Aspect Design Pattern for Non Functional Requirements

T5062

Developed by:

Ansar Siddique
Fazal-e-Amin
MS(Software Engineering Fall 05)

Supervised by:
Dr. Hafiz Farooq Ahmad

DEPATRTMENT OF COMPUTER SCIENCE
FACULTY OF BASIC & APPLIED SCIENCES
INTERNATIONAL ISLAMIC UNIVERSITY ISLMABAD

In the Name of

ALLAH

The Most Merciful

The Most Beneficent

PROJECT IN BRIEF

Project Title:	Aspect Design Pattern for Non Functional Requirements
Organization:	International Islamic University Islamabad, Pakistan.
Objective:	The objective of the project is to fulfill the degree requirement of MS in Software Engineering.
Undertaken By:	Ansar Siddique 70-FAS/MSSE/F05 Fazal-e-Amin 84-FAS/MSSE/F05
Supervised By:	Dr. Hafiz Farooq Ahmed Associate Professor SEECS, NUST Rawalpindi
Started On:	February 2007
Completed On:	June 2008
Research Area	Aspect Oriented Software Development (AOSD)

International Islamic University, Islamabd
Faculty of Basic & Applied Sciences
Department of Computer Science

Dated: 30 August, 2008

FINAL APPROVAL

It is certified that we have read the thesis, entitled "Aspect Design Pattern for Non-Functional Requirements", submitted by Mr. Ansar Siddique 70-FAS/MSSE/F05 and Mr. Fazal-e-Amin 84-FAS/MSSE/F05 it is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University Islamabad for the award of MS degree in Software Engineering.

PROJECT EVALUATION COMMITTEE:

External Examiner
Zafar Iqbal Malik
Director EMIS
Ministry of Education

[illegible]

Internal Examiner
Muhammad Usman
Lecturer
Department of Computer Science
International Islamic University

[illegible]

Supervisor
Dr. Hafiz Farooq Ahmed
Associate professor
SEECS, NUST
Rawalpindi

[illegible]

A thesis submitted to the Department of Computer Science,

Faculty of Basic & 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

OUR DEAREST PARENTS & RESPECTED TEACHERS

Their efforts and guidance

Made us able to achieve this endeavor,

Without

Their prays and support

This dream could have never come true

Declaration

We hereby declare and affirm that this thesis neither as whole nor as part thereof has been copied out from any source, we have provided proper references and citations wherever required. It is further declared that we have completed this thesis 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.

Ansar Siddique
70-FAS/MSSE/F05
Fazal-e-Amin
84-FAS/MSSE/F05

ACKNOWLEDGEMENTS

First of all we express our sincere thanks to ALLAH and bestow all praise and appreciation to Almighty Allah, The most Merciful and Compassionate, The Most Gracious and beneficent, whose bounteous blessings enabled us to pursue and perceive higher ideals life, who bestcwed us good knowledge to complete our work successfully. And Specially we are grateful to His Holy Prophet Muhammad (SAW) who enabled us to recognize our creator and provide us a true path to follow for success in this world and for hereafter.

Secondly we wish to express our profound gratitude to our supervisor **Dr.Hafiz Farooq Ahmed**, whose suggestions led us throughout this thesis. This thesis would not have been possible without the kind support, the trenchant critiques, the probing questions, and the remarkable patience of our research advisor.

Dr. Naveed Ikram for his inspiring attitude and kind behavior throughout the project effort. And to advisory committee especially to **Mr. Usman** who provide us guidance and support to accomplish this challenging task.

Finally we are thankful to our parents who ever provide us warm encouragement, love and moral support during our entire academic career.

Ansar Siddique
70-FAS/MSSE/F05
Fazal-e-Amin
84-FAS/MSSE/F05

Sr. No.	Topic	Page No.
	Chapter-1 Introduction to Aspect Orientation	
1.1	Introduction	1
1.2	Motivation for AOP	1
1.3	Aspect Oriented Programming	4
1.3.1	Aspects	6
1.3.2	Why Aspects?	6
1.3.3	Types of Crosscutting	6
1.3.4	Development & Production Aspects	8
1.3.5	Aspect Weaving	9
1.4	Design Pattern	9
1.4.1	Brief history of design patterns	9
1.4.2	Design Pattern	9
1.4.3	Elements of Design Pattern	11
1.5	Design pattern Dcscription	11
1.5.1	Categories of Patterns	12
1.5.1.1	Structural patterns	12
1.5.1.2	Creational patterns	12
1.5.1.3	Behavioral patterns	13
	Chapter 2 Literature Survey	
2.1	Introduction	14
2.1.1	Terminologies	14
2.2	Aspect Model	16
2.3	Implementation of Design Pattern using AOP	17
2.4	Aspect Oriented Technology and Design Patterns	18
	Chapter 3 Definition of the Problem	
3.1	Problem definition	19
3.2	Research questions	20
3.3	Research objectives	20
3.4	Comparative analysis of Aspects & design pattern	20
3.5	Output	23
3.6	Research method	23
3.7	Validation of pattern	23
3.8	Scope	24
	Chapter 4 Proposed Solution	
4.1	Identification of Recurring Aspect	25
4.2	Identification of Pattern	25
4.3	Proposed Pattern	25
4.4	Discussion on Proposed Pattern	26
4.5	Description of Pattern Elements	27

Chapter 5 Implementation

5.1 Case Study 1(Shopping Cart) 30
5.1.1 Concerns 30
5.1.2 Basic course of action: Narrative style 31
5.1.3 Application of Proposed Solution on Case Study.......... 32
5.2 Case Study 2(Student Registration System) 35
5.2.1 Concerns 35
5.2.2 Basic course of action: Narrative style 36
5.2.3 Application of Proposed Solution on Case Study.......... 38
5.3 Case Study 3(Motorway Toll System) 41
5.3.1 Concerns 41
5.3.2 Basic course of action: Narrative style 42
5.3.3 Application of Proposed Solution on Case Study.......... 43

Chapter 6 Validation

6.1 Quality of proposed pattern.......... 46
6.1.1 Encapsulation.......... 46
6.1.2 Generativity.......... 46
6.1.3 Abstraction.......... 47
6.1.4 Openness.......... 47
6.1.5 Composability.......... 48
Conclusion.......... 49
References
Appendix A Publication
Appendix B Terminologies

List of Figures

Sr. No.	Name	Page no.
1.1	Working of Aspect Oriented Programming	4
1.2	Join point illustration	5
2.1	Aspect Model	16
3.1	Abstract diagram of proposed solution	24
4.1	Abstract diagram of proposed solution	27
4.2	Structure diagram	28
5.1	Use Case diagram of shopping cart case study	31
5.2	Structure of classes case study-1	34
5.3	Use Case diagram of student registration system	36
5.4	Structure of classes case study-2	39
5.5	Use Case diagram of Motorway toll system	40
5.6	Structure of classes case study-3	44

ACRONYMS & ABBREVIATIONS

OOP:	Object Oriented Programming
AOP:	Aspect Oriented Programming
AOSD:	Aspect oriented Software Development
OOA:	Object Oriented Analysis
OOD:	Object Oriented Design
OMT:	Object Modeling Technique
AP:	Adaptive programming
CF:	Composition Filter
SOP:	Subject Oriented Programming
SOC:	Separation of Concern
MDSOC:	Multi Dimensional separation of Concerns
ASOC:	Advance Separation of Concern
GOF:	Gang of Four
API:	Application Programming Interface
GUI:	Graphical User Interface

ABSTRACT

Aspect Oriented Technology has emerged in recent years with the invent of the aspect oriented programming language and provided the solution to the problems that were not handled by the existing Object Oriented and Procedural approaches of software development. Initially the software practitioners used the aspect oriented language for the implementation (coding) afterwards the researchers and software developers found that it is not enough to use the aspect oriented technology just for the coding, rather this technology should be used throughout the software development to get the true benefits of this technology and an other motivation for this thought was the gap which remain between different software artifacts as the design is supported by the standard modeling languages which have no representation of aspects. So, the aspects have no representation at design level and the coding is in aspect-oriented language that is why the trace-ability between software artifacts is an issue to be solved. Currently there is a lot of work in progress to identify the aspects at the analysis level and to represent them at design level. We have contributed in this effort by developing aspect oriented design pattern. This pattern deals with crosscutting concerns like logging. And the pattern bridges the gap existing between different stages of aspect oriented software development life cycle.

Chapter-1
Introduction to Aspect Orientation

1.1 Introduction

Software engineering is a relatively a new field, and it has continued to evolve rapidly since its inception. In order to improve the ability to produce qualitative software systems researchers and theorists are always seeking for new dimensions. The basic desire is to produce engineering methods that allow for the efficient creation and maintenance of software. There are several key software quality attributes that software engineers desire to improve:

• The **modularity** of the software
• The **usability** and **reusability** of the software
• The **readability** and **understandability** of the implementation
• The **correctness** and **testability** of the software

There are already remarkable improvements in the field of software engineering like object-oriented technology has made great progress in improving above mentioned qualities, but still there is much room for improvements, even in a well-implemented object-oriented program, there is often functionality spread over most of the modules in a system. This functionality can include such things as **security handling, logging, synchronization, memory management** and other more advanced functionality like **state management**. These and other properties are called **crosscutting concerns**.
A **crosscutting concern** is a characteristic of a software implementation that is spread throughout the implementation, instead of being modularized. Devising methods to modularize these properties is the chief concern of **Aspect-Oriented Programming** [1].

1.2 Motivation for AOP

Before discussing the details of aspect oriented programming we will see the potential uses of this technology and the particular problems that it tries to resolve. A very basic problem is the need to use development code, such as logging or contract reporting code, to help create and test software. Inserting this type of code into a software implementation can be a tedious and time-consuming process. To compound the problem, much of this code is only useful during the development phases of a project, and leaving this code in a production build is often undesirable because it can create performance problems, as well as inadvertently introduce software defects [2].

With the use of aspect-oriented programming techniques, it is possible to implement this development code in a modularized fashion, so that it can be inserted easily and efficiently into the production code. Additionally, because this code is well modularized, it can be easily maintained or removed as well [1].

Overall, this usage of aspect-oriented programming can help to decrease development time, improve production system performance and minimize the chance of development code introducing defects into the implementation [3].

However, note that this is a trivial usage of aspect-oriented programming as software developers can accomplish this functionality with macros or other similar tools.

There are other more complicated problems with software development that aspect-oriented programming is trying to solve, such as state management, synchronization, and session management [3]. With existing software development tools, it is difficult to modularize this type of functionality. To explain, the difficulty with object-oriented design methods is that they rely on the modularizing the system into components based on decomposition into functional units, represented by classes. A clean object-oriented design may then have to be modified to add a feature, such as state management, which will involve several functional units. Therefore, the code that implements that feature will need to be placed in all of those components[3].

The intent of aspect-oriented programming is to create language mechanisms that allow all the functionality present in the system to be modularized, including the functionality that is scattered throughout multiple components of the system[2]. If this is done, it can greatly enhance the maintainability, reusability and other qualities of the software. Although this is similar to the goals stated earlier dealing with development code, the specific goal here is to allow all of the main functionality of an implementation, whether or not it is a **crosscutting concern,** to be quickly and efficiently modularized.

It is easy to see how modularizing aspects can increase the maintainability of a program since it allows changes to particular functionality to happen in only onc location, and this can lead to more reusable code. The idea is that, for example, an aspect that controls the screen updates for one drawing can be used again, without modification, to provide the same functionality to another similar program. This is where the most exciting potential of aspect oriented technologies rests.

Another, added benefit of aspect-oriented programming is to produce software that is efficient to run, without sacrificing other qualities, such as the **readability** and **maintainability** of the code. As an illustration of this, consider the example explored in the paper Aspect-Oriented Programming, by Kiczales,et.al. paper describes an experiment in which the authors create three separate implementations of an algorithm that is part of a graphics-filtering program [2].

The first implementation is a hand-coded algorithm that is well modularized, using procedural techniques, and easy to read. Unfortunately, this implementation is highly inefficient in both execution time and storage requirements.

The second implementation is a hand-optimized version of the latter. This version, while much more efficient in both execution time and space requirements, is very difficult to read and understand by anyone, including the original author. The reason is that since many different concerns are tangled within a very complicated procedure.

The third implementation makes use of aspect-oriented techniques to construct a working unit of code that is both easy to read and maintain, and also roughly as efficient as the hand optimized solution.

So far, only functional uses of aspect-oriented programming have been discussed, but there are many other uses of aspect-oriented programming that are not specifically related to functionality. An example of a non-functional use includes enhancing the readability of the implementation. The readability of the code is important because it can affect many other qualities, such as the understandability, maintainability, correctness and reusability of the code. The techniques of aspect-oriented programming have the potential to create a more readable code base, because they can physically separate the different functional concerns in the code In effect, aspect-oriented techniques create layers of functionality in a software implementation. Each new aspect introduces a new functional characteristic of the program, and each layer can be read separately from the others. If properly done, this can make understanding and implementation much easier.

1.3 Aspect-Oriented Programming

Before the influence of aspect-oriented techniques on software engineering are discussed, it is important to understand the basis of these techniques. There are many different languages that take advantage of aspect-oriented technology, all at various stages of development. AspectJ is used because it is one of the more mature projects.

AspectJ is "a simple and practical extension to the Java programming language that builds upon the object model of Java with enhancements that allow aspect-oriented programming techniques to be used" [1]. It is compiled into standard Java bytecode, and it is able to run on any Java platform.

Aspect-Oriented programming is a method of software engineering that is intended to build upon the earlier successes of procedural, functional and object-oriented programming by introducing aspect-oriented techniques to these programming paradigms. It does not intend to replace these programming techniques, but rather to augment and improve their abilities [4]. The aim of aspect-oriented programming is to allow the clean modularization of crosscutting concerns using aspects.

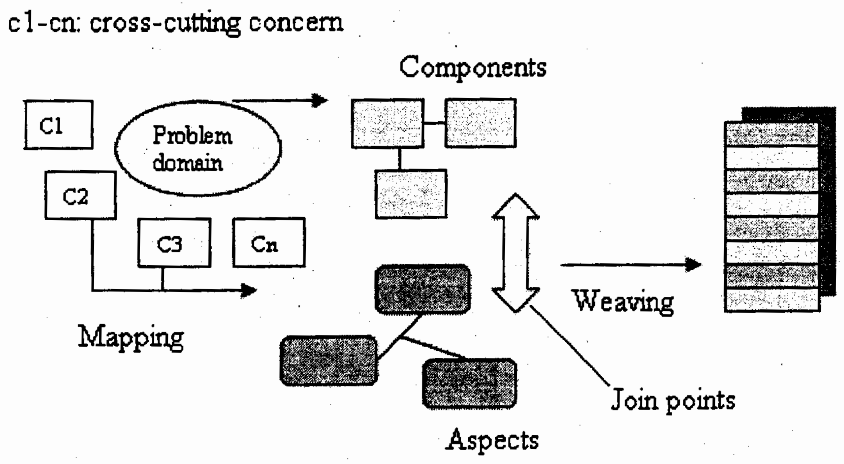


Figure 1.1: Working of Aspect Oriented Programming

AOP Mechanisms

Abstraction Mechanism:

An *aspect description languages* used to encapsulate crosscutting concerns into modules according to the *join point model*

Composition Mechanism:

A *weaver* is used to merge the aspects and components into an application that only contains traditional language constructs.

Joinpoint Model

- *Identifying joinpoints*
    - Points in the execution of components where aspects should be applied i.e. method invocation or object construction

- *describe behavior at join points*
    - Wrap join points with extra code just before or jus after execution. i.e. log before and after method or lock and unlock shared data.

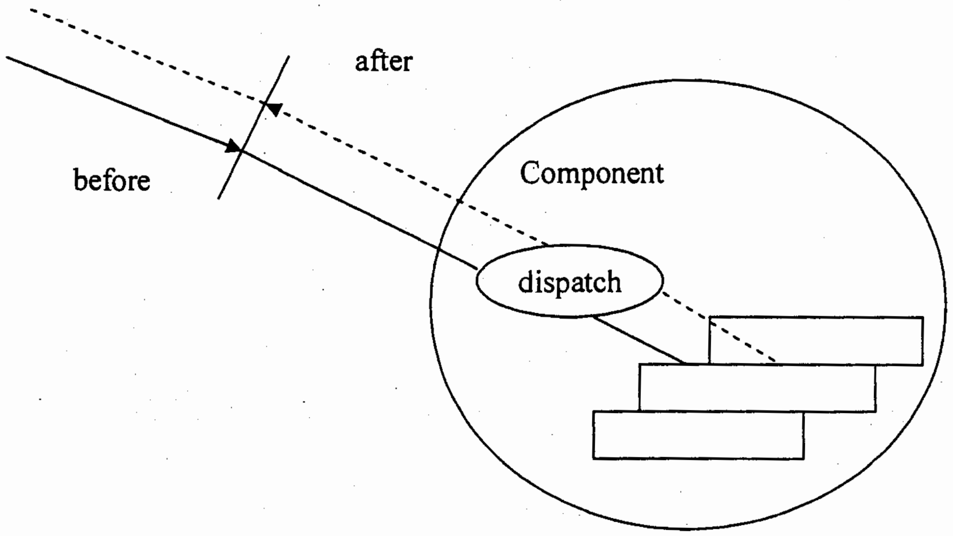


Figure 1.2 Join Point Illustration

1.3.1 Aspect

Aspect-Oriented techniques provide mechanisms that allow crosscutting concerns to be expressed as separate units from the main implementation. These units are referred to as **Aspects**, and they are the basic unit of modularization for crosscutting concerns in aspect-oriented programming. However, as aspect-oriented programming is only intended as an extension to existing programming methods, aspects work in conjunction with a base implementation represented with other constructs, such as classes or procedures more specifically known as **Component** [4].

1.3.2 Use of Aspects

Aspects are designed to allow crosscutting concerns to be easier to maintain, and more reusable. For example, in the case of logging, the programming statements that generate log entries for the entire implementation can be maintained in one aspect, and changes to those statements need only be made in just one place, versus having to modify the entire code base. In languages such as AspectJ, aspects are represented in structures that are very similar to classes.

Following are some of the advantages of the aspects:

- A system concern is treated in one place and can be easily changed.
- Evolving requirements can be added easily with minimal changes to previous version
- Configurable components become practical ("on demand computing")
- Reuse of code that cuts across usual class hierarchy to augment system in many places.

1.3.3 Types of Crosscutting

There are two types of crosscutting that an aspect can facilitate. The first type is called **dynamic crosscutting**. Dynamic crosscutting makes it possible to "define additional implementation to run at certain well-defined points in the execution of the program" [4]. Dynamic crosscutting, contrary to the appearance of the name, does not mean that the code is modified at runtime.

Instead, dynamic crosscutting refers to the selective modification of the primary abstraction at certain points of the program without affecting the static type signature of the program [1].

There are different methods used to define dynamic crosscutting in aspect-oriented programming. AspectJ, and languages similar to it, use the concept of a **join-point** to facilitate the introduction of aspect code into the primary abstraction. **Join-points** are the "Well-defined points in the execution of a program". Put simply, join-points are places in the program code that are easily distinguishable from each other and the rest of the code. Examples of join-points include the beginning and end of a method or function, an object instantiation, and an exception handler execution.

When dynamic crosscutting is used in an aspect, it has two crucial parts. These parts are the new implementation code to add to the primary abstraction, and a specification of where to add it. In the AspectJ language, these parts are called the **advice** and the **point-cut**, respectively. To be more specific, a piece of advice is a method or procedure-like construct used to define additional behavior at a join-point, and point-cuts are a means of referring to collections of join points [4].

As this definition suggests, a **point cut** can refer to more than one join point in the primary abstraction. The process of inserting an aspect's advice into the places designated by the point cut is commonly referred to as aspect weaving.

When a developer writes a piece of advice, they specify which point-cut or point-cuts that the advice should be inserted at, as well as the temporal ordering of the insertion of the advice. To that effect, there are three types of advice, called **before**, **after**, and **around** advice. The different types of advice correspond to the temporal placement of the advice at the join-points defined by the point cuts. For example, if a before advice is inserted at a join-point which refers to the start of a method, then the advice is inserted before the rest of the method body. The temporal placement of before and after advice is clear, but around advice requires some explanation. Around advice is advice that can selectively preempt the normal computation at the join-point. [4] This means that the advice can run instead of, or in addition to, the code at the join point [1].

In addition to dynamic crosscutting, aspects can modify the static structure of other elements in a program, a process called **static crosscutting**.

This type of crosscutting, referred to as introduction in the AspectJ language, is similar to dynamic crosscutting in that it introduces additional implementation into the primary abstraction. However, instead of modifying the behavior of the primary abstraction at a Join-point, it defines or modifies new members in the primary abstraction. For instance, in AspectJ introductions can add methods or fields to an existing class, modify an existing class to inherit from another, implement an interface in an existing class, and convert checked exceptions into unchecked exceptions. [1] This is a powerful use of aspect-oriented programming, because it not only changes the behavior of components in an application, but also changes their relationship. [1]

1.3.4 Development and Production Aspects

Aspects can be used at many points in the system development life cycle, but generally there are two types of aspects. One type is called a development aspect. A development aspect is intended only for use during the development of software, and is expected to be removed from the final application [3]. This means that the functionality that the aspect provides will not be included in a production release. A good example of a development aspect is one that deals with execution logging or contract checking. Generally speaking, a developer will only need a contract checking aspect while he or she is trying to develop and test the software, and would not necessarily want that aspect to be included in the final product.

The other type of aspect is called a production aspect. Unlike a development aspect, these aspects deal with code that is intended to be used in the normal operation of the software. [3] The classic example of this type of aspect is an aspect that controls screen updates for a system, similar to the role of the observer in the observer pattern. In fact, many of the classic design patterns can be implemented with aspect-oriented techniques [9]. These are the types of aspects that are of the most interest to researchers. Aspect-oriented programming has the potential to make it easy to modularize these types of operations to make them easier to create and maintain for future developers.

1.3.5 Aspect Weaving

Finally, there are two ways in which aspects are currently woven into the primary abstraction. The first method is when the weaving process takes place at compile time, rather that at runtime [3]. This is sometimes called static aspect weaving. Static aspect weaving is the method that AspectJ uses to weave aspects into the primary abstraction [1]. The other method is a weaving process that occurs at the program run-time, sometimes referred to as dynamic aspect weaving. This type of aspect weaving has the advantage of allowing aspects to be removed from the primary abstraction, or "unwoven", at runtime [3].

1.4 Design Patterns

1.4.1 Brief History of Design Patterns

Design patterns were first described by architect Christopher Alexander in his book *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977). The concept he introduced and called *patterns*, abstracting solutions to recurring design problems, caught the attention of researchers in other fields, especially those developing object-oriented software in the mid-to-late 1980s.[5]

Research into software design patterns led to what is probably the most influential book on object-oriented design: *Design Patterns: Elements of Reusable Object-Oriented Software* ,by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995); These authors are often referred to as the "Gang of Four" and the book is referred to as the Gang of Four (or GoF) book.

1.4.2 Design Patterns

In "Understanding and Using Patterns in Software Development", Dirk Riehle and Heinz Zullighoven give a nice definition of the term "pattern" which is very broadly applicable:

"A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts"[6].

The above authors point out that, within the software patterns community, the notion of a pattern is "geared toward solving problems in design." More specifically, the concrete form which recurs is that of a solution to a recurring problem. But a pattern is more than

just a battle-proven solution to a recurring problem. The problem occurs within a certain context, and in the presence of numerous competing concerns. The proposed solution involves some kind of structure which balances these concerns, or "forces", in the manner most appropriate for the given context. Using the pattern form, the description of the solution tries to capture the essential insight which it embodies, so that others may learn from it, and make use of it in similar situations. The pattern is also given a name, which serves as a conceptual handle, to facilitate discussing the pattern and the jewel of information it represents. So a definition which more closely reflects its use within the patterns community is:
"A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces". [6]

A slightly more compact definition which might be easier to remember is:

"A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns". [6]
"Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves".[6]

Alexander describes patterns in [17] as:

Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

"The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing"[6]

Documenting good patterns can be an extremely difficult task, *good* **patterns** do the following:

- **It solves a problem:** Patterns capture solutions, not just abstract principles or strategies.
- **It is a proven concept:** Patterns capture solutions with a track record, not theories or speculation.
- **The solution isn't obvious:** Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns generate a solution to a problem indirectly, a necessary approach for the most difficult problems of design.
- **It describes a relationship:** Patterns do not just describe modules, but describe deeper system structures and mechanisms.
- **The pattern has a significant human component**: All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

1.4.3 Elements of Pattern

The Gang of Four described patterns as "a solution to a problem in a context". These three things problem, solution, and context are the essence of a pattern. For documenting the pattern it is additionally useful to give the pattern a name, to consider the consequences using the pattern will have, and to provide an example or examples.
Different catalogers use different templates to document their patterns. Different catalogers also use different names for the different parts of the pattern. Each catalog also varies somewhat in the level of detail and analysis devoted to each pattern.[5]

1.5 Design Patterns Description

Design Patterns uses the following template (GoF Form):

* **Pattern name and classification:** A conceptual handle and category for the pattern

* **Intent:** What problem does the pattern address?

* **Also known as:** Other common names for the pattern

* **Motivation:** A scenario that illustrates the problem

* **Applicability:** In what situations can the pattern be used?

* **Structure:** Diagram using the Object Modeling Technique (OMT)

* **Participants:** Classes and objects in design

* **Collaborations:** How classes and objects in the design collaborate

* **Consequences:** What objectives does the pattern achieve? What are the tradeoffs?

* **Implementation:** Implementation details to consider, language-specific issues

* **Sample code:** Sample code in Smalltalk and C++

* **Known uses:** Examples from the real world

* **Related patterns:** Comparison and discussion of related patterns

1.5.1 Categories of Patterns:

There are three categories of patterns namely ***Structural, Creational,*** *and* ***Behavioral*** patterns.

1.5.1.1 Structural patterns:

Structural patterns prescribe the organization of classes and objects. These patterns are concerned with how classes inherit from each other or how they are composed from other classes.

Common structural patterns include Adapter, Proxy, and Decorator patterns. These patterns are similar in that they introduce a level of indirection between a client class and a class it wants to use. Their intents are different; Adapter uses indirection to modify the interface of a class to make it easier for a client class to use it. Decorator uses indirection to add behavior to a class, without unduly affecting the client class. Proxy uses indirection to transparently provide a stand-in for another class[5].

1.5.1.2 Creational patterns

Creational patterns prescribe the way that objects are created. These patterns are used when a decision must be made at the time a class is instantiated. Typically, the details of the classes that are instantiated, what exactly they are, how and when they are created and are encapsulated by an abstract superclass and hidden from the client class, which knows only about the abstract class or the interface it implements. The specific type of the concrete class is typically unknown to the client class.

The Singleton pattern, for example, is used to encapsulate the creation of an object in order to maintain control over it. This not only ensures that only one is created, but also allows lazy instantiation; that is, the instantiation of the object can be delayed until it is actually needed.

This is especially beneficial if the constructor needs to perform a costly operation, such as accessing a remote database.[5]

1.5.1.3 Behavioral patterns

Behavioral patterns prescribe the way objects interact with each other. They help make complex behavior manageable by specifying the responsibilities of objects and the ways they communicate with each other.

Behavioral patterns include *observer, strategy, template, concurrency patterns.* [5]

Chapter-2

Literature Survey

2.1 Introduction

We have studied a lot of literature but in this chapter only those papers are included which are directly related to our problem definition. The first paper of this chapter is Aspect Oriented Programming by Kiczales, et al. [2]. He introduced the aspect oriented elements and laid the foundation of Aspect Oriented Software Development (AOSD). He extracts these elements by the implementation of examples, details of his work are mentioned below.

Kickzales et al., in his paper "Aspect-Oriented programming", 1997 coined the term aspect and distinguished the properties as Aspect and components, a component is the property of a system that can be cleanly encapsulated in a generalized procedure, where as the aspect can not be cleanly encapsulated. AOP allows separate specification of the components as well as of the aspects and combine these specifications automatically through the process of weaving. Furthermore they have elaborated the concepts of AOP with help of image processing example. By comparative analysis between the procedural approach and AOP they concluded that AOP implementation of that system was smaller than its OOP counterpart. And given some future direction which was the development of a collection of the Aspects and Components for different applications, languages and study of current system to examine the presence of AOP elements in their design, and development of theoretical support and training methods for AOP and integration of AOP with current approaches.

2.1.1 Terminologies

Following are some of the Terminologies introduced by them:

With respect to a system and its implementation using a GP-based language, a property that must be implemented is:

A component, if it can be cleanly encapsulated in a generalized procedure

(i.e. object, method, procedure, API). By cleanly, it means well localized, and easily accessed and composed as necessary. Components tend to be units of the system's functional decomposition, such as image filters, bank accounts and GUI widgets.

An aspect, if it can not be cleanly encapsulated in a generalized procedure.

Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronization of concurrent objects.

Using these terms it is now possible to clearly state the goal of AOP: To support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system.

In general, whenever two properties being programmed must compose differently and yet be coordinated, we say that they ***cross-cut*** each other

Aspect weaver

Aspect weavers must process the component and aspect languages, composing them properly to produce the desired total system operation. Essential to the function of the aspect weaver is the concept of ***join points***, which are those elements of the component language semantics that the aspect programs coordinate with.

2.2 Aspect Model

Christina et al. in their paper "Theory of Aspects for Aspect-Oriented Software Development"[7] presented an Aspect model of Aspect-Oriented Software Development which is based on the concept of separation of concern and modularity and provided terminologies and concepts of the basic elements of the Aspect Oriented Software development. This paper presents a disciplined, yet still informal, *theory of aspects* – a conceptual framework for aspect-oriented programming that provides consistent terminology and basic semantics for thinking about a problem in terms of the *concepts* and *properties* that characterize the AOP style as an emerging paradigm to software development. These concepts and properties have already been described informally. Following is the pictorial representation of their aspect model.

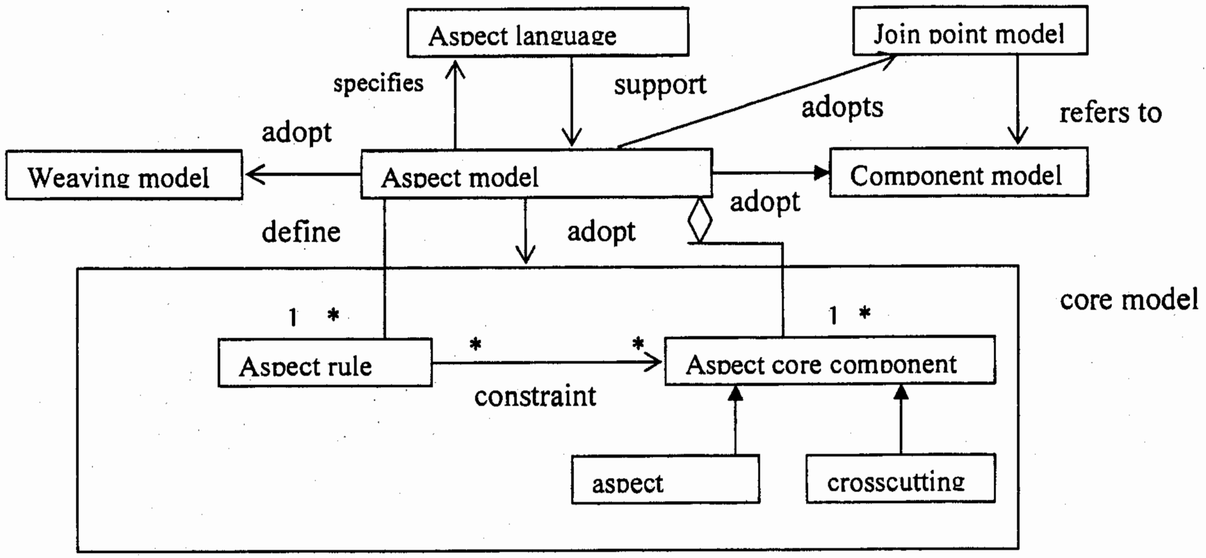


Figure 2.1: The Aspect Model

2.3 Implementation of Design Patterns using AOP

Oufa Hachani et al. in "Using Aspect Oriented Programming for Design Patterns Implementation" [11] has implemented visitor design pattern using AspectJ programming language and proposed the solution to the problems related to the visitor pattern in its implementation using object oriented programming.

Jan-Hannemann et al. "Design pattern Implementation in java & AspectJ" [8] have implemented the GOF pattern in both JAVA and AspectJ and concluded that the implementation in AspectJ resulted in better code locality, reusability and composeability and said that these improvements are directly related to the removal of the crosscutting structures in the patterns.

The degree of improvement in implementation modularity varies, with the greatest improvement coming when the pattern solution structure involves crosscutting of some form, including one object playing multiple roles, many objects playing one role, or an object playing roles in multiple pattern instances.

Deepak Dahiya et al. has raised the issues in "Approaches to Aspect oriented Design: A Study " [10] that there is a gap between the requirements, design and code. The extension of AOP to the modeling level can bridge the gap and provide better traceability throughout requirements to source code.

The author mentioned, for aspect oriented software development (AOSD) to live up to being a software engineering paradigm, there must be support for the separation of crosscutting concerns across the development lifecycle including traceability from one lifecycle phase to another. Concerns that have a crosscutting impact on software (such as distribution, persistence, etc.) present well documented difficulties for software development. Since these difficulties present throughout the development life cycle.

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

Peter Coad in "Object Oriented Patterns" [9] stated about the patterns raised the question that how we can find patterns and presented some object oriented analysis and design patterns with examples and guidelines to use patterns.

2.4 Aspect Oriented Technology and Design Patterns

Oufa Hachani et al. in "On Aspect-Oriented Technology and Design Patterns" [12] has criticized and commented that Aspect oriented implementation improves the design patterns because AO was meant to do so, and can we claim such implementation is AO design pattern and concluded that there is a need to find new patterns related to the Aspect Orientation.

The author commented that it is not very amazing that aspect-oriented programming mechanisms that have been introduced with the hope to provide better programming techniques than the previously existing object-oriented ones can improve object-oriented design patterns.

Chapter-3
Definition of the Problem

3.1 Problem Definition

While observing both the design patterns and AOP, we observed that the design patterns lies at higher level of abstraction and the solution provided by the AOP is at the lower level of abstraction. Here when we say level of abstraction it has various meanings, if we consider the people working on software then the design patterns are used by the analysts and software designers and programmers who implement the design patterns. On the other hand AOP is a programming technique and the programmer is responsible for coding of software, in current situation when design patterns are implemented through AOP it depends on the wisdom of the programmer that how well he understand and translate the software design into AOP code. So we can say that there is a gap between the design patterns and Aspects. As mentioned in [10] the gap between the Aspects and Design Patterns can be bridged by extending Aspects to the modeling level.

But both the design patterns and Aspects have a common characteristic that they provide solution to recurring problem at two different levels of abstraction. Adding to it AOP provides solution to the recurring aspects within an application as well as the same aspect recur in different applications and behave similarly. For example the concerns like security, logging, and error checking etc. are the causes of code scattering and tangling that is why these are treated as aspects in an application on the other hand these aspects recur in different applications.

The work mentioned in [8] and [11] considered the AOP for the implementation of Design Patterns to produce better results which could work well but the question is that the object oriented design patterns are meant to use the concept of object oriented and provide better design solution in object oriented paradigm. On the other hand AOP addresses the limitations of object oriented technology. Due to this difference there are issues of traceability and consistency between different software artifacts. In [12] Hanchi et al. have raised the question that, can we say such implementation of design pattern as aspect oriented design pattern and concluded that there is a need to find design patterns that are directly related to the aspects.

Therefore, we are going to develop an Aspect Pattern for Non Functional Requirements, in which it will be tried to cover the gap between the requirements and design. And we have raised the issues like:

3.2 Research Questions

- How to represent aspect as design pattern?

The first question which rose after the literature survey was about the representation of the aspect i.e. can we represent the aspect as there is a representation format is available for design patterns (the elements of design pattern).

- Does aspect pattern for non functional requirements help at design level?

Can we claim that the aspect pattern for non functional pattern will help at the design level?

- Identification of recurring aspects and their representation at design level and creation of aspect oriented patterns.

3.3 Research Objective

- Comparative analysis of aspect and design patterns.
- Finding elements to describe the Aspects.

As the design patterns are represented through the elements of design pattern likewise can we find the elements which represents an aspect or can we find equivalent description of aspects.

- Comparing the elements of design patterns & aspects.
- Equivalent description of aspects in form of design pattern elements.

3.4 Comparative analysis of aspect and design patterns

After the studying the nature aspects and design patterns, we found the similarity and differences between them on following points:

- Recurring Problem
- Scope
- Representation
- Implementation
- When to use/ guidelines
- Understandability
- Modularity

- Modifiability
- Reuse
- Maintainability

Recurring Problem

The first common characteristic of design pattern and aspect is that both provide solution to the recurring problems, the difference is that the design patterns solve the design problems at design level and AOP provides the solution to the recurring aspects on code level.

Scope

Scope of the design patterns is limited to the design phase and the pattern vanishes in code, so it is difficult to find/see design pattern in code. Design pattern become trivial after coding it. Aspects are not trivial and have a vast scope than the design patterns and aspects remain observable all the time.

Representation

Design patterns are represented by the elements of design patterns; object oriented design patterns uses the constructs of object oriented languages. Aspects are represented by the constructs available in aspect oriented languages.

Implementation

Design patterns are implemented with code and ultimately they vanish in code. After the implementation of design pattern programmer can not trace what patterns were applied and what design decisions were made. Weaving mechanism is used to weave the components and Aspects. Components and Aspects are implemented separately and weaving is the process which combines them together for this purpose aspect oriented languages use weaver.

When to use/ guidelines

Experts advise that don't use patterns prematurely. In other words, use design patterns to optimize the design solution. As the design patterns are more mature area so there are certain guidelines are available for the use of design patterns. But aspect oriented software development is in evolving phase so there are no concrete guidelines when to use the aspects.

Understandability

Design patterns may increase or decrease the understandability of design or implementation. They can decrease understandability by adding indirection or increasing the amount of code, and can increase the understandability by improving modularity, better separation of concern and easing description. Aspects can improve understandability by removing the redundant code and tangled code which is unclear and difficult to understand.

Modularity

Modularity is one of the key design features along with the others like abstraction, decomposition, encapsulation, information hiding, and separation of concern.

The root of both the Aspects and design patterns is same (Object Oriented) and modularity is a one of the characteristic of object oriented, so aspects and design patters are modular by nature. Aspects take the concept of modularity one step further. AOP provided solution to the problems/shortcomings of the object oriented technique (crosscutting).

Maintainability

After the implementation of design pattern programmer can not trace what patterns were applied and what design decisions were made. Thus, when changes have to be made the entire design has to be almost entirely reconstructed.

Tangled code is difficult to change because one has to find all the code involved and to be sure to change it consistently.

Reuse

According to the classical definitions of the design patterns design patterns facilitates the reuse of architecture and design of the software. Aspects support the reuse at the lower level of abstraction i.e. the components and the aspects because the aspects and components are implemented separately and have loose coupling between them.

Design patterns have poor maintainability because they vanish in code and hard to identify again.

3.5 Out put

- Aspect design pattern, which will help to model the aspects at design level.

3.6 Research Method

Our research methodology is literature survey based analysis of the existing aspects and design patterns, to find relationship between aspects and design patterns and the inclusion of aspects at design level.

3.7 Validation of pattern

- We will use the criteria present in literature to evaluate/validate the design patterns.
- And case study to validate the results.

Doug Lea in his paper [13] has enlisted some properties that a good pattern should exhibit which are as following:

- Encapsulation
- Abstraction
- Openness
- Variability
- Generativity
- Compose ability

While defining these properties he mentioned that a pattern should encapsulate a well defined problem and provide a solution at the abstract level, we compared the proposed aspect design pattern with this definition and we found that it is encapsulating a well define problem and providing the solution at an abstract level. The proposed pattern is quite simple and it can be composed with other patterns and it has generativity in the sense that it can be used by all software development participants. Aspect design pattern has the property of openness because it can be used for other non functional requirements. It has variability in itself as it is independent in terms of implementation the only limitation is the use of an aspect oriented programming language.

3.8 Scope

Our scope limits to the design level representation of aspect pattern for non functional requirements, and for this purpose we will consider Logging (aspect) and the discovery of pattern particularly for non functional requirements.

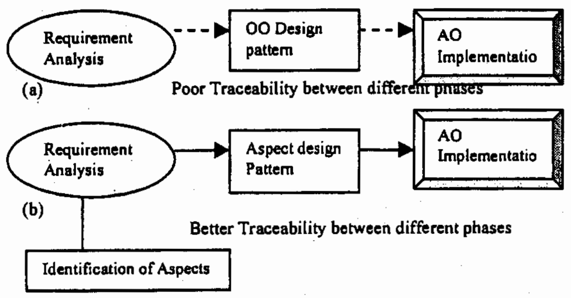


Figure: 3.1 Abstract diagram of proposed solution

Description of Figure 3.1

This figure depicts that there is a gap between analysis level and implementation level. This gap is due to the absence of AOP constructs in traditional design pattern description. The focus of this work is to bridge this gap by describing design pattern description with inclusion of AOP constructs. As represented by the part (b) of the figure.

Chapter-4
Proposed Solution

4.1 Identification of Recurring Aspect

There are concerns that crosscut the core functionality of the system and become the cause of code tangling and code scattering. Code tangling is the situation when more than one concern are handled by one module and code scattering is the case when one concern is implemented in one module.

Logging is a concern which crosscut other modules of the system and become a cause of code tangling and code scattering. Basically this concern has a recitative nature. It repeats in an application that is why it is treated as Aspect

We have observed that this aspect also repeat in a number of applications, so we consider it as a recurring aspect.

Because it recurs in a number of applications so it may have a pattern.

As peter code mentioned in his paper [9] raised a question that "how to find pattern" and answered it as "look more closely what is repeating there".

Keeping in mind this view of peter cod we observed and identified that logging is repeating in number of applications.

We also observed its general behavior that in most of the cases there is a need to record information at entry or exit point.

4.2 Identification of Pattern

We have found by considering various systems that logging required either on entry or exit. It is common that systems record information of event, when it enters or exit. Therefore we derived a pattern of logging that it will occur on entry and exit because we determined that this is a general behavior of logging entry/exit may be at any level system level or on lowest point function level.

4.3 Proposed Pattern

In [9] Peter Coad raised the question that how to find the pattern and referred [17] to answer which says, look more carefully that what is repeating there [17].

Christopher Alexander who is considered the pioneer in field of design patterns describes the patterns as "each pattern is a three part rule, which expresses a relation between a certain context, a problem and a solution".

Mainly four forms of patterns are available in literature the Alexander form (Alexander et al. 1977) which contain the elements *pattern name, problem statement, context, forces, solution, example, force resolution, design rationale*, GoF Patterns (Gamma et al. 1994) which have the elements *intent, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses, and related patterns*, Coplien form (Coplien et al 1995) with the key elements of *problem, context, forces, and solution*, POSA form (POSA Book 1996) which is similar to GoF form but have different names of elements as *summary, example, context, problem, solution, structure, dynamics, implementation, example resolved, variants, known uses, see also and consequences* [13]. All these forms of design pattern contain the basic categories name, problem statement, context, description of forces and solution.

Our studies lead us to the solution and the solution emerged in form following elements. Our proposed pattern has the following elements:

- **Pattern Name**
- **Problem Statement**
- **Forces**
- **Context**
- **Participants**
- **Structure**
- **Implementation**
- **Join Points**

4.4 Discussion on proposed pattern

As described in the previous chapter of problem definition that we have found from literature that there are issues like tractability, consistency and a gap between artifacts of the software engineering process. The reason behind is that early stages of software development life cycle are based on object orientation, and later stage (implementation) is accomplished by using Aspect Oriented Programming (AOP).

Our proposed pattern handles the non functional requirements at the design level up to a significant level by capturing these Aspects in design level document. The combination of identification of aspects and our proposed design pattern will help to solve the above mentioned issue as depicted by the following figure.

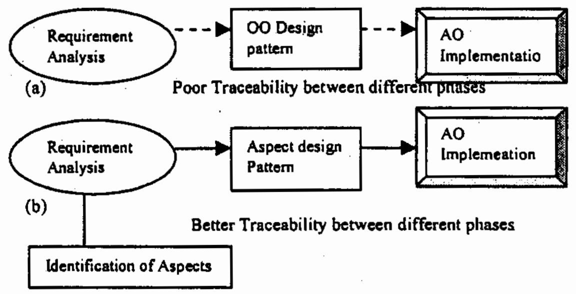


Figure 4.1 Abstract diagram of proposed solution

4.5 Description of Pattern Elements

Following is the description of the elements of pattern.

- **Pattern Name**

The first element of the pattern template is the pattern name according to the best practices in the field of design pattern. The name should be short and descriptive.

- **Problem Statement**

Problem statement describes the problem for which the pattern is going to provide the solution. In our case "How to handle recurring aspect".

- **Forces**

Force is the element that describes rationale behind the use of this pattern in case of our proposed pattern the existence of aspect i.e. any implicit or non functional requirement is the force.

- **Context**

Context contains the scenario which illustrates the situations where this pattern is applicable, as we have established in chapter-3 that the aspects recur in different applications and behave almost in same manner, so it is quite simple to illustrate such situation or context.

- **Participants**

Participants are the entities (classes) that can be affected by the presence of an aspect.

- **Structure**

The structure element of the aspect design pattern describes the abstract structure of the pattern as depicted in following figure.

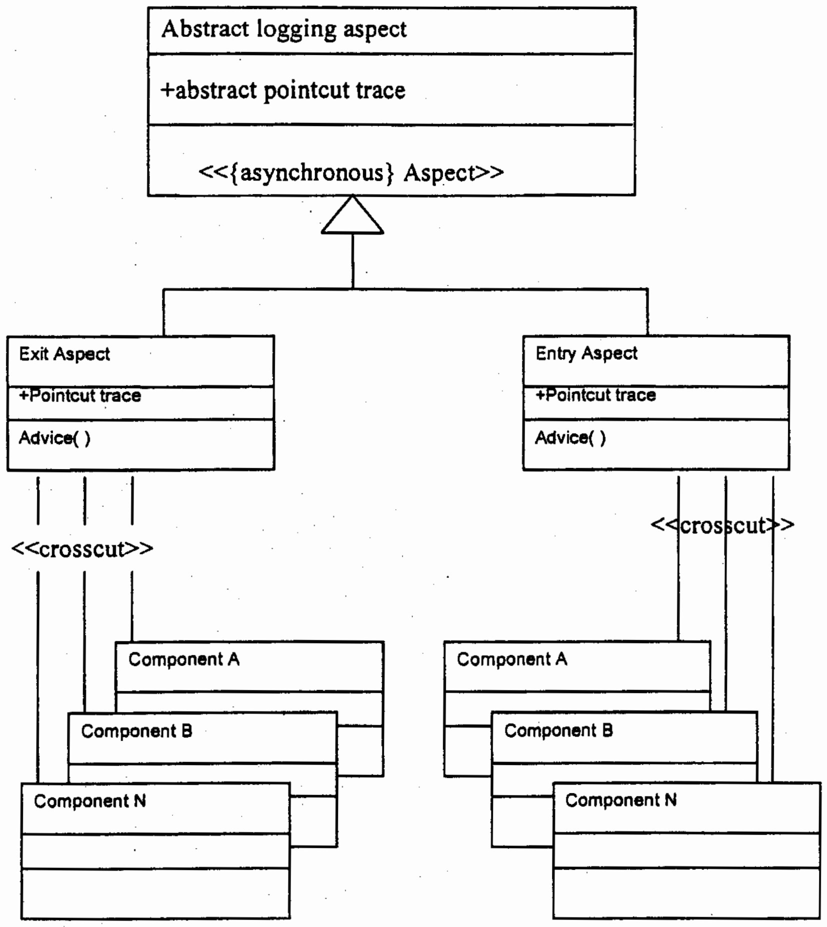


Figure 4.2 Structure diagram

The generalized behavior of logging is captured by abstract aspect logging, with an abstract pointcut trace. Entry and Exit aspects are inherited by the base aspect logging. Both inherited aspects contain separate implementation of the abstract pointcut trace. The association between aspects and components is represented by link with a stereotype <<crosscut>>.

- **Implementation**

Implementation contains the general implementation of the pattern or may have some sample code.

Sample code of pattern
Following is the sample code of pattern

```
public abstract aspect Logging {
      public abstract pointcut trace();


      before() : trace(){
            //advice
      }
      after(): trace(){
            //advice
      }}
public aspect AspectEntry extends Logging(
      public pointcut trace( ):call (desired methods);

      before(): trace(){
      //advice code

      }}

public aspect AspectExit extends Logging{
      public pointcut trace( ):execution (desired methods);

      after(): trace(){
      //advice code
      }}
```

- **Join Points**

The last element of the design pattern is join points which are well defined points in the execution of a program or a point where the requirements crosscut each other. Join point is a construct of aspect oriented languages. The inclusion of this element makes the pattern unique because it has incorporated a construct of the aspect oriented languages into the design pattern. Now the join points are visible at the design level which will result in a better tractability between the artifacts of the design and implementation level.

Chapter-5
Implementation

TH-5062

In this chapter we have implemented our proposed pattern on three case studies which are as following:

5.1 Shopping Cart

A customer selects the items she wants to purchase and adds to the cart, customer may remove the item from the cart or she can empty the cart. These are the operations a customer can perform. When customer add an item to the cart, the cart operator updates the stock and remove that item from the total and when customer remove item from the cart, cart operator adds that item to the stock and when customer empty the cart, cart operator again update the stock and add all items removed from the cart to the stock. As any of the above mentioned operation is performed, it is recorded in a log file. The information has to be recorded in a log file that what item is added, removed and update in stock on which date and time.

5.1.1 Concerns

We have identified the following concerns of customer and cart operator from the above mentioned case study

Customer is concerned with:

1- Add item to cart

2- Remove item from cart

3- Empty the cart

Cart Operator is concerned with:

1-Updating the Stock in case of add, remove, empty (this concern of the cart operator is crosscutting the concerns of customer). Whenever there is an update in stock is it has to be recorded in the log file.

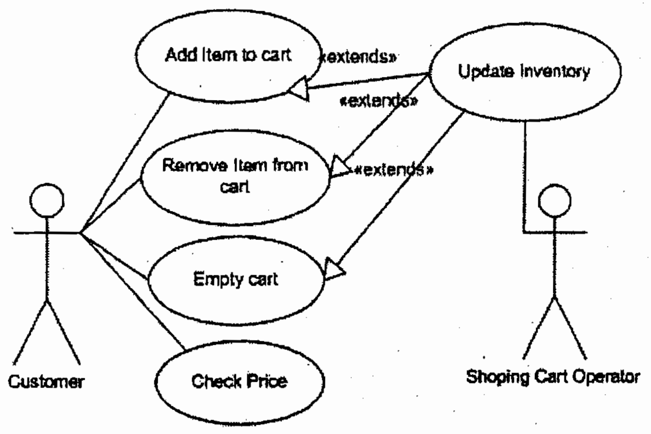


Figure 5.1 Use Case diagram of shopping cart case study

5.1.2 Basic course of action: Narrative style

- Customer wants to purchase an item
- He view the list of item
- Customer add item into cart
- Customer view the selected items
- Customer removes the item from the selected items
- Customer view the selected items
- Customer can remove all items from the cart
- Customer can check the price of an item
- Add remove or empty are updated in the stock
- In case of add, added items are deducted from the stock in case of empty and remove item, items are added into stock

Name	Add item to cart
Description	Customer add an item to cart which he want to purchase
Precondition	Customer should log in the system Item should be available in stock
Post condition	The item is added to cart and the information is recorded in a log file

Name	Remove item from cart
Description	Customer can remove item from cart
Precondition	Customer should log in the system Item should be in cart
Post condition	Item is removed from the cart and information is recorded into log file

Name	Empty cart
Description	Customer can empty the cart
Precondition	Customer should log in the system Items should be in cart
Post condition	Items removed from the cart, information of the event is logged into the log file

Name	Check price
Description	Customer can check prices of items
Precondition	Customer should be log in the system
Post condition	Price is displayed

Name	Update inventory
Description	Items purchased by the customer should be updated in stock
Precondition	Customer must have selected one or more items
Post condition	Stock is updated Information of the event is logged into log file

5.1.3 Application of Proposed Solution on Case Study

I-Name

Aspect design pattern (for Logging)

II- **Problem Statement**

"How to handle logging"

In our case study whenever there is an update in the stock it has to be written in a log file or a log of all transactions is required to be maintained. This requirement of maintaining the log is crosscutting the customer concerns.

III- **Forces**

The logging requirement is crosscutting multiple modules. So, it is necessary to capture it at design level. Logging does have an occurrence pattern that is log is maintained at the entry and exit point.

IV- **Context**

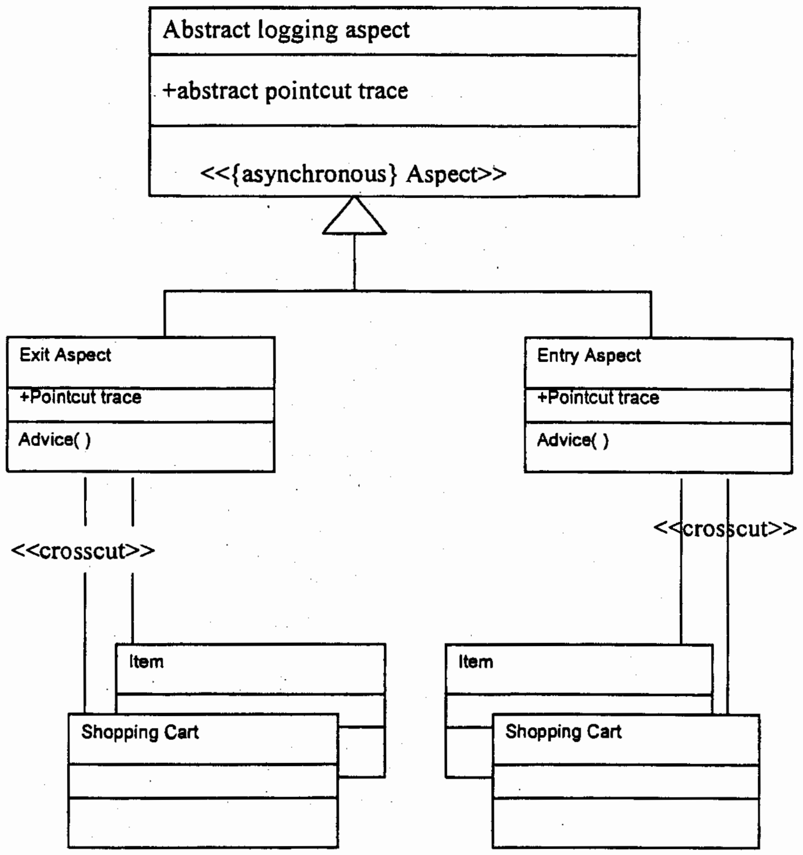
Our case study is serving the purpose of context here.

V- **Participants**

Classes involved in logging or the classes affected by logging are the participants, in this case study four classes are involved but only two are affected by the logging so, the item class and shopping cart class are the participant.

VI- **Structure** :

Following is the structure diagram for the case study.



5.2 Structure diagram for case study

VII- **Implementation**

This pattern can be implemented with any aspect oriented language. Pattern uses the construct of aspect oriented language. Following code is in AspectJ language:

```
public abstract aspect Logging {
      public abstract pointcut trace();


      before() : trace(){
            //advice
      }
      after(): trace(){
            //advice
      }}
```

```
public aspect AspectEntry extends Logging{
      public pointcut trace( ):call (desired methods of Item
Class/Shopping Cart Class);

      before(): trace(){
      //advice code

      }}

public aspect AspectExit extends Logging{
      public pointcut trace( ):execution (desired methods of Item
Class/Shopping Cart Class);

      after(): trace(){
      //advice code
      }}
```

VIII- **Join point**

The three operations of the customers i.e. add item, remove item empty and update are the join points because the update operation occurs at the same time as any of them is initiated.

5.2 Student Registration System

Student contact the student registration office & present his registration card, registration officer verifies that he is a valid student & offer him the courses. Student selects the courses. Registration officer verifies that the fee is paid and the prerequisite courses are passed & then register the student. And update the list. Whenever student select course and registration officer perform verification, register the student and update the list, information of all these events is recorded in a log file. So, logging concern is crosscutting the student concern (select course) and registration officer concern.

5.2.1 Concerns

Student is concerned with the

1- Presentation of card

2- Selection of courses

Registration officer is concerned with

1- Verification of student

2- Checking the fee & prerequisite

3- Register the student

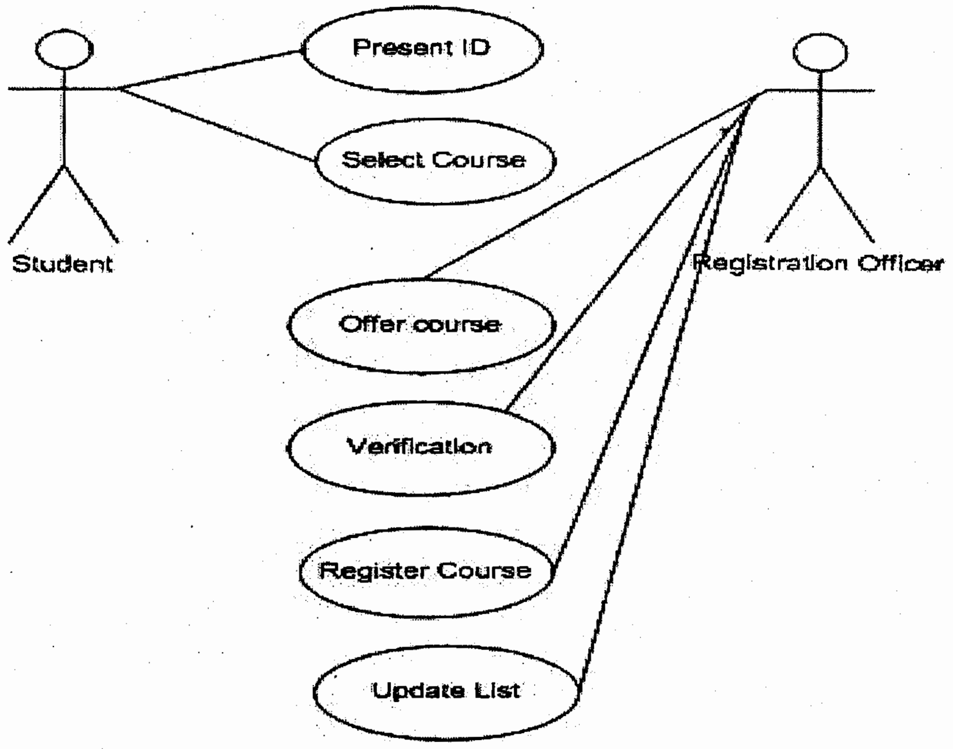


Figure 5.3 Use Case diagram of student registration system

5.2.2 Basic course of action: Narrative style

- Student contact the registration officer
- Present his university card
- Registration officer offer him courses
- Student selects the courses
- Registration officer verifies that fee is paid and prerequisites are passed
- After verification student is registered
- Registration officer update the list of registered students.

Name	Present university card
Description	Student will contact the registration officer and present his university

	card
Precondition	Student should have a valid university card
Post condition	Card is accepted as valid

Name	Select course
Description	Student will select the course from the list of offered courses
Precondition	Student should be authenticated
Post condition	Student select the desired course and information is recorded in a log file

Name	Offer course
Description	Registration officer offer list of courses to the student
Precondition	Student present his identity
Post condition	Student is provided with the list of offered courses

Name	Verification
Description	Registration officer perform the verification that fee is paid and pre requisite courses are passed.
Precondition	Student must have selection of courses
Post condition	Successfully verified and information is recorded in a log file .

Name	Register course
Description	Officer register the student for the selected courses
Precondition	Student is verified and have a selection of courses
Post condition	Student is registered and information is logged in log file.

Name	Update list
Description	Officer update the list of registered students
Precondition	A new student is registered with some courses

Post condition	List is updated and information is logged .

5.2.3 Application of proposed pattern on case study

I- Name

Aspect design pattern (for Logging)

II- Problem statement

In this case logging concern is crosscutting the student and registration officers concerns, which lead to concern scattering and tangling.

III- Forces

The requirement of logging the events is force for this pattern.

IV- Context

Our case study (student registration) serving the purpose of context here.

V-Participants

There are total three classes involved in this case study and classes affected by the aspect is Registration Officer class and student class.

VI- Structure

Following is the structure diagram for case study.

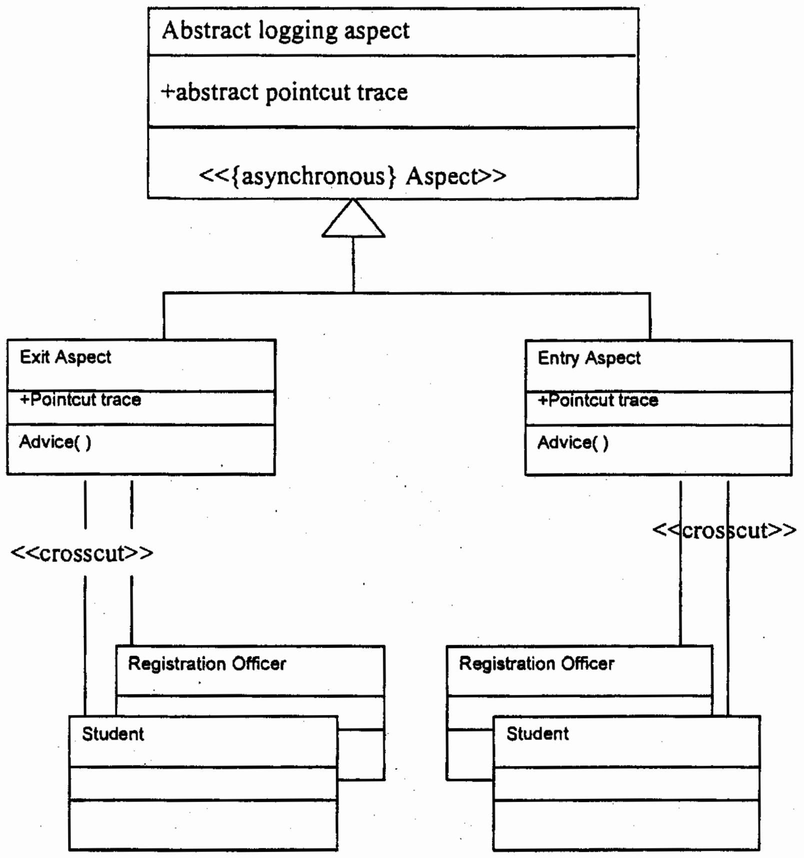


Figure 5.4 Structure diagram for case study-2

VII- Implementation

This pattern can be implemented in any aspect oriented programming language like ApectJ. Pattern uses the constructs of aspect oriented language.

```
public abstract aspect Logging {
      public abstract pointcut trace();


      before() : trace(){
            //advice
      }
      after(): trace(){
            //advice
      }}
public aspect AspectEntry extends Logging{
      public pointcut trace( ):call (desired methods of Registration
Officer/Student Class);

      before(): trace(){
```

```
      //advice code

      }}

public aspect AspectExit extends Logging{
      public pointcut trace( ):execution (desired methods of Item
Class/Shopping Cart Class);

      after(): trace(){
      //advice code
      }}
```

VIII- Join point

The logging concern is crosscutting the concerns of the student at the join point select course and registration officer at the points offer course, verification, register, and update list.

5.3 Motorway Toll System

When a car enters the motorway a card is issued to the driver with car no., time & entry point. On exit the driver show his card, toll collection officer calculate the time, distance and total toll money his toll is displayed on screen, driver pay the toll & exit.

5.3.1 Concerns

Driver is concerned with

1- Receive the card

2- Show the card at exit

3- Pay the toll

Toll Collection Officer is concerned with

1- Calculate the total time, total distance

2- Calculate the total toll money

3- Receive the toll

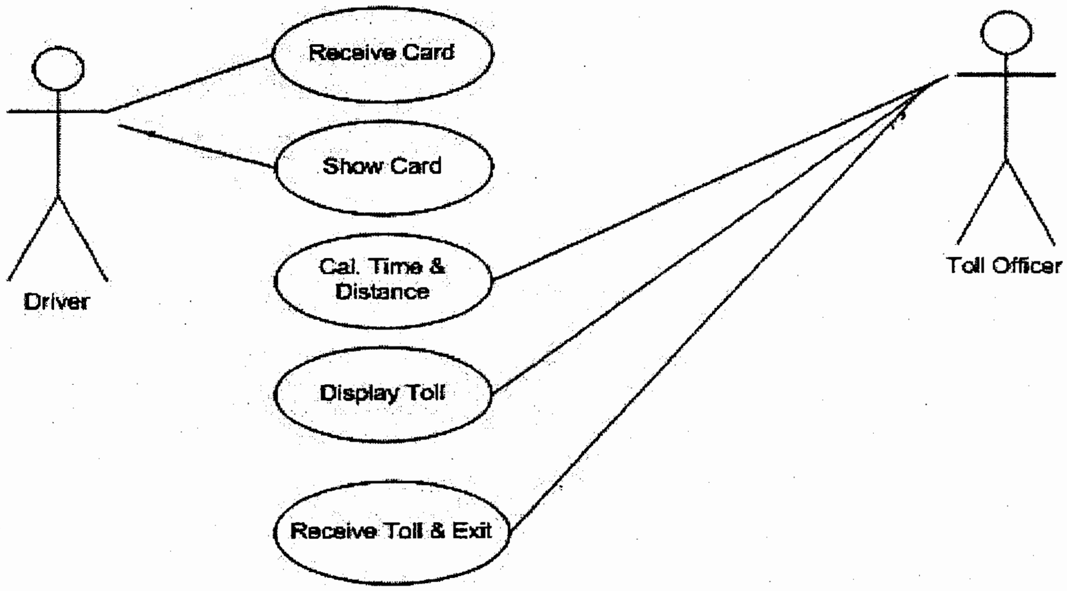


Figure 5.5 Use Case diagram of Motorway toll system

5.3.2 Basic course of Action

- Vehicles are entered to travel as motorway
- Vehicles present their visibility on entry point
- Driver receive the card to travel
- Vehicle driver show card at exit point toll collection officer calculate time and distance
- Toll collection officer receive money and exit the vehicle.

Name	Receive card
Description	The driver enters into the toll plaza and receive card from toll collector. Information of the vent is logged
Precondition	Present vehicle
Post condition	Card is issued. Information is of the event is logged

Name	Calculate time and distance
Description	Vehicle driver present the card to the toll officer. Officer performs calculation of time and distance and generates receipt. Information of the event is logged
Precondition	Card should be valid
Post condition	Receipt generated successfully, information is logged

Name	Show card
Description	The vehicle driver present card to toll collection officer
Precondition	Present vehicle
Post condition	Card accepted

Name	Receive toll and exit

Description	The toll collection officer receives the toll money. Information is logged.
Precondition	Receipt generated
Post condition	Receive money and exit car, information is logged.

5.3.3 Application of proposed pattern on Case study-3

I- Name

Aspect design pattern

II- Problem statement

In this case logging concern is crosscutting the other concerns, which lead to concern scattering and tangling.

III- Forces

The requirement of logging the events is force for this pattern.

IV- Context

Our case study (motorway toll system) is serving the purpose of context here.

V- Participants

The participants of this system are *toll officer, card* and the logging *aspect*.

VI- Structure

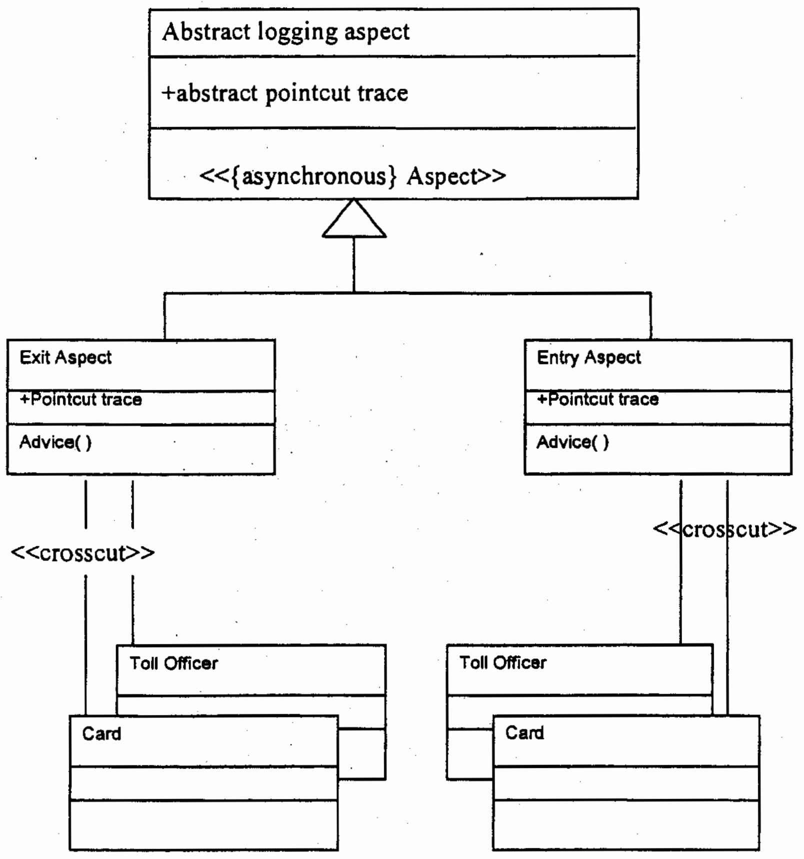


Figure 5.6 Structure of classes case study-3

VII- Implementation

This case study can be implemented by any aspect oriented programming language.

```
public abstract aspect Logging {
      public abstract pointcut trace();


      before() : trace(){
            //advice
      }
      after(): trace(){
            //advice
      }}
public aspect AspectEntry extends Logging{
      public pointcut trace( ):call (desired methods of Toll
Officer/Card Class);
```

```
      before(): trace(){
      //advice code

      }}

public aspect AspectExit extends Logging{
public pointcut trace( ):execution (desired methods of Toll
Officer/Card Class);

      after(): trace(){
      //advice code
      }}
```

VIII- Join point

All operations of driver and toll officer represent join points.

Chapter-6

Evaluation

6.1Quality of Proposed Pattern

Doug Lea in his paper [13] has enlisted some properties that a good pattern should exhibit which are as following. We analyzed our pattern regarding these parameters and found that it fulfill the criteria.

- Encapsulation
- Abstraction
- Openness
- Variability
- Generativity
- Compose ability

6.1.1 Encapsulation

Each pattern encapsulates a well-defined problem/solution. Patterns are independent, specific, and precisely formulated enough to make clear when they apply and whether they capture real problems and issues, and to ensure that each step of synthesis results in the construction of a complete, recognizable entity, where each part makes sense as an in-the-small whole.

Our pattern has a well defined problem how to handle the aspect and provide solution for it.

6.1.2 Generativity.

Each entry contains a local, self-standing process prescription describing how to construct realizations. Pattern entries are written to be usable by all development participants, not merely trained designers. Many patterns are unashamedly ``recipes", mirroring the ``unselfconscious" procedures characteristic of traditional method less construction. An expert may still use a pattern in the same way that an expert chef uses a cooking recipe -- to help create a personal vision of a particular realization, while still maintaining critical ingredients and proportions.

Our pattern exhibit the generativity as it includes the constructs of AOP, which are helpful for the coding community and a developer with less experience may use the pattern more efficiently.

6.1.3 Abstraction

Patterns represent abstractions of empirical experience and everyday knowledge. They are general within the stated context, although not necessarily universal. (Each entry in Patterns is marked with a ``universality" designation of zero to two stars.) Pattern construction is an iterative social process collecting, sharing, and amplifying distributed experience and knowledge. Also, patterns with a structural basis in or similarity with natural and traditionally constructed artifacts exploit well adapted partitioning of the world. Sometimes, patterns may be constructed more mechanically, by merging others and/or transforming them to apply to a different domain. And some patterns are so tied to universals that they emerge from introspection and intuition uncontaminated by formalism. Heuristics based on participatory design, introspection, linkage to existing artifacts, and social consensus all increase the likelihood of identifying central fixed and variable features, and play a role even when that environment is purely internal and/or artificial, but where each part helps generate a context for others.

Proposed pattern defines problem and provide solution at an abstract level and have the concept of abstraction.

6.1.4 Openness

Patterns may be extended down to arbitrarily fine levels of detail. Like fractals, patterns have no top or bottom -- at the lowest levels of any design effort, some are merely opaque and/or fluid (e.g., plaster, concrete). Patterns are used in development by finding a collection of entries addressing the desired features of the project at hand, where each of these may in turn require other sub patterns. Experimentation with possible variants and examination of the relationships among patterns that together form the whole add constraints, adjustments and situation-specific specializations and refinements. For example, while only a small set of patterns would typically apply in the design of a certain housing community, each house will itself be unique due to varying micropatterns. Because the details of pattern instantiations are encapsulated, they may vary within stated constraints. These details often do impact and further constrain those of other related patterns. But again, this variability remains within the borders of higher-level constraints.

Pattern is providing solution on a higher level of abstraction and it is independent of implementation tool and languages. So, it enhances the flexibility of solution. And the pattern may be used for other recurring aspects.

6.1.5 Composibility.

Patterns are hierarchically related. Coarse grained patterns are layered on top of, relate, and constrain fine grained ones. These relations include, but are not restricted to various whole-part relations. Most patterns are both upwardly and downwardly composible, minimizing interaction with other patterns, making clear when two related patterns must share a third, and admitting maximal variation in sub-patterns. Pattern entries are arranged conceptually as a language that expresses this layering. Because the forms of patterns and their relations to others are only loosely constrained and written entirely in natural language, the pattern language is merely analogous to a formal production system language, but has about the same properties, including infinite nondeterministic generativity.

Pattern can be composed with other patterns because it is providing the solution without other entities of the system.

The proposed pattern is quite simple and it can be composed with other patterns and it has generativity in the sense that it can be used by all software development participants. Aspect design pattern has the property of openness because it can be used for other non functional requirements. It has variability in itself as it is independent in terms of implementation the only limitation is the use of an aspect oriented programming language.

Conclusions

In this thesis we have identified that there is a gap between different stages of Aspect Oriented Software Development life cycle. We analyzed both the design patterns and aspects and found that there is one thing common that they provide solution to the recurring problems. We surveyed different forms of design patterns and proposed a representation of aspects at design level as a design pattern. We have used pattern description elements to describe the aspect. This kind of representation of aspect will help to remove the inconsistencies between different software artifacts. Aspect design pattern improves the traceability between analysis, design and code because the aspects are identified in requirement analysis phase and represented at design level, that design is translated into code using aspect oriented language. We have answered the questions which we have raised in the chapter 3 that whether an aspect can be represented as a design pattern and such representation helps at design level. We have examined our pattern under the quality criteria and found that it has the quality characteristics such as encapsulation, abstraction, genrativity, openness, variability, compose ability.

Our research focus is on two things first is the description of pattern and second is pattern. We observed that all the aspects have same kind of effect that is they modify the behavior of system on specific join points.

Our pattern is providing solution to a problem "how to handle the aspect (logging)". In software world currently the mechanism to document or to provide the solution the way used is patterns. We have used this way; we have used the elements to provide the solution. Our pattern elements describe the problem and provide the solution and bridge the gap between different artifacts of software engineering, no such element is available currently which consider the constructs of the AOP at design level in design pattern our pattern does. Description of pattern is generic we may handle other aspects using this description.

References

[1] AspectJ Programming Guide. 2003, Xerox Corporation.

[2] Gregor Kiczales et al, "Aspect-Oriented Programming", Proceedings of European Conference on Object-Oriented Programming, 1997

[3]. Harbulot B et al."An Investigation ofAspect-Oriented Programming", in Department of Computer Science. 2002, University of Manchester UK.

[4] Gregor Kiczales et al. "An Overview of AspectJ". in 15th European Conference on Object-Oriented Programming. 2001.

[5] "Java design patterns 101", David Gallardo

[6] "Patterns and Software: Essential Concepts and Terminology" by Brad Appleton http://www.bradapp.net 02/14/2000

[7] Christina et al. "A Theory of Aspects for Aspect-Oriented Software Development" Proceedings of the 7th Brazilian Symposium on Software Engineering, 2003.

[8] Jan Hannemann et al. "Design Pattern Implementation in Java and AspectJ", Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2002

[9] Peter Coad, "Object Oriented Pattern", Communication of ACM vol. 35, No. 9 1992

[10] Deepak Dahiya et al. "Approaches to Aspect Oriented Design: A study" ACM SIGSOFT Software Engineering Notes, v.31 n.5, 2006

[11] Ouafa Hachani et al. "Using Aspect-Oriented Programming for Design Pattern Implementation", Workshop of the 8th International Conference on Object-Oriented

Information Systems OOIS, 2002

[12]] Oufa Hachani et al. "On Aspect-Oriented Technology and Object-Oriented Design Pattern" ECOOP 2003 AAOS Workshop, 2003

[13] Doug Lea, "Christopher Alexander: an Introduction for Object Oriented Designers" ACM SIGSOFT Software Engineering Notes, v.19 n.1, p.39-46, 1994

[14] "Survey of Analysis and Design Approaches" Ruzanna Chitchyan et al. version 1.0, 2005

[15] Sue et al. "Aspect Oriented Software Development: Towards A Philosphical Basis", Technical Report TR-06-01, Department of Computer Science, King's College London, 2006

[16] "Software Engineering A Practitioners Approach" written by Roger S. Pressman 6th edition pg 267

[17] "The Timeless Way of Building" written by Christopher Alexander, Oxford University Press, 1979

[18] IBM Dictionary of Computing International Edition

[19] http://trese.cs.utwente.nl

[20] Ruzanna Chitchyan et al. "Survey of Analysis and Design Approaches" version 1.0, 2005

[21] Gefei Zhang, "Toward Aspect-Oriented Class Diagrams" proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)

[22]Martin Fowler,"Writing Software Patterns" available at http://www.martinfowler.com/articles/writingPatterns

[23] AspectJ Documentation Page. 2003, Xerox Corporation.

Appendix A
Publication

Aspect Design Pattern for Non Functional Requirements

FAZAL-E-AMIN[1], ANSAR SIDDIQ[2], HAFIZ FAROOQ AHMAD[3]
[1, 2]International Islamic University Islamabad, Pakistan
[3]NUST Institute of Information Technology, Chaklala Scheme III, Rawalpindi
[1]fazal_amin@hotmail.com [2]ansar_siddique@yahoo.com

Abstract
Aspect oriented technology is created to address the problems (crosscutting) that were not effectively solved through object oriented techniques, the current research has contributed to fill the gap between Aspect Oriented Programming (AOP) and other phases of software development life cycle this gap is now decreasing. In object oriented software design, patterns are tools of software engineers to solve the recurring design problems while at programming level AOP is the solution to recurring aspects. The proposed Aspect Design Pattern for non functional requirements considers the aspect at design level and provides a design level solution that handles the aspects and mainly non functional requirements.

Keywords: Software design patterns, aspect oriented design, non functional requirements

1-Introduction

Object oriented is well known and established methodology and software engineers all over the world are using it in all phases of software development life cycle from over three decades. Previously it was assumed that object oriented technology can provide solution to almost all the real world problems, which were not solved by the procedural approaches. As the systems started growing larger it was observed that some problems can not be effectively solved by the object oriented or procedural approaches [1]. And this point became the starting point for Aspect Oriented Software Development (AOSD), and AOSD addressed the limitations of the object oriented [2].

Divide and conquer is considered as a key software design principle. According to this principle a problem is divided into smaller units, these smaller problem units are then solved and combined to form a solution, these smaller units are called modules.

A module is a "separately named and addressable component" [3]. Modularity can be defined as "the extent to which a system is composed of modules" [4].

A system can have different concerns; a concern is area of interest or property of a system that must be implemented to have a successful solution of a problem [5]. Traditional software engineering is involved in the identification of concerns and these concerns are used to modularize a system [2]. These concerns are divided into two categories: Aspect and Components. If a concern can be cleanly encapsulated in a module it will be a component, and it will be an aspect if the concern crosscut and can not be cleanly implemented in a single module [1], these separate specification of aspects and components are then combined to provide the solution by the process of weaving [1][6]. The implementation of a single concern over more than one module is crosscutting [3] that create the problem of concern/code tangling and scattering. Aspect Oriented Software Engineering introduced a new mechanism to modularize a system and separating the crosscutting concerns [2] [6].

Concern scattering is a situation when one concern is implemented in more than one module and concern tangling is a situation when more than one concern is implemented in a single module [3].

In software design phase patterns have great importance because they not only provide solution to the recurring design problems but also optimize the solution as well. Design patterns can be defined as: "Design patterns focus more on reuse of recurring architectural design theme, while frameworks focus on detailed design---and implementation." (Copeland, Schmidt 1995).

"A pattern addresses a recurring design problem that arises in a specific design solution and presents a solution to it." (Bushman et al. 1996).

"Patterns identify & specify abstractions that are above the level of single class and instance, or of component" (Gamma et. al. 1995).

By observing these definitions it can be inferred that object oriented design patterns provide solution to recurring design problems in a specific context. After the identification of design pattern in a problem situation, the pattern is then implemented using object oriented language. As mentioned earlier the aspect oriented languages provide better and efficient implementation than the object oriented languages, a lot of work is done to implement object example resolved, variants, known uses, see also and consequences [16]. All these forms of design pattern contain the basic categories name, problem statement, context, description of forces and solution.

On the basis of the above we have proposed our Aspect Design Pattern for Non Functional Requirement, which mainly constitutes on Alexandrian form with some of the elements of other pattern forms.

Our proposed pattern has the following elements:

- Pattern Name
- Problem Statement
- Forces
- Context
- Participants
- Structure
- Implementation
- Join Points

4-Description of Pattern Elements

The first element of the pattern template is the pattern name according to the best practices in the field of design pattern. The name should be short and descriptive. Problem statement describes the problem for which the pattern is going to provide the solution. A force is the element that describes rationale behind the use of this pattern in case of our proposed pattern the existence of aspect i.e. any implicit or non functional requirement is the force. Context contains the scenario which illustrates the situations where this pattern is applicable, as we have established in section 2 that the aspects recur in different applications and behave almost in same manner, so it is quite simple to illustrate such situation or context. Participants are the entities (classes) that can be affected by the presence of an aspect. The structure element of the aspect design pattern describes the abstract structure of the pattern as depicted in figure 2. Implementation contains the general implementation of the pattern or may have some sample code. The last element of the design pattern is join points which are well defined points in the execution of a program or a point where the requirements crosscut each other. Join point is a construct of aspect oriented languages. The inclusion of this element makes the pattern unique because it has incorporated a construct of the aspect oriented languages into the design pattern. Now the join points are visible at the design level which will result in a better tractability between the artifacts of the design and implementation level.

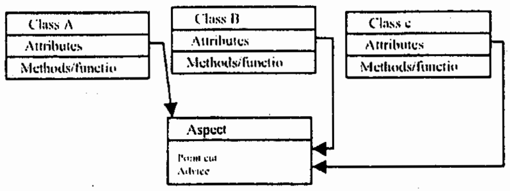


Figure 2: Abstract structure diagram of pattern

5-Case Study

We have considered the scenario of a shopping cart with a focus on logging (non functional requirement) for the application of proposed pattern. We have used minimum number of classes to represent the case study and used the approach given in [15] to represent the aspect.

5.1 Shopping Cart case study

A customer selects the items she wants to purchase and adds to the cart, customer may remove the item from the cart or she can empty the cart. The customer can add an item to the cart, the cart operator updates the stock and remove that item from the total and when customer remove item from the cart, cart operator adds that item to the stock and when customer empty the cart, cart operator again update the stock and add all items removed from the cart to the stock. As any of the above mentioned operation is performed, it is recorded in a log file.

Concerns

We have identified the following concerns of customer and cart operator from the above mentioned case study

Customer is concerned with

1- Add item to cart

2- Remove item from cart

3- Empty the cart

Cart Operator is concerned with

1-Updating the Stock in case of add, remove, empty (this concern of the cart operator is crosscutting the concerns of customer). Whenever there is an update in stock is it has to be recorded in the log file.

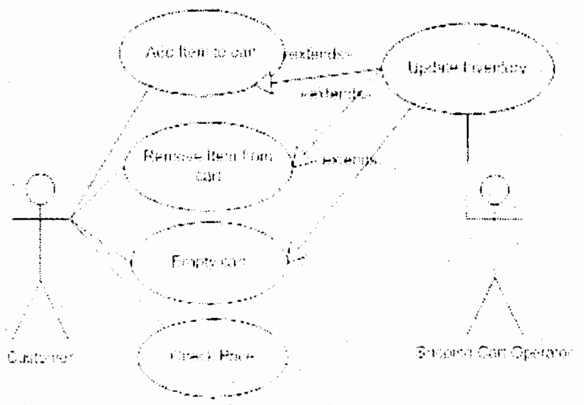


Figure 3: Use case diagram of the system

5.2 Application of Proposed Solution on Case Study

I-**Name**

Logging Pattern

II- **Problem Statement**

Whenever there is an update in the stock it has to be written in a log file or a log of all transactions is required to be maintained. This requirement of maintaining the log is crosscutting the customer concerns.

III- **Forces**

The logging requirement is crosscutting multiple modules. So, it is necessary to capture it at design level. Logging does have an occurrence pattern that is log is maintained at the entry and exit point.

IV- **Context**

Our case study is serving the purpose of context here.

V- **Participants**

Classes involved in logging or the classes affected by logging are the participants, in this case study four classes are involved but only two are affected by the logging so, the item class and shopping cart class are the particip.

VI- **Structure**

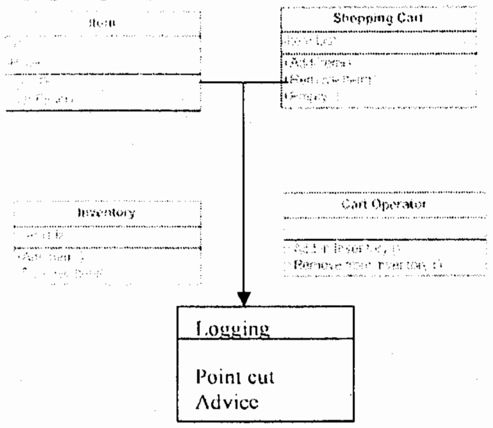


Figure 4: Class level representation

VII- **Implementation**

This pattern can be implemented with any aspect oriented language. Pattern uses the construct of aspect oriented language.

VIII- **Join point**

The three operations of the customers i.e. add item, remove item and empty are the join points because the update operation occurs at the same time as any of them is initiated.

6-Quality of Proposed Pattern

Doug Lea in his paper [13] has enlisted some properties that a good pattern should exhibit which are as following:

- Encapsulation
- Abstraction
- Openness
- Variability
- Generativity
- Compose ability

While defining these properties he mentioned that a pattern should encapsulate a well defined problem and provide a solution at the abstract level, we compared the proposed aspect design pattern with this definition and we found that it is encapsulating a well define problem and providing the solution at an abstract level. The proposed pattern is quite simple and it can be composed with other patterns and it has generativity in the sense that it can be used by all software development participants. Aspect design pattern has the property of openness because it can be used for other non functional requirements. It has variability in itself as it is independent in terms of implementation the only limitation is the use of an aspect oriented programming language.

7-Conclussion

In this paper we have proposed a representation of aspects at design level as a design pattern. We have used pattern template to describe the aspect. This kind of representation of aspect will help to remove the inconsistencies between different software artifacts. Aspect design pattern improves the traceability between analysis, design and code because the aspects identified in requirement analysis phase is represented at design level and that design is translated into code using aspect oriented language. We have answered the questions which we have raised in the section 2 that whether an aspect can be represented as a design pattern and such representation helps at design level. We have examined our pattern under the quality criteria and found that it has the quality characteristics such as encapsulation, abstraction, genrativity, openness, variability, compose ability.

8- Future Work

Our future direction is to find more patterns for other aspects, implementation of aspect design pattern on large scale software development to analyze the impacts and to refine the pattern. Such work will help to improve the quality of pattern.

References

[1] Gregor Kiczales et al, "Aspect-Oriented Programming", Proceedings of European Conference on Object-Oriented Programming, 1997

[2] Sue et al. "Aspect Oriented Software Development: Towards A Philosophical Basis", Technical Report TR-06-01, Department of Computer Science, King's College London, 2006

[3] Roger S. Pressman," Software Engineering A Practitioners Approach" 6th edition pg 267

[4] IBM Dictionary of Computing International Edition.

[5] http://tresc.cs.utwente.nl
Conference on Object-Oriented Information Systems OOIS, 2002

[9] Peter Coad, "Object Oriented Pattern", Communication of ACM vol. 35, No. 9 1992

[10] Christopher Alexander, "The Timeless Way of Building" Oxford University Press, 1979

[11] Deepak Dahiya et al. "Approaches to Aspect Oriented Design: A study" ACM SIGSOFT Software Engineering Notes, v.31 n.5, 2006

[12] Oufa Hachani et al. "On Aspect-Oriented Technology and Object-Oriented Design Pattern" ECOOP 2003 AAOS Workshop, 2003

[6] Christina et al. "A Theory of Aspects for Aspect-Oriented Software Development" Proceedings of the 7th Brazilian Symposium on Software Engineering, 2003.

[7] Jan Hannemann et al. "Design Pattern Implementation in Java and AspectJ", Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2002

[8] Ouafa Hachani et al. "Using Aspect-Oriented Programming for Design Pattern Implementation",Workshop of the 8th International

[13] Doug Lea, "Christopher Alexander: an Introduction for Object Oriented Designers" ACM SIGSOFT Software Engineering Notes, v.19 n.1, p.39-46, 1994

[14] Ruzanna Chitchyan et al. "Survey of Analysis and Design Approaches" version 1.0, 2005

[15] Gefei Zhang, "Toward Aspect-Oriented Class Diagrams" proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)

[16] Martin Fowler,"Writing Software Patterns" available at http://www.martinfowler.com/articles/writingPatterns

Appendix B
Terminologies

Terminology

Concern: "something the developer needs care about" (e.g functionality, requirement)

Problem Concern: a feature from application domain.

Solution Concern: A feature resulting from the solution technique.

Separation of Concern: handling of each concern separately

Code Scattering: One concern in many modules

Code Tangling: One module many concern

Join Point: Join Point are elements of the component language semantics that aspects coordinate with.

Static Join Point: Static join point is a location in the structure of a component.

Dynamic Join Point: Dynamic join point is a location in the execution of a component program.

Point Cut: Collection of well defined points (Joinpoint). Point cut is a predicate that matches join point.

Advice: Aspect functional implementation

Aspect: a property or concern that can not be cleanly encapsulated in generalized procedure.

Component: a property or concern that can be cleanly encapsulated in generalized procedure (i.e. object, method, procedure, API).

Aspect Weaver: Aspect weaver combines aspects and components at the join points specified by the aspect.

Crosscutting: Crosscutting is defined as a phenomenon that is observed whenever two properties being programmed must compose differently and yet be coordinated.

Crosscutting Context: Join Point may expose additional information related to context where they show up.

Crosscutting Features: are attributes and operations that describe enhancements to the structure and behavior of components.

Crosscutting Interface: Set of join points specified inside the aspect.

In-Place Modification: is destructive, that is original component code is no longer available after weaving.

Client Migration: means that both the original component and the woven version are available.

Weaving Time: Weaving can be static (compile time or load time) or dynamic (run time).

Quantification: In programs P, whenever condition C arises, perform action A over "conventionally" coded programs *P*. This implies three major dimensions of concern for the designer and implementer of an AOP system: *QUANTIFICATION, INTERFACE WEAVING.*

Obliviousness: forgetfulness