# REPRESENTING SHARED JOIN POINTS WITH STATE CHARTS: A HIGH LEVEL DESIGN APPROACH

T-4030

*Developed by*

**Muhammad Naveed**
**Muhammad Khalid Abdullah**

*Supervised by*

**Prof. Dr. Khalid Rashid**

**Department of Computer Science**
**Faculty of Basic and Applied Sciences**
**International Islamic University, Islamabad.**
**(2007)**

**International Islamic University, Islamabad**
**Faculty of Applied Sciences**
**Department of Computer Science**

***Dated:*** September 08, 2007

## FINAL APPROVAL

It is certified that we have read the thesis, entitled ***"Representing Shared Join Points with State Charts: a High Level Design Approach"***, submitted by Mr. Muhammad Naveed 53-FAS/MSSE/F04 and Mr. Muhammad Khalid Abdullah 63-FAS/MSSE/F04. It is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University Islamabad for MS Degree in Software Engineering.

## PROJECT EVALUATION COMMITTEE

**External Examiner:**

Dr. Arshad Ali Shahid
Professor (CS)
NU-FAST,
Islamabad, Pakistan.

[illegible]

**Internal Examiner:**

Dr. Naveed Ikram
Department of Computer Science,
Faculty of Basic and Applied Sciences,
International Islamic University,
Islamabad, Pakistan.

[illegible]

**Supervisor:**

Prof. Dr. Khalid Rashid
Department of Computer Science,
Faculty of Basic and Applied Sciences,
International Islamic University,
Islamabad, Pakistan.

[illegible]
8/9/07

*In*
*the*
*Name*
*of*

**ALLAH**

**The Most Merciful**
**The Most Beneficent**

*A Thesis Submitted to the Department of Computer Science, Faculty of Basic and Applied Sciences, International Islamic University, Islamabad, Pakistan, as a Partial Fulfillment of the Requirements for the Award of the Degree of*

**MS in Software Engineering**

*To*
*The Holiest man ever born,*
***PROPHET MUHAMMAD (PEACE BE UPON HIM)***
*& To*
***OUR DEAREST PARENTS & FAMILY***
*Who are an embodiment of diligence and honesty,*
*Without their prayers and support*
*This dream could have never come true*
*& To*
***PRECIOUS FRIENDSHIP***
*That has made us laugh, held us when we cried*
*and always, always, be among us*

# DECLARATION

We hereby declare and affirm that this thesis neither as a whole nor as part thereof has been copied out from any source. It is further declared that we have completed this thesis and accompanied software application entirely on the basis of our personal efforts, made under the sincere guidance of our supervisor. If any part of this report is proven to be copied out or found to be a reproduction of some other, we shall stand by the consequences. No portion of the work presented in this report has been submitted in support of an application for other degree or qualification of this or any other University or Institute of learning.

**Muhammad Naveed**
*53-FAS/MSSE/F04*
**Muhammad Khalid Abdullah**
*63-FAS/MSSE/F04*

# ACKNOWLEDGEMENTS

We bestow all praise to, acclamation and appreciation to Almighty Allah, The Most Merciful and Compassionate, The Most Gracious and the Beneficent, Whose bounteous blessings enabled us to pursue and perceive higher ideals of life, who bestowed us good health, courage and knowledge to carry out and complete our work. Special thanks to His Holy Prophet Muhammad (SAW) who enabled us to recognize our Lord and Creator and brought us the real source of knowledge from Allah, the Qur'an, and who is the role model for us in every aspect of life.

We consider it a proud privileged to express our deepest gratitude and deep sense obligation to our reverend supervisor **Professor Dr. Khalid Rashid** who kept our morale high by his suggestions and appreciation. His motivation leads us to this success without his sincere and cooperative nature and precious guidance; we could never have been able to complete this task.

It would not be out of place to express our profound admiration to **Dr. Farooq Ahmad** for his dedication, inspiring attitude, untiring help, and kind behavior through out the project efforts.

Finally we must mention that it was mainly due to our family's moral support during our entire academic career that enabled us to complete our work dedicatedly. We once again would like to admit that we owe all our achievements to our most loving parents, who mean most to us, for their prayers are more precious then any treasure on the earth. We are also thankful to our truly, sincere and most loving brothers, sisters, friends and class fellows who mean the most to us, and whose prayers have always been a source of determination for us.

**Muhammad Naveed**
*53-FAS/MSSE/F04*
**Muhammad Khalid Abdullah**
*63-FAS/MSSE/F04*

# PROJECT IN BRIEF

| | |
|---|---|
| **Project Title:** | Representing Shared Join Points with State Charts: A High Level Design Approach |
| **Organization:** | International Islamic University, Islamabad, Pakistan. |
| **Objective:** | The objective of the research in this area of Aspect Oriented Software Development is to provide sufficient design support for Share Joint Point Issues related to Aspect Oriented Programming. |
| **Undertaken By:** | Muhammad Naveed<br>*Reg. No. 53-FAS/MSSE/F04*<br>Muhammad Khalid Abdullah<br>*Reg. No. 63-FAS/MSSE/F04* |
| **Supervised By:** | Prof. Dr. Khalid Rashid<br>Department of Computer Science,<br>Faculty of Basic and Applied Sciences,<br>International Islamic University, Islamabad. |
| **Started On:** | January 2006 |
| **Completed On:** | September 2007 |
| **Research Area:** | Aspect Oriented Programming, Aspect Oriented Design, UML |
| **Tools:** | AspectJ, Eclipse 3.2, Rational Rose, MS Access 2003 |

# ABSTRACT

Aspect Oriented Programming promises many advantages at programming level by incorporating the cross cutting concerns into separate units, called aspects. Join Points are distinguishing features of Aspect Oriented Programming as they define the points where core requirements and crosscutting concerns are (inter)connected. Currently, there is a problem of multiple aspects' composition at the same join point, which introduces the issues like ordering and controlling of these superimposed aspects. Some strategies are required to handle these issues as early as possible. State chart is an effective modeling tool to capture behavior at high level design. This thesis provides methodology to formulate the strategies for multiple aspects' composition at high level design, which helps to better implement these strategies at coding level. It also highlights the need of designing shared join point at high level, by providing the solutions of these issues using state chart diagrams in UML 2.0. High level design representation of shared join points also helps to implement the designed strategy in systematic way. A case study is implemented based on the proposed methodology. The results of case study after and before applying proposed methodology are discussed. Future areas related the domains are also highlighted to encourage the researchers.

# TABLE OF CONTENTS

Chapter 1

# INTRODUCTION

# 1. Introduction

Aspect oriented software development (AOSD) is new software development paradigm. It has been evolved since 1997. But it is not an entirely a new software development approach. It is similar the way as object oriented software development was introduced as a new concept in software community but there were programming languages and tools already under use before object oriented design became recognized and adopted by the software community. There have been tools and development environments that support some of the capabilities for some times now that are being categorized under the heading of aspect oriented software development. Like Java, aspect oriented software development is becoming a commonly adopted and standard approach in practicing and implementing the older ideas that can be followed to almost at the beginning of software development [1] [2].

Aspect oriented software development environments and aspect orientation supported tools that weave code, program instructions, and even debuggers all contain some of the behavior that underlies the aspect-oriented approach. The main difference is concepts behind the approach and how the concepts drive the technology and tools. Aspect oriented software development is not about any one of these technologies on its own, though it is a new and more modular implementation of the advantages that these technologies have brought to their own domains in the past [2].

Aspect Oriented Programming (AOP) is a programming paradigm in Aspect Oriented Software Development (AOSD) techniques that describes the cross cutting concerns of a software system in modular way so that the underlying design intent remains clear in the source code [1]. This technique was developed by a team at PARC led by Gregor Kiczales, now a full time professor of computer science at the University of British Columbia [2].

Kiczales et al. introduced first time the idea of separation of concerns [3]. According to them, *"there are many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions that program must implement. This forces the* *implementation of those design decisions to be scattered throughout the code, resulting in "tangled" code that is excessively difficult to develop and maintain. The properties which these decisions address are aspects, they are hard to capture because they cross-cut the system's basic functionality".* They further highlighted the importance of Aspect Oriented Programming by comparing three different types of implementations of a real application that is *"easy to understand but inefficient, efficient but difficult to understand, and an AOP-based implementation that is both easy to understand and efficient. This presentation is based on three analogous but simplified implementations [3]".*

Kiczales et al. introduced the new programming constructs like Join Points, Pointcuts, Crosscutting, and Aspects etc for aspect oriented programming. They also presented of weaving mechanism of aspects and class [3]. They also developed the most powerful general purpose programming language called AspectJ for Aspect oriented Programming, which enables both dynamic and static types of crosscutting behavior [4].

Before going further into the details, it is worth briefly introducing some of the concepts and terms that lie under the concept of aspect orientation.

## Cross-Cutting Concerns

In aspect oriented software development cross-cutting concerns are basic concerns of the system which affect the other concerns in the system. These concerns can not be properly decomposed or modularized from the rest of the system at both design and implementation level. These concerns crosscut the system functionality throughout the program. As a result, there is possibility of either scattering or tangling of the program.

The basic principle of aspect-oriented programming is to enable developers to express modular *cross-cutting concerns* in their software. We can describe cross-cutting concern as *"a behavior, and often data that is used across the scope of a piece of software".* It may be a type constraint that is a characteristic of the software or simply behavior that every class or set of classes of system must perform [2].

The most common example of a cross-cutting concern is logging. "*Logging is almost the "Hello World" of the aspect-oriented approach. It is a cross-cutting concern that affects many areas across the software system and intrudes the business logic. It is potentially applied across many classes of the system, and this is the form of horizontal application of the logging aspect that gives cross-cutting its name [2]*".

## Aspects

Aspects are part of program that actually crosscut the core concerns or main concerns of the system. The term *aspect* is used for cross-cutting concerns and they basically violate the separation of concerns. In other words, aspects are modular units of crosscutting concerns [4]. *"In aspect orientation, aspects provide a mechanism by which a cross-cutting concern can be specified in a modular way"*. Aspects can be of multiple dimensions. They allow both functional and non-functional behavior to cross cut any other concerns. They are not just the mapping of non-functional concerns to functional requirements. In order to get maximum benefits from aspects we need to have some basic concepts in place to allow in specifying and applying aspects in a generic manner. We should be able to [2]:

- *Define the aspects in a modular fashion*
- *Apply aspects dynamically*
- *Apply aspects according to a set of rules*
- *Provide a mechanism and a context for specifying the code that will be executed for that particular aspect*

*"The aspect-oriented approach provides a set of semantics and syntactical constructs to get together these demands so that aspects can be specified and applied generically despite of the type of software being developed. These constructs are advices, join points, and pointcuts [2]"*, and are discussed next.

## Advices

An advice is a mechanism which describes that certain code should be executed again a specific join point in system [4]. In other words, the code that is executed when an aspect is invoked is called *advice* [1] [2]. It may be called as an additional functionality that needs to be performed whenever an aspect is invoked or gets called. An advice contains its own set of rules which specify that when it will be invoked in regard to the join point that has been triggered.

There are currently three forms of advices that AspectJ supports i.e. *before*, *after* and *around advice*. *Before advice* is executed before the core requirement, *after advice* is executed after the core requirement and *around advice* is executed both before and after core requirement.

## Join Points

A join point is where the main program and the aspect interconnect each other. It is very important concept in aspect oriented programming. *Join points* are simply specific points within the application that may or may not invoke some advice. The specific set of available join points is dependent on the tools being used and the programming language of the application under development. The following join points are supported in AspectJ [2]:

- *Join when a method is called*
- *Join during a method's execution*
- *Join when a constructor is invoked*
- *Join during a constructor's execution*
- *Join during aspect advice execution*
- *Join before an object is initialized*
- *Join during object initialization*
- *Join during static initializer execution*
- *Join when a class's field is referenced*
- *Join when a class's field is assigned*
- *Join when a handler is executed*

Among all aspect oriented language constructs, join point is more important. It is defined as *"a well defined execution point in a program [5]"*.

Join points represent the key concept in Aspect-Oriented Software Development (AOSD). Join point also defines the places where two concerns i.e. core and aspectual, crosscut each other [3] [4] [5] [6] [7] [8] [9].

Because of the criticality of specifying join points, the main task of aspect-oriented designers is to identify suitable set(s) of join points, where two concerns interconnect to each other, and also provide some suitable representation for join points [8] [9] [10].

### Pointcuts

*"A pointcut is a set of join points"*. Whenever the program execution reaches one of the join points described in the pointcut, a piece of code associated with the pointcut (called advice) is executed.

P*ointcuts* are the AspectJ mechanism for declaring an interest in a join point to initiate a piece of advice. They encapsulate the decision-making logic that is evaluated to decide if a particular piece of advice should be invoked when a join point is encountered. The concept of a pointcut is crucial to the aspect-oriented approach because it provides an abstract mechanism by which to specify an interest in a selection of join points without having to tie to the specifics of what join points are in a particular application [2].

## 1.1 Shared Join Points

A shared join point is a special type of join point that is being discussed at the moment. In many cases, *"a join point is superimposed by multiple aspects at the same join point"*, known as a shared join point [4] [8] [10]. There are many example scenarios where multiple aspects are being superimposed on the same join point [8] [10] [11]. Currently, there are problems with shared join points at implementation level due to uncertain execution behavior of superimposed aspects [8] [10] [11]. Since multiple aspects are being superimposed, it becomes difficult to judge what will be the exact execution order? If an aspect does not work; how to control the execution order of other aspects? In short, there come issues like ordering and controlling of multiple aspects and are discussed in detail in chapter 2. These problems exist in practice for many years but for the first time they were discussed in research by Nagy et al. in 2005. The issues related to shared join points are discussed only at implementation level [8] [10]. There is not sufficient support available for these issues at implementation level, but there are some indirect support and recommendation details for AOP languages [4] [8] [11]. In normal scenario their behavior is unpredictable. Any aspect can execute before the rest. It requires some support to handle their execution which is currently not available but some indirect support is provided by aspect oriented programming languages. For example, AspectJ provides precedence construct for ordering and do not provide any direct support for controlling [4] [8]. There are few recommendations for new language constructs to control the unpredictable behavior. One of them is proposed by Nagy et al. in 2005. There is some work available on ordering and controlling of aspects in research at implementation level, but it still require some efforts to find better solutions. In other words, still there is a room for research in this area. Since shared join point presents complex scenario, so there is a need to discuss these issues of shared join points at early software development, particularly at high level design. This study proposes a methodology that represents the shared join points in early software development life cycle where it is easy to order and control the multiple aspects. The proposed methodology is discussed in detail in chapter 3.

## 1.2 Software Design in Brief

Software design is an important activity in software development life cycle. The "design" of an application expresses how the application is to be constructed [12]. It describes the parts involved and how they are to be assembled. A design consists of a set of documents: Typically, these are diagrams, together with explanation of what the diagrams mean. A design specification is produced from system requirements. It excludes code. There are many useful notations for design documentation found in the Unified Modeling Language [7].

In software development process, a design phase provides an opportunity for software developers and designers to reason about a required software system as defined by the functional and non functional requirements of system. The objective of this whole activity is to achieve system goals by implying thoughts of necessary behavior, and to model a corresponding structure to support that behavior. An example in object-oriented software system could be, that the behavior of system is taken in terms of interaction diagrams and state diagrams, while the structure can be taken as class and object diagrams by the designers. These diagrams representing system behavior can be of different level of abstractions. The final output of design phase is models of system that specify the behavior of system. These models are normally in the form of design notations [13]. For this, designers require to follow some standard notations for designing their systems as the diagrams or notations need to be communicated or shared with other stakeholders of the system like programmer, client etc. the design notation are normally language construct of graphical languages. The graphical languages are being evolved since 1980s, but most prominent among them is UML (Unified Modeling Language) [14].

## UML

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [7] [13]. The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the fall of 1995, Ivar Jacobson and his Objectory Company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method. The efforts of Booch, Rumbaugh, and Jacobson resulted in the releases of the UML 0.9 and 0.91 documents in June and October of 1996 respectively. Then after different versions, it is the UML 2.0, the latest version of UML. UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. UML plays a very important part in developing object oriented software and the software development process. UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

## 1.3 Aspect Oriented Design

The objectives of Aspect-oriented design (AOD) are identical as that any software design activity. This is to characterize and specify the behavior and structure of the software systems. As Aspect Oriented Software Development (AOSD) is to strive for better modularization of the software systems, so, a unique contribution of Aspect-oriented design (AOD) is to software design relates to extensions to modularity capabilities. In AOSD, some concerns of a software system that are normally scattered and tangled in non-AOD approaches can be modularized in better way. As a result, cohesiveness is enhanced, and module coupling is reduced in corresponding module. An Aspect Oriented Design language provides language constructs that support for modularization of crosscutting concerns regardless of their impacts. It also supports related specification of concern composition. Apart from that, the design of each individual modularized concern is developed in such a way that it reflects software design standards [13].

Like other non AOD approached, the Aspect Oriented Design approach also includes a *process* and a *design language. "An AOD process is one that takes requirements as input and produces a design model that may partially or fully recognize architecture. The AOD process represents separate concerns and relationships between these concerns in AOD design models. These models are specified at an abstract for implementation which can occur on an AOP platform or otherwise [13]"*. An AOD *language* includes constructs that can describe the elements to be used in design. These elements also represent the relationships that can exist between those elements. A specific AOD language constructs provide necessary support to modularization the concerns [13].

Recently, Aspect Oriented Software Development is making strong progress on the implementation level, but the extensive support at design level is still insufficient [15].

## UML and Aspect Oriented Modeling

Since UML provides numerous diagrams to model the software systems, it also provides modeling means for aspect oriented software paradigm as well. One of the main features of UML is stereotype, which helps to modify the objects in the diagrams. The relationship of UML with Aspect Oriented Modeling will be briefly described in chapter 2. UML diagrams, particularly behavioral diagrams can be very useful to describe the behavior of the system. Among these diagrams, State Charts are very important means to model the system behavior. They can help to early identify the problems related to shared join points. The proposed methodology is based on state charts and is given in chapter 3.

## 1.4 Validation of Research

This is an important point about the research in the area of design that how the proposed design methodology will be validated? To validate the proposed methodology, a simple case study of university course registration module is taken. This case study is first implemented without designing or representing the ordering and controlling issues of shared join point, and then with designing these issues. This helped to validate the soundness of proposed study. Detailed functionality of case study is discussed in chapter 4. The research is validated in two steps; firstly, a shared join point scenario will be modeled using proposed methodology and secondly, the modeled shared will be implemented using AspectJ, a prominent Aspect Oriented Programming Language [4].

## 1.5 Thesis Overview

The thesis is structured as: chapter 2 presents the brief literature survey which we had during our research. It also defines the problem statement that is developed during the literature survey. The literature survey of only selected and relevant papers with respect to problem and solution is presented in this chapter; chapter 3 discusses the proposed methodology using state charts of UML. The proposed methodology is discussed at abstract and detail level. In detail level, some example scenarios are discussed and then these scenarios are designed in detail view of the proposed methodology; chapter 4 provides implementation overview of case study implemented on the bases of proposed methodology. Firstly, the proposed methodology is applied on the case study at design level. Secondly, the case study is implemented using AspectJ on the basis of designed solution using proposed methodology. This chapter also describes functional specifications and other design artifacts like architecture and database model of case study and finally, chapter 5 provides results and discussions that include the benefits achieved from the proposed methodology. Finally, it concludes the overall work about the proposed methodology. It also highlights the future work that can be done related to the research presented in this thesis.

# Chapter 2

# LITERATURE SURVEY AND PROBLEM STATEMENT

# 2. Literature Survey and Problem Statement

This chapter provides a brief overview of the literature survey done during this study. It also includes the hypothesis and problem statement based on literature survey. The significance of research in the area of aspect oriented programming and its design is also highlighted.

## 2.1 Literature Survey

A detail literature survey is carried out in the field of aspect oriented programming, its design and state charts of UML.

The programming constructs of AspectJ are introduced by Kiczales et al in 2001. The application of advices of two conceptually and semantically independent aspects at the same join point is addressed. It also described that the programmer does not need to control relative ordering of such advice [4].

The concepts of Join Point as Static Join Point and Dynamic Join Point are addressed. UML association classes (along with their new features), ports and connectors are used among components for modeling [7].

A detailed analysis of the problem aroused by shared join points is discussed by Nagy et al. in 2005. Multiple aspects' superimposition on the same join point affects the functionality of each other due to different execution orders among them. Software engineering perspective of Shared Join Point problems is also discussed. It is recommended that, to offer one solution which satisfies only a single case is not preferred. AOP languages should offer a rich set of language mechanism for composition specifications, so that, the developers may choose the right specification for their problem. It is important to identify conflicts among aspects at shared Joint point for the safety and correctness. The already presented core model [10] is enhanced by adding more constraints and the composition rules for multiple constraints. The integration of the purposed model with AspectJ is also presented. This model can be used with AspectJ, if AspectJ support the named advices. Also the Join Point interface has to be extended for this purpose. For ordering, AspectJ uses declare precedence construct and for controlling, the construct presented in Core Model needs language support [8] [10].

Anis C. et al. (2006) presented an interaction model on the basis of Interaction Specification Language (ISL) for modularizing crosscutting concerns of component based applications. The main idea of interactions is to rewrite a method body using the reaction (advice). The interaction model is used to handle the issues arouse by the Shared Join Point in a way, that the composition mechanism generates an advice, which is the result of merging all advices at that join point. Whenever a shared join point is reached one single advice is executed, which is semantically equivalent to the composed advice. The merging mechanism is based on a finite set of merging rules. The software engineering properties such as analyzability and predictability can also be achieved by using this tool. Testing and verification becomes much simpler [11].

Mahoney et al. (2004) described the importance of extended Finite State Machines in order to capture the dynamic behavior of systems [16]. A state chart is connected to a class that specifies all behavioral aspects of the objects in that particular class. They also describe that Aspect Oriented Modeling which can help in bridging the gap between software design and implementation through the use of advanced features of state charts. They have proposed a framework which helps in simplifying the design of core requirements and cross cutting concerns.

Mahoney also elaborated the need of crosscutting concerns of reactive systems using state machines. State Charts are used to describe the dynamic behavior of separate concerns. The core and aspectual requirements are represented by state in different orthogonal regions. He addressed the communication mechanism in orthogonal regions through broadcast events. The broadcast events are used as a mechanism for implicit weaving of aspect and core model in state charts [17].

Mohamed Mancona Kande et al. (2001) explained the basic concepts of AspectJ, a state of the art Aspect Oriented Programming Language. Standard UML is used for modeling these concepts and limitations of UML are highlighted. Some extensions to UML are proposed to overcome these limitations. A bottom up approach is followed for designing classes and aspects of Aspect Oriented Programming [18].

Stein D. et al. (2003 and 2004) presented an approach to model the join points with the help of Join Point Indication Diagram (JPID) and Join Point Designation Diagram (JPDD). JPID is presented for the indication of join points in core model while JPDD is presented for the indication of join points in aspects, but it does not address any solution for Shared Join Points [9] [15] [20].

Wesley Coelho and Gail C. Murphy (2004) presented a dynamic aspect modeling. The use colors for representing interactions, "+" sign for dynamic expansion relationships and arrow icons are used to represent crosscutting phenomena [21].

A. Rashid et al. (2002) presented a model to capture Aspect-Oriented Requirement. They argue that "a non-functional requirement is crosscutting if it transverse more than one use-cases". UML1.4 diagrams (Use case model, interaction Diagrams and scenarios) are used to handle separation of cross cutting concerns (aspects) from core model at requirements level. The focus of Separation of concern is only on non-functional requirements as they argue that non functional requirements are global properties of the system (i.e. Performance) [22]. Although presented methodology is very excellent for Capturing Crosscutting requirements but the good old experience is always an essential requirement for the identification of Aspects. The identification of Aspects in functional requirements is not presented here that can be used as a future research area.

A. Rashid et al. (2005) highlighted the fundamental basis for a high level Join Point Model based on state transitions and thus a Join Point Model is presented using UML state machines. A case study of "safety-critical automatic driverless train system" is discussed in which states of the system are represented in use case diagram. The State-based Join Points are categorized as dependency-based Join Points, Scope-based Join

Points and Transition Flow Join Points. The importance of design, the importance of Join Point Model, and also the importance of State-based Join Point model is discussed [23].

There is massive work on modeling, modeling in join points as well as on aspect oriented programming where as the work on shared point is only at implementation level. There is no solution presented by the researchers to represent particular issues of shared join point at high level in formulation of the suitable design strategies for shared join points.

## 2.2 Hypothesis

Since there is lack of research in the field of aspect oriented design, though aspect oriented programming is making strong progress at implementation [15]. Hypothesis that we establish is to model features of aspect oriented programming, particularly at high level design. For this reason, we have had most of the literature survey about modeling of different elements of aspect oriented programming like aspects, pointcuts, advices, join points. After sufficient literature survey we find that there is need to model the join points though there are some join point model available in research at the moment. This is important to model join points because it identify the places where both core and aspectual requirements interconnect each other. Join points represent the key concept in Aspect-Oriented Software Development (AOSD) as Join Point is defined as a well defined execution point in a program [5]. The main task of aspect-oriented designers should be to identify set(s) of join points, where two concerns i.e. core and aspectual requirements interconnect to each other, and provide suitable representation for join points [8] [9] [10]. That is why importance of designing of the join points is very important. So, we establish our hypothesis in designing join points of aspect oriented programming. Further we found that there is specialized type of joint points called shared join points, which is having new dimension in the research at the moment. Our hypothesis further directs toward the designing of shared join points. Shared join points are novel in research community. A shared join point is a point in the execution of system where multiple aspects are superimposed on the same join point. A shared join point raises the issues like ordering and controlling. There is no direct support available to handle these issues at implementation level, but there are some indirect support and recommendation details for AOP languages available [4] [8] [11]. For example, AspectJ provides precedence construct for ordering and do not provide any direct support for controlling [4], We assume that there should be a solution for these issues of ordering and controlling of shared join points at early software development stages. Hypothesis we establish is to specify any methodology to model the shared join point as early as possible in software development lifecycle, particularly at high level design. We also assume that it could reduce the development cost by identifying the issues regarding shared points like ordering and controlling. It also includes that methodology or proposed solution should be so simple that it can help designer to specify the shared join points and also could help the programmer to easily implement and solve the issues with least development cost. It was also assumed that state chart diagrams of UML could be helpful in this regard as the elements of the state charts could provide the solutions of the shared join point issues like ordering and controlling by modeling it at early design stages, because state charts are very good modeling diagram for representing the behavior of a complex scenario.

## 2.3 Problem Statement

In many cases, a join point is superimposed by multiple aspects at the same time, known as a shared join point [4] [8] [10]. There are many example scenarios, where multiple aspects are being superimposed on the same join point [8] [10] [11]. An example is shown in Figure 2.1.

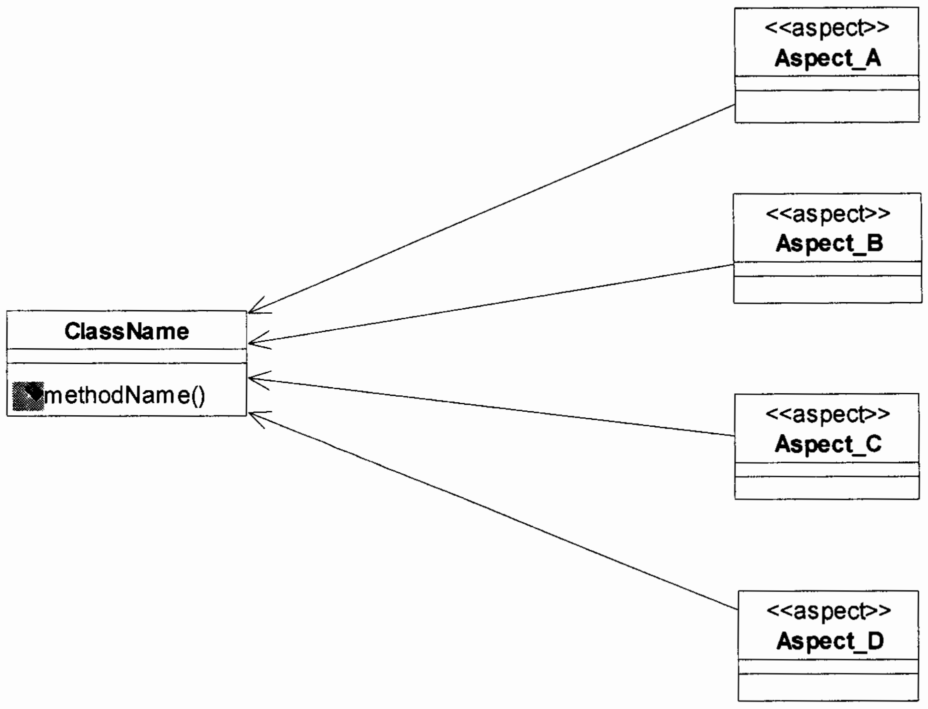

**Figure 2.1 Multiple aspects superimposed on the same Join Point**

In the figure 2.1, multiple aspects i.e. *Aspect_A*, *Aspect_B*, *Aspect_C*, and *Aspect_D* are superimposed on *methodName()* method. They are required to execute against *methodName()* method call. Since aspects are executed regardless how they are created? i.e. their execution sequence is unpredictable and they execute in any sequence at programming level. And also how much dependencies exist among them? If we see in the figure above, we can find that all the aspects are sharing a single join point, which is now called as shared join point. At this shared join point there are two problems one is ordering of the multiple aspects which share this join point and second is the controlling of these aspects. Both problems are discussed one by one in detail.

Requirement of ordering the aspects is that they should run in the following order; first *Aspect_A* aspect should execute and then aspect *Aspect_B* should be executed after *Aspect_B* is executed successfully then *Aspect_C* should be executed and at the end *Aspect_D*. This is the ordering of multiple aspects on shared join point. There is some language support available at programming level like *precedence* construct in AspectJ for ordering of multiple aspects, but this requirement of ordering should be modeled or specified at early software development stages i.e. at analysis or design phase. So, that developer should know about the exact execution sequence of aspectual requirements? If they are unable to know the exact flow, there will be an assumption based implementation of these aspectual requirements, which might create a situation where developer do not be able to implement the business logic according to requirements. Finally, the developers will have to implement those aspectual requirements again according to the actual requirements. This can increase the cost of development. There is a great need to find some mechanism at early software development life cycle to represent these aspects in the specific order that require to be implemented by developers. Currently, the issues of shared join points are discussed at programming level only not at early software development stages. Since shared join points presents a scenario which should be according to the requirements specified, so they need to be address as soon as possible.

Now, we discuss the controlling of multiple aspects. There are dependencies among the aspects like if *Aspect_B* and *Aspect_C* is dependant on *Aspect_A* and *Aspect_D* is dependant on the *Aspect_A, Aspect_B* and *Aspect_C.* we need to control them on the basis of their outputs. For example, if *Aspect_A* is executed successfully then the entire aspects dependant on *Aspect_A* should be executed, and if *Aspect_A* is not executed successfully then all the aspects dependant on *Aspect_A* should not be executed. Similarly, if *Aspect_D* is dependant on *Aspect_A, Aspect_B* and *Aspect_C,* then *Aspect_D* should be executed only if all the aspects are executed successfully and if any of the aspects does not run successfully then *Aspect_D* should not be executed. Here comes the issue of controlling of multiple aspects on the shared join point. It raises the question that how to stop the execution of the aspects which depends on other aspects? In normal scenario all superimposed aspects will execute independent of each other, even though they have dependency with respect to system functionality. To make them execute according to the requirement of system, they need to be controlled. There is not sufficient support available for this issues at implementation level i.e. aspect oriented language provide any keyword to handle the controlled execution of multiple aspects. Like, AspectJ provides precedence construct for ordering and do not provide any direct support for controlling [4]. These languages' constructs help in solving the issues regarding shared join point in some context, but there is need to find suitable keyword which could provide direct support for ordering and controlling of multiple aspects. There can be some indirect mechanism available to solve the controlling problem of multiple aspects as programming level. For example some boolean variables can use to get the outputs of aspects and these outputs can be evaluated by conditional statements like *"if"* statement in programming languages. So, we can say that it requires some strategies to implement the multiple aspects on shared join point in controlled manner. And these strategies should be formulated at early software development level like at analysis of design phase of software development life cycle. But there is no research work related to this issue at early software development life cycle that how to represent multiple aspects related to a shared join point in controlled manner.

So, currently no work has been done for shared joint points' issues like ordering and controlling discussed earlier at design level. The issues identified by Nagy et al. in 2005, are still being discussed at programming level. Some recommendations and suggestions in software engineering perspective are discussed, but no discussion is made about these issues at design level. Since these issues affect the functionality of the system [10], so they need to be discussed at early software development in order to find suitable solutions to handle the issues. These issues at later stages of development could cause increase in time and cost.

## 2.4 Significance of the Research

Research in this area of aspect oriented software development (AOSD) paradigm will help greatly to improve the aspect oriented design by early identifying and providing the solutions of shared join point issues, which does not have direct solution available even at implementation stages. The proposed study will help designer to design the complex issues regarding shared join point at an early stage. This early representation of such issues will help to formulate the strategies for their solution. As a result the programmers will be able to implement these solutions easily. The research in this area will bring more improvements in the aspect oriented design which lacks support at the moment. This will also help to recommend some keywords for aspect oriented programming languages to solve the issues related to shared join point. This study will make aspect oriented paradigm more powerful by having strong language support for critical issues of shared join points in programming languages as well as in design.

# Chapter 3

# PROPOSED METHODOLOGY

# 3. Proposed Methodology

This chapter explains the proposed methodology in detail. The proposed methodology is based on the state charts of the unified modeling language (UML) 2.0 to model the shared join points. This methodology can be used at two levels i.e. abstract and detail.

The Unified Modeling Language (UML) has become industry standard for modeling general as well as for specific purpose software artifacts. State charts in UML 2.0 are very important means of modeling and capturing the dynamic behavior of objects [14]. State chart related to a class, can specify all the behavioral aspects in that class [16]. A state chart diagram is represented through a state machine which models the individual behavior of the object. State machines throughout the UML versions remained almost the same [19]. However, some new elements like entry and exit point are introduced in UML 2.0 [14]. Some of the elements of UML 2.0 like, composite state, choice pseudo-state and terminate pseudo-state are very important means to model the dynamic behavior. Terminate pseudo-state construct is not available in the previous versions of UML.

## 3.1 Abstract View of Proposed Methodology

The proposed design methodology mainly uses composite state language elements of UML state charts for the multiple aspects' composition at the shared join point. The methodology at abstract view consists of three main composite states. Composite states are different from simple states in the way that they could contain multiple sub states i.e. they could be composed of multiple sub states. In the proposed methodology these composite states are composition of the aspectual and core requirements i.e. they compose both aspectual and core requirements of system in composite states. A composition state which is composed of core requirement is called *coreCompositeState* as shown in figure 3.1. The aspectual requirements are composed by aspectual composite states. Since, aspectual requirements are further subdivided into two different types i.e. the aspectual requirements that need to implement before the core requirement and the aspectual requirements that need to implement after the core requirements. There is another type of aspectual requirements that can be executed both before and after the core requirements at the same. They are normally implemented by the around device in aspects. This type of aspectual requirement is not modeled in this proposed methodology. It can be one of the future areas to work on. The proposed methodology is used to model on those aspectual requirements that will be either executed before the core requirement or after the core requirements. The composite states that compose these requirements are also subdivided into two categories i.e. *beforeCompositeState* and *afterCompositeState.* The multiple aspects that superimposed before the core requirement, their ordering and controlling can be handled in *beforeCompositeState* and the multiple aspects that are superimposed after the core requirement, their ordering and controlling can be handled in *afterCompositeState* in the proposed solution. This means that each composite state is responsible for handling issues related to superimposed aspects contained by that composite state. The core requirement is composed in the *coreCompositeState*. These composite states are named as just for understanding, they can be named whatever is convenient to understand the concept. At the abstract level, the aspectual and core requirement compositions are handled in composite states. The big picture of proposed methodology is shown in figure 3.1.

At the abstract level, proposed methodology orders multiple aspects by placing them in their respective composite state. Further more, the aspects in these composite states can be ordered by placing in sequence they will be executed for a specific core requirement. The proposed methodology controls the multiple aspects by evaluating their boolean guard values at choice pseudostate as shown in figure 3.1. Boolean guard values are evaluated as *"true"* or *"false"*, if it is *"true"* then next aspect or requirement will be executed and if it is *"false"* the object will be discarded. There is a terminate pseudostate in state charts of UML 2.0 for this purpose.

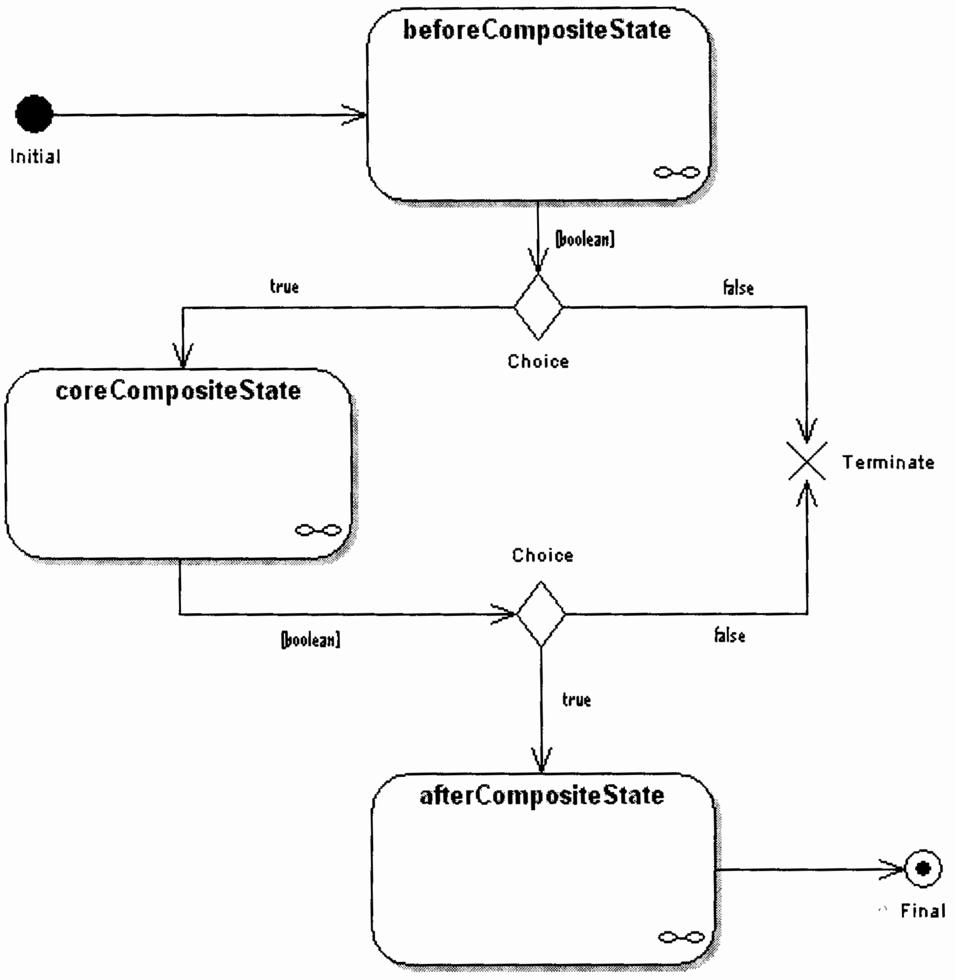

**Figure 3.1 Abstract view of proposed methodology**

## 3.2 Detailed View of Proposed Methodology

We further detail the proposed methodology by composing the aspects in their relevant composite states. In detail view of proposed methodology guard value from each step will be evaluated and on the basis of this decision to execute next aspect will be taken. For this, we need to discuss the core requirement and aspectual requirements described earlier in problem statement of chapter 2. On the basis of problem discussed in chapter 2, we assume some requirement scenarios to represent the detail view of the proposed methodology. This also includes formulating strategy against a specific requirement scenario which represents dependencies among the multiple aspects. For this purpose, two example scenarios are discussed. These scenarios are with respect to core and aspectual requirements of the system. They also represent the strategies required to handle the issues of ordering and controlling. We explain the proposed methodology in detail in the context of the problem defined in chapter 2.

The main requirement of the system is to implement *methodName()* method of class *ClassName*, but apart from that, the functionality of this method should not be completed unless it fulfills the other requirements that can be said as pre-requisite or post-requisite requirements. These requirements can be treated as aspectual requirements in the context of problem. These requirements are handled in *Aspect_A, Aspect_B Aspect_C,* and *Aspect_D.*

Let us suppose that *Aspect_A*, *Aspect_B* and *Aspect_C* are should be executed before the core requirement i.e. *methodName()* and *Aspect_D* should run after the core requirement. Now, we can see that three aspects i.e. *Aspect_A*, *Aspect_B* and *Aspect_C* are superimposed on the shared join point i.e. *methodName()* that have dependencies among them as shown in Figure 3.2.

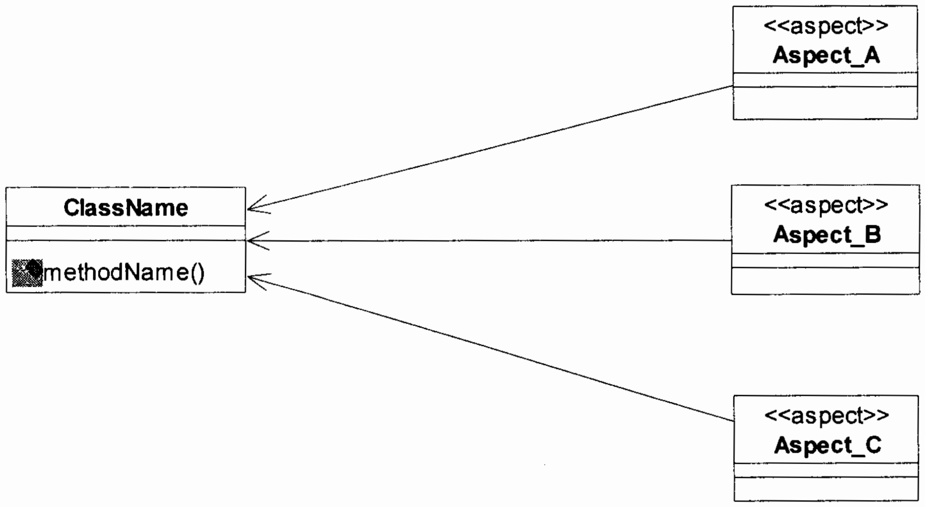

**Figure 3.2 Conflicting Aspects in Shared Join Point that require ordering and controlling**

They present serious issues of ordering and controlling these superimposed aspects. The *Aspect_D* will be executed after the core requirement so there should be no problem of ordering and controlling related to this aspect as it does not have any dependency with other aspects.

### 3.2.1 Example Scenario 1

Scenario one is that *Aspect_A* should run always first, then *Aspect_B,* and then *Aspect_C.* Now, this defines dependencies among the aspects. Here comes the issue of ordering of these aspects i.e. to make sure that *Aspect_A* should always run first. This needs to be model in this way that the design clearly represents that *Aspect_A* will be executed first. Since *Aspect_A* will be executed first, the other two aspects are depending upon the output of *Aspect_A* i.e. if the *Aspect_A* does not produce desired results, the *Aspect_B* should not be executed. It raises another issue i.e. the controlling of dependant aspects i.e. if execution decisions about the aspects that are dependant upon the output of another aspect or aspects.

A strategy formulated for scenario one is shown in figure 3.3. Here, we can see that the logical flow describe ordering of conflicting aspects which further describes that *Aspect_A* will always be executed first. A boolean guard value will be passed to choice pseudostate. This choice pseudostate will evaluate the boolean value. If the boolean guard value is true then *Aspect_B* will be executed, and if the boolean value is false then the object will be discarded and no further aspect or requirement will be executed. But, if the value is evaluated as true then *Aspect_B* is executed. The boolean guard value after the execution of *Aspect_B* will be evaluated and decision will be taken about next aspect's execution. Ultimately, we can execute core requirement after the controlled and successful execution of multiple aspects. In this way, we can handle controlling issue by applying proposed design methodology.

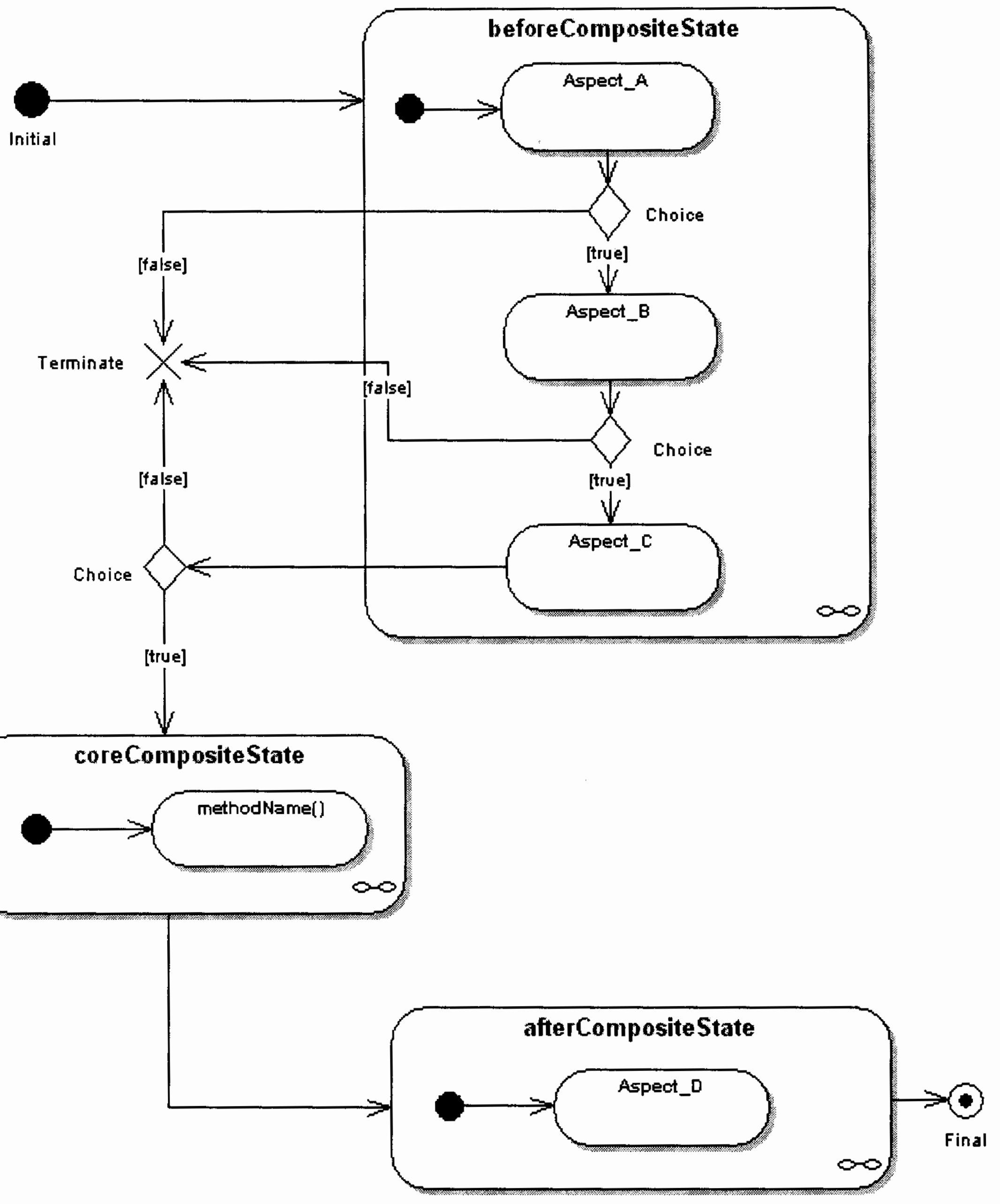

**Figure 3.3 Detail view of proposed methodology with scenario one**

### 3.2.2 Example Scenario 2

Suppose scenario two is that *Aspect_A* should run always first then *Aspect_B* and *Aspect_C* can be executed simultaneously or in any order. Here we can see that there is dependency between *Aspect_A* and two other aspects i.e. *Aspect_B* and *Aspect_C.* We again need to order these aspects. But one thing is different as compared to the previous scenario that we don't need ordering between *Aspect_B* and *Aspect_C* as they can be executed in any order or in parallel. Also, we require controlling of aspect in a different way now.

A strategy formulated for scenario two is shown in figure 3.4. Again we can see that the logical flow describe ordering of conflicting aspects, which further describes that the *Aspect_A* will always be executed first. In this scenario a new composite state name *synCompositeState* is introduced which contains *Aspect_B* and *Aspect_C.* The reasoning of introducing this state is that it can execute the two aspects in parallel and the result of both aspects will be evaluated at choice pseudostate after being passing through join element of state charts. In this scenario, aspects' execution policy can be elaborated as first of all *Aspect_A* will be executed. If it returns true value then the control will be passed to next composite state that is *synCompositeState.* And if it returns false value then the object will be directed toward discard state which destroys the object. Since the *Aspect_B* and *Aspect_C* will be executed in parallel, so their outputs will be joined at join pseudo state. After the both aspects return true value only then their joint result will be true. And if any of the two aspects return false value then their joint result will be false and ultimately when value will be evaluated at choice pseudostate then object will be discarded. In case, their joint result is true then core requirement will be executed and hence before the core requirement the entire aspects are ordered and controlled as per the required strategy.

Since the *Aspect_D* is supposed to run after the core requirement, there is no need to order or control this aspect. This representation allows the designers to show a high level ordering and controlling mechanism for superimposed aspects. It also shows how to resolve ordering and controlling issues? Detailing proposed methodology on different scenarios represents that it provides flexibility to designers to apply any of the strategies given in [9] at high level design. By selecting a suitable strategy to resolve the issue at high level design will help the programmers to implement the strategy in an ideal way. It provides additional features of handling the new aspectual requirement in systematic way. The proposed design solution of shared join point allows the designer to represent shared join point independent of the implementation details i.e. at abstract level which is one of the most important features while modeling join points [22].

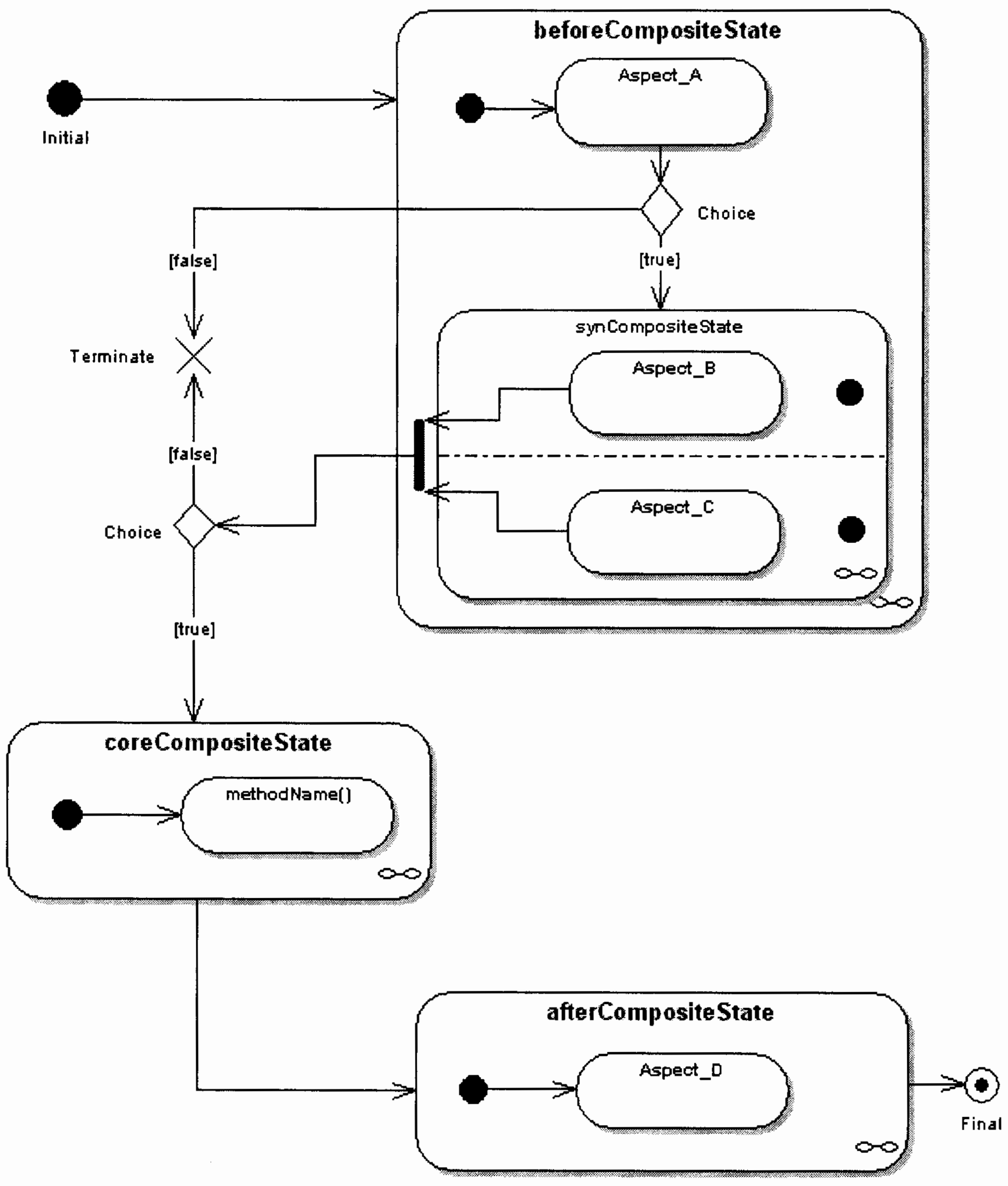

**Figure 3.4 Detail view of proposed methodology with scenario one**

The design at such an abstract level can provide benefits like scalability, by representing new conflicting aspects in the model, reusability, reusing the design strategy for other similar shared join points, and maintainability, if any of the conflicting aspects to be removed or to be added by identifying its effect on the other aspects. Early representation of ordering and controlling issues reduces the work cost by identifying the complexity of issues and formulating suitable strategies to solve the issues accordingly. Early the problem is identified cheaper to solve.

# Chapter 4

# IMPLEMENTATION OF CASE STUDY

# 4. Implementation of Case Study

In order to provide the evidence of the soundness of proposed design methodology, a case study has been considered and implemented. The case study is about the university registration module. This chapter provides detail discussion about the case study with respect to design using proposed methodology presented in chapter 3 and with respect to implementation in AspectJ programming language. The case study has been explained in section 4.1. In order to fulfill the registration requirements some other prerequisites are needed to be developed as well. The main scope of application is to handle the multiple aspects on *registerCourse()* method as per proposed design methodology. This chapter also describes some of the functional requirements, architecture diagram, collaboration diagrams, class diagram and database model of the case study. It discusses the ordering and controlling issues in design and then in implementation perspective of the case study.

## 4.1 Functional Requirements of the Case study

A simple case study about the university course registration system has been selected. Only the requirements that are relevant to shared join point are discussed. The functional requirements are divided into core and aspectual requirements, which are discussed in the following sections:

### Core Requirement

Course registration is the main requirement of the system which will be implemented by *registerCourse()* method, but apart from that, a course can not be registered unless it fulfills the other requirements like fee submission checking whether pre requisite courses are also passed or not? These are the aspectual requirements which are discussed in next section.

## Aspectual Requirements

Following aspectual requirements must be performed in order to register courses for a student successfully:

***Requirement 1:*** System should be able to maintain log of each method invoked. This aspectual requirement is implemented by *Logging* aspect.

***Requirement 2:*** Before the course registration, system must check whether the fee has been paid or not? This Requirement is implemented by *CheckFee* aspect.

***Requirement 3:*** Before the course registration system must check whether the pre requisites of the course being registered have been satisfied or not? This is implemented by *CheckPreRequisite* aspect.

***Requirement 4:*** This requirement is to check whether the course has been registered and database is updated or not? This is implemented by *DBPersistance* aspect.

In objected oriented systems functional requirement are described by use cases. Aspectual requirements can be represented in form of use cases like functional requirements [24]. The above discussed aspectual requirements and other functional requirements of the case study are described in the form of use cases. The aspectual requirements are shown in filled uses in diagram 4.1. This section first describes the use cases in the form of use case specifications and then all the use cases are shown in the form of diagram.

### 4.1.1 Use Case Specifications

Use Case specification describes how a specific scenario will be performed by system.

| | |
|---|---|
| **GUI Screen Reference:** | Add Faculty Screen |
| **Use Case ID:** | UC-01 |

| | |
|---|---|
| **Use Case Name:** | Add Faculty |
| **Actor(s):** | Administrator |
| **Goal:** | Administrator should be able to add new Faculty. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

| **Actor Actions:** | | **System Response:** | |
|---|---|---|---|
| 1 | User selects Faculty from main menu. | 2 | System displays menu items. |
| 3 | User selects Add Faculty from menu item. | 4 | System displays Add Faculty screen. |
| 5 | User enters following information:<br>• Faculty Code,<br>• Faculty Name,<br>• Faculty Location,<br>• City,<br>• Country,<br>• Faculty Notes. | 6 | System takes input data. |
| 7 | User presses Add button. | 8 | System saves record and shows message. |

**Alternative Course of Actions:**

| | | | |
|---|---|---|---|
| 5a | If Faculty Code or Faculty Name is not entered. | 6a | System displays an error message. |
| 7a | User presses Close button. | 8a | System closes Add Faculty screen. |

**Post Conditions:**
New Faculty has been added successfully.

**GUI Screen Reference:** Add Department Screen

| | |
|---|---|
| **Use Case ID:** | UC-02 |
| **Use Case Name:** | Add Department |
| **Actor(s):** | Administrator |
| **Goal:** | Administrator should be able to add new Department. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

| **Actor Actions:** | | **System Response:** | |
|---|---|---|---|
| 1 | User selects Department from main menu. | 2 | System displays menu items. |
| 3 | User selects Add Department from menu item. | 4 | System displays Add Department screen. |
| 5 | User Selects Faculty Name from List. | 6 | System displays selected faculty from list. |
| 7 | User enters following information:<br>• Department Code,<br>• Department Name,<br>• Department Location, | 8 | System takes input data. |

- City,
- Country,
- Department Notes.

| | | | |
|---|---|---|---|
| 9 | User press Add button. | 10 | System saves record and shows message. |

**Alternative Course of Actions:**

| | | | |
|---|---|---|---|
| 5a | If Faculty Name is not selected. | 6a | System displays an error message. |
| 7c | If Department Code or Department Name is not entered. | 8a | System displays an error message. |
| 9a | User can press Close button. | 10a | System closes Add Department screen. |

**Post Conditions:**
New Faculty has been added successfully.

**GUI Screen Reference:** Add Program Screen

| | |
|---|---|
| **Use Case ID:** | UC-03 |
| **Use Case Name***:* | Add Program |
| **Actor(s):** | Administrator |
| **Goal:** | Administrator should be able to add new Program. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

| **Actor Actions:** | | **System Response:** | |
|---|---|---|---|
| 1 | User selects Program from main menu. | 2 | System displays menu items. |
| 3 | User selects Add Program from menu item. | 4 | System displays Add Program screen. |
| 5 | User Selects Faculty Name from List. | 6 | System displays corresponding List of Department Names. |
| 7 | User Selects Department Name from List | 8 | System displays selected department from list. |
| 9 | User enters following information: • Program Code, • Program Name, • Program Duration, • Program Notes. | 10 | System takes input data. |
| 11 | User press Add button | 12 | System saves record and shows message. |

**Alternative Course of Actions:**

| | | | |
|---|---|---|---|
| 5a | If Faculty Name is not selected. | 6a | System displays an error message. |
| 7a | If Department Name is not selected | 8a | System displays an error message. |
| 9a | If Faculty Code or Faculty Name is not entered. | 10a | System displays an error message. |
| 11a | User can press Close button. | 12a | System closes Add Program screen. |

**Post Conditions:**

New Program has been added successfully.

**GUI Screen Reference:** Add Batch Screen

| | |
|---|---|
| **Use Case ID:** | UC-04 |
| **Use Case Name:** | Add Batch |
| **Actor(s):** | Administrator |
| **Goal:** | Administrator should be able to add new Batch. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

| **Actor Actions:** | | **System Response:** | |
|---|---|---|---|
| 1 | User selects Batch from main menu. | 2 | System displays menu items. |
| 3 | User selects Add Batch from menu item. | 4 | System displays Add Batch screen. |
| 5 | User Selects Faculty Name from List. | 6 | System displays corresponding List of Department Names. |
| 7 | User Selects Department Name from List | 8 | System displays corresponding List of Program Names. |
| 9 | User Selects Program Name from List | 10 | System displays selected program from list. |
| 11 | User enters following information: • Batch Code, • Batch Name, • Program Notes. | 12 | System takes input data. |
| 13 | User press Add button | 14 | System saves record and shows message. |

**Alternative Course of Actions:**

| | | | |
|---|---|---|---|
| 5a | If Faculty Name is not selected from List. | 6a | System displays an error message. |
| 7a | If Department Name is not selected from List. | 8a | System displays an error message. |
| 9a | If Program Name is not selected from List. | 10b | System displays an error message. |
| 11a | If Batch Code or Batch Name is not entered. | 12a | System displays an error message. |
| 13a | User can press Close button. | 14a | System closes Add Batch screen. |

**Post Conditions:**
New Batch has been added successfully.

**GUI Screen Reference:** Add Course Screen

| | |
|---|---|
| **Use Case ID:** | UC-05 |
| **Use Case Name:** | Add Course |
| **Actor(s):** | Administrator |

|  |  |
|---|---|
| **Goal:** | Administrator should be able to add new Course. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

| **Actor Actions:** |  | **System Response:** |  |
|---|---|---|---|
| 1 | User selects Course from main menu. | 2 | System displays menu items. |
| 3 | User selects Add Course from menu item. | 4 | System displays Add Course screen. |
| 5 | User Selects Faculty Name from List. | 6 | System displays corresponding List of Department Names. |
| 7 | User Selects Department Name from List | 8 | System displays corresponding List of Program Names. |
| 9 | User Selects Program Name from List | 10 | System displays selected program from list. |
| 11 | User enters following information: • Course Code, • Course Name, • Course Credit Hours, • Course PreRequisite, • Course Fee, • Course Notes. | 12 | System takes input data. |
| 13 | User press Add button | 14 | System saves record and shows message. |

**Alternative Course of Actions:**

|  |  |  |  |
|---|---|---|---|
| 5a | If Faculty Name is not selected from List. | 6a | System displays an error message. |
| 7a | If Department Name is not selected from List. | 8a | System displays an error message. |
| 9a | If Program Name is not selected from List. | 10a | System displays an error message. |
| 11a | If Course Code or Course Name is not entered. | 12a | System displays an error message. |
| 13a | User can press Close button. | 14a | System closes Add Course screen. |

**Post Conditions:**
New Course has been added successfully.

**GUI Screen Reference:** Add Student Screen

|  |  |
|---|---|
| **Use Case ID:** | UC-06 |
| **Use Case Name:** | Add Student |
| **Actor(s):** | Administrator |
| **Goal:** | Administrator should be able to add Student. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

**Actor Actions:** **System Response:**

| | | | |
|---|---|---|---|
| 1 | User selects Student from main menu. | 2 | System displays menu items. |
| 3 | User selects Add Student from menu item. | 4 | System displays Add Student screen. |
| 5 | User Selects Faculty Name from List. | 6 | System displays corresponding List of Department Names. |
| 7 | User Selects Department Name from List | 8 | System displays corresponding List of Program Names. |
| 9 | User Selects Program Name from List | 10 | System displays corresponding List of Batch Numbers. |
| 11 | User Select Batch Number from List. | 12 | System displays selected batch from list. |
| 13 | User enters following information: • Student Name, • Father Name, • Registration No, • Date of Joining, • Date of Birth, • ID Card No, • Address, • City, • Country, • Email Address, • Notes. | 14 | System takes input data. |
| 15 | User press Add button | 16 | System saves record and shows message. |

**Alternative Course of Actions:**

| | | | |
|---|---|---|---|
| 5a | If Faculty Name is not selected from List. | **6a** | System displays an error message. |
| 7a | If Department Name is not selected from List. | 8a | System displays an error message. |
| 9a | If Program Name is not selected from List. | 10a | System displays an error message. |
| 11a | If Batch Number is not selected from List. | 12a | System displays an error message. |
| 13a | If Student Name or Registration No is not entered. | 14a | System displays an error message. |
| 15a | User can press Close button. | 16a | System closes Add Student screen. |

**Post Conditions:**
New Student has been added successfully.

**GUI Screen Reference:** Submit Fee Screen

**Use Case ID:** UC-07

| | |
|---|---|
| **Use Case Name:** | Submit Fee |
| **Actor(s):** | Administrator |
| **Goal:** | Administrator should be able to submit Fee for a particular course against the student. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

| **Actor Actions:** | | **System Response:** | |
|---|---|---|---|
| 1 | User selects Fee from main menu. | 2 | System displays menu items. |
| 3 | User selects Submit Fee from menu item. | 4 | System displays Submit Fee screen. |
| 5 | User Selects Faculty Name from List. | 6 | System displays corresponding List of Department Names. |
| 7 | User Selects Department Name from List | 8 | System displays corresponding List of Program Names. |
| 9 | User Selects Program Name from List | 10 | System displays corresponding Lists of Batch Numbers and Course Titles |
| 11 | User Selects Batch Number from List. | 12 | System displays corresponding List of Student Names. |
| 13 | User Selects Student Name from List | 14 | System displays selected name from list. |
| 15 | User Selects Course Title from List | 16 | System display Course Fee in read only format. |
| 17 | User Select Is Fee Submitted Status from List. | 18 | System displays selected option from list. |
| 19 | User press Add button | 20 | System saves record and shows message. |
| **Alternative Course of Actions:** | | | |
| 5a | If Faculty Name is not selected from List. | 6a | System displays an error message. |
| 7a | If Department Name is not selected from List. | 8a | System displays an error message. |
| 9a | If Program Name is not selected from List. | 10a | System displays an error message. |
| 11a | If Batch Number is not selected from List. | 12a | System displays an error message. |
| 13a | If Student Name is not selected from List. | 14a | System displays an error message. |
| 15a | If Course Title is not selected from List. | 16a | System displays an error message. |
| 17a | If the fee is already submitted for the selected course. | 18a | System displays an error message. |
| 19a | User can press Close button. | 20a | System closes Submit Fee screen. |

**Post Conditions:**

Fee has been added successfully.

**GUI Screen Reference:** Add Result Screen

| | |
|---|---|
| **Use Case ID:** | UC-08 |
| **Use Case Name:** | Add Result |
| **Actor(s):** | Administrator |
| **Goal:** | Administrator should be able to add Result for a particular course against the student. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

| **Actor Actions:** | | **System Response:** | |
|---|---|---|---|
| 1 | User selects Result from main menu. | 2 | System displays menu items. |
| 3 | User selects Add Result from menu item. | 4 | System displays Add Result screen. |
| 5 | User Selects Faculty Name from List. | 6 | System displays corresponding List of Department Names. |
| 7 | User Selects Department Name from List | 8 | System displays corresponding List of Program Names. |
| 9 | User Selects Program Name from List | 10 | System displays corresponding Lists of Batch Numbers and Course Titles |
| 11 | User Selects Batch Number from List. | 12 | System displays corresponding List of Student Names. |
| 13 | User Selects Student Name from List | 14 | System displays selected name from list. |
| 15 | User Selects Course Title from List | 16 | System displays selected course from list. |
| 17 | User Select Is Fee Submitted Status from List. | 18 | System displays selected option from list. |
| 19 | User press Add button | 20 | System saves record and shows message. |

**Alternative Course of Actions:**

| | | | |
|---|---|---|---|
| 5a | If Faculty Name is not selected from List. | 6a | System displays an error message. |
| 7a | If Department Name is not selected from List. | 8a | System displays an error message. |
| 9a | If Program Name is not selected from List. | 10a | System displays an error message. |
| 11a | If Batch Number is not selected from List. | 12a | System displays an error message. |
| 13a | If Student Name is not selected from List. | 14a | System displays an error message. |
| 15a | If Course Title is not selected from List. | 16a | System displays an error message. |
| 17a | If the result is already added for the selected course. | 18a | System displays an error message. |
| 19a | User can press Close button. | 20a | System closes Add Result screen. |

**Post Conditions:**
Result has been added successfully.

**GUI Screen Reference:** Register Course Screen

| | |
|---|---|
| **Use Case ID:** | UC-09 |
| **Use Case Name:** | Register Course |
| **Actor(s):** | Administrator |
| **Goal:** | Administrator should be able to register a particular course for the student. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

| **Actor Actions:** | | **System Response:** | |
|---|---|---|---|
| 1 | User selects Result from main menu. | 2 | System displays menu items. |
| 3 | User selects Add Result from menu item. | 4 | System displays Add Result screen. |
| 5 | User Selects Faculty Name from List. | 6 | System displays corresponding List of Department Names. |
| 7 | User Selects Department Name from List | 8 | System displays corresponding List of Program Names. |
| 9 | User Selects Program Name from List | 10 | System displays corresponding Lists of Batch Numbers and Course Titles |
| 11 | User Selects Batch Number from List. | 12 | System displays corresponding List of Student Names. |
| 13 | User Selects Student Name from List | 14 | System displays selected name from list. |
| 15 | User Selects Course Title from List | 16 | System displays selected option from list. |
| 17 | User enters following information: <br>• Course Registration Date, <br>• Course Registration Notes. | 18 | System takes input data. |
| 19 | User press Add button | 20 | System displays menu items. |
| **Alternative Course of Actions:** | | | |
| 5a | If Faculty Name is not selected from List. | 6a | System displays an error message. |
| 7a | If Department Name is not selected from List. | 8a | System displays an error message. |
| 9a | If Program Name is not selected from List. | 10a | System displays an error message. |
| 11a | If Batch Number is not selected from List. | 12a | System displays an error message. |
| 13a | If Student Name is not selected from List. | 14a | System displays an error message. |
| 15a | If Course Title is not selected from | 16a | System displays an error message. |

| | | | |
|---|---|---|---|
| | List. | | |
| 17a | If the course is already registered for the selected course. | 18a | System displays an error message. |
| 19a | User can press Close button. | 20a | System closes Register Course screen. |

**Post Conditions:**
Course has bee registered successfully.

**GUI Screen Reference:** All View Screens of the System

| | |
|---|---|
| **Use Case ID:** | UC-10 |
| **Use Case Name:** | View Specific Record |
| **Actor(s):** | Administrator |
| **Goal:** | Administrator should be able to any specific record. |
| **Preconditions:** | Main form of application is opened and user is on main form. |

| **Actor Actions:** | | **System Response:** | |
|---|---|---|---|
| 1 | User selects a specific Item from main menu. | 2 | System displays menu items. |
| 3 | User selects View a specific sub menu item. | 4 | System displays available corresponding records on screen. |

**Alternative Course of Actions:**

**Post Conditions:**
Record is displayed successfully.

## 4.1.2 Use Case Diagram

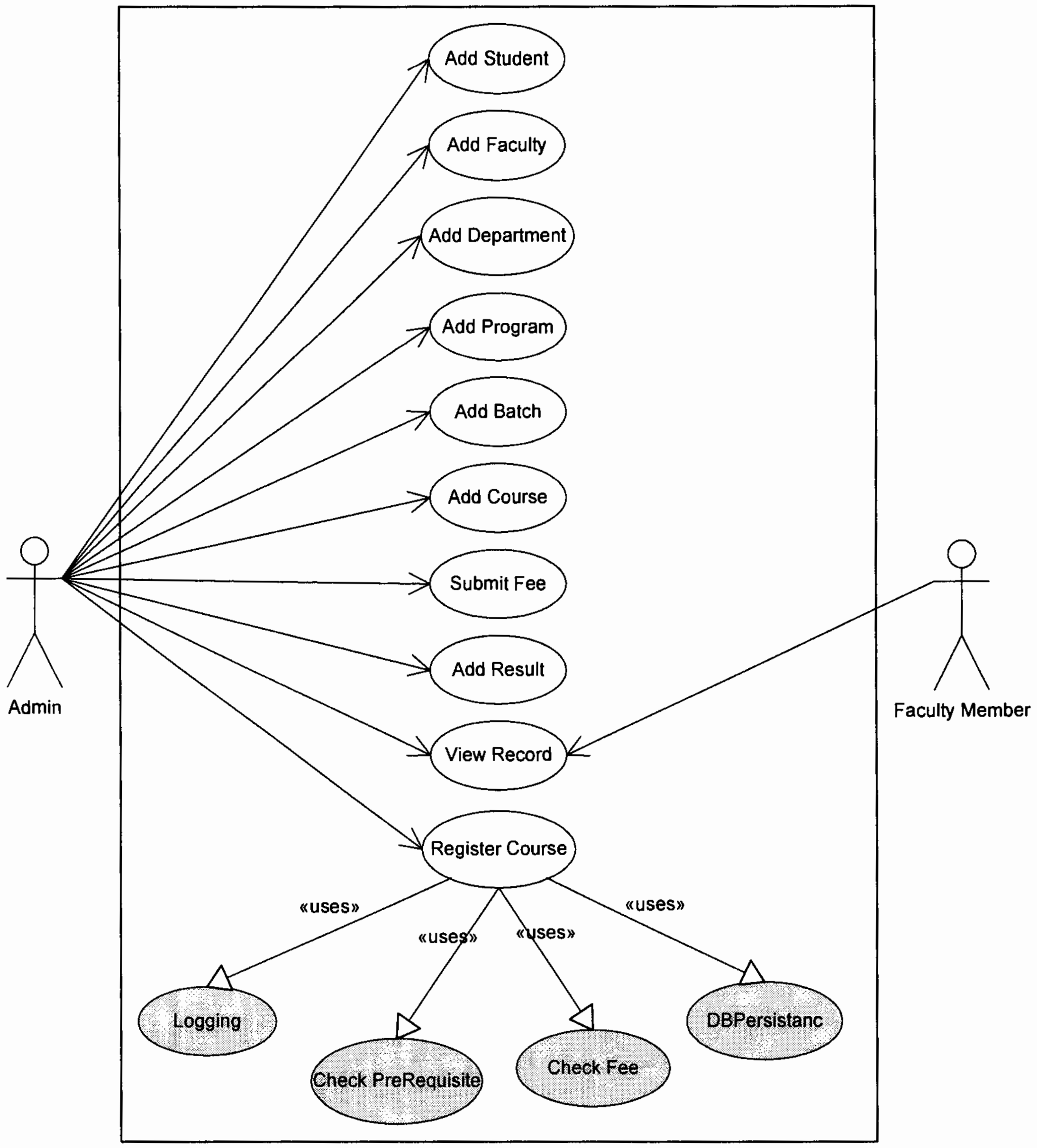

**Figure 4.1 Use Case Diagram for the Case Study**

## 4.2 Shared Join Point Problem in Case Study and Design Solution

This section identifies and describes ordering and controlling issues of shared join point in the case study presented in section 4.1. Design solution for shared join point problems is presented using proposed methodology presented in chapter 3.

### 4.2.1 Multiple Aspects at Shared Join Point

Above discussed functional requirements of the case study highlight some problems with respect to shared join point.

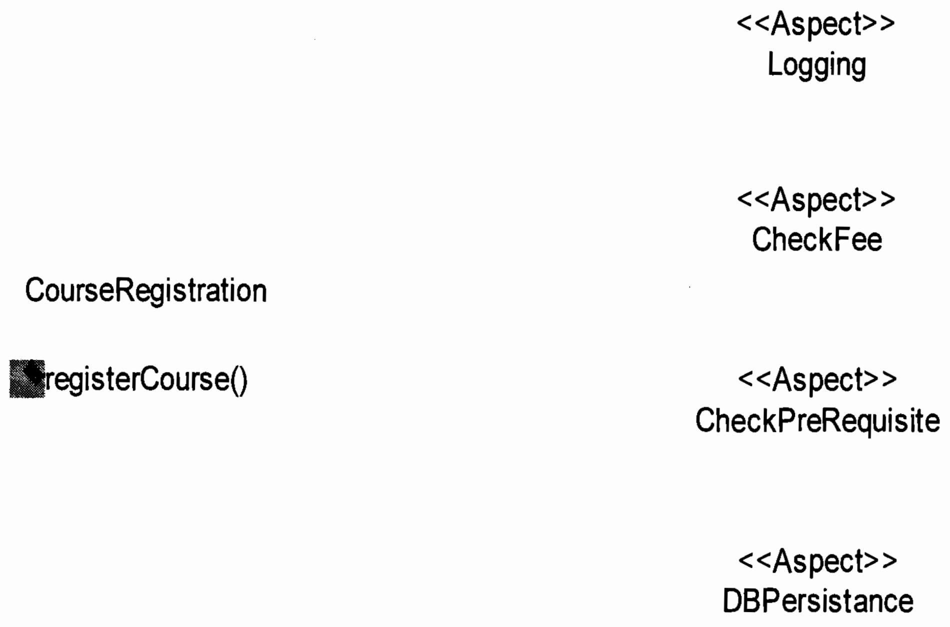

**Figure 4.2 Multiple aspects superimposed on a shared join point**

In the above figure 4.2, multiple aspects i.e. *Logging*, *CheckFee*, *CheckPreRequisite*, and *DBPersistence* are superimposed on *registerCourse()* method. They are required to execute again *registerCourse()* method call. They present issue of ordering and controlling of these superimposed aspects on the same join point. Ideally, their order of execution should be; first *Logging* aspect should execute and if *Logging* aspect gets successfully executed then next aspect will be executed, if *Logging* gets unsuccessful, the other aspects should not be executed. Two main issues arise;

- Ordering of the multiple aspects i.e. make sure that *Logging* aspect should always run first.
- Controlling of succeeding aspects (to ensure that if first aspect *Logging* does not execute properly); how to stop the execution of other aspect?

These issues have already been discussed in detail in previous chapters 2 and 3. In normal scenario all superimposed aspects are executed independently, even though they have dependencies with respect to system functionality. They need to control in order to implement according to the requirements of the system.

### 4.2.2 Design with Proposed Methodology

Since, the requirement of logging is implemented by the aspect named *Logging*. The responsibility of *Logging* aspect is to maintain the log of every entrance to a method, so *Logging* aspect should run before the *registerCourse()* join point which is now well defined point in the program.

The other requirements like student fee checking and course prerequisites checking for the course being registered by the student are implemented by *CheckFee* and *CheckPreRequisite* aspects respectively as described earlier. These aspects should also run before the *registerCourse()* join point. All three aspects will be executed on the *registerCourse()* join point at the same time. In other words the join point *registerCourse()* is being shared by the multiple superimposed aspects.

The problem of ordering and controlling among these three aspects arises, because in normal scenario any aspect can be executed by the compiler. If either of the aspects does not provide the desired results, the course should not be registered. The last requirement is to check the database persistence implemented by *DBPersistence* aspect, which ideally should run at the end when a course has been registered.

Suppose that database is up and *DBPersistence* aspect is working perfectly. This aspect should inform the user about the course registration, whether it has been successful or not? It is the only aspect which will execute after the core requirement gets completed and do not have any conflict with other aspects.

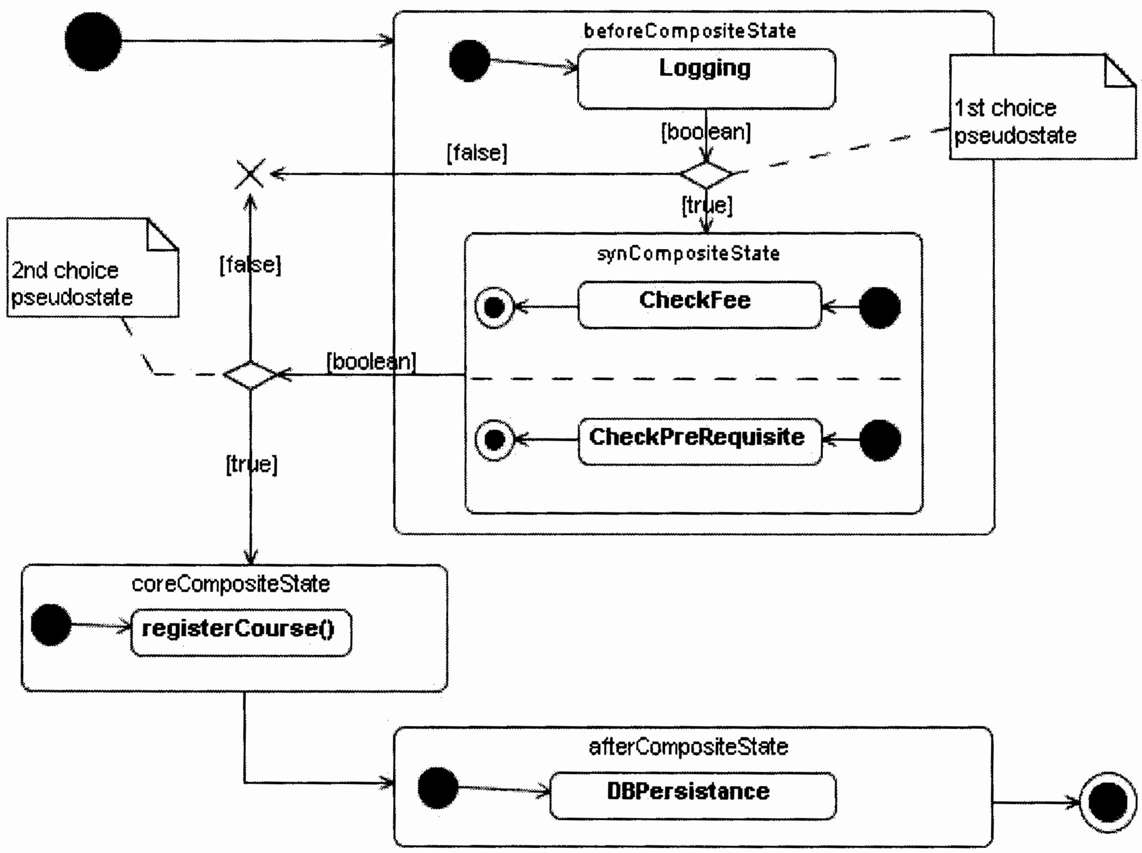

**Figure 4.3 Detailed case study design of proposed methodology**

The superimposition of three aspects requires great efforts to handle their ordering and controlling issues. In order to define better ordering and controlling strategies, we need to model shared join points at high level design. In figure 4.3, two different categories of composite states are defined; one implementing the core requirement and other implementing the aspectual requirement.

We are more concerned with the first composite state i.e. *beforeCompositeState* that contains the aspects need to be executed before the core requirement and on the bases of these results the core requirement will be implemented. This composite state contains an aspect *Logging* and a sub-composite state *syncCompositeState* which contains two aspects *CheckFee* and *CheckPreRequisite* in its orthogonal region for their concurrent execution. The concept of *synCompositeState* is the same like *beforCompositeState* and *afterCompositeState* but differ in this way that aspects contained by this composite state will be executed concurrently as they both are independent to each other. If any of the aspects contained by this state produce undesired results, the combined output of both will be false. The true results of this state only can guarantees the execution of core requirement. The aspects in *beforeCompositeState* are superimposed and required to be ordered and controlled. If we consider the above discussed scenario, the *Logging* aspect should always run first. This *Logging* aspect will transmit boolean guard value to first choice pseudostate as input. This pseudostate evaluates these boolean guard values and decides whether to transmit the object or not? If boolean guard value is evaluated as true, the object will be forwarded to *syncCompositeState* which contains two aspects namely *CheckFee* and *CheckPreRequisite.* Otherwise it will be sent to terminate pseudostate, which destroys the object action and as a result other requirements will not be fulfilled. This allows controlling of the other aspects if the first does not work properly. If it works properly, then the object is passed into *syncCompositeState* and boolean guard value has been referred to *CheckFee* and *CheckPreRequisite* aspects in the orthogonal regions, both will synchronously run and their combined result will be transmitted as boolean guard value to second choice pseudostate that evaluates the boolean guard value. If the value is true, it implements the course registration core requirement otherwise object will be sent to terminate pseudostate. Synchronous handling of two aspects takes place because they don't have any ordering constraint and they are not dependant on each other as well i.e. any one of the aspects in *syncCompositeState* could run after the other. Also the synchronous handling of these aspects in the orthogonal regions will be handled by default as run to-completion, in which each event is handled before the next occur [18].

The *DBPersistence* aspect is an aspect that should run after core requirement, so there is no need to order or control this aspect. Its role is to just inform whether the database is updated or not.

## 4.3 Architecture

The architectural view for the application is basically layered as shown in figure 4.4. The core and aspectual requirements are implemented at the same layer. The *"gui"* layer is responsible for users' actions handling and other related operations. The *"db"* is responsible to data connection opening/closing and data storage.

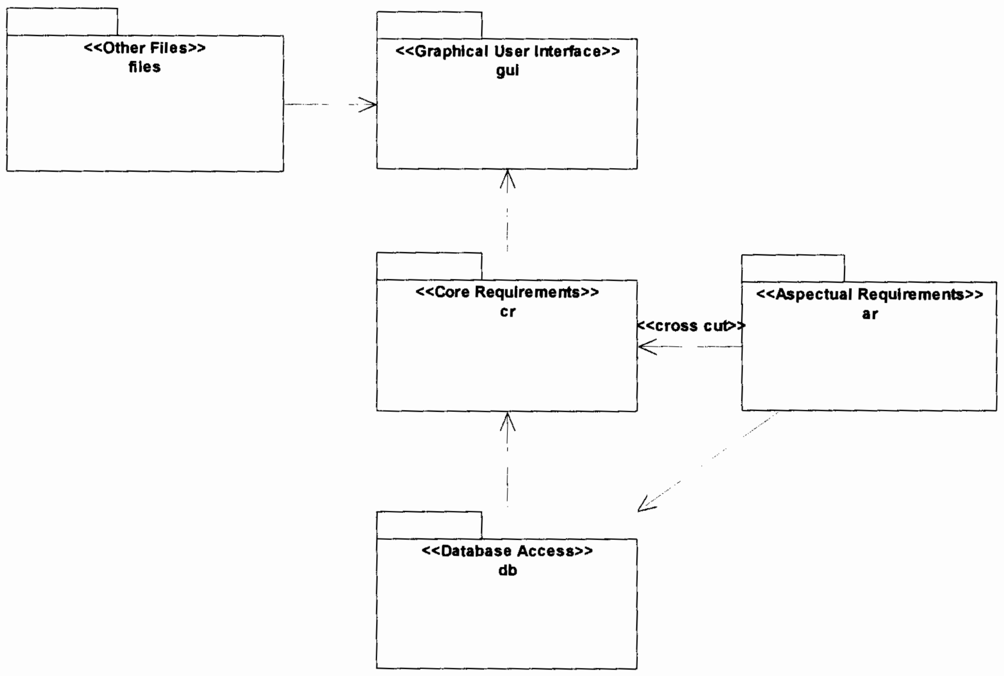

**Figure 4.4 System Architecture Diagram**

## 4.4 Collaboration Diagrams

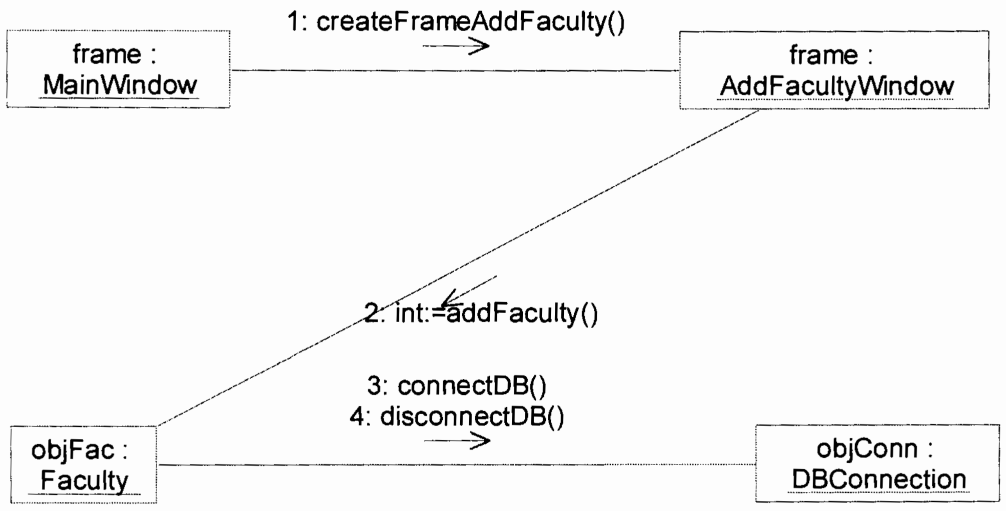

**Figure 4.5 Collaboration Diagram for Add Faculty**

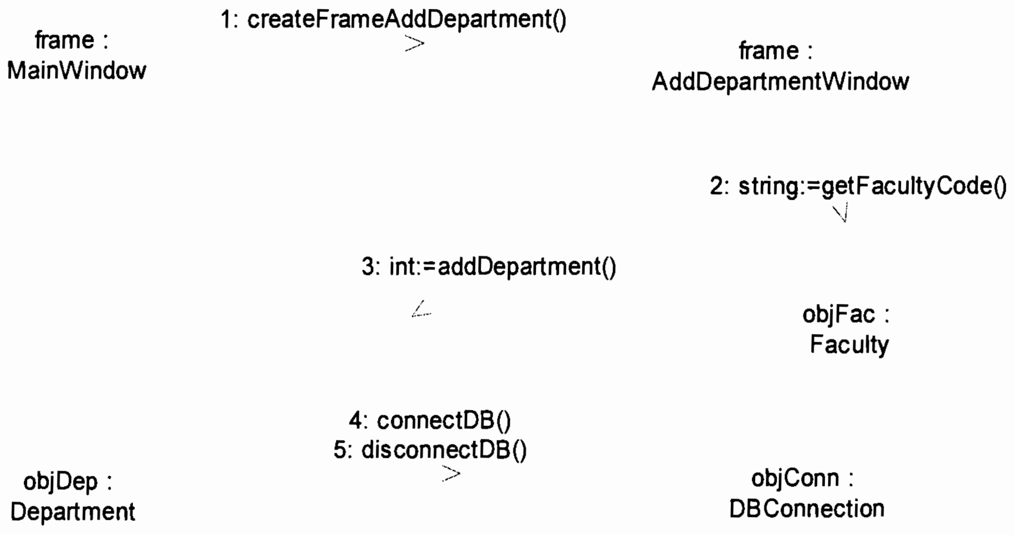

**Figure 4.6 Collaboration Diagram for Add Department**

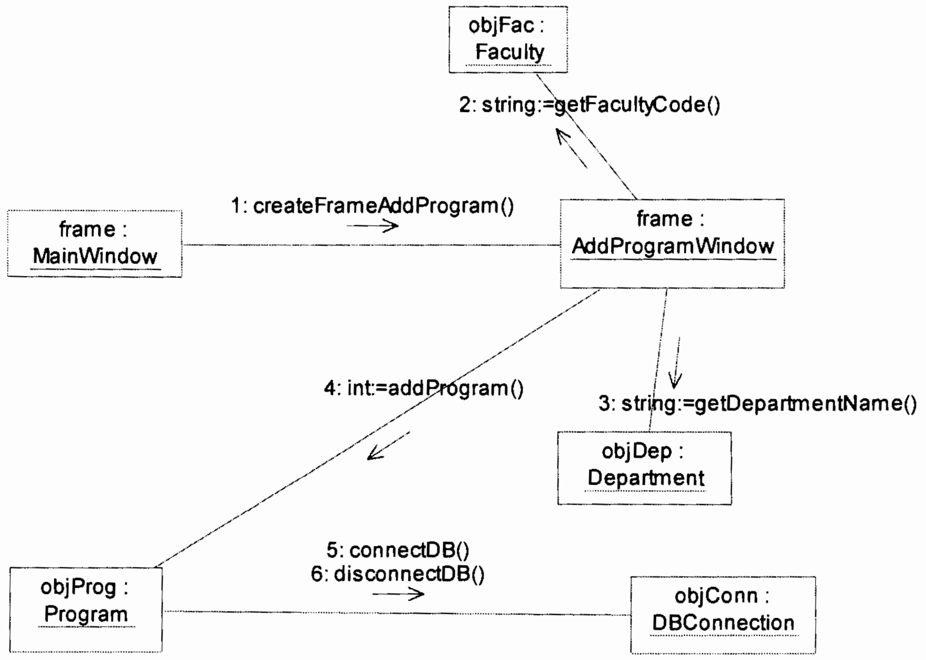

**Figure 4.7 Collaboration Diagram for Add Program**

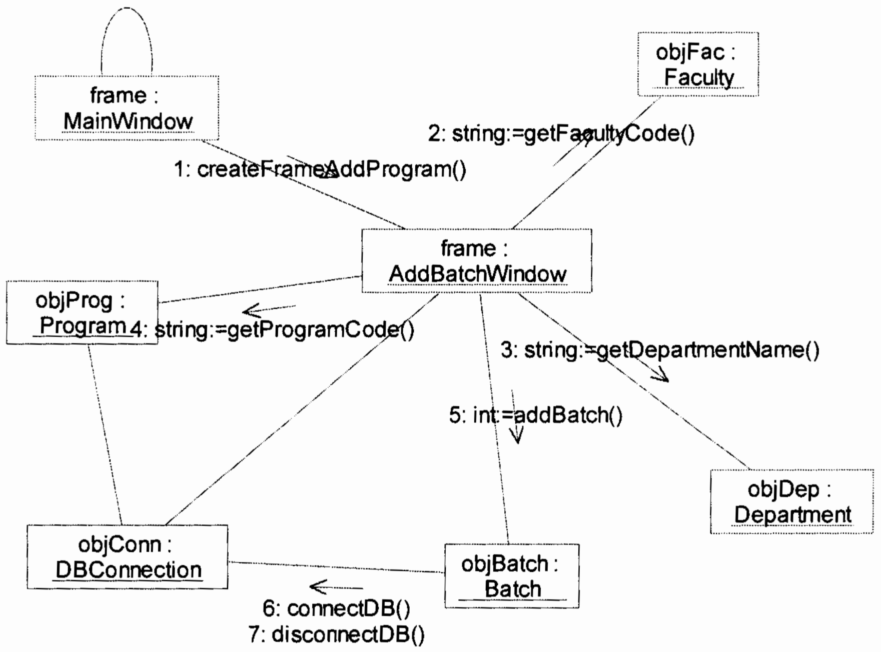

**Figure 4.8 Collaboration Diagram for Add Batch**

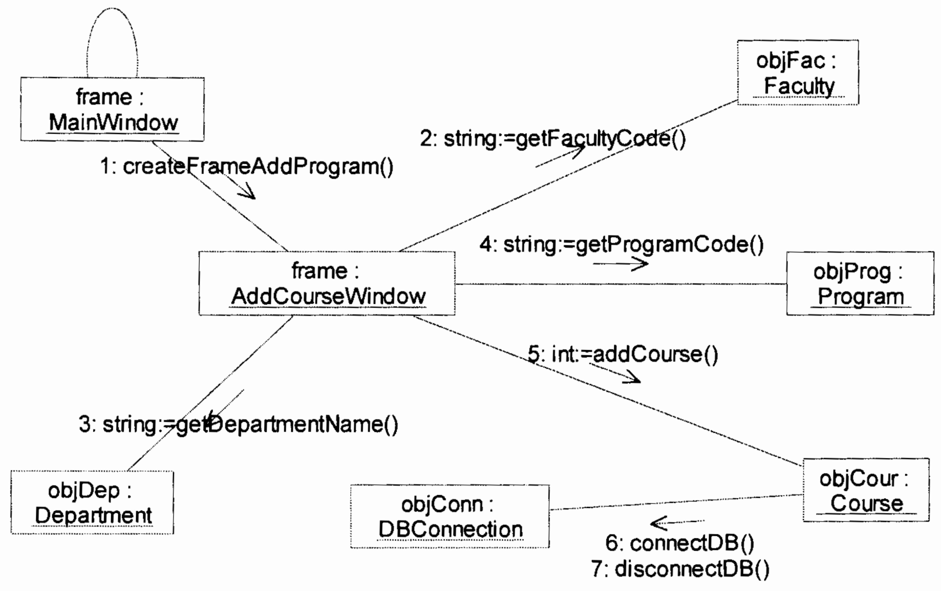

**Figure 4.9 Collaboration Diagram for Add Course**

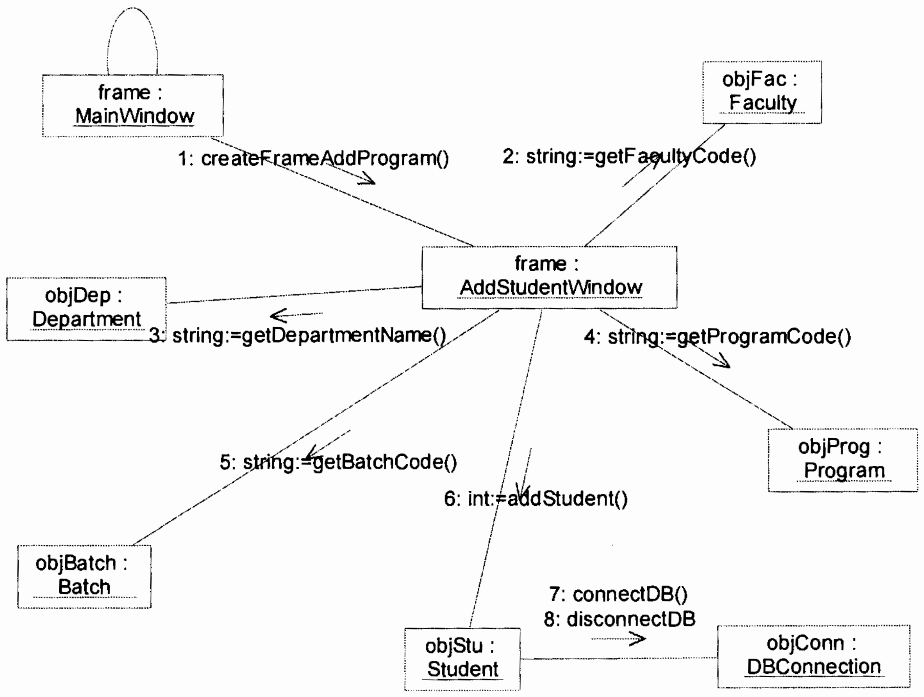

**Figure 4.10 Collaboration Diagram for Add Student**

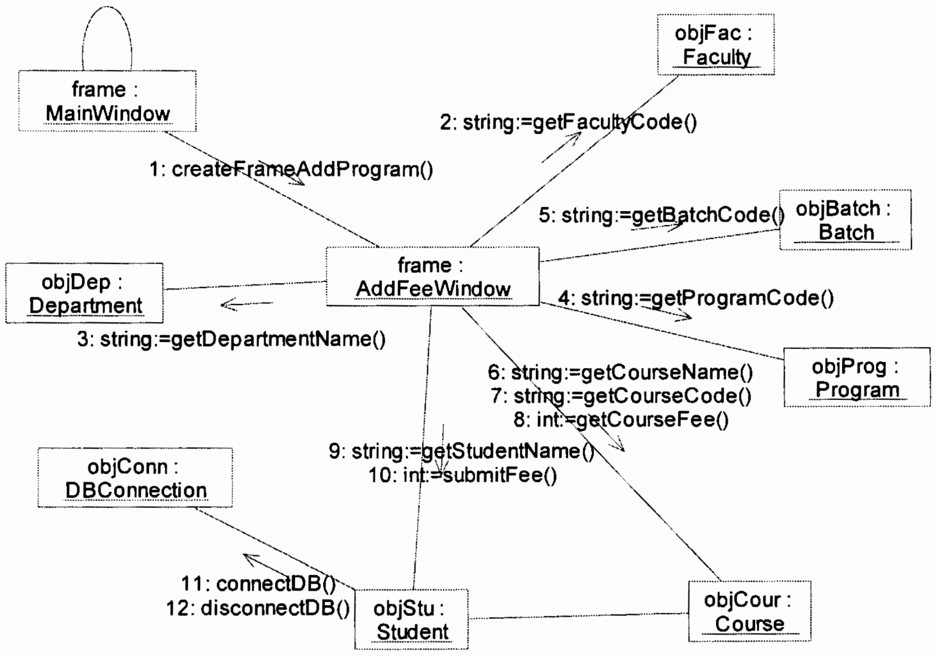

**Figure 4.11 Collaboration Diagram for Submit Fee**

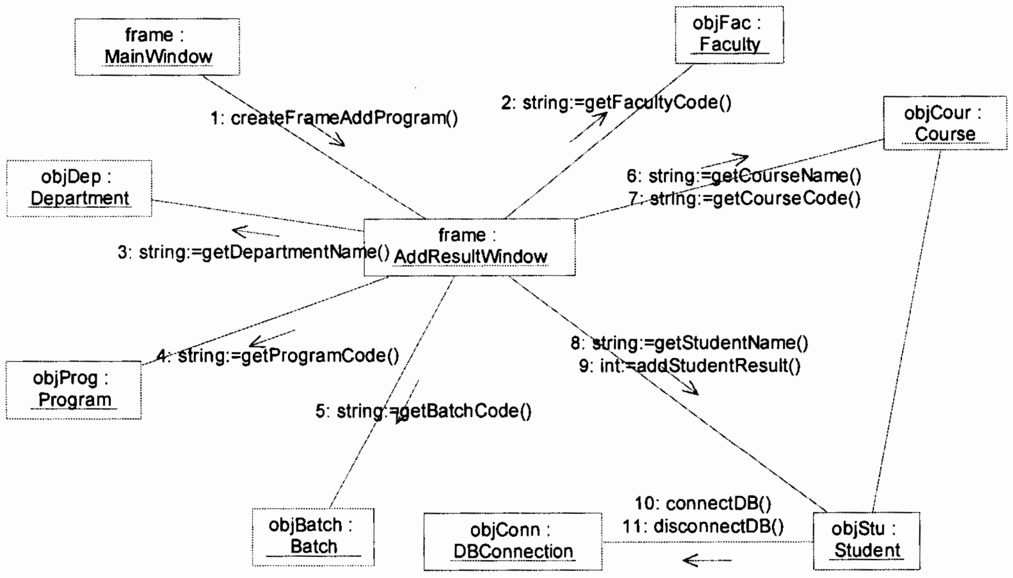
frame : MainWindow
objFac : Faculty
1: createFrameAddProgram()
2: string:=getFacultyCode()
objCour : Course
objDep : Department
6: string:=getCourseName()
7: string:=getCourseCode()
3: string:=getDepartmentName()
frame : AddResultWindow
4: string:=getProgramCode()
objProg : Program
5: string:=getBatchCode()
8: string:=getStudentName()
9: int:=addStudentResult()
10: connectDB()
11: disconnectDB()
objBatch : Batch
objConn : DBConnection
objStu : Student

**Figure 4.12 Collaboration Diagram for Add Result**

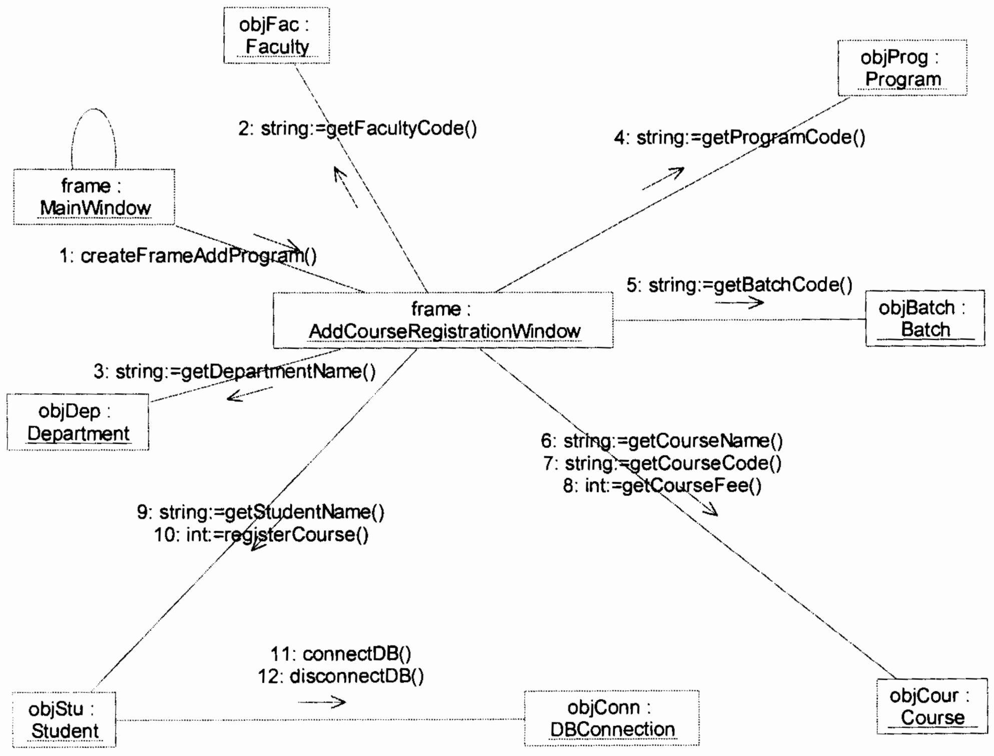
objFac : Faculty
objProg : Program
frame : MainWindow
2: string:=getFacultyCode()
4: string:=getProgramCode()
1: createFrameAddProgram()
5: string:=getBatchCode()
frame : AddCourseRegistrationWindow
objBatch : Batch
3: string:=getDepartmentName()
objDep : Department
6: string:=getCourseName()
7: string:=getCourseCode()
8: int:=getCourseFee()
9: string:=getStudentName()
10: int:=registerCourse()
11: connectDB()
12: disconnectDB()
objStu : Student
objConn : DBConnection
objCour : Course

**Figure 4.13 Collaboration Diagram for Register Course**

## 4.5 Class Diagram

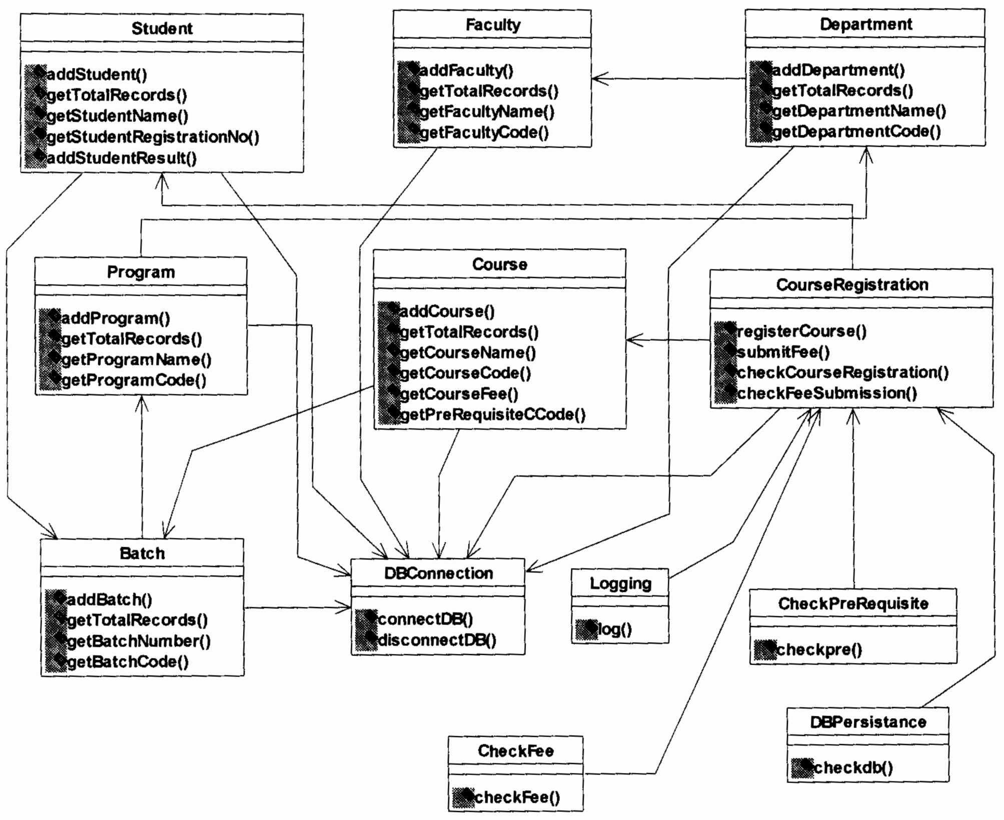

**Figure 4.14 Class Diagram**

Figure 4.5 shows only classes that are related to core requirements, aspectual requirements and database connection but not graphical user interface classes.

## 4.6 Database Model

MS Access 2003 is used as database. Database model is given in the following section. In architecture diagram, db layer contains database file and class named *DBConnection* which is responsible for opening and closing of database connections.

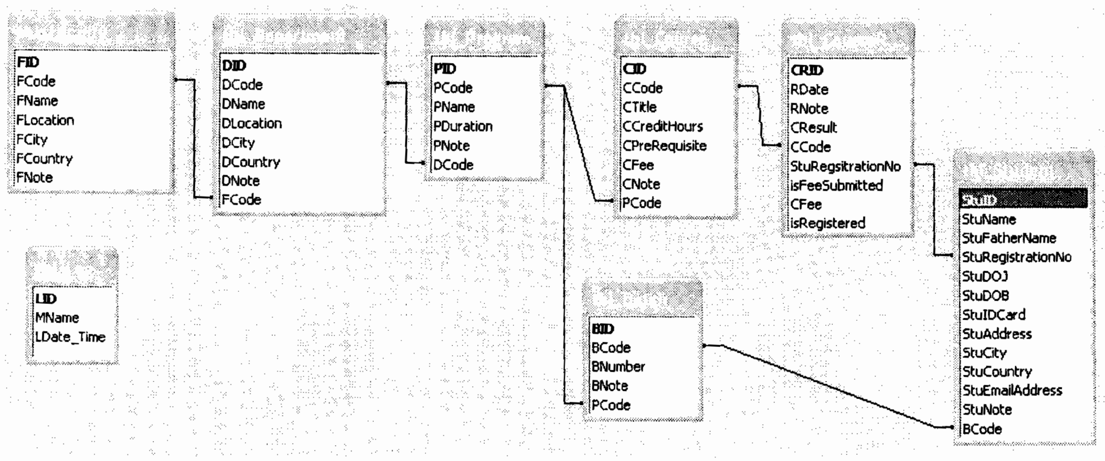

**Figure 4.15 Database Model**

## 4.7 Ordering and Controlling Issues Handling in Code

This section discusses the implementation (in AspectJ) of the case study with the help of proposed methodology. AspectJ is one of the prominent languages of aspect oriented software development [3] [4]. This section also describes how controlling and ordering issues can be better implemented with the help of proposed methodology?

For ordering of superimposed aspects at the same join point precedence construct is available in AspectJ. It is very important to understand the execution order of the superimposed aspects in the program. In the absence of precedence construct the execution order of the program is unpredictable. Any of aspects can be executed before the rest, when there are more than one superimposed aspects on the shared join point with before or after advices.

### 4.7.1 Ordering

To ensure that the *Logging* aspect should run before any other aspect, precedence construct with respect to aspect sequence (ordering) is declared in each aspect. When program begins execution and control reaches at any of aspects, the compiler can find the ordering flow of the aspects. Following statement is written in all aspects such as *Logging, CheckFee, CheckPreRequisite,*

***declare precedence:Logging*;***

Since in the case study, we are making sure that only *Logging* aspect should run first. Both aspects i.e. *CheckFee* and *CheckPreRequisite* can execute in any order after the successful execution of *Logging* aspects. So, we will use only above mentioned statement for ordering the aspects. When we also need to order other aspects, the following statement can be used;

***declare precedence: Logging, CheckFee,*;***

When control reaches in any of these aspects, this statement informs the compiler that first of all *Logging* aspect will have to be executed. The statement given above describes that after *Logging* aspect, *CheckFee* aspect will be executed. This statement is declared in all aspects define order of aspects. In this fashion ordering of the superimposed aspects on the same shared join point can achieved as shown in table 4.1 and 4.2.

## 4.7.2 Controlling

Ordering of superimposed aspects at same join point is the first step toward controlled execution of the aspects.

When execution of the aspects has been done in an ordered manner, it is the right time to introduce the strategy for controlling the aspects. For this reason some static boolean variables are used, which are responsible for storing the updated values so that the execution of relevant aspects can be controlled depending on current values in the static boolean variables. The whole phenomenon is explained:

In the *CourseRegistration* class the controlled variables are;

***public static boolean logging=true ;***

***public static boolean checkFee=true;***

***public static boolean checkPreReq=true;***

***public static boolean dbPersistance=true;***

Default values of these variables are set *"true"*. If the execution of any of the aspects produces negative result for other aspects than the corresponding static boolean variable are changed from *"true"* to *"false"*.

In such a case the next aspects first checks the value of the controlled variable (which effects the execution of this variable) and if the value is *"false"* than the execution of current aspect skips itself and the value of referring controlled variable becomes *"false"* and vice versa. See *'Logging'* aspect in table 4.1.

```
package ar;

import java.sql.Statement;
import javax.swing.JOptionPane;
import db.DBConnection;
import cr.CourseRegistration;

public aspect Logging {
        declare precedence:Logging*;//Ordering the aspects

        pointcut log(String crDate, String crNote, String cCode, String stuRegistrationNo):
        call(int cr.CourseRegistration.registerCourse(String, String,String, String))
                        && args(crDate, crNote, cCode, stuRegistrationNo);

        before(String crDate, String crNote, String cCode, String stuRegistrationNo) :
                log(crDate, crNote, cCode, stuRegistrationNo){
                try{
                        int result=0;
                        Statement statement=null;
                        DBConnection dbconn= new DBConnection();

                        dbconn.connectDB();

                        java.util.Date now = new java.util.Date();
                        statement = dbconn.conn.createStatement();
                        String sql = "INSERT INTO tbl_Log (MName" +
                                   ",              LDate_Time)              VALUES
('"+thisJoinPoint.getSignature( ).toString()+"'" +
                                                 "','"+now.toString()+"')";

                    result=statement.executeUpdate(sql);
                    dbconn.disconnectDB();

                    if (result>0){
                         System.out.println("Successful Logging!");
                         CourseRegistration.logging=true;
                    }else{
                         CourseRegistration.logging=false;
                         JOptionPane.showMessageDialog(null,"Logging   Failed!   try
later");
                    }
               }
     catch(Exception e) {
               System.out.println(e.getMessage());
               CourseRegistration.logging=false;
     }
     }
}
```

**Table 4.1 Logging aspect**

The first statement in bold text in the above aspects is responsible for the ordering issue of the superimposed aspects at the shared join point *registerCourse().* This has been discussed in the previous section. The next bold text statement is responsible for setting boolean values *"true"* as the logging functionality is successfully evaluated by the *"if"* statement above. The next and the last statement in bold text is used to set static boolean variable *"false"* as it is the else part of *"if"* statement which represents failure of logging functionality. In this way boolean static variables can be set *"true"* or *"false"* based on the success or failure of respective aspect.

Now, we see that how the values of boolean static variable are represented in proposed methodology design and declared in *CourseRegistration* class can be evaluated. In the proposed design methodology, these values were being evaluated by choice pseudostate UML state charts. In AspectJ programming, these values are evaluated by *"if"* statement of programming language. As shown in table 4.2 for *CheckFee* aspect:

```
package ar;
import javax.swing.JOptionPane;
import cr.CourseRegistration;

public aspect CheckFee {
declare precedence:Logging*;//Ordering the aspects
pointcut checkFee(String crDate, String crNote, String cCode, String stuRegistrationNo):
                                                  call(int
cr.CourseRegistration.registerCourse(String, String,String, String))
                                                              && args(crDate, crNote, cCode,
stuRegistrationNo);

        before(String crDate, String crNote, String cCode, String stuRegistrationNo) :
                                checkFee(crDate, crNote, cCode, stuRegistrationNo){

                if (CourseRegistration.logging==false){
                        //code for stopping this aspect to run
                        CourseRegistration.checkFee=false;
                }else{
                        CourseRegistration objCR=new CourseRegistration();
                        int chkValue=0;
                        chkValue=objCR.checkFeeSubmission(cCode, stuRegistrationNo);

                        if (chkValue>0){
                                CourseRegistration.checkFee=true;
                                System.out.println("Fee is submitted!");
                        }else{
                                CourseRegistration.checkFee=false;
                                //System.out.println("Fee is not submitted!");
                                JOptionPane.showMessageDialog(null,"Fee is not submitted!");
                        }
                }
        }
}
```

**Table 4.2 CheckFee aspect**

The first statement in bold text in table 4.2 represents an *"if"* statement which evaluates boolean static variable values of logging aspect. If boolean static variable value is evaluated as *"false"* the code in body of *"if"* statement sets *checkFee* boolean static variable as *"false"* which will be executed by the next aspect, and if it is evaluated as *"true"* the corresponding functionality of aspect will be implemented. This way controlling of the aspect will be done.

Chapter 5

# RESULTS AND DISCUSSIONS

# 5. Results and Discussions

This chapter mainly discusses the results achieved by applying proposed methodology on case study and without applying proposed methodology on case study. Benefits of the proposed methodology at design and implementation level are discussed. This chapter also concludes the overall study. Future work is also presented.

## 5.1 Results

In order to see whether the proposed methodology helps in ordering and controlling of the multiple aspects superimposed on the shared join point at design level, some execution scenarios with respect to aspectual requirement are represented.

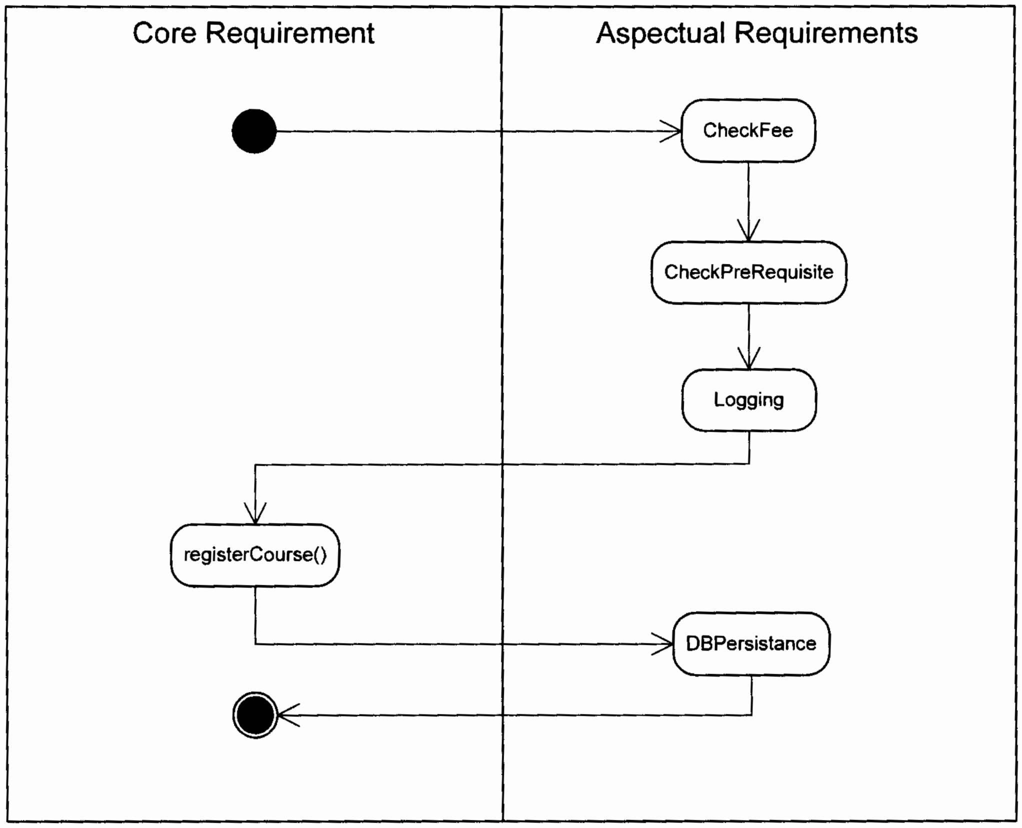

**Figure 5.1 Execution sequence of aspects without ordering and controlling**

Figure 5.1 represents an execution scenario where aspects are not ordered and controlled. It clear represents that aspects are not executing according to the aspectual requirements presented in chapter 4 (see 4.1). This is normal behavior of multiple aspects at shared join point that is why it presents the issues of ordering and controlling. The proposed methodology presented in chapter 3 and 4 helps to resolve these issues at design level. Then this design helps in implementing the aspectual requirements as per desired strategy. Table 5.1 and 5.2 represent the comparison of case study results achieved without ordering and controlling the multiple aspects through proposed methodology and with ordering and controlling the multiple aspects through proposed methodology.

| **Issues** | **Required Aspect Execution Sequence** | **Aspects' Execution Sequence without Ordering and Controlling** | | | |
|---|---|---|---|---|---|
| | | Case # 01 | Case # 02 | Case # 03 | Case # 04 |
| **Ordering of Aspects** | Logging, CheckFee, CheckPreRequisite, DBPersistance. | CheckFee, CheckPreRequisite, Logging, DBPersistance. | CheckFee, CheckPreRequisite, DBPersistance. | CheckPreRequisite, Logging, DBPersistance. | CheckFee, Logging, DBPersistance. |
| **Controlling of Aspects** | This is successful scenario as Logging aspect runs first of all. After the successful execution of Logging aspect either CheckFee or CheckPreRequisite aspect run. If Logging aspect is unsuccessful then no other aspect or core requirement should be executed. DBPersistance will always run after core requirement. | All the aspectual requirements are fulfilled in unordered and uncontrolled way. | Logging aspect is failed to maintain log even then other dependant aspects executed which violated overall strategy. | CheckFee aspect is failed to check the fee submitted even then other dependant aspects executed which violated overall strategy. | CheckPreRequisite aspect is failed to check the fee submitted even then other dependant aspects executed which violated overall strategy. |

**Table 5.1 Ordering and controlling without proposed methodology**

| Issues | Required Aspect Execution Sequence | Aspects' Execution Sequence with Ordering and Controlling | | | |
|---|---|---|---|---|---|
| | | Case # 01 | Case # 02 | Case # 03 | Case # 04 |
| **Ordering of Aspects** | Logging, CheckFee, CheckPreRequisite, DBPersistance. | Logging, CheckFee, CheckPreRequisite, DBPersistance. | Logging (Failed)* <br><br> *Execution stopped | Logging (Successful), CheckFee (Failed)* <br><br> * Execution stopped | Logging(Successful), CheckFee(Successful) CheckPreRequisite(Failed)* <br><br> * Execution stopped |
| **Controlling of Aspects** | This is successful scenario as Logging aspect runs first of all. After the successful execution of Logging aspect either CheckFee or CheckPreRequisite aspect run. If Logging aspect is unsuccessful then no other aspect or core requirement should be executed. DBPersistance will always run after core requirement. | All the aspectual requirements are fulfilled in ordered and controlled way. | Logging aspect is failed to maintain log so other aspects did not execute according to the strategy. | CheckFee aspect is failed to check fee submitted so other aspect(s) did not execute according to the strategy. | CheckPreRequisite aspect is failed to check pre requisite course so other aspects did not execute according to the strategy. |

**Table 5.2 Ordering and controlling with proposed methodology**

## 5.2 Benefits

Major benefits of the proposed methodology are presented. These benefits cover different perspectives of users and software systems.

### For designer

The proposed methodology helps designer to design solution for complex scenarios of shared join points presented in case study in previous chapters. This will also help them to solve related issues as early as possible.

## For Programmers

After designer formulates the suitable strategies for ordering and controlling, it is programmer's responsibility to implement that. The proposed methodology helps programmer to implement the designed strategy in an ideal way. The proposed methodology clearly specifies how to handle the ordering and controlling issues of shared join point. Programmer would not find any problem to implement the superimposed aspects.

## Scalability

The designed solution using proposed methodology is scalable in the sense that it can help in representing new conflicting aspects in the model. For example, if there is a new aspectual requirement which again uses a same join point that is already being used by some other aspect as shown in diagram below.

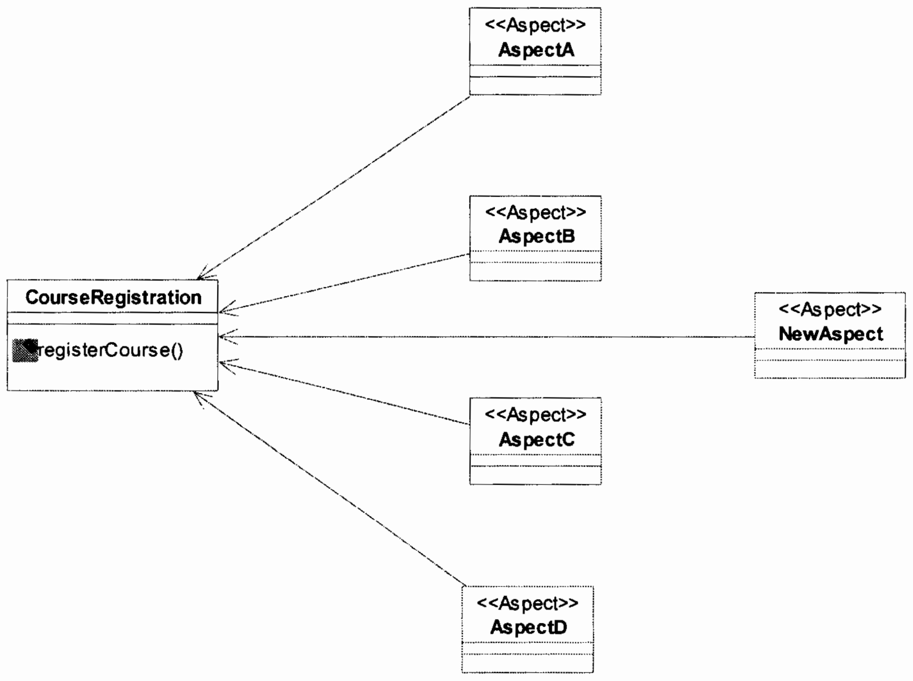

**Figure 5.2 Scalability**

Figure 5.1 shows that there are four aspects which have already superimposed on the same join point. And there is another aspect with the name of *NewAspect* which is being superimposed in same join point. The proposed methodology can help designers to effectively handle this situation.

### Reusability

The proposed methodology can be reused for solutions of similar complex problems. A similar problem is presented in [8]. This proposed design solution can be used for that problem as well.

### Independent of implementation detail

The presented design methodology of shared join point allows the designer to represent shared join point independent of the implementation details at abstract level [19]. This would help to implement the designed solution in any of the programming languages like AspectJ (one of the most prominent aspect oriented language) or AspectSharp (provide framework to implement aspect oriented programming Microsoft C#).

### Maintainability

The proposed methodology can also help in maintaining the software systems. If there is an aspect that conflicts other aspects and required to be removed or updated, the proposed methodology can help to maintain the design and then implementation. In future, if there is a need to add a new aspectual requirement, even then state chart model of proposed methodology can help to add new aspect corresponding to new aspectual where it is required to be. The proposed methodology at high level design can help in maintaining the software systems designed and developed in aspect oriented software development (AOSD) paradigm.

### Less rework

Early representation of ordering and controlling issues reduces the work cost by identifying the complexity of issues and formulating suitable strategies to solve the issues accordingly. Early the problem is identified cheaper to solve.

### Helps in formulating strategies for complex scenarios

The proposed methodology helps formulating suitable strategies required to solve shared join point issues like ordering and controlling discussed earlier. It provides flexibility for designers to apply any of the strategies given in [9] at high level design.

## 5.3 Conclusions

Previously shared join points were discussed at implementation level and to the best of our knowledge there is no research done for this particular problem at design level. The research thesis describes the need of modeling shared join point at high level design. In this regard, state charts of UML 2.0 have been used for modeling at high level. The main elements of state charts are composite states which compose the superimposed aspects at shared join point. At abstract level, the composite states are used to categorize into core and aspectual requirements. The strategies for ordering and controlling are implemented through detailing the state charts using the choice pseudostate as dynamic selection element and guard values to evaluate the next transition. State charts represent a better way to handle shared joint point Issues at high level design. This allows the designers to design issues regarding the shared join points at early design stages, and programmers can implement these strategies accordingly. A case study is presented with proposed methodology, which shows in detail how issues(s) regarding shared join point(s) can be represented at high level design. Some of the benefits like scalability and maintainability are also discussed.

## 5.4 Future Work

Future work can be done for superimposed aspects having around advice which needs to execute both before and after the core requirement. For example, there could be another composite state that can contain the superimposed aspects having around advices. Currently, work can be done on this issue and how it can be represented and integrated in the proposed methodology. The around advice has not been discussed in the proposed methodology, so this could be one of the area to work on in future.

Since shared join points represents the problem which occur again and again. So, they represent a problem that can require solution in the form of design pattern. This work can further be studied with respect to discover new design pattern for the shared join point problem.

The proposed methodology can be used to recommend a suitable language construct to control the execution of multiple aspects superimposed on the same join point.

# REFRENCES & BIBLIOGRAPHY

# References & Bibliography

[1] Aspect-Oriented Software Development. http://www.aosd.net.

[2] Russell Miles; *"AspectJ Cookbook"*. O'Reilly. December 2004.

[3] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, Ch., Loingtier, J.and Irwin, J.: "*Aspect-Oriented Programming*". In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP ′97) (Yväskylä, Finland, June 9-13, 1997). Springer-Verlag, Berlin Heidelberg, 1997, LNCS 1241, Pages 220–242.

[4] Kiczales, G, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold: *"An Overview of AspectJ*". In Proceedings of ECOOP 2001, LNCS 2072, Springer Verlag, 2001.

[5] Stein, D., Hanenberg, St., Unland, R..: *"An UML-based Aspect-Oriented Design Notation For AspectJ*". In Proc. Of AOSD '02 (Enschede, the Netherlands, Apr. 2002), ACM, pp. 106-112.

[6] Boucke N., Holvoet T.: *"State-based join points: Motivation and requirements*". In Filman, R. E., Haupt, M., Hirschfeld, R. (eds): Proceedings of the Second Dynamic Aspects Workshop (2005) 1-4.

[7] Eduardo Barra Zavaleta, Gonzalo Génova Fuster, Juan Llorens Morillo: *"An approach to Aspect Modelling with UML 2.0*". Workshop on Aspect Oriented Modeling, October 11, 2004, Lisbon, Portugal, held in Conjunction with the 7th International conference on the Unified Modeling Language- UML 2004, October 10-15, 2004, Lisbon, Portugal.

[8] I. Nagy, Lodewijk Bergmans and Mehmet Aksit: *"Composing Aspects at Shared Join Points*". Proceedings of International Conference NetObjectDays, NODe2005.

[9] Stein, D., Hanenberg, S. and Unland, R.: *"On Representing Join Points in The UML*". In Proceedings of the 2nd Workshop on Aspect Modeling with UML at the Fifth International Conference on the Unified Modeling Language and its Applications (UML 2002), (Dresden, Germany, 30 September – 4 October, 2002).

[10] I. Nagy, L. Bergmans, M. Aksit.: *"Declarative Aspect Composition"*. Technical Report, http://trese.ewi.utwente.nl/publications/publications.php?action=showPublication&pub_id=346 University of Twente, (April 2005).

[11] Anis Charfi, Michel Riveill, Mireille Blay-Fornarino, Anne-Marie Pinna-Déry: *"Transparent and Dynamic Aspect Composition"*. In Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT), Bonn (Germany), 21 march 2006.

[12] Eric Braude: *"Software Design: From Programming to Architecture"*. John Wiley & Sons, Inc. 2004.

[13] A. Rashid et al.: *"A Survey of Aspect-Oriented Analysis and Design Approaches"*. 18 May 2005.

[14] Object Management Group: *"Unified Modeling Language"*. Superstructure, version 2.0 formal/05-07-04.

[15] Stein, D.; Hanenberg, S.; Unland, R.: *"Modeling Pointcuts. Early Aspects"*. Workshop on Aspect-Oriented Requirements Engineering and Architecture Design, AOSD 2004, Lancaster, UK, March 22, 2004.

[16] Mahoney, M., Bader, A., Aldawud, O., Elrad, T.: *"Using Aspects to Abstract and Modularize Statecharts."* The 5th Aspect-Oriented Modeling Workshop in Conjunction with UML 2004. http://www.cs.iit.edu/~oaldawud/AOM/mahoney.pdf.

[17] Mark Mahoney: *"Modeling Crosscutting Concerns in Reactive Systems with Aspect-Orientation"*. Doctoral Symposium at MoDELS/UML 2005, Montego Bay Jamaica, October 2005.

[18] Mohamed Mancona Kande, Jorg Kienzle and Alfred Strohmeier, *"From AOP to UML- A Bottom-Up Approach"*, Swiss Federal Institute of Technology Lausanne, Switzerland. [2001].

[19] Michelle Crane, Juergen Dingel: *"UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal"*. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005).

[20] Stein, D.; Hanenberg, S.; Unland, R.: *"Position Paper on Aspect-Oriented Modeling: Issues on Representing Crosscutting Features"*. 3rd International

Workshop on Aspect-Oriented Modeling with UML, AOSD 2003, Boston, MA, March 18, 2003.

[21] Wesley Coelho and Gail C. Murphy: *"Modeling Aspects: An Implementation-Driven Approach".* Workshop on Best Practices for Model Driven Software Development at OOPSLA 2004.

[22] A. Rashid, Araujo J., A. Moreira, and I. Brito: *"Aspect-Oriented Requirements with UML*". Workshop on Aspect-Oriented Modeling with UML (held with UML 2002).

[23] A. Rashid and N. M. Ali: *"A State-based Join Point Model for AOP*". Workshop on Views, Aspects and Roles — VAR (held with ECOOP 2005).

[24] Ivar Jacobson, Pan-Wei Ng: *"Aspect-Oriented Software Development with Use Cases"*. Addison Wesley Professional. December 2004.

## Appendix A

## PUBLICATION

# Representing Shared Join Points with State Charts: A High Level Design Approach

Muhammad Naveed, Muhammad Khalid Abdullah, Khalid Rashid, and Hafiz Farooq Ahmad

***Abstract*—**Aspect Oriented Programming promises many advantages at programming level by incorporating the cross cutting concerns into separate units, called aspects. Join Points are distinguishing features of Aspect Oriented Programming as they define the points where core requirements and crosscutting concerns are (inter)connected. Currently, there is a problem of multiple aspects' composition at the same join point, which introduces the issues like ordering and controlling of these superimposed aspects. Dynamic strategies are required to handle these issues as early as possible. State chart is an effective modeling tool to capture dynamic behavior at high level design. This paper provides methodology to formulate the strategies for multiple aspect composition at high level, which helps to better implement these strategies at coding level. It also highlights the need of designing shared join point at high level, by providing the solutions of these issues using state chart diagrams in UML 2.0. High level design representation of shared join points also helps to implement the designed strategy in systematic way.

***Keywords*—**Aspect Oriented Software Development, Shared Join Points.

## I. Introduction

ASPECT Oriented Programming [1] [2] is a new software development paradigm which enables to increase the comprehensibility, adaptability and reusability by modularizing the crosscutting concerns into the units called "aspects" [3] [4]. It provides solutions of many real time problems that neither the object oriented nor procedural languages can sufficiently handle [2] [5]. "Aspect" in AOP is like a class entity which mainly differs in instantiation and inheritance [3]. Other constructs of AOP are join points, pointcuts, advices and introductions [2] [5], among all, join point is more important. Join point is defined as a well defined execution point in a program [3].

This work is a part of our research project for Department of Computer Science at International Islamic University, Islamabad, Pakistan.

Muhammad Naveed is a student of MS (Software Engineering) at International Islamic University Islamabad, Pakistan. Also working as a Software Engineer in ESOLPK, 12 SNC Center, Blue Area, Islamabad, Pakistan (e-mail: naveed_2334@yahoo.com).

Muhammad Khalid Abdullah is a student of MS (Software Engineering) at International Islamic University Islamabad, Pakistan (e-mail: am.khalid@yahoo.com).

Khalid Rashid is with Faculty of Applied Science, International Islamic University Islamabad, Pakistan (e-mail: drkhalid@yahoo.com).

Hafiz Farooq Ahmad is with Department of Computer Science, International Islamic University Islamabad, Pakistan (e-mail: farooq@comtec.co.jp).

Join points represent the key concept in Aspect-Oriented Software Development (AOSD). Join points define the places where two concerns i.e. core and aspectual, crosscut each other [2] [3] [5] [6] [7] [8] [9]. Main task of aspect-oriented introductions [2] [5], among all, join point is more important. Join point is defined as a well defined execution point in a program [3].

Join points define the places where two concerns i.e. core and aspectual, crosscut each other [2] [3] [5] [6] [7] [8] [9]. Main task of aspect-oriented designers is to identify set(s) of join points, where two concerns interconnect to each other, and provide suitable representation for join points [8] [9] [10].

In many cases, a join point is superimposed by multiple aspects at the same time, known as a shared join point [5] [8] [10]. There are many example scenarios (one discussed in section 4), where multiple aspects are being superimposed on the same join point [8] [10] [11]. Currently, there are problems with shared join points at implementation level due to uncertain execution behavior of superimposed aspects [8] [10] [11]. Since multiple aspects are being superimposed, it becomes difficult to judge what will be the exact execution order? If an aspect does not work; how to control the execution order of other aspects? There is not sufficient support available for these issues at implementation level, but there are some indirect support and recommendation details for AOP languages [5] [8] [11]. For example, AspectJ provides precedence construct for ordering and do not provide any direct support for controlling [5].

These issues are novel and being discussed at implementation level only [8] [10]. These issues require the strategies to order and control the superimposed aspects at run time. The strategy presented at implementation level requires to be modeled for shared join points at the early software development stage [12] [13]. Due to the significance of join points, particularly, the shared join points, the representation of issues regarding the shared join points in an aspect-oriented development environment is a major task for aspect-oriented designers at high level design. This high level representation can reduce the work cost by the formulation of early design decisions.

Software design is an important activity in the software development. It is like a blueprint of the software to be built [14].Recently, Aspect Oriented Software Development is making strong progress on the implementation level, but the extensive support at design level is still insufficient [13].

Unified Modeling Language (UML) [7] of OMG group is one of the most popular modeling languages to design different artifacts of the software systems. UML [15] provides numerous diagrams to model properties of a software system [16]. It has become an industry standard now for a while. It provides a verity of diagrams that can be used to model software for aspect oriented development paradigm [17] [18]. Among these diagrams, state charts are very important to model the dynamic behavior of the system. When the decision on what action is to take in response to a given input, the state chart is an effective design tool [18]. This work explains how the shared join points can be modeled using state charts of UML 2.0 at high level design. It also proposes a methodology to formulate the strategies based on dynamic decisions at high level design and finally the formulated strategy is implemented in AspectJ, one of the most prominent Aspect Oriented Programming languages [5].

The rest of the paper is organized as follows; Section 2 provides literature review. Section 3 presents proposed methodology for modeling shared join points with state charts at high level design. Section 4 describes the application of the proposed methodology to a case study, and finally, Section 5 concludes the paper.

## II. Literature Review

A detailed analysis of the problem aroused by shared join points is discussed by Nagy et al. in 2005. Multiple aspects' superimposition on the same join point affects the functionality of each other due to different execution orders among them. Software engineering perspective of Shared Join Point problems is also discussed. It is recommended that, to offer one solution which satisfies only a single case is not preferred. AOP languages should offer a rich set of language mechanism for composition specifications, so that, the developers may choose the right specification for their problem. It is very important to identify conflicts among aspects at shared Joint point for the safety and correctness. The already presented core model [10] is enhanced by adding more constraints and the composition rules for multiple constraints. The integration of the purposed model with AspectJ is also presented. This model can be used with AspectJ, if AspectJ support the named advices. Also the Join Point interface has to be extended for this purpose. For ordering, AspectJ uses declare precedence construct and for controlling, the construct presented in Core Model needs language support [8] [10].

Anis C. et al. presented an interaction model on the basis of Interaction Specification Language (ISL) for modularizing crosscutting concerns of component based applications. The main idea of interactions is to rewrite a method body using the reaction (advice). The interaction model is used to handle the issues arouse by the Shared Join Point in a way that, the composition mechanism generates an advice, which is the result of merging all advices at that join point. Whenever a shared join point is reached one single advice is executed, which is semantically equivalent to the composed advice. The merging mechanism is based on a finite set of merging rules. The software engineering properties such as analyzability and predictability can also be achieved by using this tool. Testing and verification becomes much simpler [11].

Mahoney et al. described the importance of extended Finite State Machines in order to capture the dynamic behavior of systems [18]. A state chart is connected to a class that specifies all behavioral aspects of the objects in that particular class. They also describe that Aspect Oriented Modeling can help in bridging the gab between software design and implementation through the use of advanced features of state charts. They have proposed a framework which helps in simplifying the design of core requirements and cross cutting concerns.

Mahoney elaborated the need of crosscutting concerns of reactive systems using state machines. State Charts are used to describe the dynamic behavior of separate concerns. The core and aspectual requirements are represented by state in different orthogonal regions. He addressed the communication mechanism in orthogonal regions through broadcast events. The broadcast events are used as a mechanism for implicit weaving of aspect and core model in state charts [19].

The programming constructs of AspectJ are introduced by Kiczales et al. The application of advices of two conceptually and semantically independent aspects at the same join point is addressed. It also described that the programmer does not need to control relative ordering of such advice [5].

Mohamed Mancona Kande et al. explained the basic concepts of AspectJ, a state of the art Aspect Oriented Programming Language. Standard UML is used for modeling these concepts and limitations of UML are highlighted. Some extensions to UML are proposed to overcome these limitations. A bottom up approach is followed for designing classes and aspects of Aspect Oriented Programming [16].

The concepts of Join Point as Static Join Point and Dynamic Join Point are addressed. UML association classes (along with their new features), ports and connectors are used among components for modeling [7].

Stein D. et al. presented an approach to model the join points with the help of Join Point Indication Diagram (JPID) and Join Point Designation Diagram (JPDD). JPID is presented for the indication of join points in core model while JPDD is presented for the indication of join points in aspects [18] [19] [20], but it does not address any solution for Shared Join Points.

There is massive work on modeling, modeling join points as well as on aspect oriented programming where as the work on shared point is only at implementation level. There is no solution presented by the researchers to represent particular issues of shared join point at high level in formulation of the suitable design strategies for shared join points.

## III. Proposed Methodology

This section presents the proposed methodology based on state charts of UML 2.0. The section is divided into two sub-sections. First describes, why to use state charts for shared join point modeling and second subsection explains the proposed design methodology.

### A. *State Charts for Shared Join Point Modeling*

The Unified Modeling Language (UML) has become industry standard for modeling general and as well as for specific purpose software artifacts. State charts in UML 2.0 [15] are very important means of modeling and capturing the dynamic behavior of objects. State chart related to a class, can specify all the behavioral aspect in that class [18]. A state chart diagram is represented through a state machine which models the individual behavior of the object. State machines throughout the UML versions remained almost same [20]. However, some new elements like entry and exit point are introduced in UML 2.0 [15]. Some of the elements of UML 2.0 like, composite state, choice pseudostate and terminate pseudostate are very important means to model the behavior.

### B. *Methodology*

The proposed design methodology mainly uses composite state of UML for the multiple aspects' composition at the shared join point. The model consists of three main composite states. These composite states are composition of the aspectual and core requirements. Aspectual requirements are further subdivided into two composite states; *beforeCompositeState* and *afterCompositeState*. If multiple aspects are superimposed before the core requirement, then their ordering and controlling will be handled in *beforeCompositeState*. And if the multiple aspects are superimposed after the core requirement, then their ordering and controlling will be handled in *afterCompositeState*. This means that each composite state is responsible for handling issue related to superimposed aspects contained by that composite state. The core requirement is composed in the *coreCompositeState*. These composite states are named as just for understanding. So, at the abstract level, the aspectual and core requirement compositions are handled in composite states. The big picture of proposed methodology is shown in Fig. 1.

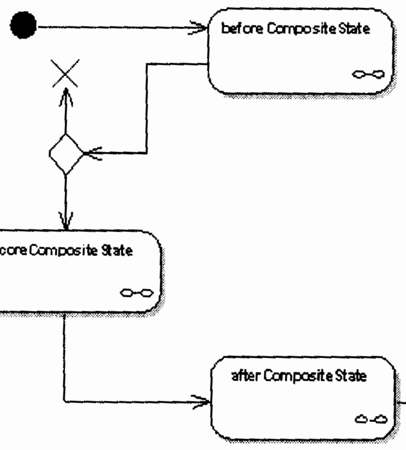

Fig. 1 Abstract view of proposed methodology

## IV. Case Study

To understand the problems of shared join point, it becomes more convenient if we consider following case study. There are also some examples related to these issues, discussed in [8] [11]. The case study is about university course registration system. There are some requirements such as, no student will be allowed to register course without prior submission of fee, and also no course will be registered if its prerequisites are not passed by the student. The system should maintain log and database persistence. In this scenario of university course registration, *CourseRegistration* class fulfils the core requirement of the system. The requirement of course registration is fulfilled by *registerCourse()* method. The other requirements are implemented in different aspects.

Suppose that requirement of logging is implemented by aspect named *Logging*. The responsibility of *Logging* aspect is to maintain the log of every entrance to a method, so *Logging* aspect should run before the *registerCourse()* join point which is now well defined point in the program.

There are other requirements of student fee checking and course prerequisites checking for the course being registered by the student. These requirements are implemented by *CheckFee* and *CheckPreRequisite* aspects respectively. These aspects should also run before the *registerCourse()* join point. Now, all three aspects will be executed on the *registerCourse()* join point at the same time. In other words this join point is being shared by the multiple superimposed aspects.

The problem of ordering and controlling among these three aspects arises. If either of the aspects does not provide the desired results, the course should not be registered. The last requirement is to check the database persistence implemented by *DBPersistence* aspect, which ideally should run at the end when course has been registered.

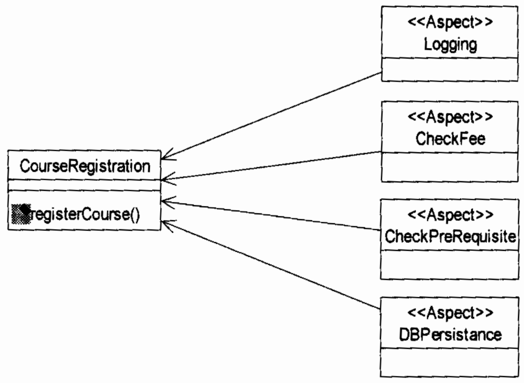

Fig. 2 Multiple aspects' composition

Suppose that database is up and *DBPersistence* aspect should work perfectly if the course has been registered. Now, the superimposition of three aspects requires great efforts to handle their ordering and controlling issues. The superimposition of the multiple aspects at the join point *registerCourse()* are shown in Fig. 2. All four aspects are superimposed with *registerCourse()* join point. The ordering and controlling issues are discussed at implementation level and some strategies and software engineering rules are also highlighted in [8] [11]. In order to define better ordering and controlling strategies, one needs to model shared join points at high level design, so that the ordering and controlling strategies can be implemented perfectly at implementation level. This reduces the work cost by identifying suitable strategies at the early software development stage.

## *A. Application of the Proposed Methodology*

The proposed methodology uses state charts, a design constructs of UML 2.0 [15] to represent shared join point for the case study discussed in university course registration scenario.

This model comprises of three main composite states; *beforeCompositeState* is to formulate the aspects that need to be run before the core functionality which will be composed in the second composite state *coreCompositeState.* Last composite state is *afterCompositeState* which should contain the aspect(s) that require(s) to be run after the core functionality as shown in Fig. 3. Purposed model can be customized. For example, there can be another composite state for those aspects which need to be executed before as well as after the core requirement i.e. around the core requirement.

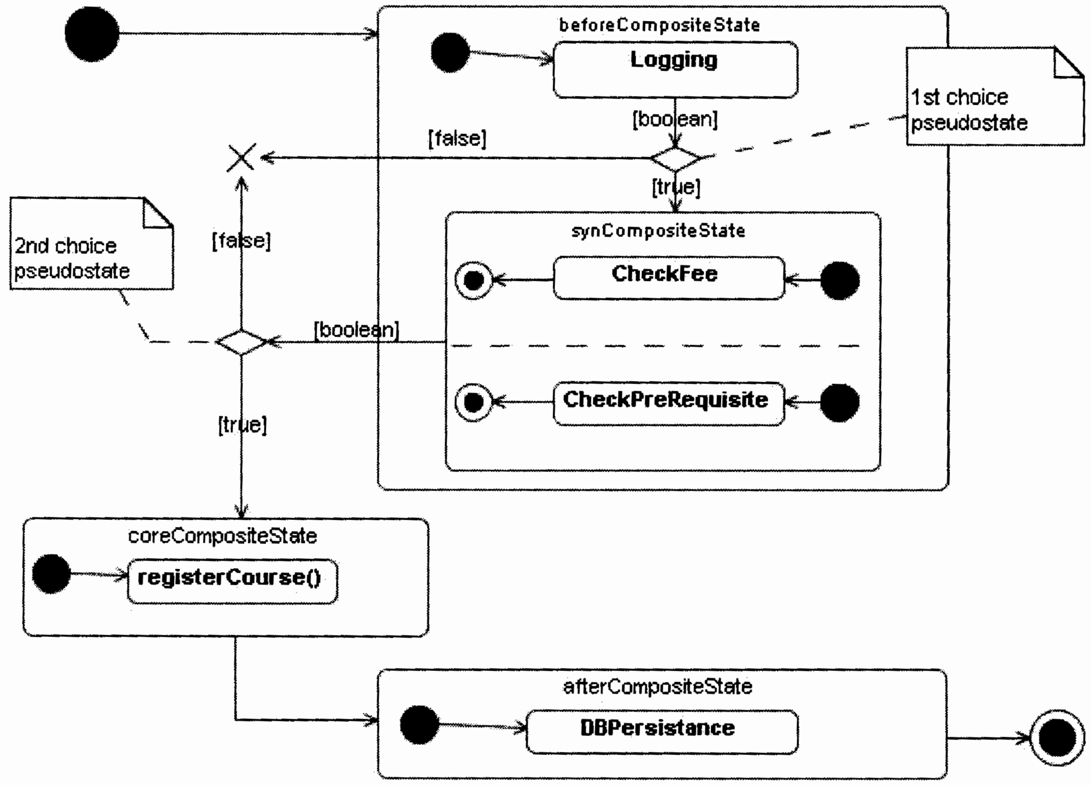

Fig. 3 Detail design of proposed methodology for case study

Two different categories of composite states are defined; one implementing the core requirement and other implementing the aspectual requirement. We further sub-categories the aspectual composite state into before and after composite states. By having three composite states at abstract level, we can easily model the core and aspectual requirements in a systematic way.

We are more concerned with the first composite state i.e. *beforeCompositeState* that contains the aspect need to be executed before the core requirement and on the bases of their results the core requirement will be implemented. This composite state contains an aspect *Logging* and a sub-composite state *syncCompositeState* which contains two aspects *CheckFee* and *CheckPreRequisite* in its orthogonal region for their concurrent execution. The aspects in *beforeCompositeState* are superimposed and required to be ordered and controlled. If we consider the above discussed scenario, the *Logging* aspect should always run first. This *Logging* aspect will transmit boolean guard value to first choice pseudostate as input, which will evaluate these boolean guard values and decides where to transmit the object. If the boolean guard value is evaluated as true, the object will be forwarded to *syncCompositeState* otherwise it will be sent to terminate pseudostate, which destroys the object action. If the object is in *syncCompositeState* and the boolean guard value has been referred to *CheckFee* and *CheckPreRequisite* aspects in the orthogonal regions, both will synchronously run and their results will be transmitted as boolean guard values to second Choice pseudostate that evaluates the boolean guard value. If the value is true, it implements the course registration core requirement otherwise object will be sent to terminate pseudostate. Synchronous handling of two aspects takes place because they don't require any ordering constraint. Any one of the aspects in *syncCompositeState* could run after the other. Also the synchronous handling of these aspects in the orthogonal regions will be handled by default as run to-completion, in which each event is handled before the next occur [20].

The *DBPersistence* aspect is an aspect that should run after core requirement, so there is no need to order or control this aspect. This representation allows the designers to show a high level ordering and controlling mechanism for superimposed aspects. It also shows how to resolve ordering and controlling issues discussed. It provides flexibility to designers to apply any of the strategies given in [9] at high level design. By selecting a suitable strategy to resolve the issue at high level design will help the programmers to implement the strategy in an ideal way. It provides additional feature of handling the new aspectual requirement in systematic way. The presented design of shared join point allows the designer to represent shared join point independent of the implementation details at abstract level [21]. The design at such an abstract level can provide benefits like scalability, by representing new conflicting aspects in the model, reusability, reusing the design strategy for other similar shared join points, and maintainability, if any of the conflicting aspects to be removed or to be added by identifying its effect on the other aspects. Early representation of ordering and controlling issues reduces the work cost by identifying the complexity of issues and formulating suitable strategies to solve the issues accordingly. Early the problem is identified cheaper to solve.

## V. Conclusion

Previously shared join points were discussed at implementation level and to the best of our knowledge there is no research done for this particular problem at design level. The paper describes the need of modeling shared join point at high level design. In this regard, state charts of UML 2.0 have been used for modeling at high level. The main elements of state charts are composite states which compose the superimposed aspects at shared join point. At abstract level, the composite states are used to categorize into core and aspectual requirements. The strategies for ordering and controlling are implemented through detailing the state charts using the choice pseudostate as dynamic selection element and guard values to evaluate the next transition. State charts represent a better way to handle shared joint point Issues at high level design. This allows the designers to design issues regarding the shared join points at early design stages, and programmers can implement these strategies accordingly. A case study is presented with proposed methodology, which shows in detail how issues(s) regarding shared join point(s) can be represented at high level design.

## References

[1] Aspect-Oriented Software Development. http://www.aosd.net.

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, Ch., Loingtier, J.and Irwin, J.: *"Aspect-Oriented Programming"*. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97) (Yväskylä, Finland, June 9-13, 1997). Springer-Verlag, Berlin Heidelberg, 1997, LNCS 1241, Pages 220–242.

[3] Stein, D., Hanenberg, St., Unland, R..: *"An UML-based Aspect-Oriented Design Notation For AspectJ"*. In Proc. Of AOSD '02 (Enschede, the Netherlands, Apr. 2002), ACM, pp. 106-112.

[4] Wesley Coelho and Gail C. Murphy: *"Modeling Aspects: An Implementation-Driven Approach"*. Workshop on Best Practices for Model Driven Software Development at OOPSLA 2004.

[5] Kiczales, G, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold: *"An Overview of AspectJ"*. In Proceedings of ECOOP 2001, LNCS 2072, Springer Verlag, 2001.

[6] Boucke N., Holvoet T.: *"State-based join points: Motivation and requirements"*. In Filman, R. E., Haupt, M., Hirschfeld, R. (eds): Proceedings of the Second Dynamic Aspects Workshop (2005) 1-4.

[7] Eduardo Barra Zavaleta, Gonzalo Génova Fuster, Juan Llorens Morillo: *"An approach to Aspect Modelling with UML 2.0"*. Workshop on Aspect Oriented Modeling, October 11, 2004, Lisbon, Portugal, held in Conjunction with the 7th International conference on the Unified Modeling Language- UML 2004, October 10-15, 2004, Lisbon, Portugal.

[8] I. Nagy, Lodewijk Bergmans and Mehmet Aksit: *"Composing Aspects at Shared Join Points"*. Proceedings of International Conference NetObjectDays, NODe2005.

[9] Stein, D., Hanenberg, S. and Unland, R.: *"On Representing Join Points in The UML"*. In Proceedings of the 2nd Workshop on Aspect Modeling with UML at the Fifth International Conference on the Unified Modeling Language and its Applications (UML 2002), (Dresden, Germany, 30 September – 4 October, 2002).

[10] I. Nagy, L. Bergmans, M. Aksit.: *"Declarative Aspect Composition"*. Technical Report, http://trese.ewi.utwente.nl/publications/publications.php?action=showPublication&pub_id=346 University of Twente, (April 2005).

[11] Anis Charfi, Michel Riveill, Mireille Blay-Fornarino, Anne-Marie Pinna-Déry: *"Transparent and Dynamic Aspect Composition"*. In Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT), Bonn (Germany), 21 march 2006.

[12] A. Rashid, N. M. Ali: *"A State-based Join Point Model for AOP"*. Workshop on Views, Aspects and Roles — VAR (held with ECOOP 2005).

[13] Stein, D.; Hanenberg, S.; Unland, R.: "Modeling Pointcuts. Early Aspects". Workshop on Aspect-Oriented Requirements Engineering and Architecture Design, AOSD 2004, Lancaster, UK, March 22, 2004.

[14] Eric Braude: Software Design: From Programming to Architecture. John Wiley & Sons, Inc. 2004.

[15] Object Management Group: "Unified Modeling Language". Superstructure, version 2.0 formal/05-07-04.

[16] Mohamed Mancona Kande, Jorg Kienzle and Alfred Strohmeier, "From AOP to UML- A Bottom-Up Approach", Swiss Federal Institute of Technology Lausanne, Switzerland. [2001].

[17] A. Rashid, Araujo J., A. Moreira, and I. Brito: "Aspect-Oriented Requirements with UML". Workshop on Aspect-Oriented Modeling with UML (held with UML 2002).

[18] Mahoney, M., Bader, A., Aldawud, O., Elrad, T.: "Using Aspects to Abstract and Modularize Statecharts." The 5th Aspect-Oriented Modeling Workshop in Conjunction with UML 2004. http://www.cs.iit.edu/~oaldawud/AOM/mahoney.pdf.

[19] Mark Mahoney: "Modeling Crosscutting Concerns in Reactive Systems with Aspect-Orientation". Doctoral Symposium at MoDELS/UML 2005, Montego Bay Jamaica, October 2005.

[20] Michelle Crane, Juergen Dingel: *"UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal"*. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005).

[21] Stein, D.; Hanenberg, S.; Unland, R.: *"Position Paper on Aspect-Oriented Modeling: Issues on Representing Crosscutting Features"*. 3rd International Workshop on Aspect-Oriented Modeling with UML, AOSD 2003, Boston, MA, March 18, 2003.