Evaluating Impact of Design Patterns on Software Maintainability and Performance

Farooq Abdullah

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Institute of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Autumn 2017

Evaluating Impact of Design Patterns on Software Maintainability and Performance

Farooq Abdullah

Evaluating Impact of Design Patterns on Software Maintainability and Performance

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Software maintainability and performance are most important non-functional quality attributes that should be taken into consideration for software development. Good and maintainable software reduce the cost of maintenance, speed up the development process and up-gradation of software becomes easy. Poor performance, on the other hand, results in serious consequences such as loss of company reputation, loss of income and project failures.

Design Patterns are the repeatable solution for commonly occurring problems. These are well tested and proven solutions. It is easy for the developers to use them in software development instead of reinventing the new design to speed up the developing process.

Our research is mainly focused on evaluating the performance and maintainability by using Design Patterns. Some web-based use cases are implemented using Gang of Four Design Patterns such as Observer, Facade, Strategy, Singleton, and Command following the Model View Controller architectural style. Implementation of these Design Patterns is evaluated by measuring the maintainability using software quality metrics such as Cyclomatic Complexity, Class Coupling, Maintainability Index, Depth in Inheritance and Lines of Code. Performance is evaluated by measuring the execution time.

It is concluded from the research that good maintainability can be achieved by implementation of Design Patterns in software development and using Design Pattern have no negative impact on performance. Further, We have concluded that the Design Patterns should be implemented in software applications with caution and probability of poor maintainability is less by use of Design patterns. There are fewer chances of performance issues in maintainable code and performance issues are easier to rectify in maintainable code than the complex code.

Acknowledgment

I would like to thank my supervisor Eric Bartley Jul for giving me the opportunity to work on his thesis topic. I would also give my sincerest gratitude for his useful suggestions both technical and related to scientific report writing. It was not possible for me to finish this thesis without his valuable tips and guidance.

I would also like to thank the University of Oslo for providing lovely campus and pleasant study environment to Informatics students. Further, I would like to thank my sister Saeeda Awan for proofreading my thesis and giving me useful suggestions to improve it.

Last but not least, I would like to thank my parents for their support and encouragement.

Blindern, November 1st, 2017
Farooq Abdullah

Contents

List of Figures

List of Tables

Part I

Introduction

1

Introduction

Software quality is one of the most important area of focus in software development. It is not necessary that the developed software that performs all its specified operations is an efficient software. There are some non-functional software quality attributes that should be considered in software development [33]. Few of the key non-functional software quality attributes include software maintainability and performance.

Maintainability means how easily we can understand, modify and retest the software. Sometimes software applications are hard to maintain and are not maintainable. It results in additional costs on software maintenance. It is essential to focus the maintainability during

development of software, so that at later stages if the modification is necessary, it can be done easily in less, cost, time and effort.

Performance is another main area of focus for efficient delivery of services. It can become a major reason of poor software quality. Poor delivery of services due to poor performance of software results in economic loss [32].

In this thesis, we perform empirical research to evaluate the maintainability and performance by implementing different Design Patterns such as Observer, Facade, Singleton, Strategy, and Command. Model-View-Controller is used as Architectural Design Pattern.

To develop the software with good practices. It is essential to focus on few attributes during the development of software. The code in the development of software should be simple, readable and easy to understand, loosely coupled, efficient, self-documented and testable.

Most of the time is spent by the developers in upgrading, modifying and improving the software. The cost, effort and time can be reduced by following good practices in software development [20].

Design Patterns are the general reusable solution of commonly occurring problem [12]. Design Patterns can be used in the development of software as these are well tested and already developed solutions, so it is easy for the developers to implement them instead of reinventing the wheel. Software development becomes easy by using these Design Patterns with better code quality. Design Patterns are known to provide more maintainable and reusable code. Design Patterns are the common language for the developers irrespective of the programming language used. The software development becomes fast and easy by using theses tested solutions to common problems, hence improve the overall software development process.

1.1 Motivation

The main motivation of our research is to evaluate software quality in terms of maintainability and performance by implementing Design Patterns. Poor quality software results in increasing effort, software maintenance cost and loss of time. We analyze the behavior of software Design Patterns to see their impact on software maintainability and performance by developing some use cases. All use cases are web-based scenarios and are developed in C# programming language following ASP.net Model-View-Controller and SQL server Database. Design Patterns such as Observer, Facade, Command, Singleton, and Strategy are used to analyze maintainability and performance quantitatively. We also discuss, how efficient code quality can be achieved with the Design Patterns. There is a need for more empirical research based on the case studies to identify the impact of software maintenance as limited evidence-based research is available in this area [39].

1.2 Goal

The Design Patterns are known to be providing good code quality in the development of software. These are well-tested solutions to common problems and reduce the risk of using the new and untested solutions.

The Design Patterns also reduce development timescales once the familiarity of the development team increase by using Design Patterns.

The research is still needed to analyze the impact of Design Patterns on software maintainability and performance. There is little research emphasize on more empirical research [39] and few papers also indicate that Design Patterns are not always good choice to use in

software applications [22, 36].

The goal is to contribute to the research to quantitatively analyze the impact of Design Patterns on software maintainability and performance that are known to be key indicators of software quality.

1.3 Approach

The approach is to evaluate Design Patterns quantitatively by doing empirical research to verify their impact on software maintainability and performance.

Maintainability is measured by using different software quality metrics such as Cyclomatic Complexity [21], Depth of Inheritance [7], Lines of Code, Maintainability Index [23] and Class Coupling [34]. We measure performance by calculating execution times on different implementation of Design Patterns.

1.4 Work Done

We develop some use cases to implement few Design Patterns from Gang of Four [12] to analyze the behavior of Design Patterns to evaluate software maintainability and performance. The implemented Design Pattern are Observer, Strategy, Singleton, Command, and Facade to study performance and maintainability. The Architectural Style followed is Model-View-Controller to develop use cases.

1.5 Evaluation and Results

For maintainability, we focus on software quality metrics such as Cyclomatic complexity, Class Coupling, Maintainability Index, Lines of Code and Depth of Inheritance. For performance, we measure the execution time for each of the Design Pattern use case implementation.

We have observed that results generated by the use cases are positive by analysis the software quality metrics and Design Patterns are helpful in making the code quality better. We also measure the execution times for each implementation of Design Pattern and notice that the there is no significant effect on performance due to the implementation of Design Patterns. We also suggest that poorly maintainable code can also be written by following the Design Patterns, So Design Patterns should be used with caution for software development to avoid maintainability and performance issues. However, the probability is less as compared to developing applications without following the Design Patterns.

1.6 Chapters Outline

Part I: Introduction

- **Chapter 1: Introduction**

  This chapter provides brief introduction to the thesis.

Part II: Background and Related Work

- **Chapter 2: Research Area and Literature Review**

  This chapter introduces the problem statement and research questions. A brief literature review on software quality, the importance of maintainability and efficiency in software development

and a need for evidence-based research to evaluate Design Patterns.

- **Chapter 3: Design Patterns**

  A brief introduction of Gang of Four Object-Oriented Design Patterns and Model-View-Controller Architectural Style.

- **Chapter 4: Software Static Analysis, Code Maintainability and Efficiency**

  A brief introduction to static analysis and importance of static analysis in measuring maintainability. Introduction to different maintainability metrics such as Cyclomatic Complexity, Class Coupling, Maintainability Index, Depth in Inheritance and Lines of code. Importance of performance as a key quality attribute.

Part III: Research Work

- **Chapter 5: Implemented Design Patterns Use Cases**

  Introduce application architecture and discussion on use cases implemented with Design Patterns such as Observer, Singleton, Facade, Command, and Strategy.

- **Chapter 6: Code Maintainability Metrics Measurements**

  This chapter discusses the maintainability metrics measurements obtained by the static analysis of use cases implemented with different Design Patterns.

- **Chapter 7: Performance Measurements**

  This chapter discusses the performance measurements obtained by calculation of execution times of different Design Patterns implemented in use cases.

- **Chapter 8: Conclusion**

  This chapter discusses the conclusion based on the results of software maintainability and performance measurements of

Design Patterns.

1.7 Summary

In this chapter, we gave a brief overview of this thesis. The motivation is to proceed evidence-based research to verify Design Patterns impact on maintainability and performance. The research methods are based on the development of use cases with Design Patterns such as Singleton, Observer, Facade, Command, and Strategy. Evaluation is based on measuring maintainability using software quality metrics such as Cyclomatic Complexity, Maintainability Index, Coupling between Objects, Lines of Code and Depth in Inheritance. Execution time is measured to verify the performance of Design Patterns. In the next chapters, we discuss literature review and implementations briefly.

Part II

Background and Related Work

2

Research Area and Literature Review

This chapter investigates the problem statement stating the main problem that is identifying the impact of Design Patterns on software maintainability and performance. Research questions are formalized to focus and investigate problems that need to be addressed in this specific area. The brief literature review is presented to identify the gap and need of further research in this area.

2.1 Problem Statement

Growing interest is developed among the software developers after the introduction of Design Patterns by Eric Gamma [12]. The reason is they provide the well-tested solution for commonly occurring problems, and they require less time and effort than designing new solution. However, research is needed to analyze them in terms of maintainability, and performance.

Software quality is most important element to consider in development of software. Good quality of software reduces the cost of maintenance, speed up the development and upgrading the software is easy.

Our research contribution consists of analyzing Design Patterns maintainability and performance. Maintainability is measured by software quality metrics such as Cyclomatic Complexity, Class Coupling, Maintainability Index, Lines of Code and Depth of Inheritance. Performance is measured by calculating the execution times. The implemented Design Patterns on different use cases are Observer, Facade, Singleton, Command, and Strategy Design Patterns and Moded-View-Controller is used as Architectural Style.

2.2 Research Questions

Based on problem statement, we have the following research questions to contribute to research.

- Can we achieve good software maintainability by implementing Design Patterns in developing software applications?
- What is the effect of using Design Patterns on the performance of the application?

- Should we use Design Patterns in development of software applications?
- Should we give priority to software maintainability over performance?

Our research is based on prototyping the use cases with Design Patterns and measure the software maintainability quantitatively. We also measure the execution time for each of the Design Pattern implemented in use cases. Our results are based on measurements from software maintainability metrics and executions time measurements for each implemented Design Pattern use case.

2.3 Related Work

Software quality is most important area to focus to make efficient, good quality software and reduce the cost of software maintenance, time and effort. The definition of software quality is different for different entities.

- For developers, the software quality is related to the quality attributes like Maintainability, Reliability, Efficiency, Security, and Size of Software.
- For users, it is how well the software works. They don't know the insights of the software. They are only concerned with the functionality.

Software quality is based on functional software quality and software structural quality. Functional quality is based on the functional requirements of the software, and structural quality is based on non-functional requirements such as maintainability and performance.

Software quality standards have been developed by ISO/IEC 9126 [33]. These standards direct us toward the measurement of software quality. The quality model proposed by ISO/IEC 9126 classify the software quality as a set of six characteristics. Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability. The ISO/IEC 9126 model is based on McCall's Model [5] with some differences.

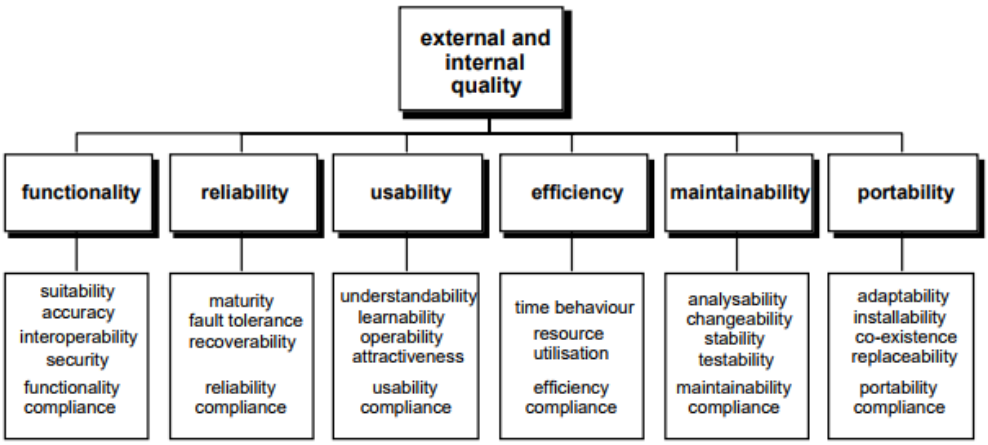

Figure 2.1: Quality model for external and internal quality [33]

Our focus in this thesis is the implementation of Design Patterns and assessment of maintainability and performance. We discuss these two quality attributes briefly.

Maintainability is defined as a set of attributes that bear an effort needed to make specified modifications (which may include corrections, improvements or adaptations of software to environmental changes and changes in the requirements and functional specifications) [33].

A lot of research is done on measuring the software maintainability quantitatively, and different software metrics are developed. Theses software metrics make Static Analysis on the code to identify maintainability of software. By Static Analysis, we mean analyzing the code without executing the software.

- Chidamber and Kemerer developed six software metrics for

Object-Oriented Design to measure the maintainability [6]. These metrics include Depth in the Inheritance tree, Number of Children, Coupling between Object classes, Response for the classes, Lack of Cohesion in Methods and Weighted Methods per class.

- Park suggested a research study on developing a framework to measure the size of software [24]. The software size is based on physical Lines of Code and Logical Lines of Code. Size is considered as a valuable metric in one research study with the conclusion that all other metrics are inconsistent apart from size and the inverse of cohesion. The study is based on analyzing four systems developed by different companies and implementing the same functionality [31].

- Cyclomatic Complexity is a measure of the complexity of the program and is suggested by Thomas J McCabe [21]. The lower value of complexity means the program is easier to maintain and vice versa. It is quantitatively measured from the program source code.

- Halstead Volume is suggested by Halstead in 1977 and is a software metric to measure the complexity of the code and is based on operator and operands [14].

- Maintainability Index is a software metric to identify maintainability of the software quantitatively [23]. The formula for Maintainability Index is based on different metrics such as Lines of Code, Cyclomatic Complexity, and Halstead volume.

Design Patterns are first introduced by Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides in Element of Reusable Object-Oriented Software [12]. Design Patterns make it easier to reuse Object-

Oriented Design in development of software. They are well tested and proven solution for commonly occurring problems in software development. Design Patterns increase the re-usability and hence improve the maintenance of software. Design pattern provides neutral language to the developers for commonly occurring problems in Object Oriented Design and can be followed in any programming language.

There has been a lot of research on evaluating the Design Patterns; still, there is a need for evidence-based research in this area.

- Cheng Zhang conducted a systematic literature review based mapping study on the effectiveness of software Design Patterns and conclude that Design Patterns are subjected to little empirical evaluation, and mostly much of it is studied indirectly [39].

- Peter Hegedus, Denes Ban, Rudolf Ferenc and Tibor Gyimothy conducted an empirical analysis to explore the connection between Design Patterns and software maintainability by using 300 revisions of JHotDraw7 and concluded that code quality is improved due to the introduction of Design Patterns. However, the study is not clear, and no Design Patterns are mentioned that are used in this study [15].

- Peter Wendorff conducted a study in the large commercial project and concluded that in some cases the implementation of Design Patterns led to negative results and inappropriately used Patterns are the aspect of re-engineering and maintenance in the future [36].

- FM Alghamdi, MR Jameel Qureshi proposed a tool to evaluate the effect of Design Patterns on software maintainability. However, the tool is survey-based, and deeper research is needed to control the effect of Design Patterns on maintainability [2].

- F Khomh, YG Gueheneuce evaluated the Design Patterns based on the empirical study using the questionnaire and concluded that Design Patterns do not always improve the quality of the system [17].
- Williams and Smith study evaluate that minor performance benefits can be achieved by choosing appropriate Design Pattern and improving design do not create any performance issues [37].

Most research is based on surveys [2], questionnaire [17] and there is a lack of empirical evidence [39] to quantitatively measure software quality attributes to evaluate Design Patterns. There are some studies, which have shown that software Design Patterns improve software quality [15] and other shows that it is not always good to use Design Patterns [17],[36].

Hence, we conducted a study that is based on empirical research to evaluate the Design Patterns. The study is based on evaluating the Design Pattern to identify their impact on non-functional quality attributes such as maintainability and performance.

2.4 Summary

In this chapter, we have presented a brief literature review, which indicates that lack of empirical, evidence-based research to quantitatively measure software quality attributed to evaluate Design Patterns. Our problem statement is stating that we are conducting quantitative analysis to evaluate software maintainability and performance on various use cases implemented with Design Patterns. Research questions identify the main problems that should be addressed in this research.

3

Design Patterns

This chapter gives an introduction to Design Patterns. We also present a general UML representations of Design Patterns used in this thesis to investigate maintainability and performance. These Design Patterns include Observer, Command, Facade, Singleton, and Strategy. Further, we also discuss Model-View-Controller Architectural Pattern.

3.1 Introduction

Design Patterns are solutions for reoccurring problems in software design. These are not complete designs that can be transmitted directly

in the code. These are blueprints to solve the reoccurring problem and can be used in many situations.

Design Patterns are first proposed as Architecture Patterns by Christopher Alexander in 1979 [1]. The popularity is gained by Design Patterns after the book Elements of Reusable Object-Oriented Software [12] published in 1994. The use of Design Patterns is known to improve maintainability, re-usability and improve the software quality and accelerates the development process. We discuss different software Design Patterns in this chapter such as Architectural Pattern Model-View-Controller, Gang of Four patterns such as Observer, Facade, Command, Singleton, and Strategy.

In this thesis, we mainly discuss the patterns that are used in the development of our use cases. Gang of Four patterns [12] present 23 Design Patterns. We use few patterns from Gang of Four such as Strategy, Command, Singleton, Observer, and Facade in our use case scenarios. We also discuss the architecture pattern that is used to describe the structure of the system such as Model-View-Controller [28] in detail.

3.2 Architectural Pattern

Architecture Patterns are used to formulate the structure of the system or part of the system. They are useful for structuring the architecture and hence improving maintainability of the software. One of the examples of Architecture Pattern is Model-View-Controller.

3.2.1 Model-View-Controller

Model-View-Controller was first proposed by Smalltalk-80 programmers. It was first introduced by Trygve Reenskaug in Smalltalk-76 during his visit to Xerox Palo Alto Research Center during 1970's [28]. Later it was implemented in Smalltalk-80 as a library, and its general concept is published in the article in 1988 [19].

Model-View-Controller is Architectural Pattern that separate the application into three components.

- Model
- View
- Controller

Each component deals with the specific aspects of application functionality. Model-View-Controller is one of the most frequently used web development frameworks. It is the decoupled architecture and is useful maintainability, scalability, and extensibility.

3.2.1.1 Models

All the data related logic is maintained by the Model component. There Model component interacts with the data and data is transferred to the Views and the Controller components. It does not depend upon the view and controller components.

3.2.1.2 Views

View represents the user interface of the application. View includes all the user interface components such as text boxes and drop downs

3.2.1.3 Controller

Controller represents a way the system interacts with the user input. Actions between the Model and View are controlled by the Controller component. The main logic of the interaction between Model and the View is based on the logic in Controller component.

3.3 Gang of Four Design Patterns

Design Patterns are introduced in book Design Patterns - Elements of Reusable Object-Oriented Software [12]. It has initiated the concept of introducing Design Patterns in software development. The authors are commonly known as Gang of Four. In Design Patterns of Reusable Object-Oriented Software [12], 23 Design Patterns are presented. These Design Patterns are classified into three categories.

- Creational
- Structural
- Behavioral

3.3.1 Creational Design Patterns

These Design Patterns provides a way for the creation of the objects. These Design Patterns deals with the creation of the objects suitable for the situation. They are used to solve the problem by creation of the objects depending on the situation.

3.3.2 Behavioral Design Pattern

These patterns describe, how the classes and objects interact with and each other and distribute responsibility. The communication between two objects is such a way that they can easily communicate with each other and should be loosely coupled.

3.3.3 Structural Design Pattern

These are used to make larger structures by providing the ways of combining objects and the classes. Structural Design Patterns are used for the composition of classes and the objects. These Design Patterns make the structure simple by identifying the relationship. The focus is on how classes are inherited from the other classes and how they are composed form other classes.

The following table shows the Design Patterns associated with the three categories.

	Creational	**Behavioral**	**Structural**
Class	Factory Method	Adapter	Interpreter, Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Fascade Flyweight Proxy	Chain of Responsibility command Iterator Mediator Momento Observer State Strategy Visitor

Table 3.1: The Category Based Classification of Design Patterns [12]

3.4 Design Patterns implemented on use cases

Here we discuss some Gang of Four patterns that we are implementing in different use cases to investigate the impact of Design Patterns on Software maintainability and performance.

3.4.1 Singleton Design Pattern

Singleton Design Pattern ensure a class only has one instance, and provide a global point of access to it. Singleton Design Pattern is used in Emerald programing language before it was introduced in the book Elements of Reusable Object-Oriented Software [12]. It was used as a language concept in objects creation [27].

Sometimes an object needs to be instantiated multiple times within the application. The operation is resource intensive due to the creation of same objects again and again within the application. Singleton ensures that only one instance of an object is created and provide a direction to access it. Singleton object is created only for the first time when it is requested and can be used many times in the application.

The general UML representation of Singleton Pattern is shown in figure 3.1. Singleton class has a private static variable with name instance. The class has a constructor declared as private, so it cannot be accessed outside the class. We can access it within the class with the GetInstance method.

GetInstance method is declared as a public static method. GetInstance method creates the singleton object for the first time only

within the application. Once it is created, it can be used within the application, and we don't need to create it again and again.

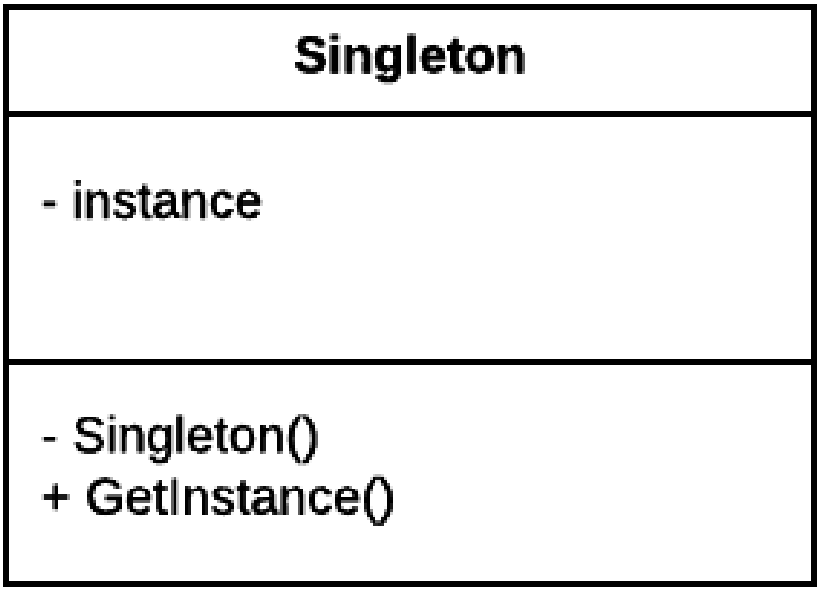

Figure 3.1: General UML Diagram for Singleton Pattern

3.4.2 Observer Design Pattern

In some scenarios, it is necessary to notify multiple objects about some event in the application. Observer Pattern can be used in such scenarios, where it is necessary to broadcast notifications about something to the different objects in the application.

Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents get notified about the change and are updated automatically [12].

The general UML diagram of Observer Pattern is shown in figure 3.2. The main objects used in Observer Pattern are Subject and

Observer. Subject has a collection of observers. When the state is changed in the Subject, all the observers are notified about the change in state. Subject state is synchronized with the Observer.

In Observer Pattern, there is an Interface with the name Observer, and it is an abstraction of the classes that receive notification in case of change is state of the Subject. It has one update method that is called automatically in case of a change in state. ConcreteObserver class is the concrete implementation of the Observer Interface. They can be more than one concrete classes to send notifications to multiple objects. All of them must implement Observer interface.

Subject Interface is an abstraction of the object that is responsible for sending the notification to the observers in case of a change in state. The ConcreteSubject class is a concrete implementation of the Subject Interface. It has three methods attach, detach and notify. Attach and detach methods are to subscribe or unsubscribe observer from the collection. Notify method is to notify the collection of observers about the change in state. There is association between Subject Interface and Observer Interface and also between ConcreteObject and ConcreteSubject

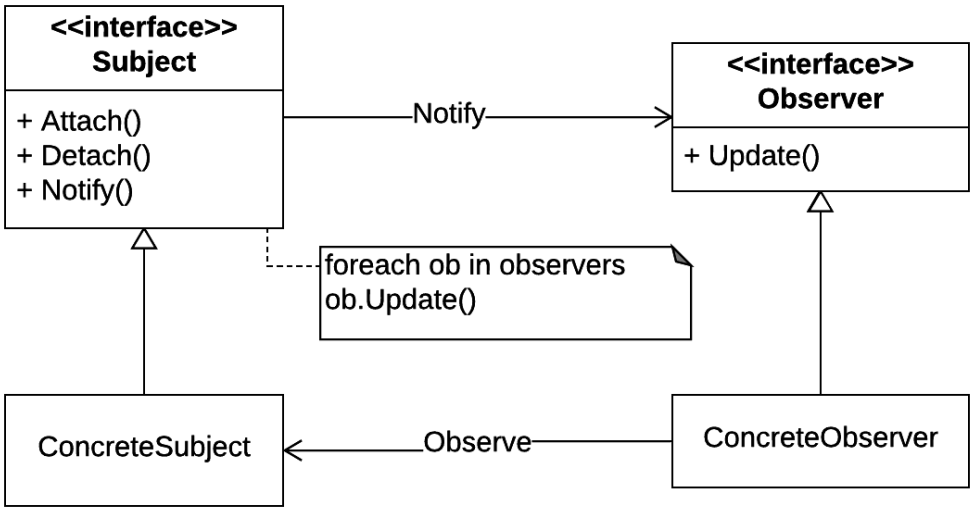

Figure 3.2: General UML Diagram for Observer Pattern

3.4.3 Command Design Pattern

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations [12].

Using Command Pattern request can be executed without knowing the operation to be requested or the receiver. Different requests can be queued, and Command Pattern provides interface to encapsulate a request as an object. Command pattern provides command objects that execute the requests and client can only place the request hence encapsulate the request as an object, so different requests can be queued. A request is passed from client to the invoker object; invoker looks for the object that is responsible for handling the command and command is passed to object responsible for executing the command.

The general UML implementation of Command Design Pattern is shown in figure 3.3. It provides decoupling between the object that invokes the request/command and the object that executes the operations.

- Client creates the cocretecommand objects and sets its receiver. The client executes the command.
- Invoker passes the request to the Icommand to execute the commands.
- Icommand is the interface implemented by the concrete command objects.
- ConcreteCommand has the association with the receiver and implements Icommand to complete the action.
- Receiver executes the operations of command to complete the request.

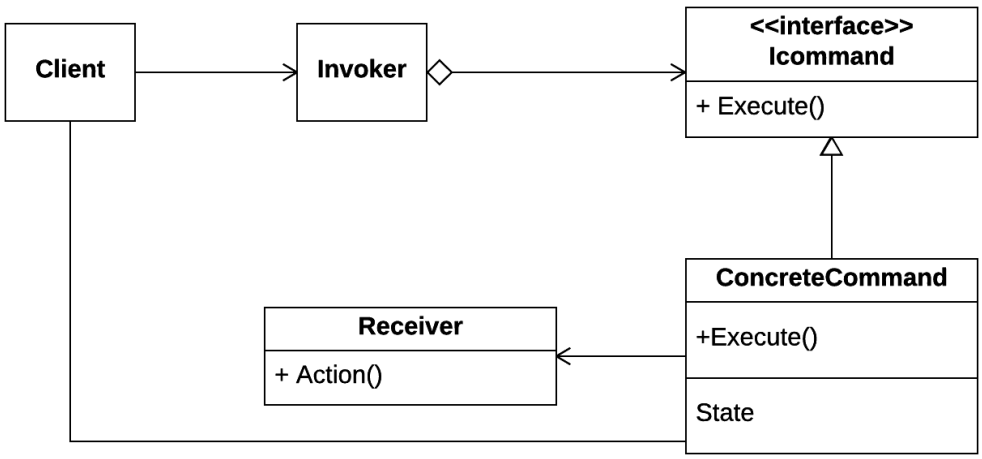

Figure 3.3: General UML Diagram for Command Pattern

3.4.4 Strategy Design Pattern

Strategy Pattern defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [12].

In Strategy Pattern, we declare objects with various strategies

that are interchangeable at runtime. The context object is used by the strategy to execute the required algorithm.

In some scenarios, there can be different strategies that need to be implemented on runtime. These strategies or algorithms need to accomplish the certain task. One way is that we can put these algorithms or strategies in one single class and anyone of the strategies can be utilized based on some condition. However, if we declare these strategies in different classes, it will make the code more maintainable as we can add more strategies later if the requirements are changed with the time. It provides decoupled architecture making the code more maintainable. Client can then initiate one single strategy at run time to complete the task. So different strategy algorithms encapsulate in different classes can be used interchangeably, and addition of more algorithms become extremely easy at later stages without affecting existing code of the software.

The general UML implementation of Strategy Pattern is shown in figure 3.4. IStretegy interface is the abstraction of different strategies implemented in different classes. It has a method ExecuteStretegy that is initiated by the context class based on strategy client want to execute. IStretegy interface is implemented by three classes, ConcreteStrategy1, ConcreteStrategy2 and ConcreteStrategy3. All these three classes encapsulate different strategies. In this way, a family of algorithms with different strategies can be created, and context class invokes one of the strategies based on the client request. Client uses context class to invoke strategy. Hence Strategy Pattern minimizes coupling and increases cohesion making the code more maintainable.

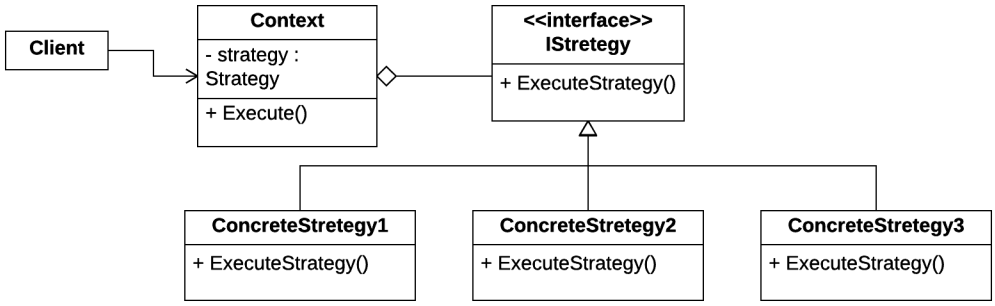

Figure 3.4: General UML Diagram for Strategy Pattern

3.4.5 Facade Design Pattern

Facade Pattern provides uniform interface to a set of interfaces in a subsystem. It is higher level interface makes the subsystem in Facade easier to use. Facade class wraps the underlying complicated subsystems with a one simple and straightforward interface [12]. Encapsulating the subsystem with a single Facade class reduce the overall complexity by decoupling the subsystems from client as client can only interact with the Facade. Without Facade, the interactions are required for the client at many places to communicate with the subsystem. Facade is very useful to remove these complexities by isolating client from the subsystem. When the client needs to interact with the subsystem, Facade takes care of interactions with the subsystem. Facade knows the subsystem and sends the request of client to the appropriate subsystem. The subsystem is implementation of the actual work and Facade is the interface between client and these subsystems to complete the task. Hence, overall benefits we can obtain by using Facade are

- It provides isolation between clients and the subsystem thus reduce the complexity by decoupling the client from the subsystems
- A loose coupling can be achieved by introduction of facade class between subsystems and the client.

A general implementation of Facade is shown in figure 3.5. Client can interact with three subsystems using Facade. The subsystems have operations to provide certain functionality to the application and client can interact these subsystems using a single Facade class.

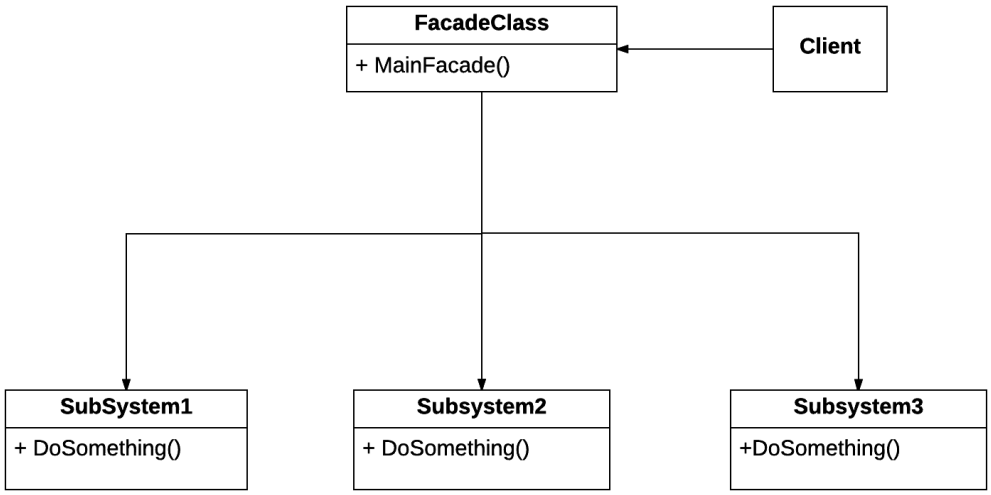

Figure 3.5: General UML Diagram for Facade Pattern

3.5 Summary

In this chapter, we have discussed Design Patterns, categories of Design Patterns such as Creational, Behavioral and Structural. We have also discussed the Model-View-Controller Architectural Pattern and brief implementation of Design Patterns used in this thesis such as Observer, Facade, Singleton, Command and Strategy. In later chapters, we discuss the use case implementation of these Design Patterns and evaluate them to analyze their impact on maintainability and performance.

4

Software Static Analysis and Code Maintainability and Efficiency

This chapter introduces the concepts of static analysis and some non-functional software quality attributes such as performance and maintainability. Various quality metrics for measuring software maintainability are discussed briefly. These software maintainability metrics include Class Coupling, Maintainability Index, Cyclomatic Complexity, Depth in Inheritance and Lines of Code.

4.1 Overview of Static Analysis

Static analysis is used to examine application code without program execution. It is used to identify many defects in the code at early stages software development. Many static analysis tools are available which helps in discovering more defects in the code and make less false warnings. Hence, static analysis approach is used to analyze the software to improve code quality and correct errors before actual executions of the software.

Static analysis on the source code is performed using the static analysis tools together with the human analysis for the code inspection. Software metrics can be obtained using these techniques to make it more maintainable, reusable and testable. This is very good approach for detecting the defects at early stages of the development process, and it proved to be more reliable and efficient in developing error-free and maintainable software.

Manual review of the code is a time-consuming process and to analyze code manually, programmers must know, what kind of errors they are supposed to diagnose in the code during software development. So using the static analysis tools together with the human inspection is very efficient way to analyze the code and perform improvement on the code based on the results generated from the static analysis.

Code review and analysis should be done at any stage of the software development process, but it is better to do it at early stages. It becomes tidy and costly to remove defects in the code at later stages of the software development lifecycle.

Software design can have many mistakes, some of them include

- Dead Code
- Poorly organized and unstructured bad code.
- Poorly Maintainable code

All of them results in poor quality software that creates problems in later stages of the software development lifecycle. Poorly maintainable code results in increasing the cost of software maintenance.

The problems associated with software design are challenging to detect with testing. The origin of problems is in requirements and design of the software [13].

4.2 Software Quality Metrics with Static Analysis

Assessment of the metrics that measures the applications source code quality are useful in analyzing the code maintainability. Measuring the metrics over time give good trends to identify the issues in software quality. Quality metrics assessment with the time gives important measures that lead towards good software quality. Some major measures include

- Repetition in Code
- Trends in Complexity of Software Maintainability
- Complexity of program structure

Analysis of these metrics with the time notifies if our matrices exceed the certain threshold that leads toward poor software quality.

Static analysis tools give us plenty of information about the software maintainability. Different metrics calculated by static analysis

provides the guide for decision-making and execution of software maintenance.

4.3 Importance of Software Maintainability for Improving Software Quality

Software maintainability provides ease to maintain the software system or components. More maintainable software means it is more convenient to improve performance, remove bugs, faults and several other attributes. It can be easily adapted to a changed environment. More maintainable software is less costly and risky to improve performance, adding new features or changing existing functionalities.

The Institute of Electrical and Electronic Engineers define the Software maintenance as

"Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performance, or adapt to a changed environment [4]." Making the software understandable to others can never be underestimated to make the software more maintainable and improve the quality of software with time in a more efficient way.

Numerous amount of benefits can be obtained by making the software more maintainable few of them are described in details.

4.3.1 Software Upgrades

Software needs upgrades to improve the quality and adding new functionalities in software with the time. These upgrades are

implemented based on reviews from the users of software and are bases of improved performance and functionality of the software. It is easy and less costly to implement upgrades to the software having a high degree of maintainability and upgrades can be implemented based on new requirements and technology.

4.3.2 Adapt to Changing Environment

Technology and business environment is changing with time in the current world. More maintainability in software means it is easier to implement current trending technologies to make it up to date according to the current technologies. Regular maintenance of the software makes it efficient and effective according to the current scenarios

4.3.3 Financial Benefits

One of the most important benefit of the maintainable software is the estimation of the budget on software expenditure. The maintainable software is less costly to perform upgrades like performance improvement, features addition and gives the better assessment of the cost estimation to perform such tasks. Hence, maintainability reduces to amount of cost to perform upgrades. On the other hand, software, which has low degree of maintainability are extremely costly to perform upgrades and improvement as more effort is required to upgrade them according to the new technologies.

4.4 Static Analysis and Measuring Code Metrics

Static analysis is used to analyze code without executing it and is used to maintain the code quality. The code is analyzed without execution analytically. Use of static analyzing tools measures various code metrics, which are helpful to identify the quality of software with the time and give handy information about code maintainability. These metrics are helpful in answering various questions related to the code maintainability such as

- Is the application easier to maintain or its maintainability is becoming harder?
- Is the application source code is tightly coupled making maintainability of the application complex?
- What is the complexity of program structure?
- How many lines of code the application or in the methods?
- What is the Maintainability Index?

There are several other metrics, which can be used to identify the software maintainability problems by analyzing the code with static analysis tools. However, we measure the following code metrics in this thesis to analyze the maintainability of code.

- Class Coupling
- Lines of Code
- Cyclomatic complexity
- Depth in Inheritance
- Maintainability Index

We measure the trends in maintainability by analyzing these code metrics on our use cases implemented with Design Patterns. Few studies suggest that there are little evidence regarding maintainability techniques and there is still scarce in validation of metric prediction models, and there is a need of few more reliable software maintainability prediction models and techniques [29]. A brief overview of all code metrics, we measure to analyze the maintainability of the application implemented with different Object Oriented Design Patterns are discussed as below.

4.4.1 Class Coupling

The concept of coupling was given by Larry Constantine in 1960 as a part of Structured Design and later in 1979 [34, 38]. Coupling is very useful metric in measuring the maintainability of code. Classes can be loosely coupled or tightly coupled. This metric is also called Coupling between Objects.

- Loosely coupled classes are less dependent on the other classes directly thus reduce interdependencies between the components
- Tightly coupled classes are more dependent on other classes

Tightly coupled classes are harder to maintain, and it reduces the maintainability of the code as in case of tight coupling it becomes very difficult to make changes in the code making the software less maintainable. However, loose coupling is very useful in maintaining the software, and it becomes easy to maintain code in large software systems.

In small applications, it is easy to identify changes and can easily be identified but in case large applications it becomes very difficult to

identify changes and software becomes extremely hard to maintain and Class Coupling plays a significant role to manage the code. Hence, Class Coupling is very useful metric to identify the dependency of one class on another.

Class Coupling is a measure of how many classes a single class use [7]. Excessive coupling makes code reuse more difficult and effects maintainability of the code. The low value of coupling is the indicator of maintainable code hence increasing the maintainability, reusability, and testability of the application. Measuring code maintainability is very useful to identify the complex structure of the application. Higher the value of Class Coupling is the indication of complexity in maintaining the application. It is considered one of the major reason of software failure [30]. The optimal value for Class Coupling should be nine [30].

In this thesis, among other software maintainability metrics, we are taking Class Coupling to identify the maintainability of different Design Patterns that we analyze in later sections. Here we are giving an example how we measure this metric.

```
1 public class Customer
2 {
3 string name;
4 int age;
5 string address;
6 }
```

Listing 4.1: Class coupling in customer class

The class customer does not have another class in place, so coupling value, in that case, is zero.

```
1  public class shopping
2  {
3  Customer c =new Customer();
4  void CustomerDetails()
```

```
5  {
6  c.name;
7  c.age;
8  c.address;
9  }
```

Listing 4.2: Class coupling in shopping class

The class has one object of customer class in place. Hence the Class Coupling of this class is equal to one.

4.4.2 Cyclomatic Complexity

Cyclomatic Complexity is a measure of the complexity of the program and is suggested by Thomas J McCabe [21]. Lower Cyclomatic Complexity makes the application easier to understand and maintain. It is the measure of linearly independent paths within the program. It is the measure of decision logic in the program hence more decisions means higher the complexity of the program. The Cyclomatic Complexity is calculated by the formula.

$$CyclomaticComplexity = E - N + 2 * p$$

Where E is the number of edges in program flow
N is the nodes in program flow
P the Number of exist point in program flow.
Here is the example of Cyclomatic Complexity measurement of a simple program.

```
1  class Program
2  {
3
4  static void Main(string[] args)
5  {
```

```
6  int a = 9;
7  string name = "world";
8  if(a > 5)
9  {
10 if (name == "world")
11 {
12
13 Console.WriteLine("Hello World");
14
15 }
16 else
17 {
18 Console.WriteLine("hei hei");
19 }
20
21 } else
22 {
23 Console.WriteLine("Hello Universe");
24 }
25 }
26 }
```

Listing 4.3: Measurement of Cyclomatic Compleixty

The Cyclomatic Complexity of the above program is calculated as four. Note that Cyclomatic Complexity of the program is two without introducing any decisions in a program.

4.4.3 Depth of Inheritance

It is a very useful metric for Object Oriented analysis, and it measures the hierarchy and depth of the class in Object Oriented designs. It is defined as the maximum length from the node to root of the tree [7]. It is also known as Depth in Inheritance Tree. Its value depends on the ancestors, class or classes on which it is dependent. If the class car is

inherited from brakes and class brakes is inherited from the shop. Then Depth in Inheritance value of class is 3.

Depth in Inheritance is dependent on the following two main points [7].

- Deeper the depth of class in hierarchy, greater are the number of methods it inherits and harder to predict the behavior of class thus reduce interdependencies between the components
- If deeper trees are involved, the complexity increases due to more number of methods and the classes

It is clear from the above points that complexity of the class increases if we increase the depth in inheritance making its reuse easier, but it increases the complexity of the code. On the other hand, if we decrease the depth in inheritance, overall complexity decreases and code reuse becomes more difficult.

The inheritance is studied for zero, five and three level of inheritance. Five is found to be harder to maintain than zero. Three level program is found to be easier to maintain than zero level [25]. Hence, depth of the class should be less to make it less complex. Increasing depth in inheritance increases the complexity of program making it more difficult to maintain.

4.4.4 Lines of Code

This metric is used to analyze the Lines of Code. The higher Lines of Code indicates that a method or type is harder to maintain. It is also an indication of too much responsibility on a single method and suggests splitting method to make the code more maintainable and readable.

Hence more Lines of Code means method get more complicated

and if more complication makes the code harder to understand and maintain. Software upgrades or bugs fixing becomes difficult, which as a result is a loss of money and time. On the other hand, a smaller code is easier to maintain. Software upgrades, in that case, are less time-consuming and cheaper.

If we count the Lines of Code, we should count the lines we produced rather than counting the lines we spent in developing the application [10]. Lines of code is a very useful metric to analyze as the same functionality in the application can be implemented with either few numbers of Lines of Code and a large number of lines. So it is appropriate to analyze this metric for the whole project as well as the methods, which are taking more number of lines in the application.

Let us take an example of a project having the same functionality, which is developed twice and one have 10,000 lines of code and other 100,000 lines of code. This metric can be useful in the sense that it gives an idea about the Lines of Code and further analysis can be done on the selection of more maintainable project and makings decisions in improving the maintainability of the selected project.

4.4.5 Maintainability Index

Maintainability Index is the metric to measure maintainability of the code and was proposed by Paul Oman and Jack Hagemeister [23]. Maintainability Index can be calculated by the following formulas. The original derived formula is

$$MI = 171 - 5.2 * ln(V) - 0.23 * (G) - 16.2 * ln(LOC) \tag{4.1}$$

We are using visual studio as a static analysis tool, which use slightly modified formula to calculate the Maintainability Index

$$MI = MAX(0, (171 - 5.2 * log(V) - 0.23 * (G) - 16.2 * log(LOC)) * 100/171) \tag{4.2}$$

Where V is the Halstead Volume

G is Cyclomatic complexity

LOC is Lines of Code

A single value of maintainability is measured using this formula and is dependent on Halstead volume, Cyclomatic Complexity and Lines of Code. Halstead complexity metrics are used to measure the complexity quantitatively and is based on operands and the operators. These complexity metrics are measured directory form the source code [14]. They are considered valuable among the early developed software metrics in order to identify the software quality.

η 1 = the number of distinct operators

η 2 = the number of distinct operands

N1=total number of operators

N2=total number of operands

Several measures are calculated using these values such as program vocabulary, program length, volume, difficulty and effort.

The Halstead volume can be calculated using the following formula

$$V = N * log2\eta \tag{4.3}$$

where N is the program length and η is program vocabulary.

Value of the η(program vocabulary) can be calculated as

$$\eta = \eta 1 + \eta 2 \tag{4.4}$$

and value of program length can be calculated as

$$N = N1 + N2 \tag{4.5}$$

4.5 Software Quality Attributes

Software Quality attributes is defined as the degree with which software possesses a desired combination of attributes [26]. The requirements that are significant in the development of quality software are divided into two main groups[26].

- Development Qualities
- Operational Qualities.

The development qualities improve the software maintainability, flexibility and understandability thus providing ease to the developers to upgrade it according to the changing environment and technology. Operational qualities are mainly related to the performance and usability of the software to improve the user experience.

The IEEE standard 610.12-1990 [9] Defines the four quality attributes as

Maintainability: The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment

Performance: The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage.

Testability: The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

Portability: The ease with which a system or component can be transferred from one hardware or software environment to another.

4.6 Performance as a Software Quality Metric

Performance is one most important key quality attribute in software development. One of six characteristics proposed by ISO/IEC 9126 software quality model is efficiency [33]. Efficiency is the capability of software product to provide appropriate performance relative to some resources used under stated conditions [33].

The poor performance of software results in serious consequences such as loss of income and even project failure. In this thesis, our focus is to identify the performance of Design Patterns by measuring their execution times.

4.7 Summary

In this chapter, we discuss the usefulness of static analysis for the development of maintainable software. How static analysis is an efficient way to calculate the quality metrics. We discuss various software quality metrics used to measure software quality such as Cyclomatic Complexity, Class Coupling, Depth in Inheritance, Maintainability Index and Lines of Code and how these metrics are useful to improve software maintainability which as a result reduce the cost of maintenance with the changing technology and environment and software upgrades. This chapter also discusses the importance of performance in software development.

Part III

Research Work

5

Implemented Design Patterns Use Cases

In this chapter, we presented different use cases implemented with Design Patterns such as Observer, Facade, Singleton, Command, and Strategy. All use cases are web-based scenarios and are developed in C# programming language following ASP.net MVC and SQL Server Database.

5.1 Introduction

Architecture to implement use cases is shown in Figure 5.1. Use cases are implemented using Gang of Four [12] Design Patterns such as Observer, Singleton, Command, Strategy, and Facade. Model-View-Controller [19] is used as Architectural Pattern.

The main logic of Design Patterns is in the Model as they are interacting with the database. Controller act as a client for these Design Patterns to complete the user request.

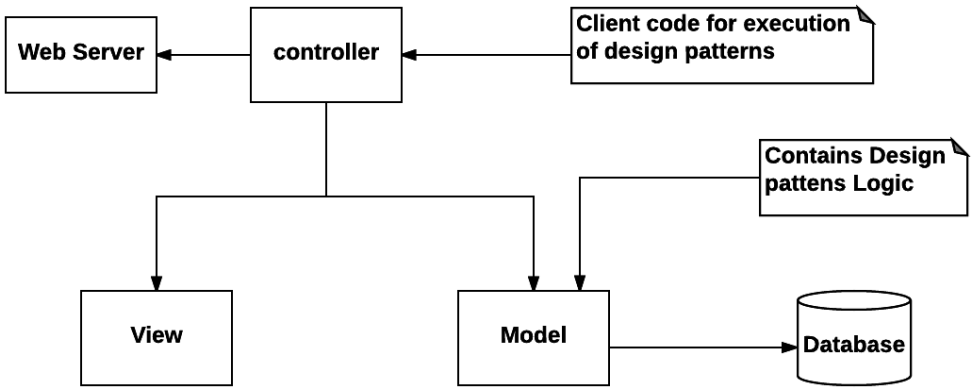

Figure 5.1: Application Architecture

5.2 Implementation of Design Patterns

In this section, we explain in detail all the use case scenarios, we have implemented in the application using Design Patterns.

5.2.1 Singleton

Singleton Pattern ensures instantiating single instance of an object. When one object is used many times to perform certain action within

the application, Singleton restricts the initiation of class to one object and can be used many times within the application.

UML for use case implementation of Singleton Pattern is shown in figure 5.2. We have implemented the exceptionHandler Singleton class. The responsibility of this class is to save the exceptions in the text file to facilitate programmers to inform them about the exception occurrences within the application. GetInstance method checks the creation of an instance of an object. If it is null, new instance is created otherwise the application uses already created object.

To make Singleton thread-safe, we have a read-only object with the name Lock that prevents the creation of multiple instances of object which is violation of Singleton Design Pattern.

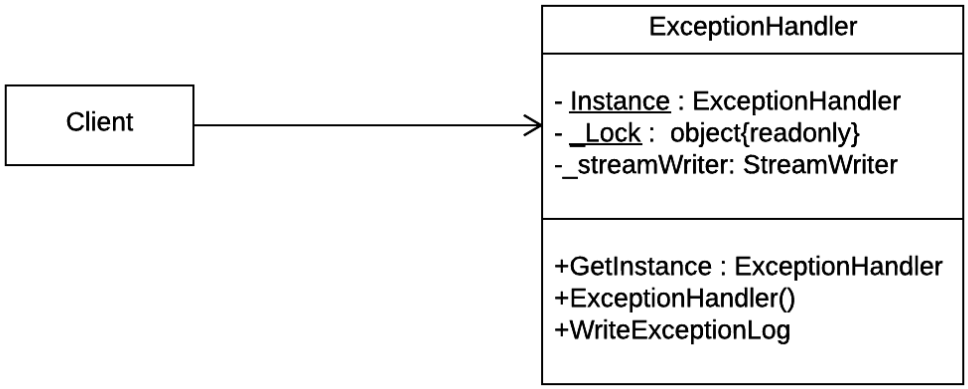

Figure 5.2: Use Case UML representation of Singleton

5.2.2 Observer

Observer Pattern describes, if modification is done in the one object, its dependent objects get notified about the event.

Our UML implementation of Observer Pattern is shown in figure

5.3. User submits the contact information and the Observers get notified on submission of information by the user. Following steps are followed to implement Observer Pattern.

- User submit the information in contact form to get feedback from administrators
- The information is submitted, and two Observer objects get notified about the information submitted by the user.
- One Observer object is responsible for sending the notification to all the administrators by email, and another object is activity tracker save the activity information with time in the database.

The ActivityNotifier and ContactNotifier are the concrete implementations of IContactObserver Interface. The ContactNotifier object is responsible for sending emails to all the administrators. Activity tracker object is to save the information being notified in the database to track activity later. IContactNotifer is the concrete implementation of MainNotifier Object. It maintains the list of IContactObservers and notifies them. The client from Controller setup the MainNotifier that is monitored by the ActivityNotifer and ContactNotifier Objects.

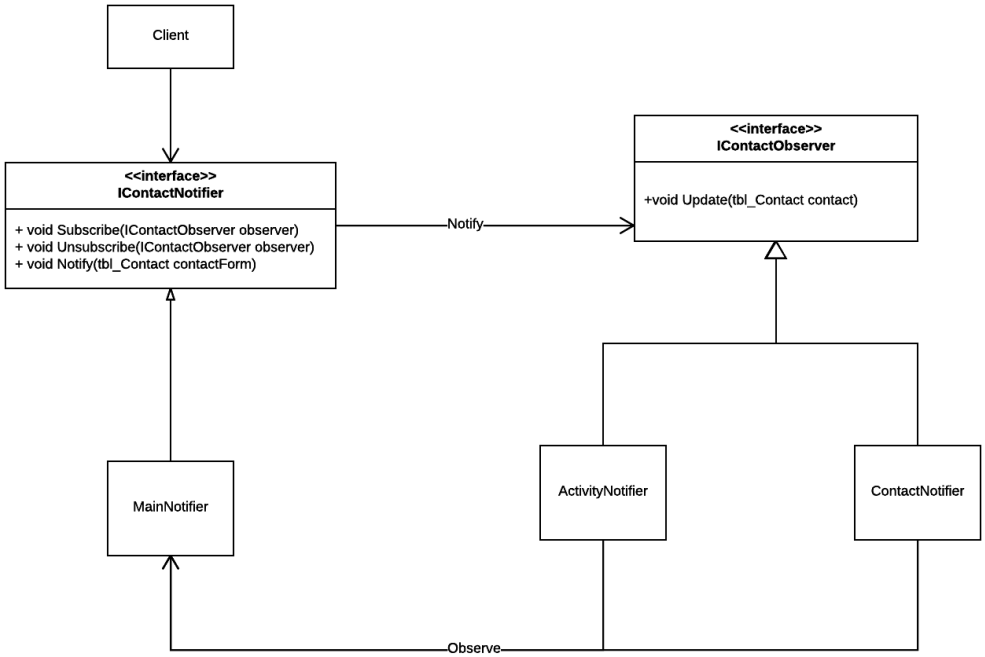

Figure 5.3: Use Case UML representation of Observer

5.2.3 Command

Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations [12].
Our use case implementation is to implement Command Design Pattern to execute the set of commands, when user tries to login and involves the following steps.

- User tries to log in to use services offered by the application.
- Command Pattern execute the set of commands on login to send email to the user about login time and to save logs in the database to track login information later by the administrators.

Our use case implementation of Command Design Pattern is shown in figure 5.4. The commands are passed from the client to Invoker. Invoker class maintains the list of commands and has execute method.

So Invoker invokes all the commands requested by the client.

SaveLog and ConfirmationLogin classes are the concrete implementations of Icommand Interface. Execute method in SaveLog and ConfirmationLogin class calls the method SavetheLog and SendEmail in LoginManager Class to complete the commands to execute.

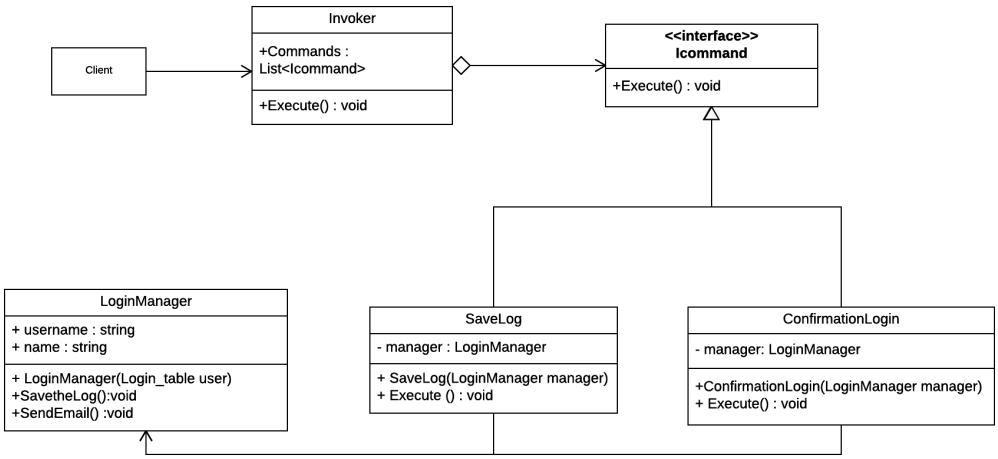

Figure 5.4: Use Case UML representation of Command

5.2.4 Strategy

There can be different strategies to complete the certain task. Strategy Pattern is all about executing Strategy algorithms to accomplish the certain task based on user selection. These strategies can be implemented in a single class, but it increases the complexity of the application. Strategy Pattern decouples code by introducing different strategies in different classes.

Our use case implementation of Strategy Pattern to filter the discount price based on the user selection and involve the following steps. Implementation is also shown in figure 5.5.

- User selects the discount option that are Loyalty discount and

student Loyalty discount.

- The implemented Strategy Pattern invokes the particular Strategy algorithm based on the user selection to show the discount price to the user.

The LoyaltyStudentDiscount class and LoyaltyDiscount class are the concrete implementations of the DiscountStrategy Abstract class. DiscountStrategy Abstract method is implemented in LoyaltyStudentDiscount class and LoyaltyDiscount class. It contains the main logic of different strategies implemented in the application. In our case, each Strategy algorithm has the different discount option. The code is maintainable, so additional Strategy algorithms can be added with the addition of one more class without touching the existing code.

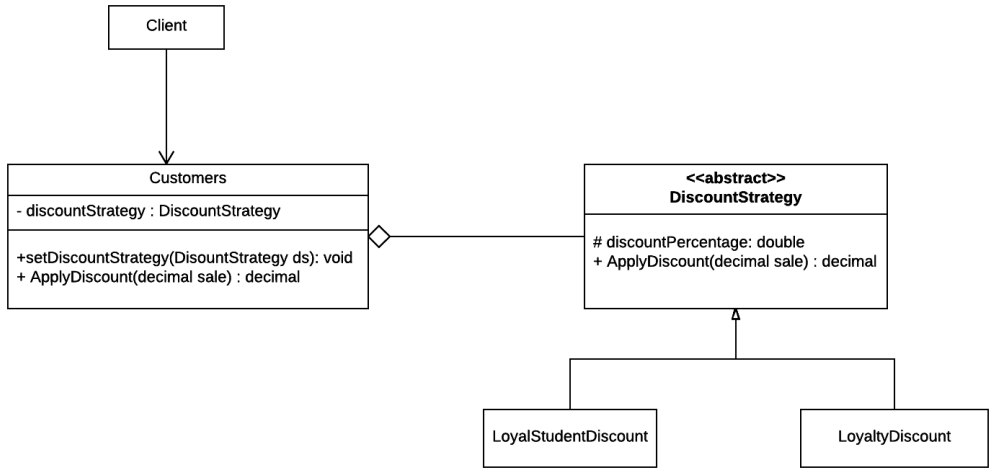

Figure 5.5: Use Case UML representation of Strategy

5.2.5 Facade

The Facade Pattern provides interface to the client hiding subsystems, which make it easier for the client to interact with the one interface instead of interacting with all the subsystems used and avoid complexity.

Our use case implementation of Facade is shown in figure 5.6. The user completes the registration by submission of data. There are three subsystems responsible for fulfilling user request. Implementation of Facade is explained in the following steps.

- User tries to complete the registration to use services.
- The client has to interact with three different subsystems to complete user request.
- Facade simplifies the process by providing one interface to interact with three subsystems to complete client request.

There are three subsystems with names RegisterData, SendEmail and ValidateUser. Registerdata is responsible for saving the registration data to the database. ValidateUser is validating the user data that is the validation of user inputs and email validation by checking it in the database and SendEmail is sending email on successful registration. To avoid the complexity of these subsystems to the client. These subsystems are managed by the RegisterFacade class that interacts with the client. The client code is written in the Controller to communicate with the RegisterFacade class that manages the subsystems to complete the client request.

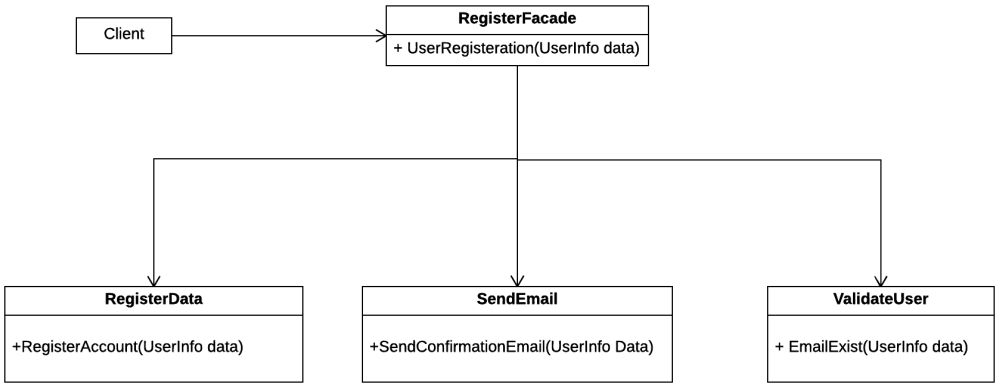

Figure 5.6: Use Case UML representation of Facade

5.3 Summary

In this chapter, our focus is to explain architecture to implement web based use case scenarios using Design Patterns. Use case scenarios are discussed in detail with UML representation implemented with Design Patterns such as Observer, Facade, Singleton, Command and Strategy Design Pattern. The architectural pattern used is Model-View-Controller. In later chapters, we investigate the impact of implemented Design Patterns use cases on software maintainability and performance.

6

Code Maintainability Metrics Measurements

In this chapter, we discuss the results obtained by static analysis on the code of use case scenarios implemented with the Design Patterns. Different quality metrics values are measured and compared with the research previously done to analyze the impact of Design Patterns on maintainability to identify good maintainability scores [8, 21, 25, 30].

6.1 Introduction

To make analyses quantitatively for measuring maintainability of use case scenarios implemented with Design Patterns. We mainly focus on the values of maintainability metrics as a result of static analysis on the code.

Following software, maintainability metrics are measured to verify maintainability of Design Patterns.

- Class Coupling
- Cyclomatic Complexity
- Depth in inheritance
- Maintainability Index
- Lines of Code

We have discussed all the maintainability metrics as mentioned earlier in section 4.4.
Good maintainability criteria for all these maintainability metrics is illustrated as below

- The optimal value for Class Coupling should be nine [30].
- McCabe suggested the value of upper limit for Cyclomatic Complexity is ten [21].
- The inheritance is studied for zero, five and three level of inheritance. Five is found to be harder to maintain than zero. Three level program is found to be easier to maintain than Zero level [25].
- We are using Visual Studio for static analysis of C# code. According to Microsoft a maintainability value between 20 to 100

inclusive is considered as high maintainable code and between 10 and 19 inclusive is considered as moderate maintainable code and between 0 to 9 is indication of the code which is hard to maintain. According to Coleman, the Maintainability Index about 85 is considered to be highly maintainable and between 85 and 65 are moderately maintainable, and components below 65 are harder to maintain [8].

- Too many Lines of Code decrease the maintainability of code by effecting understandability and testability.

6.2 Metrics Measurements

In this section, we mainly discuss the results of maintainability metrics. These maintainability metrics are calculated by doing static analysis on use cases that are implemented with different Design Patterns. Design Patterns can be implemented in several scenarios. Our results are limited to the implementations discussed in chapter 5.

The main purpose of doing static analysis is to measure the values of different maintainability metrics; we mentioned in section 6.1 to verify the usefulness of implementing software Design Patterns on software maintainability. We measure MI(Maintainability Index), CBO(Class Coupling), CC(Cyclomatic Complexity), LOC(Lines of Code) and DIT(Depth in Inheritance) for different implementations of Design Patterns.

6.2.1 Singleton Pattern Quality Metrics Measurements

Maintainability metrics for implementation of Singleton is illustrated in table 6.1. It is observed that all the maintainability metrics for Singleton Design Pattern are within limit of maintainability thresholds and good scores for maintainability are observed for Singleton Design Pattern.

	MI	**CBO**	**CC**	**LOC**	**DIT**
Exceptionhandler	73	8	6	16	1

Table 6.1: Singleton Pattern Software Quality Metrics Measurements

6.2.2 Observer Pattern Quality Metrics Measurements

The different maintainability metrics for Observer Pattern are shown the table 6.2. The table illustrates metrics values for different classes and interfaces of Observer Design Pattern. We have observed that the ContactNotifier class has Cyclomatic Complexity of fourteen which is above the threshold value that is ten. It is due to the code written to send emails to notify multiple administrators about the event. The activityNotifier class has code to track the event to save it in the database and has a Maintainability Index value of 76. However, the overall maintainability scores for the Observer Pattern leads toward the improvement of maintainability.

	MI	CBO	CC	LOC	DIT
IContactObserver	100	1	1	0	0
IContactNotifier	100	2	3	0	0
ActivitiyNotifier	76	8	2	6	1
MainNotifier	91	6	6	7	1
ContactNotifier	62	6	14	14	1

Table 6.2: Observer Pattern Software Quality Metrics Measurements

6.2.3 Command Pattern Quality Metrics Measurements

As shown in table 6.3, Good maintainability scores are observed for Command Design Pattern. Among different classes and interfaces used to implement Command Design Pattern, the lowest score of Maintainability Index is of LoginManager Class which has the code for saving the logs in the database and sending email to the user logged in to the system. However, measurements of metrics indicate that implementation of Command Pattern improves the maintainability of the software.

	MI	CBO	CC	LOC	DIT
ICommand	100	0	1	0	0
ConfirmationLogin	94	2	2	3	1
SaveLog	94	2	2	3	1
Invoker	90	4	5	6	1
LoginManager	66	11	5	18	1

Table 6.3: Command Pattern Software Quality Metrics Measurements

6.2.4 Strategy Pattern Quality Metrics Measurements

Different components maintainability metrics for Strategy Design Pattern are shown in table 6.4. Very good maintainability stores are observed for Strategy Design Pattern.

	MI	CBO	CC	LOC	DIT
DiscountStrategy	96	1	4	3	1
LoyaltyDiscount	82	2	2	4	2
StudentDiscount	82	2	2	4	2
Customers	93	2	3	4	2

Table 6.4: Strategy Pattern Software Quality Metrics Measurements

6.2.5 Facade Pattern Quality Metrics Measurements

Overall good maintainability scores are observed for different maintainability metrics for the use case implementation of Facade Design Pattern. The results for Facade Design Pattern also conclude that its implementation does not affect software maintainability. The lowest score of maintainability is observed in RegisterFacade class which is also in good maintainability score range. The highest coupling value twelve is observed in ValidateUser class which is above the threshold value of Class Coupling that is Nine.

From the metrics values generally, Facade Pattern implementation improves the maintainability. Table 6.5 Illustrates the different maintainability metrics measurement for Facade Design Pattern.

	MI	CBO	CC	LOC	DIT
RegisterFacade	71	4	3	11	1
RegisterData	83	4	2	5	1
SendEmail	68	5	3	11	1
ValidateUser	94	12	3	12	1

Table 6.5: Facade Pattern Software Quality Metrics Measurements

6.3 Conclusion

In this chapter, we discuss the maintainability metrics for different Design Patterns, we have implemented. It is observed that the Design

Patterns impact positively on maintainability of software applications. We generate results for different maintainability metrics such as Class Coupling, Depth in Inheritance, Maintainability Index, Lines of Code and Cyclomatic Complexity. We conclude that the implementation of Design Patterns leads toward good scores of maintainability and their implementation is beneficial. However, it is also concluded that code implementations following the Design Patterns should be done with caution as we have observed that metric values in some classes are above threshold limits which are indication of Code Smells [11] in some classes of Design patterns but there is less probability of poor maintainability by implementing Design Patterns.

7

Performance Measurements

Apart from software maintainability, performance is also one of the most important quality attribute, which should be taken into the consideration in software development. The poor performance of software results in loss of income, project failure due to the incapability of meeting performance objectives, the high cost of redesign and loss of business reputation. It is therefore important to investigate the impact of Design Patterns on performance of the application.

In this chapter, we mainly discuss the impact of Design Patterns on performance. The discussion is on calculated execution time for different Design Patterns implemented in use cases (See Chapter 5). The results of execution time for different implementation of Design

Patterns is compared with the software maintainability metrics results discussed in chapter 6 to identify the effect of maintainability on performance.

7.1 Execution Time for Design Patterns

The performance of different Design Patterns use case implementations discussed in Chapter 5 is analyzed by calculating the execution time in milliseconds. To perform calculations, the machine used has system specifications shown in table 7.1.

Stopwatch class is used and present in System.Diagnostics name space

Processor	Intel Core i7 4400U
RAM	8Gb
Operating system	Windows 10
Web Server	IIS

Table 7.1: Hardware Specification for Measuring Execution Time

in .Net Framework offered by Microsoft to calculate execution time in milliseconds for different Design Pattern use cases.

The limitation to measure the performance is that execution time might be different for each calculation due to many reasons. To get the more accurate value of execution times in our measurements. We have taken five test for each Design Pattern use case implementation discussed in Chapter 5. The reason is that execution time can be altered by many factors such as Garbage Collection, caching, the computer might perform other tasks during program execution, Tests are performed on Windows 10 which is not the real operating system and .Net provides managed code that can alter execution time.

Due to all these factors and to get most accurate value, it is more appropriate to average the execution time values by performing many

tests. We have taken five tests to calculate the execution time for each Design Pattern use case. The behavior can be execution time measurements is visualized in the graphs.

7.1.1 Singleton

Execution times for Singleton Design Pattern is shown in figure 7.1. The average execution time for Singleton Design Pattern is 293.5 milliseconds. The value is the average of five tests. However, in the first test, Singleton object is created, and other four tests used the same object which is created in the first test.

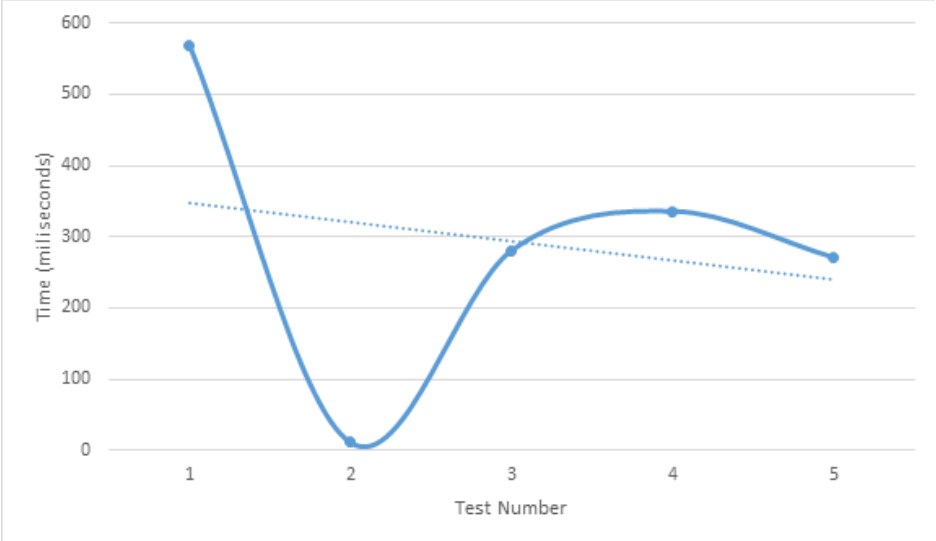

Figure 7.1: Execution Time for Singleton

7.1.2 Observer

The execution times for Observer Design Pattern is shown in figure 7.2. The average value of execution time for five different tests on Observer Design Pattern is 684.6 milliseconds. Observer Design Pattern contains several functionalities such as sending emails to

multiple administrators and multiple queries are involved for saving information to database. The execution time of 684.6 is very good score in terms of performance.

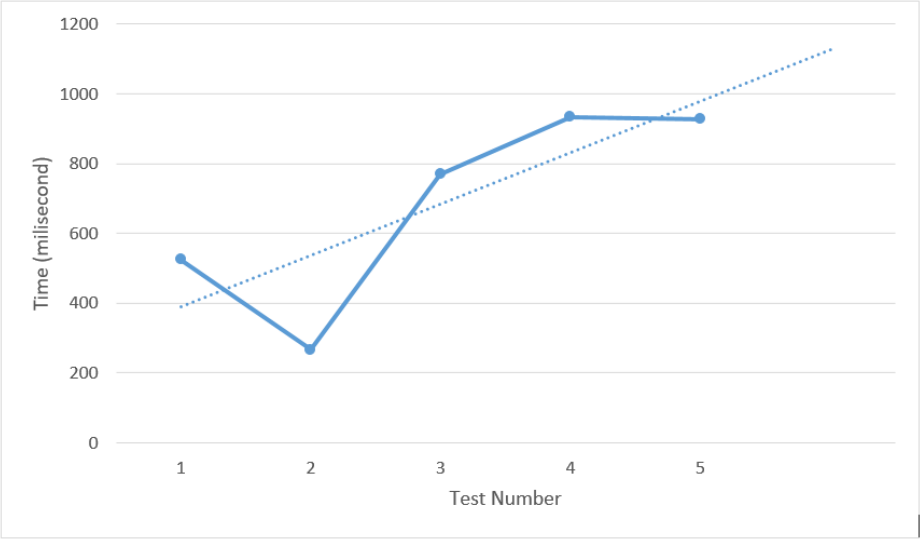

Figure 7.2: Execution Time for Observer

7.1.3 Command

The execution times for five different tests on Command Design Patterns is shown in the figure 7.3. The average execution time for Command Design Pattern is 331.2 milliseconds.

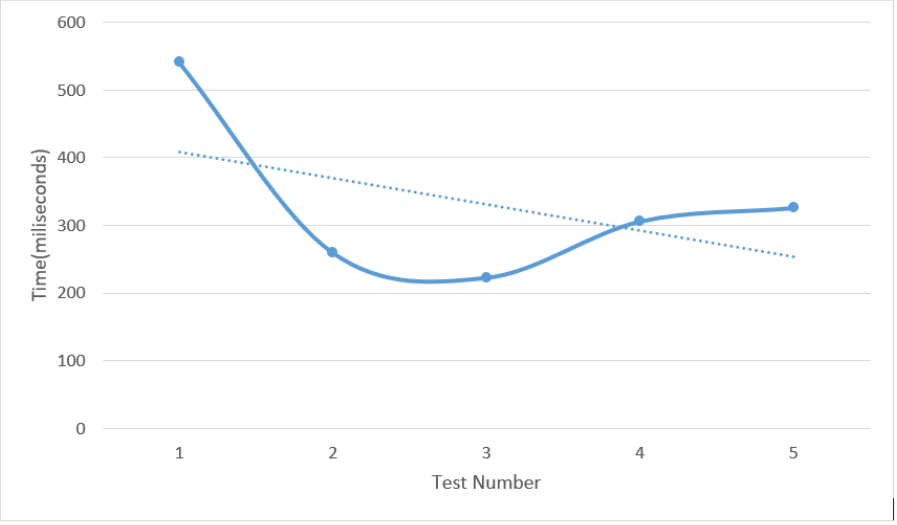

Figure 7.3: Execution Time for Command

7.1.4 Strategy

The execution time for use case implementation of Strategy Design Pattern is shown in figure 7.4. The average execution time for strategy pattern is 130 milliseconds.

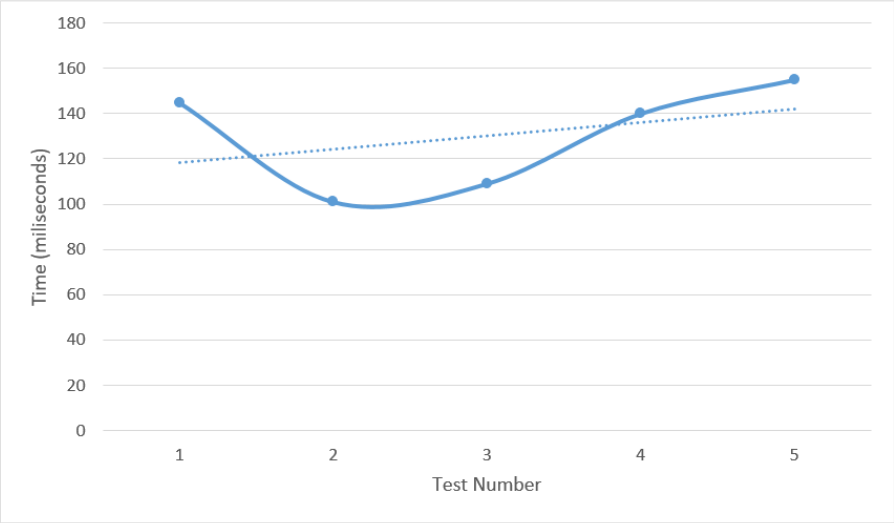

Figure 7.4: Execution Time for Strategy

7.1.5 Facade

The execution times for Facade Design Pattern is shown in the figure 7.5. The average execution time calculated for five different tests is 277.2 milliseconds.

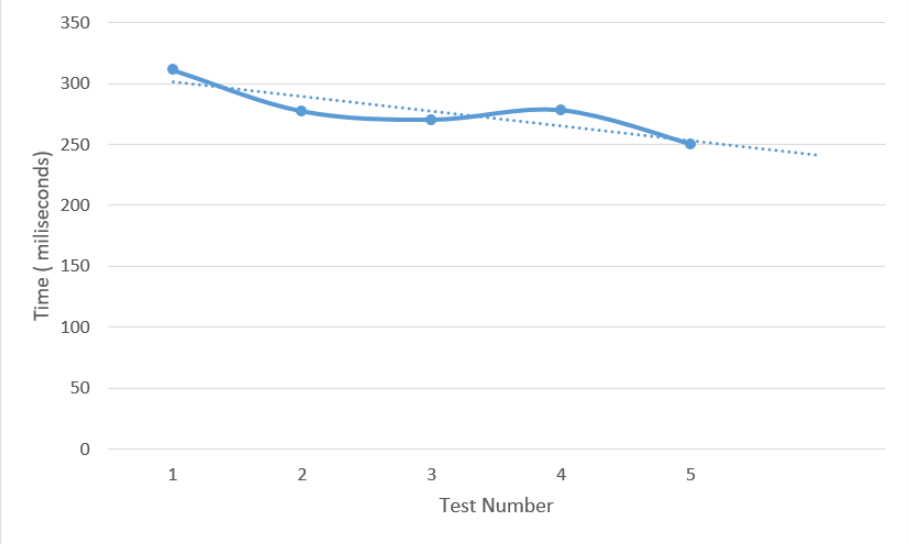

Figure 7.5: Execution Time for Facade

7.2 Discussion on Results

Average execution time for all Design Patterns implemented is shown in figure 7.6. It is observed from the results that execution time for most of the Design Patterns is from 100 to 350 milliseconds except Observer pattern. From the data and analysis, it can be concluded that the Design Patterns do not affect the performance of the application. Observer pattern is observed as slowest with 684.6 milliseconds. In table 6.2, maintainability metrics for Observer Design Pattern are mentioned. It is observed that ContactNotifier class implemented in Observer Design Pattern use case have a slightly low value of Maintainability Index and higher value of Cyclomatic Complexity. Same

is the case with Command Design Pattern as seen in section 6.2.3, the lowest Maintainability Index 66 and high coupling value 11 is observed for LoginManager Class and average execution time observed is 331.2 milliseconds. The Strategy Pattern has excellent scores for maintainability and average execution time for Strategy Pattern is lowest that is 100 milliseconds.

It can be concluded from the results that Design Patterns itself do not affect the performance of the application. Overall scores for execution time for all the Design Patterns are good. However, the code written to implement different functionalities following the Design Patterns can affect the performance of the application and also the maintainability. To identify these Code Smells [35] maintainability metrics discussed in chapter 6 are very useful.

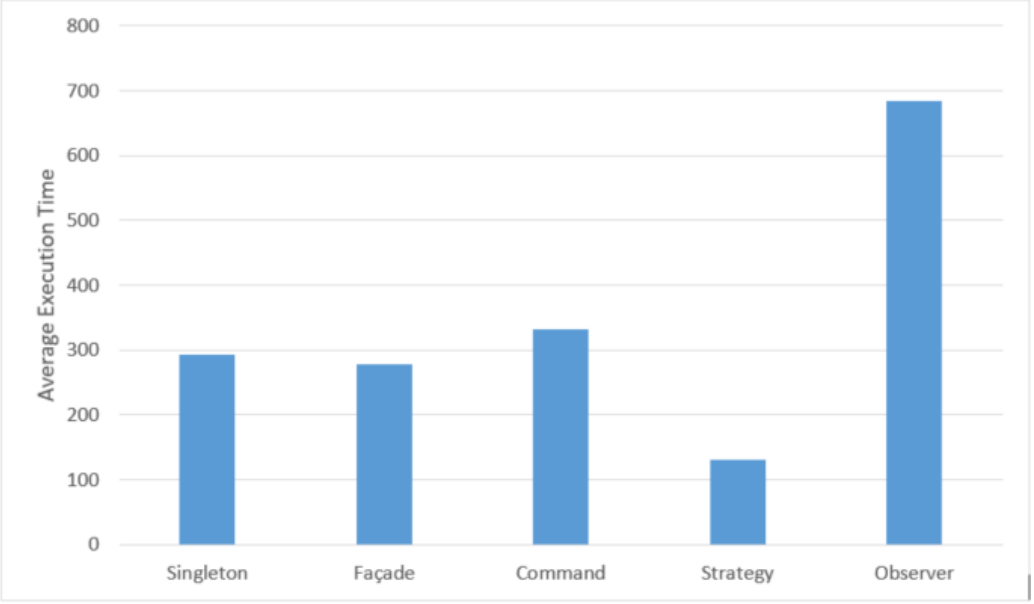

Figure 7.6: Average Execution Time for Design Patterns

7.3 Limitations

There can be several scenarios, where Design Patterns can be implemented. Our measurements are limited to the Use cases, We have implemented using different Gang of Four Design patterns and Model-View-Controller as Architectural Design Pattern. We already discussed in 7.2 that we have taken five measurements of executions time for each implementation of Design Patterns to measure the most accurate trend of performance. The reason is the execution time is altered by many reasons such as Garbage Collection, caching, the computer might perform other tasks during test execution and use of non-real Operating System.

7.4 Conclusion

In this chapter, we mainly discuss the execution times for different use case implementation of Design Patterns discussed in Chapter 5. The average execution time for different Design Patterns are calculated. It is observed that overall the Design Patterns implementation do not have the negative impact on the performance of the application. Good scores are observed for different implemented Design Patterns. However poorly maintainable code can be written even following the Design Patterns, and it can affect the performance of the application. Software maintainability metric scores are very useful to identify and rectify issues with the poorly maintainable code.

8

Conclusion

In this thesis, our focus is to evaluate the Design Patterns to measure their impact on non-functional quality attributes such as software maintainability and performance. We develop different use cases with Design Patterns such as Observer, Facade, Singleton, Command, and Strategy. All use cases are web-based scenarios. Model-View-Controller is followed as architectural Pattern in development of use cases. We formalized some research questions, and empirical research methodology is performed to evaluate Design Patterns impact on maintainability and performance.

To evaluate maintainability, we measure software maintainability metrics such as Cyclomatic Complexity, Class Coupling, Lines of code,

Maintainability Index and Depth in Inheritance on use case implementation of different Design Patterns such as Observer, Singleton, Facade, Command, and Strategy. Maintainability is measured with software quality metrics using Static Analysis. It is observed that good scores are achieved for maintainability metrics from Design Pattern use case implementations. The impact of Design Patterns on maintainability is positive.

However, in some classes, we observe some metrics values are above the threshold limits. In Observer Pattern, high Cyclomatic Complexity is observed for ContactNotifier class (See section 6.2.2). In Command Pattern, high classes coupling is observed for LoginManager Class, and in Facade pattern, high coupling values are observed for RegisterFacade Class.

We have concluded that the Design Patterns should be implemented in software applications with caution and probability of poor maintainability is less by use of Design patterns.

To measure performance, average execution time is calculated for each use case implementation of Design Pattern. It is concluded that implementation of Design Patterns in software applications does not affect the performance. We also suggest that poorly maintainable code can be written by following Design Patterns and It can affect the performance of the application.

Hence, we conclude that Design Patterns are very useful for the development of software applications. We have observed that good scores for performance and maintainability are achieved in our study, and there is less probability of poorly maintainable code in application by use of Design Pattern.

We further concluded that there are fewer chances of performance

issues in maintainable code than the complex and poor maintainable code and performance issues are easier to rectify in maintainable code than the complex code. So Design Patterns implementation is beneficial to improve the performance of the applications.

Appendix

The code written for experimentation purpose can be obtained from the following URL.

`https://github.com/farooqspecials/DesignPatterns`

Bibliography

[1] Christopher Alexander. *The timeless way of building*. Vol. 1. New York: Oxford University Press, 1979.

[2] Fatimah Mohammed Alghamdi and M Rizwan Jameel Qureshi. “Impact of Design Patterns on Software Maintainability”. In: *International Journal of Intelligent Systems and Applications* 6.10 (2014), p. 41.

[3] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. “A validation of object-oriented design metrics as quality indicators”. In: *IEEE Transactions on software engineering* 22.10 (1996), pp. 751–761.

[4] Keith H Bennett and Václav T Rajlich. “Software maintenance and evolution: a roadmap”. In: *Proceedings of the Conference on the Future of Software Engineering*. ACM. 2000, pp. 73–87.

[5] Joseph P Cavano and James A McCall. “A framework for the measurement of software quality”. In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 7. 3-4. ACM. 1978, pp. 133–139.

[6] Shyam R Chidamber and Chris F Kemerer. “A metrics suite for object oriented design”. In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[7] Shyam R Chidamber and Chris F Kemerer. “A metrics suite for object oriented design”. In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[8] Don Coleman et al. “Using metrics to evaluate software system maintainability”. In: *Computer* 27.8 (1994), pp. 44–49.

[9] IEEE Standards Coordinating Committee et al. “IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos”. In: *CA: IEEE Computer Society* (1990).

[10] Edsger W Dijkstra et al. “On the cruelty of really teaching computing science”. In: *Communications of the ACM* 32.12 (1989), pp. 1398–1404.

[11] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[12] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[13] Ivo Gomes et al. “An overview on the static code analysis approach in software development”. In: *Faculdade de Engenharia da Universidade do Porto, Portugal* (2009).

[14] Maurice Howard Halstead. *Elements of software science*. Vol. 7. Elsevier New York, 1977.

[15] Péter Hegedűs et al. “Myth or reality? analyzing the effect of design patterns on software maintainability”. In: *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity* (2012), pp. 138–145.

[16] John A Hoxmeier and Chris DiCesare. “System response time and user satisfaction: An experimental study of browser-based applications”. In: *AMCIS 2000 Proceedings* (2000), p. 347.

[17] Foutse Khomh and Yann-Gael Gueheneuce. “Do design patterns impact software quality positively?” In: *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE. 2008, pp. 274–278.

[18] G Krasner and S Pope. *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80 J. Object Oriented Program., 1, 26-49*. 1988.

[19] Glenn E Krasner, Stephen T Pope, et al. “A description of the model-view-controller user interface paradigm in the smalltalk-80 system”. In: *Journal of object oriented programming* 1.3 (1988), pp. 26–49.

[20] Robert Leitch and Eleni Stroulia. “Assessing the maintainability benefits of design restructuring using dependency analysis”. In: *Software Metrics Symposium, 2003. Proceedings. Ninth International*. IEEE. 2003, pp. 309–322.

[21] Thomas J McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[22] William B McNatt and James M Bieman. “Coupling of design patterns: Common practices and their benefits”. In: *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*. IEEE. 2001, pp. 574–579.

[23] Paul Oman and Jack Hagemeister. “Metrics for assessing a software system’s maintainability”. In: *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE. 1992, pp. 337–344.

[24] Robert E Park. *Software size measurement: A framework for counting source statements*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1992.

[25] Lutz Prechelt et al. “A controlled experiment on inheritance depth as a cost factor for code maintenance”. In: *Journal of Systems and Software* 65.2 (2003), pp. 115–126.

[26] Usman Rafi et al. “US-Scrum: A Methodology for Developing Software with Enhanced Correctness, Usability and Security”. In: ().

[27] Rajendra K Raj et al. “Emerald: A general-purpose programming language”. In: *Software: Practice and Experience* 21.1 (1991), pp. 91–118.

[28] Trygve Reenskaug. “Models-views-controllers”. In: *Technical note, Xerox PARC* 32.55 (1979), pp. 6–2.

[29] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. “A systematic review of software maintainability prediction and metrics”. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society. 2009, pp. 367–377.

[30] Raed Shatnawi. “A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems”. In: *IEEE Transactions on software engineering* 36.2 (2010), pp. 216–225.

[31] Dag IK Sjøberg, Bente Anda, and Audris Mockus. “Questioning software maintenance metrics: a comparative case study”. In: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM. 2012, pp. 107–110.

[32] Connie U Smith and Lloyd G Williams. “Performance solutions: a practical guide to creating responsive, scalable software”. In: (2001).

[33] International Organization for Standardization and International Electrotechnical Commission. *Software Engineering–Product Quality: Quality model*. Vol. 1. ISO/IEC, 2001.

[34] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. “Structured design”. In: *IBM Systems Journal* 13.2 (1974), pp. 115–139.

[35] Eva Van Emden and Leon Moonen. “Java quality assurance by detecting code smells”. In: *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE. 2002, pp. 97–106.

[36] Peter Wendorff. “Assessment of design patterns during software reengineering: Lessons learned from a large commercial project”. In: *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. IEEE. 2001, pp. 77–84.

[37] Lloyd G Williams and Connie U Smith. “Performance evaluation of software architectures”. In: *Proceedings of the 1st international workshop on Software and performance*. ACM. 1998, pp. 164–177.

[38] Edward Yourdon and Larry L Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc., 1979.

[39] Cheng Zhang and David Budgen. “What do we know about the effectiveness of software design patterns?” In: *IEEE Transactions on Software Engineering* 38.5 (2012), pp. 1213–1231.