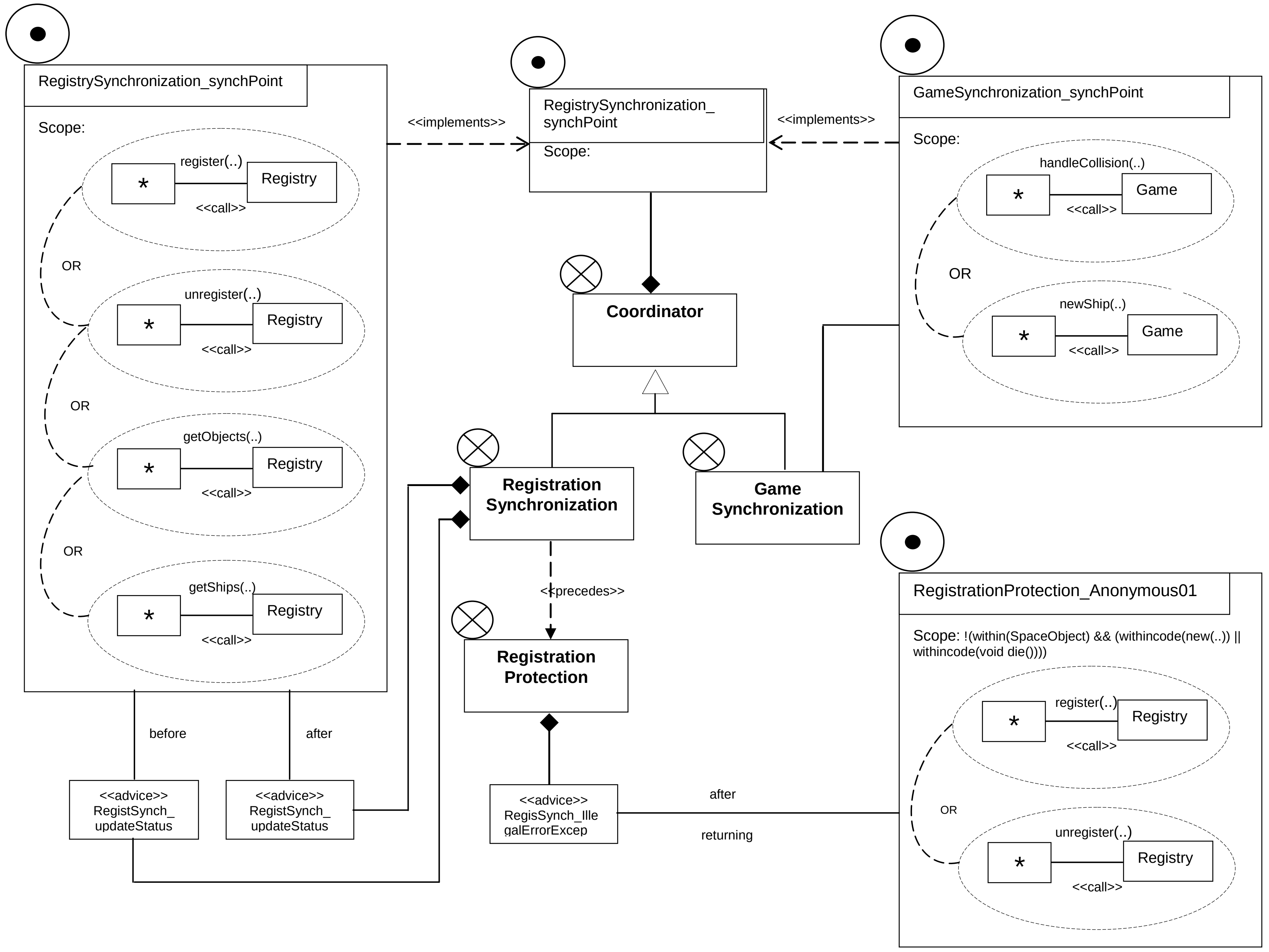

RegistrySynchronization_synchPoint
Scope:
register(..)
Registry
<<call>>
OR
unregister(..)
getObjects(..)
getShips(..)
<<implements>>
RegistrySynchronization_ synchPoint
Coordinator
Registration Synchronization
Game Synchronization
<<precedes>>
Registration Protection
GameSynchronization_synchPoint
handleCollision(..)
Game
newShip(..)
before
after
<<advice>> RegistSynch_ updateStatus
<<advice>> RegisSynch_IllegalErrorExcep
after
returning
RegistrationProtection_Anonymous01
Scope: !(within(SpaceObject) && (withincode(new(..)) || withincode(void die())))

Figure 5.42: Pointcut Composition Model of *Coordinator*, *RegistrySynchronization*, *GameSynchronization* and *RegistryProtection* aspects

5.3.2.4. Pointcut Table

Table 5.2 shows a pointcut table that contains specification of the pointcuts of all the aspects included in the designed system. The system does not have any clashes among aspects so the Precedence column is blank.

Table 5.2 - Pointcut Table for *SpaceWar* game

	<<aspect>> **GameSynchronization**		**<<aspect>>** ***EnsureShipIsAlive***	**<<aspect>>** **DisplayAspect**					**Precedence**
	<<advice>> updateStatus (before)	<<advice>> updateStatus (after)	<<advice>> checkShipStatus (around)	<<advice>> modeSelection (after)	<<advice>> displayPlayer (after)	<<advice>> displayChange (after)	<<advice>> displayChange (after)	<<advice>> displayElements (after)	
Class Game									
game()				call					
clockTick()								call	
handleCollisions(Ship s, SpaceObj so)	call	call							
handleCollision(SpaceObj so, SpaceObj soj)	call	call							
Class Ship									
helmCommandsCut(Ship)			call						
newShip()	call	call							
Class Player									
player()					call				
Class Display									
display()						call			
setSize()							call		
Pointcut Definition	call(void Game.handleCollisions(..)) || call(Ship Game.newShip(..))	call(void Game.handleCollisions(..)) || call(Ship Game.newShip(..))	Ship.helmCommandsCut(ship)	call(Game+. new(String))	call(Player+. new(..))	call(Display+.new(..)	call(void Display. setSize(..)	call(void Game.clockTick())	
Pointcut Name	**synchronizationPoint()**	**synchronizationPoint()**	**Anonymous_01()**	**Anonymous_01()**	**Anonymous_02()**	**Anonymous_03()**	**Anonymous_04()**	**Anonymous_05()**	

5.4. Discussion

The purpose of this chapter was to demonstrate the application of AODL notations and design diagrams to case studies. This chapter applies AODL techniques on two case studies which are well-known in aspect-oriented research community and contain enough complexity to address almost all types of aspect-oriented modelling issues.

The first case study, **Car Crash Crisis Management System**, is a well-known example which was specially designed to assess aspect-oriented modelling approaches. It was a standard case study for a special edition of Transactions on Aspect-Oriented Software Development VII (Shmuel and Mezini, 2010). The case study contains an adequate number of aspects and a detailed model of their interaction with the base classes. The second case study is a game, ***SpaceWar*** (AspectJ, 2012), which is part of the AspectJ sample code on the eclipse plug-in. This case study is again a well-known system which has been adopted by a number of research articles as a standard example. The complexity of the game makes it a good choice to assess the efficacy of AODL. The reason behind selecting these two case studies is to evaluate AODL in both forward and reverse engineering styles. The crisis management case study is not an implemented study so it has been adopted to design a system from the scratch using AODL techniques. The game case study, on the other hand, is an implemented system, which has been designed in AODL to demonstrate a reverse engineered design of a system.

The case studies have been successfully designed in this chapter. The application demonstrates that AODL can cover all types of design requirements of an aspect-oriented system and provides design notations and diagrams to model a comprehensive design of a complex system. Besides designing the aspect-oriented constructs, AODL also provides support to identify and resolve aspect interferences and conflicts. The pointcut table and pointcut composition models help in identifying the shared join point problems (Nagy et al., 2005) and provide a priority mechanism to set aspects' precedence during the design phase. The pointcut table also specifies aspects in a detailed manner along with interacting base classes and their features.

Findings:

The following findings have been gathered from the application of AODL in this chapter:

- AODL can be used to design a complex aspect-oriented system and can be used to address some of the aspect interferences (such as Shared Join Points) effectively.
- AODL can be used to capture the design of a legacy system using a reverse engineering technique.
- AODL can address all of the design requirements and can help in designing all kinds of aspect-oriented constructs.
- AODL offers a Pointcut Composition Model that helps in designing join point predicates of pointcuts and also helps in resolving the shared join point problem at the design level.

5.5. Chapter Summary

This chapter has presented an evaluation of AODL by its application to case studies. The chapter starts with the introduction of the purpose of the chapter and an explanation about the case studies and their selection. The case studies are explained further in the following sections with details about the systems and identified aspects and base constructs. The application of AODL to these case studies has been described in detail, with commentary on every design notation and design diagram which has been adopted in the system design. A discussion section concludes the chapter by explaining the purpose of the chapter and the findings gathered from the chapter.

Chapter 6: Qualitative Evaluation of AODL

The chapter is about a qualitative evaluation of Aspect-Oriented Design Language. The chapter starts with description of the qualitative criteria. The criteria discussedare applied to AODL and eight other selected aspect-oriented design approaches. A comparison is made among the approaches and the strengths and weaknesses of AODL are identified. The purpose of this evaluation is to judge the efficacy and maturity of AODL among contemporary design approaches.

6.1. Introduction

In this chapter, the language is assessed qualitatively by selecting a set of criteria which evaluates efficacy and performance of the language. The criteria are inspired from similar studies conducted by Wimmer et al. (2011) and Chitchyan et al., (2005) to assess the quality of an aspect-oriented design language. Each criterion included in the set of criteria evaluates the language from a specific perspective and judges its ability to effectively provide a design solution to a particular need of the system. The language is also compared with other related modelling languages using the same evaluation criteria. A comparison with the existing approaches provides an insight into AODL and reveals its strengths and weaknesses.

The rest of the chapter is structured as follows: section 6.2 explains the qualitative criteria and outlines how the approaches are evaluated using these criteria, section 6.3 explains the evaluation and provides results, section 6.4 provides the discussions on the acquired results and 6.5 concludes the chapter with a summary.

6.2. The Qualitative Evaluation Criteria

Software design consists of a set of models which are developed to specify and design constructs included in the system. A software design methodology usually contains a design process and a language (Chitchyan et al., 2005). The design process defines the set of activities and the order in which these activities are performed to develop a design model.

The language defines the design notations and design diagrams to help in producing the design model in terms of representation of the designed constructs and relationships among them. AODL includes an informal design process and a design language. The design process is in the form of guidelines about the usage of the design diagrams defined in 4.3.1. The language of AODL is explained in detail in Chapter 4, which contains design notations and design diagrams to specify, represent, design and document aspect-oriented design constructs.

To qualitatively evaluate AODL, criteria have been proposed, which can be applied to any design language that models aspect-oriented constructs. Some of the criteria have also been used by similar studies conducted by Wimmer et al. (2011) and Chitchyan et al. (2005). The selection of criteria and description of each criterion is provided below.

6.2.1. Evaluation Criteria

1. **Basic Design**

   There are four parameters identified in this criterion. The description of each parameter is provided below:

   a) **Platform and Language Dependencies:** Most of the existing aspect-oriented design languages are based on UML and AspectJ. The reason behind UML's selection is the de-facto standard status of the language for designing object-oriented systems. Since both the UML and OOP are the most familiar languages for designers as well as implementers so the choice is inevitable. AspectJ was the first technology proposed for the implementation of aspect-oriented systems which makes it the ultimate choice as a foundation technology. The criterion will assess AODL and eight other selected approaches to find out the dependency of each approach. The criterion has been inspired from the similar kind of criteria proposed by Reina et al. (2004) and Wimmer et al (2011).

   b) **Comprehensiveness:** A design language has to be comprehensive in a sense that it covers all the constructs and related elements in the design. This criterion evaluates if the language offer notations and diagrams to specify, represent and design all the aspectual elements such as aspects, pointcuts, join points, advices, inter-type declarations and composition semantics.

c) **Representation of Structural and Behavioural Crosscutting:** This criterion evaluates whether the language offers design diagrams to design both structural and behavioural crosscutting of the design constructs. A design language for aspect-oriented systems must provide a means to design structural and behavioural crosscutting along with dependencies among the constructs.

d) **Design Process:** A design process is a set of activities or processes to design the system. This criterion will evaluate if the language has a defined design process for modelling aspectual constructs step by step. The criterion is inspired from the similar type of approaches adopted by Op de beeck et al. (2006) and Wimmer et al (2011).

2. **Design Language**

   There are four parameters included in this criterion. The description about each parameter is provided below:

a) **Design Notations:** An AOD design language must offer distinct design notations to represent aspects and their constituent elements. The design notations also ought to depict the characteristics of the constructs. This criterion will evaluate if the language offers effective design notations for all the basic aspectual constructs.

b) **Design Representation (Rules, Models or Diagrams):** This criterion evaluates the language in terms of its ability to offer a set of rules, models or design diagrams to specify and design aspectual elements along with their relationships and associations. This criterion has been used by both Chitchyan et al., (2005) and Wimmer et al., (2011).

c) **Design Semantics:** The language must provide explanations of the semantics of the proposed notations. This criterion evaluates if the language provides manuals or guides explaining semantics, syntax and usage of the notations.

3. **Concern Representation**

   This criterion consists of two parameters, descriptions of which are provided below:

a) **Symmetric vs Asymmetric:** AOD languages can be distinguished in terms of representation of concerns. There are two different types of approaches that are followed by AOD languages for representing concerns. One is the symmetric approach where both crosscutting and non-crosscutting concerns are designed and the other is the asymmetric approach, where only crosscutting concerns are represented and designed.

b) **Concern representation**: this criterion will evaluate if the language provides:
    i. An Abstract Concern Model
    ii. An Aspectual model
    iii. Structural and behavioural representation of Aspects, Join Points, Pointcuts and Advices.

4. Concern Composition:

This criterion evaluates if the language offers:

a) **Aspect Composition:** Support for composition of aspects as well as aspectual elements and techniques to compose aspectual elements with each other. For instance, support for pointcut-pointcut, pointcut-advice and aspect-aspect composition.

b) **Rules to resolve conflicts:** The rules to resolve aspect interference and conflicts must also be provided by the language. This criterion has also been proposed by Blair et al. (2005) and Wimmer et al., (2011).

5. Efficacy and Maturity

There are five parameters that assess the quality of the approaches against this general criterion. The description of each parameter is provided below:

a) **Extensibility**

There are two types of extensibility:

i. **Heavy Weight Extension:** The criterion evaluates if the language supports extension of the language with the introduction of components other than aspects. For example, the UML is a language that supports heavy weight extension with the help of UML metamodels.

ii. **Light Weight Extension:** This criterion evaluates if the language offers to extend the components with attributes, properties or associations other than those mentioned in the specification of the language. UML profiling is an example of this type of extension provided by UML.

b) Traceability:

There are two types of traceability:

i. **External Traceability:** The traceability of design components (aspects) from the requirements engineering phase to the implementation phase of the development life cycle is called external traceability. This criterion will evaluate if the language provides a technique to trace aspects from any of the earlier phases to the later phases in the development life cycle.

ii. **Internal Traceability:** This type of traceability refers to the techniques to trace a component from an abstract model to a refined model.

c) Scalability: Scalability measures whether a language has the ability to handle growth. The language will be evaluated on the basis of available literature indicating implementation of small and large systems with equal expressivity.

d) Comprehensibility: This criterion helps in evaluating whether all the artefacts and semantics of the language are comprehensible and provide logical explanation about their notations and representations.

6. Tool Support:

The following parameters are included in this criterion. The criteria are inspired from Wimmer et al., (2011).

a) **Modelling Support** – The criterion measures whether the language allows models to be designed using a tool. All the design notations, design diagrams and associations included in the language ought to be supported by the tool.

b) **Composition Support –** The criterion evaluates whether the language offers complete visualization and simulation of the composition process.

c) **Code Generation:** This criterion measures whether a tool cangenerate code against a visualized model.

The following table presents a summary of the criteria used in this evaluation:

Table 6.1 - Summary of Evaluation Criteria

Basic Criterion	Extensions	Explanation
Basic Design	Platform and Language Dependencies	This criterion discusses the dependency of the design methodology on design and programming frameworks.
	Comprehensiveness	This criterion evaluates the design notations and design models provided by the design methodology.
	Representation of Structural and Behavioural Crosscutting	This criterion assesses the design methodology in terms of support for designing and modelling both structural and behavioural types of crosscutting.
	Design Process	This criterion evaluates whether the design methodology offers a design process to follow.
Design Language	Design Notations	This criterion finds out the notational support provided by the design methodology for representing the system constructs.
	Design Representation	This criterion assesses the rules, models and diagrammatic support provided by the design methodology.
	Design Semantics	This criterion finds out whether the design methodology provides explanation of design semantics of the notational constructs.
Concern Representation	Symmetric vs Asymmetric	This criterion finds out the type of the design methodology in terms of supporting concerns of the system.
	Concern Representation	This criterion evaluates the ability of the design approach in terms of providing support for representation of an aspect and its constituent key elements.
Concern Composition	Aspect Composition	This criterion evaluates whether the design approach provides strategies, rules and design models to compose aspects with each other and with base constructs.
	Rules to Resolve Conflicts	This criterion evaluates whether the design approach provides rules to resolve conflicts which arise as a result of aspect compositions
Efficacy and Maturity	Extensibility	This criteria assesses the support for both heavy-weight and light weight extensions provided by the design approach.
	Traceability	This criterion assesses the support for internal and external traceability provided by the design methodology.
	Scalability	This criterion finds out the scalability of the design approach by assessing the provided literature on the approach.
	Comprehensibility	This criterion assesses the logical explanation of the design notations provided by the design approach.
Tool Support	Modelling Support	This criterion measures whether the design approach provide

		tool support to model all the design constructs
	Composition Support	This criterion evaluates whether the language provides automated support to compose design components.
	Code Generation	This criterion assesses the ability of the tool provided by the design approach to generate code against visualized models.

6.3. Evaluation of AODL

The qualitative evaluation of AODL has been carried out by assessing the language against the criteria which are discussed above. From the literature, eight existing approaches have been selected which will also be assessed against each criterion. These approaches are AODM (Stein et al., 2002c), Theme/UML ((Baniassad and Clarke, 2004b), AOSD/UC (Jacobson and Ng, 2005), Motorola Weavr (Cottenier et al., 2007a), AAM (France et al., 2004), JAC Design Notations (Pawlak et al., 2002, 2005), Klein's Weaving Approach (Klein et al., 2006, 2007) and SUP Approach (Aldawud et al., 2001; Elrad et al., 2005). These approaches have been discussed in detail in Chapter 3. The selection of the approaches is based on the maturity and efficacy of the approaches and their similarity with AODL. A detailed description about each approach is also provided in Chapter 3. The assessment of AODL along with these approaches will provide a way to analyse the quality of the language and its effectiveness against contemporary approaches.

The criteria discussed above are now applied to AODL and selected approaches below:

6.5.2. Basic Design

a) Platform and Language Dependencies:

Existing AO design approaches have been designed targeting particular aspect-oriented programming languages. Most of the approaches were initially dependent on AspectJ, the first programming language for aspect-oriented development. The choice was obvious as AspectJ was designed by the very team that developed aspect-oriented programming in the first place (Kiczales et al., 1997). For modelling aspect-oriented constructs and aspect-class compositions, UML was widely adopted because of its popularity and extension mechanisms which can be adopted for introducing new design components, elements, attributes and properties.

In Table 6.2, we evaluate AODL along with other selected AOD approaches in this criterion to find out the platform and language dependencies.

Table 6.2 - Comparison of all approaches based on Platform and Language Dependencies criterion

Approach	Platform and Language Dependencies
AODL	AODL is dependent on AspectJ programming technology. It offers design solutions for modelling constructs included in AspectJ. The language also relies on UML 2.4.1 and extends some of its notations and diagrams to develop new artefacts.
AODM	AODM was initially developed to provide design notations for AspectJ. It uses semantics of AspectJ and provides notations and diagrams to design these semantics. Later, it was evolved to be a generic approach by providing support for other technologies as well, such as composition filters and adaptive programming.
Theme/UML	The approach was developed initially to support subject-oriented programming paradigm. Later, it was evolved to accommodate composition filters, AspectJ and HyperJ technologies.
AOSD/UC	The approach is influenced by AspectJ and HyperJ technologies. The notations and semantics of both the technologies have been mentioned in the literature and used in the development of proposed techniques.
Motorola Weavr Approach	A general-purpose modelling approach has been proposed, which is based on Motorola Weaver technology.
AAM	The approach has been developed as a platform-independent approach. The notations and diagrams have been borrowed from UML 2.0.
JAC Design Notations	JAC design notations were proposed for JAC design framework so they depend heavily on the framework's notations and semantics.
Klein's Weaving Approach	The approach is based on a scenario-based language called message sequence charts.
SUP Approach	The approach is not based on any particular platform and is also independent of implementation technology.

We can observe that most of the approaches are either dependent on AspectJ or some other programming languages such as HyperJ and Motorola Weavr. There are only two approaches, AAM and SUP which are language-independent. Regarding the modelling dependability, most of the approaches depend heavily on UML. They either use the notations and diagrams of the language or extend them to propose new design constructs.

AODL is also based on AspectJ programming language. It provides modelling solutions for notations and design constructs of AspectJ using newly proposed constructs which have been specially designed for each construct, including an aspect and its constituent elements. The proposed notations and diagrams are developed along with UML design models thus providing a unified design approach where both aspects and objects can be designed in parallel.

b) Comprehensiveness:

This is an important criterion because most of the design languages are incomplete because they do not provide design solutions for each aspect-oriented construct. An AOD language or design approach should ideally provide support for designing every construct and their composition with the base program. Table 6.3 provides an assessment of AODL along with the selected approaches regarding the comprehensiveness of the design techniques.

Table 6.3 - Comparison of all approaches based on Comprehensiveness criterion

Approach	**Comprehensiveness**
AODL	AODL is comprehensive as far as availability of design solutions for majority of the design constructs is concerned. It provides distinct design notations for aspects join points, advices, pointcuts and inter-type declarations. The language also proposes design diagrams to specify, design and compose these constructs. The strategies and design solutions proposed by the language for composition of constructs also address composition-related conflicts and aspect interference. It proposes design solutions to reduce aspect interference and resolve conflicts. A design process has also been proposed along with the language to formalize the design activity. The only deficiency is lack of tool-support which is part of the future research.
AODM	The approach does not provide design notations or diagrams to represent some constituent elements of an aspect such as pointcuts and advices, although representation for join points and introductions has been supported. The approach lacks a design process.

	The approach lacks scalability as no examples have been provided to suggest the opposite.
Theme/UML	The approach lacks design representation for aspectual elements such as join points, pointcuts, advices and inter-type declarations.
AOSD/UC	The approach provides support to design aspects comprehensively but aspectual elements are not separately represented and designed.
Motorola Weavr Approach	The approach is quite comprehensive regarding providing support for representation of all constructs; however, the only problem is that no design process or guidelines are provided to support procedural way of designing.
AAM	The approach was primarily proposed for architectural solutions for aspect-oriented systems. It lacks detailed design support for concern representation and composition. The composition strategies proposed by the approach focus on architectural composition of concerns only.
JAC Design Notations	The approach has not matured yet. There is no support for designing aspectual elements. The approach only uses class diagrams to develop structural diagrams. There is no support for developing detailed design models. It does not offer a design process either.
Klein's Weaving Approach	The approach only offers modelling solutions for composition of aspects. There is no method proposed to design structural or behavioural crosscuttings and there are no design process or guide lines available to formalize the modelling.
SUP Approach	The approach does not provide higher-level abstraction and has not been implemented on complex systems to suggest scalability.

Most of the approaches discussed in Table 6.3 lack a design process. There are no guidelines provided by these approaches on how to carry out modelling of the constructs in given order. The examples of these approaches are AODM, Motorola Weavr approach, JAC Design Notations and Klein's Weaving Approach. On the contrary, AODL provides a design process which defines the order of development of each design construct from specification to composition.

The second problem is lack of design solutions for all aspect-oriented constructs and elements. This deficiency has been observed in almost all of the discussed approaches except AODL. There is a distinct design notation for each construct in

AODL and a separate design diagram has been proposed to model these constructs as well.

c) Representation of Structural and Behavioural Crosscutting

The aspects crosscut structurally when they introduce inter-type declarations (previously known as introductions). This type of crosscutting adds new behaviour to the base constructs and has to be designed properly in the design model. Similarly, when aspects are composed with base constructs dynamically, they insert new behaviour into the base code. This type of crosscutting is known as behavioural crosscutting and it also has to be designed in behavioural models during the aspect-oriented design process.

This criterion evaluates AODL and other selected approaches in Table 6.4 to find out if both structural and behavioural crosscuttings are supported by each approach.

Table 6.4 - Comparison of all approaches based on Structural andBehavioural Crosscutting Criterion

Approach	Structural Crosscutting	Behavioural Crosscutting
AODL	Aspect Design Diagrams are used to define structural crosscutting among constructs.	Communication diagrams and activity diagrams are used to depict behavioural representation of crosscutting concerns.
AODM	Class diagrams and sequence diagrams are used to capture structural crosscutting.	Collaboration diagrams are used to model behavioural crosscutting of the system constructs.
Theme/UML	The approach uses package diagrams and class diagrams for representing the structural crosscutting in the system.	Sequence diagrams are used for depicting behavioural crosscutting.
AOSD/UC	In the design phase, component diagrams are transformed into class diagrams to depict structural crosscutting among the models.	For representing behavioural crosscutting, sequence diagrams have been utilized.

Motorola Weavr Approach	Composite structure diagrams and class diagrams are used to model structural crosscutting.	Transition-oriented state machines are used to depict behavioural characteristics of the concerns.
AAM	The class diagram templates are utilized to design structural representation.	Communication diagram templates are used to depict behavioural representation of concerns.
JAC Design Notations	The structural representation is made using class diagrams.	No support for behavioural representation is available.
Klein's Weaving Approach	There is no method provided for structural representation of concerns.	The approach uses sequence diagrams for behavioural representation of concern compositions.
SUP Approach	The class diagrams are used to show the structural dependencies.	State machines, use cases and collaboration diagrams are utilized to depict behavioural characteristics of the concerns

Almost every approach has provided solutions for designing structural and behavioural crosscutting as is evident from Table 6.4 except JAC Design Notation approach and Klein's Weaving Approach. The former has not provides any method to design behavioural crosscutting while latter has not provided any design solutions for structural crosscutting.

AODL proposes design diagrams for both types of crosscutting. An Aspect Design Diagram has been proposed to design structural representation of an aspect. The model also provides support for depicting the structural crosscutting of aspects with the base classes (for details see section 4.3.2.3.2).

For behavioural crosscutting, activity diagrams and communication diagrams are adapted to design join point behavioural models and dynamic models of composition (for details see sections 4.3.2.1.2 and 4.3.2.3.3).

d) Design Process:

The design process of an AOD approach defines a set of rules, order of designing models and guidelines about the usage of the diagrams. A design process supports step-wise designing of aspectual constructs which ultimately helps in addressing issues at different levels of modelling.

The following table (Table 6.5) discusses the design processes proposed by AODL and other selected AOD approaches.

Table 6.5 - Comparison of all approaches based on Design Process criterion

Approach	Design Process
AODL	A three phase design process is proposed. Join points are identified and modelled in the first phase. Aspects, pointcuts and inter-type declaration are specified and designed in the second phase. The aspect composition with base construct is modelled in the third phase (For details see section 4.3).
AODM	There is no specified design process provided by AODM. There are no guidelines available to establish the order of design diagrams.
Theme/UML	The approach provides a design process that helps in designing the concerns from the requirement analysis phase to the implementation phase.
AOSD/UC	AOSD/UC follows a design process which separates concerns from the analysis phase down to implementation in the form of use case slices.
Motorola Weavr Approach	No design process has been proposed yet.
AAM	There is no formal design process provided by the approach but some suggestions have been made in the publications to carry out the design process in a certain way (France et al., 2004).
JAC Design Notations	There is no defined design process proposed by the approach.
Klein's Weaving Approach	There is no formal design process provided by the approach.
SUP Approach	A design process is provided for the approach which comprises of step-wise activities to design system modules.

AODL along with three other approaches provide a design process. The rest of the approaches have not proposed a design process or guidelines about the modelling of constructs. AODL provides a Diagrammatic Model which proposes a three phase development of design models. Each phase designs a separate design construct and provides an input to the next phase until all the design constructs are composed together in the final phase.

Summary

Table 6.6 summarizes the Basic Design criterion. The table indicates the weaknesses and strengths of each approach against each criterion.

Table 6.6 - Summary of evaluation of the approaches against Basic Design criteria

	Basic Design				
	Platform & Language Dependency	Comprehensiveness	Representation		Design Process
			Structural Crosscutting	Behavioural Crosscutting	
AODL	AspectJ, UML	Middle-High	✓	✓	✓
AODM	AspectJ, UML	Middle-High	✓	✓	✕
Theme/UML	UML	High	✓	✓	✓
AOSD/UC	UML	Middle-High	✓	✓	✓
Motorola Weavr Approach	Motorola Weaver	Middle-High	✓	✓	✕
AAM	UML	Middle-High	✓	✓	✕
JAC Design Notations	UML	Middle	✓	✕	✕
Klein's Weaving Approach	MSC	Low	✕	✓	✕
SUP Approach	Independent	Middle	✓	✓	✓

Legend:✓ = Supported, ✕ = Not Supported, **Rating**: Low, Middle, Middle-High, High

Just like other approaches, AODL depends heavily on AspectJ and UML. AODL is categorized as High as far as comprehensiveness of the approach is concerned. Only Theme/UML is the other approach which provides representation and composition supports for all the aspectual constructs (these criteria are discussed in the following sections). Though only two approaches, JAC Design Notations and Klein's weaving approach, do not support both structural and behavioural crosscutting, AODL is among all those approaches which provide design solutions for representing both types of crosscuttings. Regarding the availability of a design process, AODL is one of the few approaches that provide a well-defined design process.

Findings:

The following findings about AODL have been established by this criterion:

Strengths of AODL:

- AODL provides notational representation for all the design constructs of aspect-oriented design model including aspects, pointcuts, join points, advices and inter-type declarations.
- AODL provides design diagrams to model each design construct and to model the composition of each construct with each other and with base constructs.
- AODL supports modelling of both structural and behavioural crosscutting of aspects.
- AODL proposes a design process that defines the design activity of aspects by adopting a step-wise development of each aspectual construct.

Weaknesses of AODL:

- AODL is dependent on AspectJ notations and semantics, which hampers its usage for other programming languages such as composition filters, Adaptive Programming and Spring AOP.
- AODL has extended UML diagrams to develop new diagrams, which again suggests the dependability of the language on a modelling language. This adaptation is intentional though because AODL has primarily been developed to unify designing of aspects and objects together in one design framework. The extended forms of UML diagrams provide comprehensibility for the designer to work with both constructs using similar type of design standards.

6.5.3. Design Language

An AOD design language must provide design notations to represent an aspect and its constituent elements. The language must also define rules, strategies and diagrammatic representation for all the design elements. Each design notation and diagram must be described semantically. All these three factors are part of this criterion.

An evaluation of AODL and the eight other selected approaches against each criterion is provided below.

a) Design Notations:

A design notation is a graphical representation of structural or behaviour characteristics of a design construct or a relationship. An AOD language is expected to provide graphical notations to visualize features and relationships of a construct. This criterion evaluates AODL and other selected approaches to find out if they have provided design notations for visual representation of design models.

Table 6.7 - Comparison of the approaches based on Design Notations criterion

Approach	Design Notations
AODL	AODL proposes new design notations for specifying, representing and designing an aspect and its constituent elements. There is a distinct design notation proposed for each construct, which reflects the characteristics and nature of the construct.
AODM	Design notations have been proposed as an extension to UML notations.
Theme/UML	New design notations have been introduced. For example, notation *Theme* is used to represent an aspect which is composed with base constructs through structural diagrams.
AOSD/UC	Use cases are used as notations for concerns.
Motorola Weavr Approach	UML notations have been extended.
AAM	UML notations are adopted and extended. Notations of Role-Based Metamodelling Language (RBML) have also been adapted.
JAC Design Notations	UML notations have been extended.

Klein's Weaving Approach	Design notations of UML's sequence diagrams have been adopted.
SUP Approach	The approach utilizes notations of UML 1.x.

As evident in Table 6.7, all the approaches compared including AODL provide design notations for aspects. Most of the approaches have either adopted or extended UML notations. AODL, on the other hand, has proposed its own notations which have been developed based on the characteristics and features of the design constructs. Except AODL and AODM, there is no other approach that provides design notations for aspectual elements. The elements such as pointcuts, advices and inter-type declarations are usually represented statically within the notation of an aspect. AODL, on the contrary, proposes new notations for pointcuts, advices and inter-type declarations.

b) Design Representation(Rules, Models or Diagrams)

The design notations are represented in visual models containing diagrams to depict their characteristics and relationships with each other and with the base constructs. The design approach ought to offer design diagrams for modelling structural and behavioural features of the design constructs.

This criterion evaluates AODL and other selected approaches to find out whether design representation has been supported. Table 6.8 provides this analysis.

Table 6.8 - Comparison of the approaches based on Design Representation criterion

Approach	Design Representation
AODL	AODL proposes new design notations to represent and design structural and behavioural features of an aspect and its constituent elements. Some new diagrams, such as Aspect Design Diagrams and Pointcut-Advice Diagrams, have been introduced, which reflect the structural characteristics of an aspect and a pointcut respectively. Some of the design diagrams extend UML diagrams and introduce new design artefacts, associations and features to reflect the distinctive nature of the designing construct or relationship.
AODM	The approach supports representation of join points with sequence diagrams and aspect representation with an extended version of class diagram. The composition has been supported with use case diagrams and collaboration

	diagrams.
Theme/UML	Theme/UML offers structural modelling with package and class diagrams and behavioural modelling with the help of sequence diagrams.
AOSD/UC	The approach utilizes the component diagram and class diagram of UML for structural representation and sequence diagram for behavioural characteristics.
Motorola Weavr Approach	Class diagrams and composite diagrams are used for structural modelling. State machines and sequence diagrams are used for behavioural modelling. The approach also utilizes sequence diagrams for generating test cases.
AAM	The approach utilizes many UML diagrams, such as package diagram templates, class templates, collaboration templates and sequence diagrams.
JAC Design Notations	The approach uses class diagrams for structural modelling.
Klein's Weaving Approach	Only the sequence diagram is used which is utilized for behavioural representation of composition of aspects.
SUP Approach	Class diagrams have been extensively used to model the structural representation of concerns. The models are then refined gradually to be represented in state charts, use cases, state machines and collaboration diagrams.

Almost all the approaches discussed in Table 6.8 provide diagrammatic support for representation of structural and behavioural characteristics of design constructs. The Klein's Weaving Approach is the only technique that does not propose any method to represent structure of an aspect diagrammatically, nor does it support behavioural representation of the aspect. Another approach, JAC Design Notations, is the only technique that does not provide diagrams to model the behavioural properties and relationships of the design constructs.

AODLsuggests new design diagrams for both structural and behavioural representation of an aspect and proposes diagrams for aspectual elements such as pointcuts and advices. The relationship between a pointcut and an advice has been captured in a diagram for the first time in AODL. The diagram is called Pointcut-Advice Diagram and it represents structural properties of both the elements and depicts their relationships diagrammatically (for details see section 4.3.2.2.3).

c) Design Semantics

A design approach proposing new design constructs, notations and diagrams has to semantically describe each artefact. The semantics of the design notations does not only provide explanation of the construct but also helps it to be extended. This criterion will evaluate AODL along with other selected approaches to find out whether such a document has been provided by the approach. Table 6.9 provides a comparison of all the approaches based on this criterion.

Table 6.9 - Comparison of the approaches based on Design Semantics criterion

Approach	**Design Semantics**
AODL	AODL explains informal semantics of each design notation. A formal set of semantics is yet to be formalized.
AODM	There is no formal manual or a document available describing semantics of the proposed notations. In one of the papers (Stein et al., 2002a), however, the purpose and motivations behind the selection of a certain notation have been described.
Theme/UML	The proposed notations are well-described semantically in available literature.
AOSD/UC	The semantics of each design notation are explained thoroughly in the available literature.
Motorola Weavr Approach	The semantics of each design notation are well-explained in the available literature.
AAM	The notations have been semantically described in the available publications.
JAC Design Notations	The notations are semantically described in the publications related to the approach.
Klein's Weaving Approach	There are no manuals or documents available describing semantics of the proposed notations.
SUP Approach	The notations are semantically described in the available literature.

Almost all the approaches provide documents explaining semantics of the design construct and design diagrams except for Motorola Weavr Approach and Klein's Weaving Approach. AODL also provides explanation about semantics of each design notation.

Summary

A summary of the comparison made against each criterion has been presented in Table 6.10. The table provides comparison of AODL against each criterion as well as each approach.

Table 6.10 - Summary of evaluation of the approaches against Design Language/Approach

	Design Language/Approach		
	Design Notations	Design Representation	Design Semantics
AODL	N + E(UML)	N + E(UML)	✕
AODM	E(UML)	A(UML)	✓
Theme/UML	N + E(UML)	N + E(UML)	✓
AOSD/UC	E(UML)	A(UML)	✓
Motorola Weavr Approach	E(UML)	A(UML)	✓
AAM	E(UML), A(RBML)	A(UML)	✓
JAC Design Notations	E(UML)	A(UML)	✓
Klein's Weaving Approach	A(UML)	A(UML)	✕
SUP Approach	A(UML)	A(UML)	✓

Legend:✓ = Supported, ✕ = Not Supported, **N** = New, **E(x)** = extension of x, **A(x)** = Adaptation of x

The summary of the comparison shows that AODL is the only approach besides Theme/UML that provides new notations and new design diagrams. Both the approaches also provide some notations extended from UML notations. Aspects are non-object-oriented constructs in nature and thus cannot be represented with UML's object-oriented notations (Iqbal and Allen, 2009). That is the reason that AODL proposes new notations, which reflect the nature and characteristics of aspectual constructs.

Only two approaches, Motorola Weavr Approach and Klein's Weaving Approach, have not provided design semantics. Other than them, all the approaches including AODL explain each design construct introduced in the approach semantically.

Findings:

On the basis of the comparison, the following strengths and weaknesses of AODL have been identified based on these criteria.

Strengths:

- AODL introduces new design constructs for aspects and their constituent elements.
- The notations are accompanied by design diagrams that capture structural and behavioural features of design constructs and their relationships with each other as well as with base constructs.
- AODL provides comprehensive explanation of the design semantics of each proposed notation.

Weaknesses

There are no weaknesses found in AODL against these criteria.

6.5.4. Concern Representation

The primary objective of an AOD language is the representation of a concern. The representation can be made in a symmetric way when all the concerns are equally represented and in an asymmetric way when only the crosscutting concerns are represented. The criteria in this section evaluate all the selected approaches and AODL to find out whether they are asymmetric or symmetric approaches. The criterion also evaluates the support for representation provided by the approaches for an abstract design model, an aspect design model and structural and behavioural models for aspectual elements.

a) Symmetric vs Asymmetric:

As described above, asymmetric approaches propose design techniques to model only crosscutting concerns and symmetric approaches provide design support for all types of concerns. This section does not discuss which type of approach is better; rather it categorizes all the selected approaches in these two types.

Table 6.11 - Comparison of the approaches based on Symmetric vs Asymmetric criterion

Approach	**Symmetric vs Asymmetric**
AODL	AODL is an asymmetric approach. The technique designs only crosscutting concerns whereas non-crosscutting concerns are modelled using UML. The approach, however, provides a unified design framework where aspects and objects are designed in parallel thus providing a better representation of interaction among them.
AODM	AODM is an asymmetric design approach. It only provides design techniques for crosscutting concerns and does not address non crosscutting concerns.
Theme/UML	Theme/UML is a symmetric design approach. It allows all types of concerns, whether they are crosscutting or non-crosscutting, to be modelled using the same approach.
AOSD/UC	AOSD/UC can be categorized as a symmetrical approach which provides support for both crosscutting and non-crosscutting concerns. Concerns are implemented as use case slices (Jacobson and Ng., 2005)
Motorola Weavr Approach	The approach is asymmetric which only provides techniques to model crosscutting concerns (aspects).
AAM	AAM is an asymmetric approach which only provides architectural design solutions for crosscutting concerns.
JAC Design Notations	JAC Design notations approach is an asymmetric approach which only provides support for modelling crosscutting concerns.
Klein’s Weaving Approach	The approach is an asymmetric approach which only addresses weaving of crosscutting concerns.
SUP Approach	The approach may be considered as a symmetric approach. It models both base and aspectual concerns in class diagrams for structural crosscutting and then the models are refined into state charts and collaboration diagrams.

As can be seen in Table 6.11, there are three approaches which are symmetric, Theme/UML, AOSD/UC and SUP Approach. Other than these, all approaches along with AODL are asymmetric approaches. The Klein’s Weaving Approach is although an asymmetric approach but it does not provide full support for representation of concerns in the structural level. The approach only provides technique to weave crosscutting concerns with base concerns.

b) Concern representation:

This criterion evaluates the representation techniques provided by all approaches. There could be three types of basic representations of a concern:

i. An Abstract Concern Model

ii. An Aspectual model

iii. Structural and behavioural representation of Aspects, Join Points, Pointcuts and Advices.

A good AOD language would support all three types of representations as these representations ensure a comprehensive design and aid in the composition process of concerns.

Table 6.12 - Comparison of the approaches based on Concern Representation criterion

Approach	**Concern representation**		
	Abstract Concern Model	**Aspectual Model**	**Aspectual Elements' Representation**
AODL	Aspect Design Diagram provides an abstract concern model for aspects. Similarly, Aspect-Class Dynamic Model provides an abstract composition model for the weaving process.	Aspect Design Diagram represents the structural features of an aspect.	Each aspectual element is represented and designed separately with a distinctive design notation and a diagram. Join Points are identified in a Join Point Identification model and are represented in a behavioural model. Pointcuts are structurally represented in Pointcut-Advice Diagram and then are refined into Pointcut-Composition Model where each pointcut is represented in a behavioural model. Intertype-declarations are also represented structurally in Aspect Design

			Diagram.
AODM	Abstract model has not been proposed.	An Aspect Design Diagram is proposed that presents a structural model of an aspect.	Aspects join points and introductions are represented with the help of design diagrams but there is no diagrammatic modelling support for advices and pointcuts.
Theme/UML	Abstract models have not been proposed.	An aspect is represented in the form of a Theme. A design diagram is proposed to show structural and behavioural characteristics of a theme.	Aspects are depicted using theme model but there is no diagrammatic representation of aspectual elements. They are rather represented statically in an aspect container.
AOSD/UC	Abstract model for concerns have been depicted using component diagrams	Aspectual models are represented in class diagrams	There are no distinct notations for aspectual elements provided by the approach.
Motorola Weavr Approach	Abstract models have been proposed which are later refined into detailed models using state machines.	An aspectual model has been proposed which is based on UML's package and class diagram.	Pointcuts are represented along with advices, though there are no separate notations for the aspectual elements.
AAM	High level model views provide high abstraction of concerns.	Aspectual models are designed using class diagram templates.	There are no separate design constructs to represent aspectual elements such as join points, pointcuts or advices.
JAC Design Notations	High level models of aspects are supported.	Aspectual models are developed using class diagrams.	There is no support proposed by the approach for modelling aspectual elements.
Klein's Weaving Approach	No support provided.	No support provided.	No support provided.
SUP Approach	Abstract concern	An aspectual model is	No separate representation of

	models are not proposed.	designed using class diagrams.	aspectual elements is supported.

As shown in Table 6.12 AODL is the only design approach that provides support for representing aspects and their key elements with the help of design notations. All the rest of approaches either support one model of representation or two and they all lack in providing support for representation of aspectual elements such as pointcuts, advices and join points. There are some approaches such as AODM and Theme/UML that does provide a diagrammatic representation of pointcuts but no design notation has been provided for any of the aspectual elements by both approaches. The Klein's Weaving Approach is the only approach which does not satisfy any category as this approach only designs the weaving process of the concerns.

Summary

AODL provides support for representation of all concerns. It is an asymmetric approach like many other approaches but it does not lack in representing key elements of aspect-oriented design as many of other approaches do. Table 6.13. Indicates that the majority of the approaches either provide no support altogether for representing aspectual elements or they provide only partial support.

Table 6.13 - Summary of evaluation of the approaches against Concern Representation criteria

	Concern Representation			
	Symmetric vs Asymmetric	Concern Representation		
		Abstract Concern Model	Aspectual Model	Aspectual Element Representation
AODL	asymmetric	✓	✓	✓
AODM	asymmetric	✕	✓	~
Theme/UML	symmetric	✕	✓	~

AOSD/UC	symmetric	✓	✓	✗
Motorola Weavr Approach	asymmetric	✓	✓	~
AAM	asymmetric	✓	✓	~
JAC Design Notations	asymmetric	✓	✓	~
Klein’s Weaving Approach	asymmetric	✗	✗	✗
SUP Approach	symmetric	✗	✓	~

Legend:✓ = Supported, ✗ = Not Supported, ~ = Partially Supported

The table also reveals that there are two approaches, Theme/UML and SUP Approach, which do not provide an Abstract Design Model for concerns. AODL and majority of the approaches do provide support for an abstract model and an aspectual model.

Findings:

The following strengths and weaknesses of AODL have been identified in these criteria:

Strengths:

- AODL provides complete representational support with the help of design notations and diagrams for aspects and their key elements.
- AODL also offers an Abstract Design Model to design a high level design structure of an aspect which can then be transformed into more detailed design models.

Weaknesses:

There is no weakness of AODL identified in these criteria.

6.5.5. Concern Composition

The crosscutting concerns are composed with base concerns after being modelled in the design phase. The aspect composition is based on the join points defined in a pointcut in the aspect. The definition of a join point indicates the base constructs to be composed with the aspect and the location where this composition will take place in the base system. The concern composition can involve aspect-aspect composition as well besides aspect-base composition. An aspect may use or refer to a pointcut defined in another aspect. The composition model designs this relationship and depicts the composition based on pointcut-pointcut interaction.

There are two sub criterions included in this general criteria, aspect composition and rules to resolve conflicts. The aspect composition criterion is further refined into two sub criterions, aspect composition that evaluates if an approach provides design support for aspect-base composition and aspect-aspect composition, and inner-aspect composition, which evaluates whether pointcut-pointcut and pointcut-base composition is supported.

a) Aspect Composition:

Aspect composition is related with integration of aspects with each other and with base constructs. The composition strategy generally includes design notations, diagrams, directives for composition and rules to avoid and resolve conflicts. In this criterion, we will evaluate AODL and other selected approaches against two sub criterions of aspect composition. In the first criterion, aspect-class and aspect-aspect composition will be assessed and in the second criterion, pointcut-advice and pointcut-pointcut composition will be evaluated. Table 6.14 given below provides the evaluation.

Table 6.14 - Comparison of the approaches based on Aspect Composition criterion

Approach	**Aspect Compositions**	**Intra-Aspect Compositions**
AODL	Aspects are composed with each other and with base constructs with the help of Aspect Composition Models. Structural composition is depicted by Aspect-Class Structural Model and behavioural composition is	Pointcuts are composed with each other using nested communication diagrams in Pointcut Composition Model. Pointcut-advicecomposition is modelled in a Pointcut-Advice Diagram.

	represented by Aspect-Class Dynamic Model and Pointcut Composition Model.	
AODM	Aspect composition is supported. A keyword <<dominates>> is used for the representation of dependency between aspects. It establishes the order of execution of aspects.	No support is provided for pointcut-pointcut and pointcut-advice compositions.
Theme/UML	A composition model has been proposed which depicts structure of aspects and base classes along with behavioural of aspects.	There is no support for composing aspectual elements.
AOSD/UC	Aspect-aspect composition is supported.	Only pointcut-advice composition is supported. There is no mechanism available for pointcut-pointcut composition.
Motorola Weavr Approach	Aspect-aspect compositions are supported. A stereotype <<follows>> has been proposed to decide the order of execution of aspects (Zhang et al., 2007d).	Inner aspect composition are well-supported by the approach (Zhang et al., 2007d).
AAM	The approach supports aspect-base and aspect-aspect composition.	A composition of pointcut-advice has been proposed by Solberg et al. (2005) and Reddy et al. (2006) but pointcut-pointcut composition is still not supported.
JAC Design Notations	Aspects are composed with each other on the structural level using class diagrams.	There is no support for inner-aspect composition provided by the approach.
Klein's Weaving Approach	The approach allows aspect-base and aspect-aspect composition with the help of sequence diagrams.	No inner-aspect composition method is proposed.
SUP Approach	The approach allows aspect-base and aspect-aspect composition.	The approach does not support inner-aspect composition.

The table 6.14 indicates that all of the approaches including AODL support aspect composition. There are different composition strategies offered by each approach but the majority have adapted class diagrams or sequence diagrams of UML. The different between AODL and other approaches lies in providing compositional support for aspectual elements. Other than AODL, Only AOSD/UC and AAM support pointcut-advice composition in a notational and diagrammatic way. The rest of the approaches represent this type of composition statically and do not address it separately from aspect composition.

Although, AOSD/UC and AAM support pointcut-advice composition, they still lack in supporting pointcut-pointcut composition. AODL is the only approach that addresses this composition at the modelling level and provides diagrammatic support in the form of a pointcut-advice diagram.

b) Rules to resolve conflicts:

During the composition of aspects with each other and with base constructs, two major problems arise, the shared join point problem (Nagy et al., 2005) and aspect interference problem (Katz and Katz, 2008). The shared join point problem arises when two or more aspects interact with the base system at the same join point simultaneously. The solution to the problem is a pre-defined precedence order of execution. The majority of the approaches leave the solution until the implementation phase and precedence is declared in the source code. For example, AspectJ provides a *declare precedence* keyword to order advices, Composition Filters (Compose, 2012) provides *Seq* operator to declare precedence, and JAC (Pawlak et al., 2005) determines the order by implementing wrappers in the classes which are filed in a wrapper file in an execution sequence (Iqbal and Allen, 2012).

The aspect interference problem is also associated with aspect composition. It arises when a composition causes drastic changes in the aspect definitions or base program, such as change in join points, change in value of variables or change in some aspect’s behaviour. This problem has also been addressed at the implementation level by most of the researchers. The notable techniques are by Lagaisse et al. (2004), Nagy et al., (2005) and Durr et al., (2005). The contemporary researchers, however, have started suggesting design solutions to this problem. For example, techniques by Reddy et al., (2006), Zhang et al., (2007) and Driver et al., (2008) are among some of the renowned examples.

This criterion evaluates if AODL and the selected approaches provide any technique or rules to resolve both the problems at the design level. Table 6.15 assess all the approaches against this criterion.

Table 6.15 - Comparison of the approaches based on Rules to resolve conflictscriterion

Approach	**Rules to resolve conflicts**
AODL	Aspect-Class Structural model provides a structural representation of aspect composition. A stereotype <<*precedes*>> has been introduced which resolves the priority problem between two executing aspects. The problem can also be overcome during the specification of pointcuts which is done in the form of a pointcut table. The table specifies each pointcut in detail thus providing a mechanism to allocate precedence to the aspects that have a clash before they are implemented. The Pointcut Composition Model allows pointcut-pointcut composition and pointcut-advice composition using communication diagrams. This model can help in avoiding aspect interference at the design level.
AODM	Only conflicts regarding priority of execution of aspects have been handled by providing <<*dominates*>> stereotype (Stein et al., 2002a). This stereotype points from an aspect whose priority is greater to the one whose priority is lesser.
Theme/UML	No conflict resolving technique has been provided by Theme/UML. It is assumed that designers compose the models by considering their ordering beforehand.
AOSD/UC	Although a clear approach for resolving conflicts has not been presented in the literature, some refactoring methods have been suggested to remove conflicts from the design models.
Motorola Weavr Approach	A conflict resolving technique has been proposed in (Zhang et al., 2007d) in which a keyword <<follows>> has been introduced to order the execution of aspects. The approach has claimed that the shared join point problem can be resolved using this technique.
AAM	Syntactical conflicts can be detected using operationalized techniques proposed by Muller et al. (2005). The paper has also introduced composition semantics and directives to help with composition and

	conflict detection.
JAC Design Notations	No rules have been proposed by the approach for resolving aspectual conflicts.
Klein's Weaving Approach	No method to resolve conflicts has been proposed yet.
SUP Approach	The state charts provide sequence of events which can be considered as a solution to the ordering problem so one can say that an implicit conflict-resolving mechanism is provided.

AODL provides methods to avoid the shared join point problem at the modelling level. The problem can be overcome with the help of a Pointcut Table where pointcuts are specified and aspects' execution is prioritized. The problem can also be addressed in the Pointcut Composition Model where a keyword <<*precedes*>> has been provided to allocate precedence to the colliding aspects. There are some other approaches as well that address this problem during the modelling phase, such as AODM, Motorola Weaver Approach and AAM. These are the approaches besides AODL, which explicitly provides techniques to resolve shared join point problem. Other approaches either don't address this problem at all or implicitly provide composition strategies to resolve such conflicts.

As far as the aspect interference problem is concerned, it has not been addressed by any approach other than AODL and Motorola Weavr Approach. Both approaches provide pointcut composition strategies that can be utilize to overcome interference. It is important to note that AODL does not claim to eradicate all kinds of interference rather it asserts that aspect interference can be reduced if the proposed techniques are adopted.

Summary

The evaluation against Concern Composition criteria has revealed that AODL is the only approach other than Motorola Weaver that fulfils all the criterions. As shown in Table 6.16, AODL provides notational methods to support both aspect and inner-aspect compositions. The approach also provides strategies to avoid conflicts that arise as a result of a composition.

Table 6.16 - Summary of evaluation of the approaches against Concern Composition criteria

	Concern Composition		
	Aspect Composition		Conflict Resolution
	Aspect Composition	Inner-Aspect Composition	
AODL	✓	P-P, P-A	✓
AODM	✓	✕	✓
Theme/UML	✓	✕	✕
AOSD/UC	✓	P-A	✕
Motorola Weavr Approach	✓	P-P, P-A	✓
AAM	✓	P-A	✓
JAC Design Notations	✓	✕	✕
Klein's Weaving Approach	✓	✕	✕
SUP Approach	✓	✕	✓

Legend: ✓ = Supported, ✕ = Not Supported, P-P = Pointcut-Pointcut, P-A = Pointcut-Advice

Findings:

The following strengths and weaknesses of AODL have been identified during this analysis:

Strengths:

- AODL offers good notational and graphical methods to compose aspects
- AODL also supports pointcut-advice composition and pointcut-pointcut composition.
- AODL provides techniques to avoid shared join point problem and aspect interferences

Weaknesses:

The techniques for conflict resolution are still immature. There is more work required so that techniques become more robust and can ensure eradication of all types of conflicts and aspect interferences entirely.

6.5.6. Effectiveness

There can be a number of parameters that can be adopted to evaluate effectiveness of a design language. We have only selected those criterions that reflect efficacy and maturity of an aspect-oriented design language. The primary qualities of an AOD Langue are comprehensibility, extensibility, traceability, and scalability. The following sections evaluate all the selected approaches against each of these criterions.

a) Comprehensibility

An AOD language is expected to comprise of design notations which are easy to understand. The characteristics of a construct must be reflected in its assigned notation. Most of the existing AOD approaches either adapt UML notations or extend them to propose notations for aspect-oriented constructs. The UML notations are object-oriented in nature and they can only depict object-oriented properties. Aspect-oriented constructs, on the other hand, are not pure object-oriented entities and need to be represented with such notations that could represent all of their features and relationships. Due to this dilemma, many notations that have been proposed by AOD researchers hinder the comprehensibility of the design due to their inability to represent themselves fully in a design model.

Table 6.17 evaluates AOD as well as all the selected approaches to find out the comprehensibility of the design notations and diagrams proposed by these approaches.

Table 6.17 - Comparison of the approaches based on Comprehensibility criterion

Approach	Comprehensibility
AODL	The language proposes design notations reflecting characteristics of the design constructs and elements. For instance, an aspect is represented by a container with a crossed circular symbol (⊗) on top to denote that aspect is a crosscutting concern and a join point is denoted by a circular symbol containing a joining dot (⊙) suggesting that an aspect will join a base construct at this point to add its behaviour. Other symbols are similarly designed reflecting the behaviour and features of the construct. The design diagrams have also been developed in such a manner that even a novice designer can easily comprehend the purpose and representation depicted by the model. For instance, Aspect-Class Dynamic Model shows the composition of aspects with the base constructs. This composition is denoted

	by a weaving association ~~() whi~~⊕ depicts the appending process of an aspectual behaviour with the base program.
AODM	The approach uses parameterized templates to represent aspects and collaborations to depict behavioural features of aspects. The approach is comprehensible in a way that it uses UML's notations but distinguishing aspectual notations from those of base constructs is not easy. The comprehensibility provided by the approach, therefore, cannot be considered as very good.
Theme/UML	Theme/UML uses composition patterns which make the individual designs comprehensible but the integrated design becomes very complex, especially for large systems (Blair et al., 2005).
AOSD/UC	The approach is comprehensible in a sense that it utilizes UML notations and diagrams but there are some relationships such as crosscutting and execution precedence among aspects, which cannot be captured by traditional UML semantics. Similarly, aspect, pointcuts, advices and inter-type declarations require new notations because of their different nature from object-oriented constructs.
Motorola Weavr Approach	Motorola Weavr approach relies heavily on the platform it is built upon, which is Motorola Weavr. There is an extensive use of state machines to represent aspects and pointcuts which is not as comprehensible as a notational representation of these constructs could be.
AAM	To make the approach more comprehensible, France et al., (2004) and Kim et al. (2004) have proposed notations based on Role-Based Metamodelling language with an additional symbol '|' to distinguish the constructs from those of the language. This approach hampers the comprehensibility even further rather than improving it as the exploited language is less-known and all the aspectual constructs are not well-represented by the proposed notations.
JAC Design Notations	Pointcuts are represented as static associations among aspects and base constructs. There is no notational support for pointcuts, advices, join points and inter-type declarations. The aspect's representation is similar to that of a class with a keyword <<aspect>>, which is also not a complete notation to represent a construct such as an aspect.
Klein's Weaving Approach	The approach only supports the weaving process and does not support modelling of the concerns.

SUP Approach	The approach denotes aspects with a container similar to a class with additional stereotype <<aspect>> which is not a good representation of an aspect as it has contrasting features and non-object-oriented nature if compared with a class. Similarly, there are no distinct notations to represent an aspectual element, which reduces the comprehensibility of the approach.

The Theme/UML approach and AODM provide new notations for aspect representation. These notations are extended version of UML which captures all the features of an aspect, but they become very complex upon integration especially when the system is large and complex. Similarly, AOSD/UC and SUP approach extend UML's class diagram notations but as discussed before, aspects cannot be fully represented with UML notations. The AAM approach and Motorola Weavr Approach have adopted notation of RBML and Motorola Framework respectively with some additional characteristics. Their notations are even more complex for new designers as both the frameworks are less common in use. The second problem common to all the approaches is the lack of representation of aspectual elements, such as pointcuts and advices. The system cannot be fully documented or designed without these elements being represented.

On the contrary, AODL has proposed notations based on the features and characteristics of design constructs. The comprehensibility of these notations is very high as each notation is distinct and only represents one design element. AODL has also proposed notations for pointcuts and advices which again makes the system more understandable and easy to be designed.

The comprehensibility of AODL has been assessed by comparing its application to Car Crash Crisis Management Case Study (discussed in section 5.2) with the similar applications by Shmuel et al., (2012), Mosser et al., (2010), Hölzl et al., (2010), Landuyt et al., (2010) and Mussbacher et al., (2010). It has been observed that distinct notational support for every aspectual construct in AODL makes it more comprehensible and improves the readability of AODL design models. A detailed comparison can be found in (Iqbal and Allen, 2012b).

b) Extensibility:

Extensibility is important for a design language as the programming languages and design paradigms keep evolving and new constructs keep creeping in with the advancement of the technology. An AOD language is expected to support an extension mechanism to adopt new elements, attributes and relationships. There are two types of extensions that are provided by an AOD language/approach, a heavy-weight extension, which allows new components other than aspects to be introduced, and a light-weight extension that allows the introduction of new elements and attributes to the existing components.

Most of the approaches discussed in this evaluation extend UML by adopting profiling or metamodelling. The difference is that the metamodelling approach allows the introduction of new components other than aspects, while profiling does not. The approaches following metamodelling may therefore allow both types of extensions but those following profiling mechanism may only allow light-weight extensions.

Table 6.18 - Comparison of the approaches based on Extensibility criterion

Approach	**Extensibility**	
	Heavy-Weight Extension	**Light-Weight Extension**
AODL	AODL extends UML MOF metamodels which can be extended to include new design constructs other than aspects.	New features, attributes and features can be introduced to the designs constructs other than what are already defined.
AODM	It does not support heavy-weight extension	The approach supports light-weight extension.
Theme/UML	Theme/UML extends UML metamodels, thus supports heavy-weight extension.	No support is provided for light-weight extensions.
AOSD/UC	The approach supports heavy-weight extension as UML 2.0 metamodels are followed.	The light-weight extension is also supported.
Motorola Weavr Approach	There is no support available for heavy-weight extension yet.	The light-weight extension mechanism is supported by the approach. New notations and

		elements can be introduced in the UML profile used by the approach.
AAM	Heavy weight extension can be made by introducing new modelling elements.	Light-weight extensions have not been supported.
JAC Design Notations	No extension mechanism has been proposed or supported by the approach.	There is no support for light-weight extension provided by the approach.
Klein’s Weaving Approach	No support for heavy-weight extension is provided yet.	No support for light-weight extension is provided yet.
SUP Approach	No support for heavy-weight extension is provided. The approach uses UML profiling which does not allow introduction of new non object-oriented constructs.	The light-weight extension is possible as UML profiling does allow introducing of new features, attributes and relationships other than what are already defined.

As shown in Table 6.18 Theme/UML and AOSD/UC follows UML metamodelling which allows these approaches to be accommodating towards the introduction of heavy-weight extension as well as light-weight extension. AODM only allows light-weight extension whereas AAM only allows heavy-weight extension. The rest of the approaches do not support any of the extensions except for SUP Approach which only supports light-weight extension.

AODL, however, supports both types of extension. AODL is based on aspect-oriented design models and design constructs which is entirely opposite to all the other approaches, which all follow either UML profiling or metamodelling. It has been discussed before that UML is a purely object-oriented design technology which does not support non object-oriented constructs. That is the reason that AODL’s extensibility is much better than other approaches.

c) Traceability:

Traceability refers to the ability of design models to be tracked from an earlier phase to a later phase in development life cycle. There are two kinds of traceability, external and internal. The external traceability allows the traceability of concerns from one phase to another while internal traceability allows detailed models to be traced to more abstract ones. This criterion assesses the traceability provided by AODL and other approaches in Table 6.19.

Table 6.19 - Comparison of the approaches based on Traceability criterion

Approach	**Traceability**	
	External Traceability	**Internal Traceability**
AODL	The external traceability is not yet supported in AODL.	There are two abstract models, Aspect Design Diagram and Aspect-Class Dynamic Model which are refined into a more detailed model, Pointcut Composition Model. Similarly, internal elements are specified in a Pointcut Table in detail which also provides support for refinement mapping.
AODM	It supports external traceability from design to implementation.	Internal traceability is not supported because there is no mechanism provided by the approach to map elements from higher abstraction to detailed design.
Theme/UML	The approach provides modelling support for Themes from requirements engineering phase to the implementation phase, hence, providing a mean to trace Themes throughout the development life cycle.	Internal traceability is also supported as the approach provides both high level and low level of abstractions.
AOSD/UC	The concerns are modelled as use case slices which can be traced from requirement engineering phase to the design and implementation phase.	The component diagrams are transformed into class diagrams. This provides a tracing capability of the approach for internal elements.
Motorola Weavr Approach	External Traceability is not supported.	The structural diagrams depicted in class diagrams are refined into composite structure diagrams, thus providing internal traceability.
AAM	The approach primarily focus on architectural representation,	As far as internal traceability is concerned, it is only limited to tracing concerns from

	hence, traceability from analysis to implementation phase is not supported.	requirement engineering approach to architectures.
JAC Design Notations	External traceability is possible on the abstract design level.	There is no support for internal traceability provided by the approach as the design models are not refined into detailed ones.
Klein's Weaving Approach	No support for external traceability is provided.	No support for internal traceability is provided.
SUP Approach	External traceability is supported from the requirements to design.	There is no support for internal traceability available.

There are only two approaches that support both external and internal traceability. Other approaches either support only one kind of traceability or support none. AODL does support internal traceability fully as abstract design models can easily be traced to more detailed ones. The external traceability is only supported partially though. Only design to implementation traceability is possible because there is no requirements analysis approach provided by the language yet.

d) Scalability:

Scalability refers to the ability of a design approach to implement a complex system as comprehensively as simpler ones. The approach must cover all aspects of system design thus supporting all types of interactions among design models, even when the design becomes bigger and more complex.

The scalability of AODL and other approaches have been assessed in Table 6.20.

Table 6.20 - Comparison of the approaches based on Scalability criterion

Approach	**Scalability**
AODL	The approach has been illustrated with the help of a detailed case study, implementation of a Car Crash Crisis Management system, which supports the scalability of the approach.
AODM	No support for modelling high-level elements provided in the approach. There are no examples of implementation of the approach on complex

	systems available either.
Theme/UML	Scalability has been demonstrated with the help of some easy to complex case studies (Clarke and Baniassad, 2005).
AOSD/UC	Besides some easy real world implementations, a complex hotel management system has been implemented using the AOSD/UC approach. This example shows that the approach is capable of modelling complex system.
Motorola Weavr Approach	The approach is scalable which is evident by the examples provided in the literature. A detailed telecom system has been implemented that shows the scalable nature of the approach.
AAM	The scalability of the approach has not been addressed in the available literature. The approach is yet to be tested on complex systems involving several concerns.
JAC Design Notations	The approach is very limited in addressing all issues related to aspect modelling. Only class diagrams have been utilized, which only provides structural view of the system. Consequently, there is no support for scalability provided by the approach.
Klein's Weaving Approach	There is no example provided by the literature supporting scalability.
SUP Approach	The approach has not provided any example proving the scalability.

There are three approaches, Theme/UML, AOSD/UC and Motorola Weavr Approach other than AODL which provides a high level of scalability. This has been proven with the help of complex implementation examples which have been provided in the available literature of the approaches. AODL has been implemented on two complex case studies, a Car Crash Crisis Management System and a Telecommunication System and on a number of small systems. The findings have indicated that the language covers all design aspects of complex systems.

Summary

A summary of all the criterions that make up Effectiveness criterion has been provided in Table 6.22.

Table 6.21 - Summary of evaluation of the approaches against Effectiveness criterion

	Extensibility		Traceability		Scalability	Comprehensibility
	Heavy-Weight	Light-Weight	External	Internal		
AODL	✓	✓	~	✓	High	High
AODM	✗	✓	~	✗	Low	Middle
Theme/UML	✓	✗	✓	✓	High	Middle
AOSD/UC	✓	✓	✓	✓	High	Middle-High
Motorola Weavr Approach	✗	✓	✗	✓	High	Middle
AAM	✓	✗	✗	✓	Middle	Middle-High
JAC Design Notations	✓	✓	~	✗	Low	Middle
Klein's Weaving Approach	✗	✗	✗	✗	Low	Low
SUP Approach	✗	✓	~	✗	Low	Middle

Legend:✓ = Supported, ✗ = Not Supported, ~ = Partially Supported, Rating: Low, Middle, Middle-High, High

Findings:

The following findings have been yielded about AODL by the evaluation in this section.

Strengths:

- AODL provides support for internal extensibility of design components as well as attributes and relationships of existing components.
- AODL provides internal mapping that indicates full support for internal traceability AODL has been tested on simple as well as complex systems which indicates the scalable nature of the language

- Design notations proposed by AODL are comprehensible in nature and improve the understandability of the overall design of the system.

Weaknesses:

- AODL does not provide complete external traceability as it is a design language and does not have a sister approach for analysis of the concerns.
- AODL is still an evolving approach and there is more application of the language required to make it a mature design language.

6.5.7. Tool Support

Tool support is imperative for a new design language to become more mature and to be adopted in the industry. The tool provides a modelling support that helps in modelling design constructs in a visual environment thus ensuring syntactical and semantic correctness of the models. The tool must also provide composition support for integrating design models and identifying and resolving conflicts among them. And the tool should provide a code generation capability so that design models are translated into source code easily.

a) Modelling Support

This criterion evaluates whether the tool provides a visual editor to model design constructs and the relationships among them. Table 6.23 evaluates all the approaches against this criterion.

Table 6.22 - Comparison of the approaches based on Modelling Support criterion

Approach	Modelling Support
AODL	No tool support for modelling concerns is provided yet.
AODM	A tool named, M4JPDD, has been developed to design Join Point Designation Diagrams (Stein and Hanenberg, 2008).
Theme/UML	Standard UML editors are used to model aspect concerns, base concerns and relationships (Clarke, 2012).
AOSD/UC	There is no tool available for the approach so no support is provided for modelling elements using an automated environment.
Motorola Weavr Approach	The approach is implemented as an extension to the Telelogic TAU MDA tool which supports modelling of aspectual elements with base elements

	using the tool.
AAM	There is no tool support available yet, although a number of proposals for tool development have been presented in some publications (Reddy et al., 2006; France et al., 2007).
JAC Design Notations	An IDE based on JAC Framework has been provided which allows modelling of the design constructs using proposed notations.
Klein's Weaving Approach	No tool support for modelling concerns is provided yet.
SUP Approach	No tool support has yet been provided for the approach.

The Theme/UML, Motorola Weaver and JAC Design Notations are the only approaches that provide a tool support for modelling of design constructs. AODL along with all other approaches does not have any tool support available yet.

b) Composition Support

This criterion evaluates all the approaches to find out if they provide a tool support for composing design models. The tool must also be able to identify conflicts arising as a result of integrations and be able to resolve them. Table 6.24 evaluates all the approaches against this criterion.

Table 6.23 - Comparison of the approaches based on Composition Support criterion

Approach	Composition Support
AODL	No tool support for composition is available yet.
AODM	Composition support has not been supported by the AODM tool yet.
Theme/UML	An Eclipse plugin has been developed to model composition of concerns (Clarke, 2012).
AOSD/UC	Composition is deferred to the implementation phase in AOSD/UC. Since there is no tool provided for the approach so no automated support is available for composition process.
Motorola Weavr Approach	An extension to the Telelogic TAU MDA has been provided to compose aspects with base constructs.

AAM	There is no tool support available yet.
JAC Design Notations	There is no support for composition at the design level by the provided tool.
Klein's Weaving Approach	There is no tool support available specifically designed for this approach. Yet, authors propose to use any UML 2.0 tool to model the weaving process. Klein et al., (2007) has proposed Omondo UML tool for modelling weaving process in sequence diagrams.
SUP Approach	No tool support has yet been provided for the approach.

Only Theme/UML and Motorola Weaver provide a tool support for composition of models. The rest of the approaches along with AODL do not have any tool support available yet.

c) Code Generation

This criterion assesses tool support provided by the approaches to find out whether tools generate code from the visualized models. Table 6.25 investigates all the approaches against this criterion.

Table 6.24 - Comparison of the approaches based on Code Generation criterion

Approach	Code Generation
AODL	No tool support for code generation from composed models is available yet.
AODM	There are tools available to generate code from JPDDs (Hanenberg et al., 2007; Stein and Hanenberg, 2008; Stricker et al., 2009).
Theme/UML	A third-party technology, openArchitectureWare, has been proposed to be used to generate code from Theme/UML models (Clarke, 2012).
AOSD/UC	There is no tool available for AOSD/UC approach.
Motorola Weavr Approach	An extension to the Telelogic TAU MDA has been developed which also supports code generation.
AAM	There is no tool support available yet.
JAC Design Notations	The IDE provided by the approach does have the capability to generate code.

Klein’s Weaving Approach	No tool support is available to generate code.
SUP Approach	No tool support has yet been provided for the approach.

AODL does not provide any tool support and it does not have support available to generate code from design models. There are four approaches, AODM, Theme/UML and Motorola Weavr and JAC Design notations that provide tools to generate code from the models.

Summary:

A summary of all the factors which are part of the Tool Support criteria is given in Table 6.26.

Table 6.25 - Summary of evaluation of the approaches against Tool Support criterion

	Tool Support		
	Modelling Support	Composition Support	Code Generation
AODL	✕	✕	✕
AODM	✓	✕	✓
Theme/UML	✓	✓	✓
AOSD/UC	✕	✕	✕
Motorola Weavr Approach	✓	✓	✓
AAM	✕	✕	✕
JAC Design Notations	✓	✕	✓
Klein’s Weaving Approach	✕	✕	✕
SUP Approach	✕	✕	✕

Legend:✓ = Supported, ✕ = Not Supported

The summary of the evaluation reveals that most of the languages do not have tool-support available. AODL does not provide tool-support either.

Findings:

The following findings have been yielded by the evaluation of AODL in this section:

Weaknesses:

AODL does not provide tool-support yet. Tool development is part of future research.

6.4. Discussion

Evaluating a design language is not an easy task. There are many factors that play an important role in making a design language effective. There are always certain kinds of factors attached with a design language. For instance, UML provides design solutions for object-oriented constructs. To evaluate UML one can think of support for encapsulation and inheritance as these are fundamental properties of object-oriented paradigm. Similarly, an aspect-oriented design language can be assessed on the parameters such as, support for representation and design of crosscuttings, level of abstraction and composition techniques.

Keeping in view these qualities, a set of criteria has been developed for the evaluation of AOD design approaches. The criteria assess fundamental properties of an AOD language from different perspectives. It includes parameters to assess not only support for designing structural crosscutting but also behavioural crosscutting. It does not only contain criterions to assess composition of aspects but also composition of aspectual key elements. A set of criteria for assessing maturity and efficacy of the language has also been included to find out how mature and scalable a language or approach is to be adopted in the industry.

Out of the existing AOD approaches, eight most popular approaches have been selected. The selection is based on maturity of the approaches and similarity with AODL apart from being popular. All these approaches have been evaluated against the set criteria along with AODL. A comparison has been made among the approaches, especially between AODL and the contemporary approaches. The purpose of this kind of evaluation was to find out the maturity of AODL and its position among existing approaches. The evaluation has also revealed strengths and weaknesses of AODL.

On average, AODL has scored well. Against many parameters, AODL has been found to be better than the existing approaches. The most remarkable finding is that AODL covers majority of the parameters included in this set of criteria.There are, however, some weaknesses of AODL that has been revealed by this evaluation. First of all, AODL is still evolving and there are still some modelling areas that could be improved. The prime example is the composition model, which is lacking a resultant composed model of the whole system that would show an overall model of the system after aspects are composed with the base constructs. The second modelling improvement can be in providing modelling support for static crosscutting (inter-type declarations). Design diagrams for static associations between aspects and base constructs can be incorporated in the Aspect-Oriented Design Diagram to complete the design representation of all inner aspectual elements. AODL is also lacking a formal description of its semantics for all design notations. Keeping in mind it is still an evolving approach, one can envisage that AODL will improve and will become more effective and mature with time.

6.5. Chapter Summary

The chapter presents a qualitative evaluation of AODL. A set of criteria has been discussed which is applied to AODL along with eight existing design methodologies. Each criterion in the main criteria assesses the design approaches from a certain perspective. The comparison is made which is then summarized in comparison tables. At the end of evaluation against each criterion, strengths and weaknesses of AODL have been discussed. The chapter closes with a discussion on the results found during the evaluation.

Chapter 7:
Conclusions and Future Work

This chapter concludes the study conducted during this research. The chapter explains how the hypothesis of the study has been proved and what methods have been employed to prove this hypothesis. The contribution to knowledge made as a result of this study has also been highlighted. The limitations and weaknesses of AODL, which have been identified in the evaluation chapters, have been discussed along with the possible improvements that can be made to rectify these issues. The chapter concludes with an explanation about the future direction of the research.

7.3. Achieved Goals of the Research

The primary hypothesis of the research was:

> *A unified design[1] approach for aspectual and non-aspectual concerns of a system improves quality[2] of the design and makes it comprehensible[3] and effective[4].*

The hypothesis is proved with the help of following findings:

1. AODL was developed to provide a unified design framework for aspects and objects. It was felt that the existing design approaches force designers to adopt two different design methodologies for aspect-oriented and object-oriented constructs. UML is the widely used design language for object-oriented constructs, and most of the designers use this language while designing the base constructs. Whereas for aspect-oriented constructs, a number of design approaches are available, the majority of which are different from UML. The designers have to use two different design languages to design one system which makes it hard for these new aspect-oriented design languages to be adopted. To fill this gap, AODL was developed to provide a unified design approach. The intention was to develop a design approach similar to UML for aspect-oriented constructs. The reason behind not selecting UML in its current state is that it does not support design of non object-oriented constructs. Therefore, the solution was to develop a similar language to UML with similar design notations and design diagrams. AODL utilizes and extends UML diagrams to represent aspects and their behaviour. The design notations are different from those of UML but are designed in the similar way.

2. An exhaustive evaluation of AODL has been conducted (explained in Chapter 5 and 6) which indicates that AODL improves quality of the design by providing a complete design support for aspectual concerns and their constituent elements in a unified design approach along with base constructs. During the evaluation of AODL, the quality of the language has been assessed with a set of 20 criteria. These criteria assess different aspects of the design approach from the quality perspective. The evaluation has also been applied to eight other contemporary design approaches and a comparison between AODL and these approaches has been made to assess the efficacy of the language over existing approaches.

3. The features and nature of each aspectual construct are reflected in its designated AODL notation which makes it comprehensible and improves the readability and understandability of the design. The AODL design diagrams have also been developed on the same principle. Each diagrammatic model contains notations and associations which reflect the purpose of the diagram. There are separate diagrams for structural and behavioural representation of aspects and their associations. For instance, the Aspect Design Diagram presents a structural depiction of an aspect and its association with the base classes, and the Pointcut-Advice diagram depicts the structural association between pointcuts and advices. For weaving associations, a structural model, entitled Aspect-Class Structural Model, has also been proposed that captures structural relationships between aspects and base classes. Similarly, there are models for behavioural representations as well. For instance, the Joint Point Behavioural Model captures join points in a behavioural representation of activities and the Aspect-Class Dynamic Model captures the dynamic process of weaving of aspects with base objects.

4. Efficacy is a difficult criterion and there is no standard way to calculate it. A number of sub criteria have been utilized in our evaluation (explained in Chapter 6) to find out the efficacy of a design methodology. Some of these sub criteria are support for Structural and Behavioural Crosscutting, Concern Representation, Extensibility, Traceability, and Concern Composition. The detailed analysis of AODL against these criteria has revealed that AODL can be considered as an effective language as far as support for these criteria is concerned. One of the major weaknesses in most of the existing strategies is support for inner-aspect representation and intra-aspect compositions. The pointcuts and advices are not well-represented in the majority of the approaches, and composition of these constructs is overlooked as well. AODL, on the other hand, provides full support for concern representation and their compositions.

7.4. Contribution

There are a number of aspect-oriented design and modelling approaches available in the industry, though none of these approaches have been adopted as a standard technique yet. As described in the motivations of this thesis in Chapter 1 and Chapter 4, there is still room for new approaches that could provide a unified design approach and that could comprehensively represent aspects and their constituent elements.

The following are some of the notable contributions of this study:

The research has analysed all the existing aspect-oriented design approaches (some of the notable ones have been explained and critiqued in Chapter 3). It has been observed that there are still limitations with the existing design approaches as far as comprehensiveness of the approach and uniformity of the design standards are concerned. The research has, therefore, focused on developing a unified approach to design both aspectual and non-aspectual concerns in a single design framework. Aspect-Oriented Design Language (AODL) is the resultant design approach that has been proposed as an answer to the problems in the existing design approaches. It contains a set of design notations to represent each aspectual concern, such as aspect, advice, pointcut, join point and weaving association, in a notational way. The notations are then complemented with design diagrams to model structural and behavioural characteristics of aspects and base constructs. Examples of such diagrams are Join Point Identification Diagrams (explained in section 4.3.2.1.2), Join Point Behavioural Diagram (explained in section 4.3.2.1.3) and Pointcut Composition Model (explained in section 4.3.2.4.4). The relationships and associations among aspectual elements and constructs can also be represented in diagrams such as Aspect Design Diagram (explained in 4.3.2.3.2), Aspect-Class Structural Model (explained in 4.3.2.4.3) and Aspect-Class Dynamic Model (explained in 4.3.2.4.2). AODL has been explained in detail in a journal paper published in 2011 (Iqbal and Allen, 2011).

Pointcuts are vital aspectual elements that define the joining of aspects with the base constructs. Pointcut modelling is considered an integral part of aspect-oriented design. AODL provides a pointcut table (explained in 4.3.2.5) to specify each pointcut with a detailed description of related aspects and base constructs (such as objects and methods).

The table also documents the related advices with the pointcuts to indicate the relationships between the two. The table also provides a mechanism to define precedence of each aspect should a join point be shared between two or more aspects. The defined order of execution can help in avoiding the shared join point problem. The table has also been explained in a conference paper (Iqbal and Allen, 2012).

The pointcuts are associated with other pointcuts and with the base program's methods. The composition of pointcuts helps in designing the joining of aspects with the base objects at run time. This composition is designed in AODL with the help of a design model, known as Pointcut Composition Model (explained in 4.3.2.4.4). Each pointcut is represented diagrammatically along with their advices and parent aspects. The model can help in resolving the aspect interference problem (Katz et al., 2008) that can arise as a result of aspect compositions. The model is also explained in a paper submitted to a journal (Iqbal and Allen, 2012).

A detailed evaluation of some of the notable aspect-oriented design methodologies has been provided in Chapter 6. A qualitative set of criteria has been implemented on these approaches and AODL to assess strengths and weaknesses of each approach. The criteria provide a set of quality attributes that are vital for an AO design approach and can be implemented on any design methodology to evaluate the efficacy and maturity of the approach. Two case studies have also been designed using AODL to provide a demonstration of applicability of the language.

During this study, the following research publications have been produced:

Journal Papers:

1. Iqbal, S. and Allen, G. (2011) 'Designing Aspects with AODL' International Journal of Software Engineering. ISSN 1687-6954
2. Iqbal, S. and Allen, G. (2012) 'Application of AODL: A Case Study', Software: Practice and Experience, (Submitted).
3. Iqbal, S. and Allen, G. (2012) 'Composition of Aspects in AODL', Journal of Systems and Software, (Submitted).

Conference Papers:

1. Iqbal, S. and Allen, G. (2012) 'Pointcut Design with AODL'. In: The Twenty-Fourth International Conference on Software Engineering and Knowledge Engineering (SEKE 2012), July 1-3, 2012. Redwood City, California, USA.

2. Iqbal, S. and Allen, G. (2010) 'Aspect-Oriented Modelling: Issues and Misconceptions'. In: Proceedings of Software Engineering Advances (ICSEA), 2010 Fifth International Conference. : IEEE. Nice, France. pp. 337-340. ISBN 978-1-4244-7788-3
3. Iqbal, S. and Allen, G. (2010) 'A notational Design of Join Points'. In: Future Technologies in Computing and Engineering: Proceedings of Computing and Engineering Annual Researchers' Conference 2010: CEARC'10. Huddersfield: University of Huddersfield. pp. 27-30. ISBN 9781862180932.
4. Iqbal, S. and Allen, G. (2009) 'On identifying and representing aspects'. In: SERP'09 - The 2009 International Conference on Software Engineering Research and Practice, July 13-16, Las Vegas, USA. pp. 497-501. ISBN 1-60132-129-5
5. Iqbal, S. and Allen, G. (2009) 'Representing Aspects in Design'. In: Theoretical Aspects of Software Engineering, 2009 TASE 2009, TheThird IEEE International Symposium on. : IEEE. China, pp. 313-314. ISBN 978-0-7695-3757-3
6. Iqbal, S. and Allen, G. (2009) 'Aspect-oriented design model.' In: Proceedings of Computing and Engineering Annual Researchers' Conference 2009: CEARC'09. Huddersfield: University of Huddersfield. pp. 137-141. ISBN 9781862180857

7.5. Limitations

Some of the limitations of AODL, which have been observed during the evaluation process, are:

- AODL designs aspects identified in the earlier phases of the software development lifecycle using any suitable aspect-oriented requirements engineering approach. AODL does not offer any techniques for capturing aspects from the requirements specifications document at the moment. This limitation affects the results of AODL design techniques if aspects are not properly identified or if some are overlooked in the requirements engineering stage.
- In some situations, some new aspects can arise during the design phase. This can happen due to the introduction of creeping requirements or any modifications to the original design decisions or business logic. AODL does not provide a means to verify a new aspect due to the absence of an aspect identification technique.
- The structural characteristics of an aspect are captured in an Aspect Design Diagram (explained in chapter 4, section 4.3.2.3.2). This diagram does provide a diagrammatic relationship between pointcuts and advices but there is no diagrammatic

representation provided for static crosscutting (inter-type declarations) at the moment. The future work will include new diagrams and design notations to represent this type of crosscutting.

- The weaving process of aspects is captured in an Aspect-Class Dynamic Model (explained in Chapter 4, section 4.3.2.4.3) and Pointcut Composition Model (explained in chapter 4, section 4.3.2.4.4). These models do depict composition of aspectual elements with the base program. However, there is no support available in AODL yet to show the resultant model that is formed as a result of composition of aspects and objects. The future work will also address this problem and support will be provided to develop these kinds of models.

7.6. Future Work

AODL is an evolving approach and there is still room for improvements and extensions. As is mentioned in section 7.3, new design notations are yet to be developed for designing inter-type declarations. The reason to have a distinctive notation for this type of construct is that AODL offers notations and diagrams for all aspectual elements so this construct must also be represented diagrammatically. Another reason is that inter-type declarations represent static crosscutting of a base program, which is required to be represented in the Aspect Design Model to capture associations between the corresponding aspect and involved base constructs.

The weaving process captured by the Aspect-Class Dynamic Model and Pointcut Composition Model in AODL can be enhanced to show a complete system design after the aspects are woven into the base system. The resultant model would show aspectual behaviour linked with the join points in the base program. It would help in modelling dynamic composition of aspects with base classes. Due to lack of time and level of complexity, this model has not been developed during the course of this research. This model can be developed in the future by extending the weaving models.

An early aspect approach will also be part of the future work. The requirements specification documents and use case diagrams can provide a means to identify crosscutting concerns in the requirements engineering phase. The plan is to extend the UML use case diagram with additional notations to identify those functional requirements that overlap others in the system. Similarly, overlapping non-functional requirements can also be identified in the requirements engineering document. A specification document for defining aspects can be developed to specify aspects and their associations.

Finally, tool-support will be provided for AODL as it is extremely important to aid application and adaptability of the language. The tool-support will also help in demonstrating the efficacy of the language over existing design methodologies. Ideally, such a tool should allow ‘round trip’ editing of both the AODL design models and /or the AspectJ source code.

7.7. Closing Remarks

It cannot be claimed that the presented work is the final product. There are a number of areas in AODL that can be evolved and extended to make them even better and more mature. One deficiency that is easily evident is that AODL is needed to be complemented with a compatible requirements engineering approach. That will complete the analysis and design of aspectual components. Another thing, which has already been mentioned in section 7.3, is provision of a detailed aspect composition model that could model and demonstrate the entire weaving process. This research will obviously not halt here. All the limitations and deficiencies will be overcome in the future research.

On a closing note, I have learnt in a great deal about aspect-oriented programming in general and aspect-oriented modelling and design in particular. I will keep on striving with the same zeal in the future as well to take this work further.

References

Albunni, N. and Petridis, M., 2008. Using UML for Modelling Cross-Cutting Concerns in Aspect Oriented Software Engineering. In the proceedings of 3rd International Conference on Information and Communication Technologies: From Theory to Applications, ICTTA 2008. April 2008. pp.1-6, 7-11.

Aldawud O., Bader A., and Elrad T., 2002. Aspect-Oriented Modelling: Bridging the Gap between Implementation and Design. Presented at Generative Programming and Component Engineering Conference (GPCE), Pittsburgh, PA, USA. pp. 189-201.

Aldawud, O., Elrad, T., And Bader, A. 2003. UML profile for aspect-oriented software development.In Proceedings of the 3rd International Workshop on Aspect Oriented Modelling.

AspectJ Team, 2012. The AspectJ Programming Guide, [online]. Available at: <http://www.eclipse.org/aspectj/doc/released/devguide/ajbrowser-navigating.html> [Accessed May 12, 2012].

Bálik, J., & Vranić, V. 2012. Symmetric aspect-orientation: some practical consequences. In Proceedings of the 2012 workshop on Next Generation Modularity Approaches for Requirements and Architecture. pp. 7-12. ACM.

Baniassad E. and Clarke, S., 2004a. Finding Aspects in Requirements with Theme/Doc. presented at Workshop on Early Aspects (held with AOSD 2004), pp. 157-167. Lancaster, UK.

Baniassad E. and Clarke, S., 2004b. Theme: An Approach for Aspect-Oriented Analysis and Design, presented at International Conference on Software Engineering.

Baniassad, E., Clements, P., Araújo, J., Moreira, A., Rashid, A., Tekinerdogan, B., 2006. Discovering Early Aspects.IEEE Software Special Issue on Aspect-Oriented Programming. 23(1): pp. 61-70.

Batory, D. S., Singhal, J. V., Thomas, S. D., Geraci, B., Sirkin M., 1994. The GenVoca Model of Software-System Generators. IEEE Software. pp. 89-94.

Blair, G. S., Blair, L., Rashid, A., Moreira, A., Araujo ´ , J., And Chitchyan, R. 2005. Engineering aspect-oriented systems. In Aspect-Oriented Software Development, R. Filman, T. Elrad, S. Clarke, and M. Aks¸it, Eds. Addison-Wesley, pp. 379–406.

Boner, J., 2004.AspectWerkz - Dynamic AOP for Java. In Karl Lieberherr,editor, 3rd Int. Conf. on Aspect-Oriented Software Development (AOSD’04), Lancaster, UK. ACM. pp. 51–62.

Bosch, J. and Aksit, M. 1992. Composition-Filters Based Real-Time Programming. Vancouver: An Evaluation of Object-Oriented Technology in Real-Time Systems: Past, Present & Future (ACM OOPSLA'92 Workshop).

Brito I., 2008. Aspect-oriented requirements analysis.Ph.D.thesis, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

Bustos, A. & Eterovic, Y., 2007.Modelling aspects with UML class, sequence and state diagrams in an industrial setting.In Proceedings of the 11th IASTED International Conference on Software Engineering and Applications (SEA 2007), 2007, pp. 403-410, Anaheim, USA.

Chitchyan, R., Rashid, A., Sawer, P., Garcia, A., Alarcon, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A. 2005.Survey of Analysis and Design Approaches. TR: AOSD-Europe-ULANC-9.

Cibran, M., D'Hondt, M. and Jonckers, V., 2003.Aspect-Oriented Programming for Connecting Business Rules.In Proceedings of the 6th International Conference on Business Information Systems (BIS'03). Colorado Springs, USA. Vol. 6, No. 7, p. 24.

Clarke, S. 2001. Composition of object-oriented software design models.Ph.D.dissertation, Dublin City University.

Clarke, S. 2002. Extending standard UML with model composition semantics. Sci. Comput. Prog. 44, 1, pp. 71–100.

Clarke, S. And Baniassad, E., 2005. Aspect-Oriented Analysis and Design The Theme Approach. AddisonWesley.

Clarke, S. and Walker, R. J., 2002.Towards a Standard Design Language for AOSD. ACM Proceedings on Aspect Oriented Software Development, pp. 113- 119.

Clarke, S., Harrison, W., Ossher, H., And TARR, P., 1999. Subject-oriented design: Towards improved alignment of requirements, design and code. In Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA’99). pp. 325–339.

Clarke, S., 2012. Theme/UML, [online] Available at:
<http://www.dsg.cs.tcd.ie/aspects/themeUML> [Accessed on May 7, 2012].

Coelho, W., & Murphy, G. C. 2006.Presenting crosscutting structure with active models.In Proceedings of the 5th international conference on Aspect-oriented software development. pp. 158-168. ACM.

Cottenier, T., Van Den Berg, A., & Elrad, T. 2007a.Joinpoint inference from behavioral specification to implementation. ECOOP 2007–Object-Oriented Programming, pp. 476-500.

Cottenier, T., Van Den Berg, A., & Elrad, T. 2007b. The Motorola WEAVR: Model weaving in a large industrial context. Aspect-Oriented Software Development (AOSD), Vancouver, Canada, pp. 32`-44.

Cottenier, T., van den Berg, A., & Elrad, T. 2007c. Motorola WEAVR: Aspect orientation and model-driven engineering. Journal of Object Technology, 6(7), pp. 51-88.

Cui, W. L. L. X. Z. and Xu, D., 2009. Modelling and integrating aspects with UML activity diagrams. In Proceedings of the 2009 ACM Symposium on Applied Computing (SAC 2009), New York, USA, 2009, pp. 403–437.

Dahiya, D. and Dahiya, S., 2008. Software Reuse in Design and Development of Aspects. Computer Software and Applications, 2008.COMPSAC '08. 32nd Annual IEEE International, vol., no., pp.745-750.

Dasgupta, S. , 1989. The structure of design processes, in Advances in Computers, Yovits, M. C., Ed., Academic Press, pp. 1–67.

Dijkstra, Edsger W., 1982. On the role of scientific thought. In Dijkstra, Edsger W.. Selected writings on Computing: A Personal Perspective. New York, NY, USA: Springer-Verlag New York, Inc.. pp. 60–66. ISBN 0-387-90652-5.

Durr, P., Staijen, T., Bergmans, L., and Mehmet A., 2005. Reasoning About Semantic Conflicts Between Aspects. Presented in 2nd European Interactive Workshop on Aspects in Software (EIWAS), Brussels, Belgiu.

Eclipse, 2011. AspectJ, [online] Available at <http://www.eclipse.org/aspectj/> [Accessed 20 December, 2011].

Eclipse, 2012. AspectJ Programming Guide, [online] Available at: <http://aspectj.org/doc/dist/progguide/index.html> [Accessed on 5th of June. 2012].

Eisenbarth T., Koschke R., and Simon D. 2003.Locating features in source code. IEEE Transactions on Software Engineering, volume 29, pp. 210-224.

Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K. and Ossher, H., 2001. Discussing Aspects of AOP.Communication of the ACM. Vol. 44, No. 10, pp. 33-38.

Elrad, T., Aldawud, O., And Bader, A. 2005.Expressing aspects using UML behavioural and structural diagrams. In Aspect-Oriented Software Development, R. Filman, T. Elrad, S. Clarke, and M. Aks¸it, Eds. Addison-Wesley, pp. 459–478.

Evermann, J., Fiech, A., and Alam, F. E., 2011. A platform-independent UML profile for aspect-oriented development, In Proceedings of the Fourth International C* Conference on Computer Science and Software Engineering, pp. 25-34, May 16-18, 2011, Montreal, Canada.

Figueiredo, E., Garcia, A., Sant'Anna, C., Kulesza, U., and Lucena, C., 2005. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. Workshop.on Quantitative Approaches in OO Software Engineering.

Filman, R. E., & Friedman, D. P. 2000. Aspect-oriented programming is quantification and obliviousness. In Workshop on Advanced separation of Concerns, OOPSLA (Vol. 2000).

Filman, R. E., 2005, Aspect Oriented Software Development. Addison Wesley

France, R., Fleurey, F., Reddy, R., Baudry, B., And Ghosh, S. 2007. Providing support for model composition in metamodels.In Proceedings of the 11th International EDOC Conference (EDOC’07). pp. 253-253.

France, R., Ray, I., Georg, G., & Ghosh, S. 2004. Aspect-oriented approach to early design modelling. In Software, IEE Proceedings- Vol. 151, No. 4, pp. 173-185.IET.

Freeman, P., 1980. The nature of design, in Tutorial on Software Design Techniques, Freeman, P. and Wasserman, A. I. Eds, IEEE, 1980, pp. 46–53.

Fuentes, L., Pinto, M., and Troya, J. M. 2007.Supporting the development of CAM/DAOP applications: An integrated development process.Software - Practi. Exp. 37, 1, pp. 21–64.

Grundy, J.C. Aspect-oriented Requirements Engineering for Component-based Software Systems, In Proceedings of the 1999 IEEE Symposium on Requirements Engineering, Limmerick, Ireland, 7-11 June, 1999, IEEE CS Press, pp. 84-91.

Grundy, J. 2000. Multi-perspective specification, design and implementation of software components using aspects. Int. J. Softw. Engi.Knowl.Eng. 20, 6. pp. 713-734.

Guessi, M., Oliveira, L. B. R., and Nakagawa, E. Y., 2011. Extensions of UML to Model Aspect-oriented Software Systems. April 2011 Special issue of best papers presented at CLEI 2010, Asunción, Paraguay. 3-3.

Gupta, P., Garg, S., and Khalon, K. S., 2011.Designing Aspects Using Various UML Diagrams in Resource-Pool Management. International Journal of Advanced Engineering Sciences and Technologies, Vol No. 7, Issue No. 2, pp. 228 – 233.

Hanenberg, S., Stein, D., and Unland, R., 2007. From Aspect-Oriented Design to Aspect-Oriented Programs: Tool-Supported Translation of JPDDs into Code. In: de Moor, O. (Hrsg.): Proc. of 6th International Conference on Aspect-Oriented Software Development (AOSD 2007), ACM, Vancouver, BC, Canada. pp. 113-118.

Harrison, W. and Harold, O. 1993.Subject-Oriented Programming - A Critique of Pure Objects. Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications. pp. 223-228.

Harrison, W., Ossher, H., & Tarr, P. 2002. Asymmetrically vs. symmetrically organized paradigms for software composition. IBM Rsch. Rpt. RC22685 (W0212-147).

Harrison, W.H., Ossher, H.L. and Tarr, P.L., 2002. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition.IBM Research Division, Thomas J. Watson Research Center. RC22685.

Ho, W.-M., Jez´ Equel ´ , J.-M., Pennaneac'h, F., And Plouzeau, N. 2002. A toolkit for weaving aspect oriented UML designs. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02). pp. 99-105.

Hölzl, M., Knapp, A., & Zhang, G. 2010.Modeling the car crash crisis management system using HiLA. Transactions on aspect-oriented software development VII, pp. 234-271.

Iqbal, S. and Allen, G., 2009.On identifying and representing aspects. In: SERP'09 - The 2009 International Conference on Software Engineering Research and Practice, July 13-16, Las Vegas, USA. pp. 113-117.

Iqbal, S. and Allen, G., 2012b. Application of AODL: A Case Study, Software: Practice and Experience, (Submitted).

Iqbal, S. and Allen, G., 2010.Aspect-Oriented Modelling: Issues and Misconceptions. In: Proceedings of Software Engineering Advances (ICSEA), 2010 Fifth International Conference. IEEE at Nice, France. pp. 337-340. ISBN 978-1-4244-7788-3.

Iqbal, S. and Allen, G., 2011. Designing Aspects with AODL. International Journal of Software Engineering. pp. 221-229. ISSN 1687-6954.

Iqbal, S. and Allen, G., 2012.Pointcut Design with AODL. In: The Twenty-Fourth International Conference on Software Engineering and Knowledge Engineering (SEKE 2012), July 1-3, 2012. Redwood City, California, USA. pp. 418-421.

Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B. and Silva, J. R. O., 2004.Documenting component and connector views with UML 2.0.Tech. Rep., 2004, cMU/SEI-2004-TR-008.

Jacobson I., 2003. Use Cases and AspectsWorking Seamlessly Together. Journal of Object Technology, vol. 2, pp. 7-28.

Jacobson, I. And NG, P.W., 2005. Aspect-Oriented Software Development with Use Cases.Addison-Wesley.

JBoss. 2012. JBoss AOP homepage, [online] Available at: <http://labs.jboss.com/portal/jbossaop/> [Accessed 10 February, 2012].

Jingyue, L., Houmb, S. H. and Kvale, A. A. 2004. A Process to Combine AOM and AOP: A Proposal Based on a Case Study. Presented at Workshop on Aspect- Oriented Modelling (held with UML 2004), Lisbon, Portugal.

Jones J.C. 1970. Design Methods: Seeds of Human Futures. (Revised edn 1981) Wiley-Interscience.

Kande, M., Kienzle, J., Strohmeyer, A. 2003. From AOP to UML: Towards an aspect-oriented architectural modelling approach Technical Report, Swiss Federal Institute of Technololgy (Lausanne, 2003).

Katara, M. And Katz, S. 2007. A concern architecture view for aspect-oriented software design.Softw. Syst. Model. 6, 3, pp. `247–265.

Katz, E. et al., 2008. Detecting Interference among Aspects.AOSD Europe Deliverable D116.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Ch., Lopes, Ch.V., Loingtier,J.-M., Irwin, J., 1997. Aspect-Oriented Programming. In: Proceedings of ECOOP 1997, Jyväskylä, Finland, June 9-13, 1997, pp. 220-242.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Ch., Lopes, Ch.V., Loingtier,J.-M., Irwin, J., 1997. Aspect-Oriented Programming," In: Proceedings of ECOOP , Jyväskylä, Finland, June 9-13, 1997, pp. 220-242

Kienzle, J., Guelfi, N., Mustafiz, S., 2010. Crisis Management Systems: A Case Study for Aspect- Oriented Modelling. Transactions on Aspect-Oriented Software Development.

Kim, D.-K., France, R. B., And Ghosh, S. 2004. A UML-based language for specifying domain-specific patterns. J. Visual Lang. Comput. 15, 3-4, 265–289.

Klein, J., Caillaud, B., & Hélouët, L. 2005. Merging scenarios. Electronic Notes in Theoretical Computer Science, 133, pp. 193-215.

Klein, J., Fleurey, F., And Jez´ Equel ´ , J.-M. 2007. Weaving multiple aspects in sequence diagrams. Transaction on Aspect-Oriented Software Development. pp. 167-199.

Klein, J., Hélouët, L., & Jézéquel, J. M. 2006.Semantic-based weaving of scenarios.In Proceedings of the 5th International Conference on Aspect-oriented software development. pp. 27-38. ACM.

Koppen, C. and Stoerzer, M., 2004. Pcdi: Attacking the fragile pointcut problem. In First European Interactive Workshop on Aspects in Software (EIWAS).

Krechetov, I., Tekinerdogan, B., Garcia, A., Chavez, C., Kulesza, U., 2006. Towards an Integrated Aspect-Oriented Modelling Approach for Software Architecture Design.8th Workshop on Aspect-Oriented Modelling (AOM'06), AOSD'06, March, Bonn, Germany.

Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., Staa, A. V., and Lucena C., 2006. Quantifying the Effects of Aspect Oriented Programming: A Maintenance Study, In Proceedings of ICSM'06. 22nd IEEE International Conference on Software maintenance. pp. 223-233. IEEE.

Lagaisse, B., Joosen, W., and Win, B. D., 2004. Managing Semantic Interference with Aspect Integration Contracts.International Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT), Lancaster, UK.

Lau, Y., Zhao, W. Peng, X. Chen, Y. and Jiang, Z., 2007. A unified formal model for supporting aspect-oriented dynamic software architecture.In Proceedings of the International Conference on Convergence Information Technology (ICCIT 2007), Los Alamitos, USA. pp. 450–455.

Lawson, B., 1980. How Designers Think, The Architectural Press Ltd., London.

Li, H., Zhang, J. and Chen, Y., 2010. Aspect-oriented modelling in software architecture pattern based on UML. Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on , vol.3, no., pp.575-578, 26-28 Feb. 2010.

Lieberherr, K. 1996. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company.

Lippert, M. and Lopes, C. V., 2000. A study on exception detection and handling using aspect-oriented programming, In Proceedings of the 2000 International Conference on Software Engineering. pp. 418-427. IEEE.

Lopes, C. V., & Bajracharya, S. K. (2005, March). An analysis of modularity in aspect oriented design. In Proceedings of the 4th international conference on Aspect-oriented software development. pp. 15-26. ACM.

Lopes, C. V., 1997. D: A Language Framework for Distributed Programming. Ph.D. Thesis, College of Computer Science, Northeastern University.

Mahoney, M., Bader, A., Aldawud, O., And Elrad, T., 2004. Using aspects to abstract and modularize statecharts. In Proceedings of the 5th Aspect-Oriented Modelling Workshop (UML'04).

Meyer, B., 1988. Object-oriented Software Construction. Prentice Hall, 1988.

Moreira, A., Rashid, A., & Araujo, J. 2005. Multi-dimensional separation of concerns in requirements engineering.In Proceedings.13th IEEE International Conference on Requirements Engineering. pp. 285-296. IEEE.

Moreira, A., Araújo, J., & Rashid, A. (2005).A concern-oriented requirements engineering model.In Advanced Information Systems Engineering. pp. 55-100. Springer Berlin/Heidelberg.

Mosser, S., Blay-Fornarino, M., & France, R. 2010.Workflow design using fragment composition. Transactions on aspect-oriented software development VII, pp. 200-233.

Muller, P.-A., Fleurey, F., And J´Ez´Equel, J.-M. 2005. Weaving executability into object-oriented metalanguages. In Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05). pp. 264-278.

Mussbacher, G., Amyot, D., Araújo, J., & Moreira, A. 2010. Requirements modeling with the aspect-oriented user requirements notation (AoURN): a case study. Transactions on aspect-oriented software development VII, pp. 23-68.

Nagy, I., Lodewijk, B., and Mehmet, A., 2005. Composing aspects at shared join points. In Andreas Polze Robert Hirschfeld, Ryszard Kowalczyk and Mathias Weske, editors, Proceedings of International Conference NetObjectDays (NODe), volume P-69 of Lecture Notes in Informatics, Erfurt, Germany, Gesellschaft fur Informatik (GI).

OMF, OMG's Meta-Object Facility, [online] Available at: http://www.omg.org/mof/
[Accessed January 5, 2012].

OMG, 2012.UML 2.4.1. Specification, [online] Available at:
<http://www.omg.org/technology/documents/modelling_spec_catalog.htm#UML>
[Accessed on February 10, 2012].

Op De Beeck, S., Truyen, E., Boucke´, N., Sanen, F., Bynens, M., And Joosen, W. 2006. A study of aspect-oriented design approaches. Tech. rep. CW435, Department of Computer Science, Katholieke Universiteit Leuven.

Ossher, H., & Tarr, P. 2002. Multi-dimensional separation of concerns and the hyperspace approach. Software Architectures and Component Technology, pp. 293-323.

Page, J. K. 1966.Conference Report, Ministry of Public Building and Works, London.

Parnas, D. L., 1972. On the criteria to be used in decomposing systems into modules. In Communications of the ACM, Vol. 15(12): pp. 1053-1058.

Paula, V. and Batista, T., 2007.Revisiting a formal framework for modeling aspects in the design phase.In Aspect-Oriented Requirements Engineering and Architecture Design, 2007. Early Aspects at ICSE: Workshops in (pp. 6-6). IEEE.

Pawlak, R., Seinturier, L., Duchien, L., Martelli, L., Legond-Aubry, F., And Florin, G. 2005. Aspectoriented software development with Java aspect components. In Aspect-Oriented Software Development, R. Filman, T. Elrad, S. Clarke, and M. Aks¸it, Eds. Addison-Wesley, pp. 343–369.

Pawlak, R.,Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L., And Martelli, L. 2002. A UML Notation for Aspect-Oriented Software Design.In Proceedings of the 1st Workshop on Aspect-Oriented Modelling with UML (AOSD'02).

Pinto, M., Fuentes, L., and Fernandez, L. 2011.Deriving Detailed Design Models from an Aspect-Oriented ADL using MDD. Journal of Systems and Software, 85(3), pp. 525-545.

Rashid, A., Sawyer, P., Moreira, A., and Araujo, J., 2002. Early aspects: A model for Aspect-Oriented Requirements Engineering. Proc. IEEE Joint Int. Conf. on Requirements Engineering, Essen, Germany, 9–13 September 2002, pp. 199–202.

Rashid, A., Moreira, A., and Araujo, J., 2003. Modularisation and Composition of Aspectual Requirements.In Proceedings of the 2nd International Conference on Aspect-oriented Software Development (AOSD). pp. 11-20. ACM.

Rausch, A., Rumpe, B., Hoogendoorn, L., 2003. Aspect-Oriented Framework Modelling, 4th AOM Workshop at UML'03, (San Francisco, CA, Oct. 2003)

Reddy, R., Ghosh, S., France, R. B., Straw, G., Bieman, J. M., Song, E., and Georg, G. 2006a. Directives for composing aspect-oriented design class models. In Transactions on Aspect-Oriented Software Development I, Lecture Notes in Computer Science, vol. 3880. Springer-Verlag, pp. 75–105.

Reddy, R., Solberg, A., France, R., and Ghosh, S. 2006. Composing sequence models using tags. In Proceedings of the 9th International Workshop on Aspect-Oriented Modelling at MoDELS'06.

Reina, A. M., Torres, J., And Toro, M. 2004. Separating Concerns by Means of UML-profiles and Metamodels in PIMs. In Proceedings of the 5th Aspect-Oriented Modelling Workshop (UML'04).

Shmuel, K. and Mezini, M,, 2010. Transactions on Aspect-Oriented Software Development VII. Lecture Notes in Computer Science (6210). Springer Verlag, Berlin, pp. 321-374. 1st Edition, ISBN 9783642160851.

Shonle, M., Tewari, N. and Rajan, H., 2005.On the criteria to be used in decomposing systems into aspects. In Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference (ESEC/FSE 2005). pp. 1053-1058. ACM Press.

Simon, H. A. , 1973. The structure of ill-structured problems, Artificial Intelligence,Vol. 4, pp. 181–200.

Solberg, A., Simmonds, D., Reddy, R., Ghosh, S., And France, R. B. 2005. Using aspect oriented techniques to support separation of concerns in model driven development. In Proceedings of the 29th Anual International Computer Software and Applications Conference (COMPSAC'05). Volume 1, pp. 121-126.

Spring, 2012.Aspect Oriented Programming with Spring: Spring Framework: [online] Available at <http://static.springsource.org/spring/docs/2.5.x/reference/aop.html> [Accessed January 10, 2011].

Steimann F. 2005. Domain Models are Aspect Free", Proc. MODELS 2005, Springer, 171-185.

Stein, D., Hanenberg, S. and Unland, R., 2003. Aspect-Oriented Modelling: Issues on Representing Crosscutting Features. Presented at Workshop on Aspect- Oriented Modelling (held with AOSD 2003), Boston, Massachusetts, USA.

Stein, D., Hanenberg, S. and Unland, R., 2002c. On Representing Join Points in the UML.Presented at Workshop on Aspect-Oriented Modelling with UML (held with UML 2002), Dresden, Germany.

Stein, D., Hanenberg, S., & Unland, R. 2002a. A UML-based aspect-oriented design notation for AspectJ.In Proceedings of the 1st international conference on Aspect-oriented software development. pp. 106-112. ACM.

Stein, D., Hanenberg, S., 2008. M4JPDD - Tool-Support for Modelling Join Point Designation Diagrams. In: Demo at AOSD 2008. Brussels, Belgium.

Stein, D., Hanenberg, S., And Unland, R. 2002b. Designing aspect-oriented crosscutting in UML.In Proceedings of the 1st Workshop on Aspect-Oriented Modelling with UML (AOSD'02).

Stein, D., Hanenberg, S., And Unland, R. 2006. Expressing different conceptual models of join point selections in aspect-oriented design.In Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06). pp. 15-26.

Stricker, V., Hanenberg, S., And Stein, D. 2009. Designing design constraints in the UML using join point designation diagrams. In Proceedings of the 47th International Conference on Objects, Components, Models and Patterns (TOOLS'09). pp. 57-76.

Sutton, S. M., Jr. and Rouvellou, I., 2005. Concern Modelling for Aspect-Oriented Software Development. in AspectOriented Software Development, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, eds. Boston, MA: Addison-Wesley, pp. 479-505.

Suvee´ , D., Vanderperren, W., Wagelaar, D., and Jonckers, V. 2005. There are no aspects. Electron.NotesTheoret.Comput.Sci. 114.

Tarr, P., Ossher, H., Harrison, H. and Sutton Jr, S., 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In 21th International Conference on Software Engineering (ICSE'99), Ontario, Canada, IEEE Computer Society. pp. 107-119.

Tsang, S. L., Clarke, S., and Baniassad, E., 2004. An Evaluation of Aspect-Oriented Programming for Java-based Real-time Systems Development. Intl. Symp. On OO RealTime Distributed Computing.

Van Landuyt, D., Truyen, E., & Joosen, W. 2010.Discovery of stable abstractions for aspect-oriented composition in the car crash management domain. Transactions on aspect-oriented software development VII, pp. 375-422.

Vanderperren, W., Suvee, D. and Jonckers, V., 2003.Combining AOSD and CBSD in PacoSuite through Invasive Composition Adapters and JAsCo. In Proceedings of Node 2003 international conference, Erfurt, Germany, pp. 36-50, ISBN 3-9808628-2-8.

Von Flach Garcia Chavez, C. 2004. A model-driven approach for aspect-oriented design. Ph.D. dissertation, Pontif´ıcia Universidade Catolica do Rio de Janeiro.

Whittle J., Araujo J., and Kim D.-K., 2003. "Modelling and Validating Interaction Aspects in UML," presented at AOSD Modelling With UML Workshop (located with UML 2003), San Francisco, USA.

Willem, R. A. , 1990. Design and science, Design Studies, Vol. 11, No. 1, pp. 43–47,1990.

Wimmer, M., Schauerhuber, A. and Kappel, G., 2011.A Survey on UML-Based Aspect-Oriented Design Modelling, ACM Computing Surveys, vol. 43, no. 4, article 28, pp. 28:1–28:33.

Win De, B., Vanhaute, B. and De Decker, B., 2002. How aspect-oriented programming can help to build secure software. Informatica Vol.26(2), pp. 141-149.

Yu Y., Leite J. C. S. d. P., and Mylopoulos J., 2004. From Goals to Aspects: Discovering Aspects from Requirements Goal Models. Presented at International Conference on Requirements Engineering, Kyoto, Japan. pp. 102-111.

Zakaria, A. A., Hosny H.and Zeid, A., 2002. An UML Extension for Modelling Aspect-Oriented Systems, Second International workshop on Aspect-Oriented Modelling with UML at UML 2002, September 30-October 4, 2002, Dresden, Germany.

Zhang, J., Cottenier T., Berg, A. V. D., and Gray, J. 2007d. Aspect Composition in the Motorola Aspect-Oriented Modelling Weaver. In Journal of Object Technology, vol. 6, no. 7, Special Issue: Aspect-Oriented Modelling, pp. 89-108.

Zhu, H. 2005. Software Design Methodology: From Principles to Architectural Styles. Elsevier.

Appendix A: AODL Metamodels

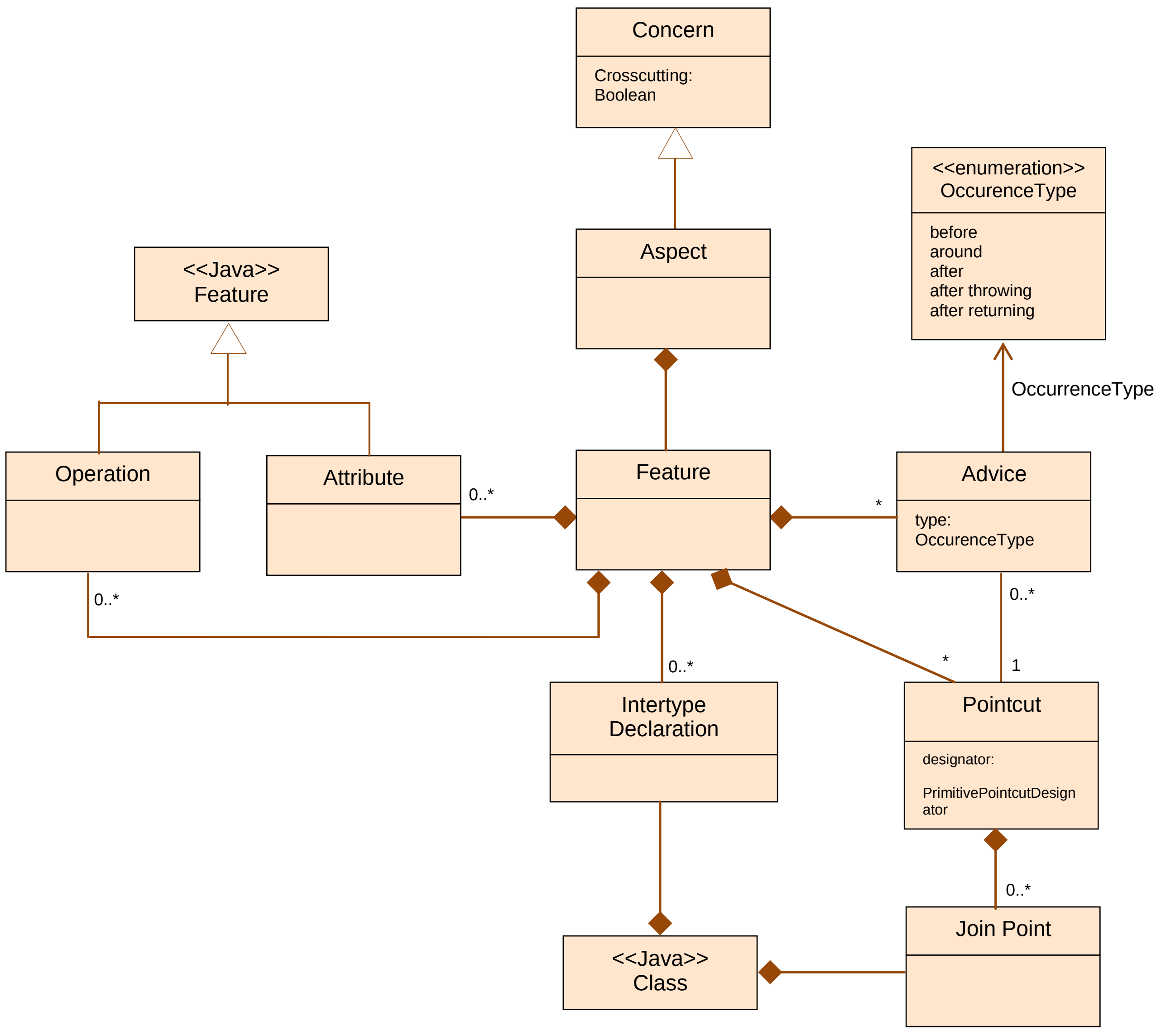


Figure A- 1: Aspect Design Metamodel

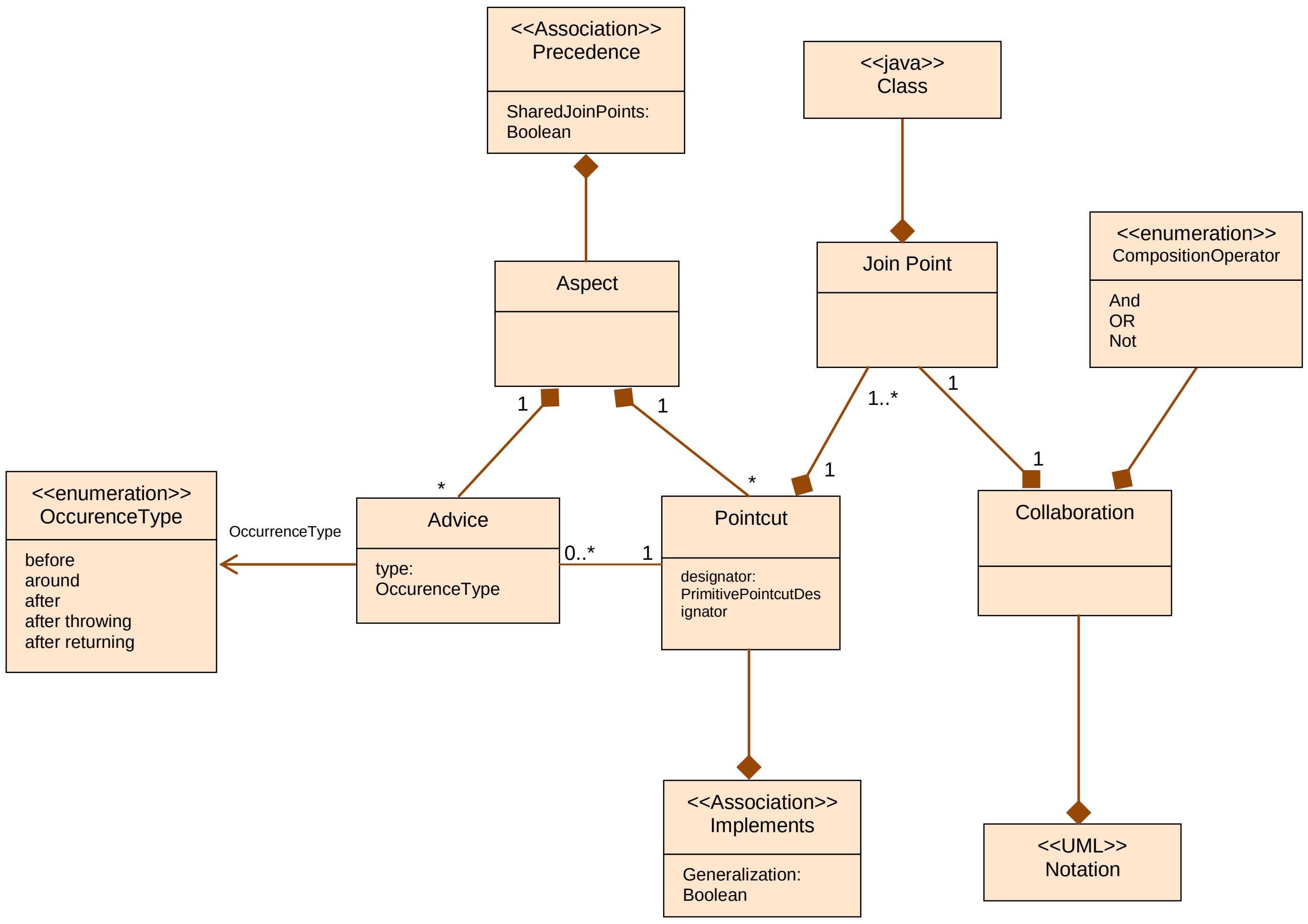


Figure A- 2: Pointcut Composition Metamodel