Aspect Design Pattern for Non Functional Requirements

FAZAL-E-AMIN¹, ANSAR SIDDIQ², HAFIZ FAROOQ AHMAD³
¹ ²International Islamic University Islamabad, Pakistan
³NUST Institute of Information Technology, Chaklala Scheme III, Rawalpindi

Abstract
A 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: A spect 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
oriented design patterns through aspect oriented languages. It has been observed in [7][8] that the aspect oriented implementation of GoF design patterns resulted in better code locality, improved modularity and traceability.

2- Requirement for an Aspect Design Pattern

While observing both the design patterns and AOP, we can say that the design patterns are at higher level of abstraction and the solution provided by the AOP is at the lower level of abstraction. But both the design patterns and aspects have a common characteristic that they provide solution to recurring problem at two different levels of abstraction. A doing 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 concerns are treated as aspects in an application on the other hand these aspects recur in different applications. The gap between the aspects and design Patterns can be bridged by extending aspects to the modeling level [11]. The work mentioned in [7] and [8] used 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 constructs 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, which is a step to cover the gap between the requirements and design phase. And we have raised the issues like:

- How can we represent aspect as design pattern?
- Does aspect pattern for non functional requirements help at design level?

Following is a pictorial representation of the approach, in figure (1-a) we have shown the traditional practice and in figure (1-b) we have shown that inclusion of aspect pattern can result in better traceability and consistency throughout the lifecycle, another thing mentioned in figure 2 is the identification of aspects at requirement analysis level for this purpose we are using the approach mentioned in [14]. According to which the construct or stereo type of <<extend>> in UML is an indicator of the crosscutting. Although the requirement analysis phase or identification of aspects at requirement level is not the primary focus of this paper, we have used this approach just to have a good start point to solve the case under consideration.

![Figure 1: generic diagram of software life cycle](image)

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 (logging).

3-Proposed Pattern

In [9] Peter Coad raised the question that how to find the pattern and referred [10] to answer which says, look more carefully that what is repeating there [10].

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 [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.

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.
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

```
<table>
<thead>
<tr>
<th>Item</th>
<th>Shopping Cart</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Item List</td>
</tr>
<tr>
<td>Price</td>
<td>Add Item()</td>
</tr>
<tr>
<td>list ID</td>
<td>Remove Item()</td>
</tr>
<tr>
<td>item Price</td>
<td>Empty()</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Inventory</th>
<th>Cart Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>List Item</td>
<td>Add in Inventory()</td>
</tr>
<tr>
<td>item ID</td>
<td>Remove from Inventory()</td>
</tr>
</tbody>
</table>
```

```
Logging
   Point cut
      Advice
```

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. A spect 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-Conclusion
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
Conference on Object-Oriented Information Systems OOIS, 2002