USE OF DESIGN PATTERNS IN NON-OBJECT ORIENTED REAL-TIME SOFTWARE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AYŞEGÜL ÇİFTCİ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JANUARY 2013

Approval of the thesis:

USE OF DESIGN PATTERNS IN NON-OBJECT ORIENTED REAL-TIME SOFTWARE

submitted by **AYŞEGÜL ÇİFTCİ** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. İsmet ERKMEN
Head of Department, **Electrical and Electronics Engineering** _____

Prof. Dr. Semih Bilgen
Supervisor, **Electrical and Electronics Engineering Dept., METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Ilkay Ulusoy Parnas
Electrical and Electronics Engineering Dept., METU _____

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Ali Hikmet Doğru
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering Dept., METU _____

Özgü Özköse Erdoğan, M.Sc.
Software Department Manager, ASELSAN _____

Date: 25.01.2013

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: AYŞEGÜL ÇİFTCİ

Signature :

ABSTRACT

USE OF DESIGN PATTERNS IN NON-OBJECT ORIENTED REAL-TIME SOFTWARE

Çiftci, Ayşegül
M.S., Department of Electrical and Electronics Engineering
Supervisor : Prof. Dr. Semih Bilgen

January 2013, 65 pages

After the book, Design Patterns: Elements of Reusable Object-Oriented Software was published in 1994, usage of design patterns in object-oriented (OO) programming has been investigated by many researchers. However, the effects of design patterns on non-object oriented (non-OO) programming have not been analyzed too much in the literature. This study focuses on various design pattern implementations using non-OO programming and investigates the benefits of design patterns upon real-time software. In order to evaluate the results, specific quality metrics were selected and performance of traditionally developed software was compared with that of software developed using design patterns.

Keywords: Design Patterns, Non-Object Oriented Programming, Software Maintainability, Real Time Software Performance

ÖZ

TASARIM KALIPLARININ GERÇEK ZAMANLI NESNE TABANLI OLMAYAN YAZILIMLARDA KULLANIMI

Çiftci, Ayşegül
Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Semih Bilgen

Ocak 2013, 65 sayfa

1994 yılında Design Patterns: Elements of Reusable Object-Oriented Software kitabının basımından sonra tasarım kalıplarının nesne tabanlı programlamada kullanımı bir çok araştırmacı tarafından incelenmiştir. Ancak, tasarım kalıplarının nesne tabanlı olmayan programlama dillerinde kullanımı çok fazla analiz edilmemiştir. Bu çalışma, tasarım kalıplarının nesne tabanlı olmayan yazılımlarda nasıl uygulanabileceğine odaklanmıştır ve bu kullanımın gerçek zamanlı yazılımlarda sağlayacağı yararlar araştırılmıştır. Sonuçları değerlendirmek için ise, uygun metrikler seçilmiş ve geleneksel yöntemlerle geliştirilmiş yazılımlar tasarım kalıpları uygulanmış yazılımlar ile karşılaştırılmıştır.

Anahtar Kelimeler: Tasarım Kalıpları, Nesne Tabanlı Olmayan Programlama, Yazılım Bakım Yapılabilirliği, Gerçek Zamanlı Yazılım Başarımı

To my mother and Mustafa Ozan...

ACKNOWLEDGMENTS

This thesis would not have been possible without the help of several individuals who in one way or another contributed their valuable help in the preparation of this study.

First and foremost, I offer my sincerest gratitude to my advisor, Prof. Dr. Semih Bilgen. I know him from my undergraduate years. I know how he fights for his beliefs and I am grateful that he had faith in me for this study. His support and valuable guidance are my inspirations as I encounter problems in the completion of this research work.

I would like to thank to ASELSAN INC for encouragements and resources that are supported for this thesis.

I really appreciate the love and support of all my friends. They make me smile when I get tired throughout this thesis.

I owe my deepest gratitude to my mother, dear sister, and all my family members for their infinite love and support. They are one of the best families one can have.

Last but not the least; I would like to express my special thanks to Mustafa Ozan for his endless love and patient. His understanding, encouragement and trust throughout this study are beyond description.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

LIST OF TABLES

TABLES

LIST OF LISTINGS

LISTINGS

LIST OF ABBREVIATIONS

ABTM	: A-B Timing Measurement
ADT	: Abstract Data Type
CC	: Cyclomatic Complexity
COM	: Comment on Module
HV	: Halstead Volume
IDE	: Integrated/Interactive Development Environment
LOC	: Line of Code
MI	: Maintainability Index
Non-OO	: Non-Object Oriented
OO	: Object-Oriented

CHAPTER 1

INTRODUCTION

1.1 Introduction to the Problem

Career-professionals, especially in the area of software engineering, work together to compose large scale systems by using small well-designed pieces and each area of specialization is considered to be accessible over well-defined interfaces to others. Detailed design and implementation processes are hidden from users. Software can be thought as a big company in which not only each part should perform its responsibility perfectly, but also they have to be used together. Therefore, providing useful interface to small software units and making them re-usable is advisable. When the OO concept was introduced in the 1980's, programmers thought that they found the solution to achieve reusability of software. It was believed that OO programming could resolve the problem by providing the features which structural programming cannot offer, like data abstraction and encapsulation. The reality showed that OO concept was just one step among many, however a very important one, toward software reusability [6].

According to [19], OO design is one of the solutions for extending the life-span of software. However, this design requires high attention to achieve this goal. On this subject, Gamma, Helm, Johnson, and Vlissides collected their experiences and provided proven solutions to the problems that every programmer encounters over and over again [14]. In [8], the author claims that developers without the knowledge of design patterns cannot understand the critical points in design issue. He suggests that every developer should understand the general idea behind design patterns in order to gain a new design point of view. Solutions proposed by design patterns are based on the idea that well-designed code is described as reusable, extensible and maintainable [17] [18].

1.2 Statement of the Problem

From the beginning of the years OO design patterns were introduced, many studies on the effects of design patterns have been carried out. Language specific implementations were given in many books as well as their effects on software quality. Furthermore, [5] explored the language dependencies of the design patterns and compared the performance of languages under the same design pattern implementations. Although that study includes only five leading programming languages and does not include C, language dependency of design patterns is an important feature that emerges in this study.

The only non-OO implementation for design patterns were given in Petersen's publications [21]. In these publications, First Class Abstract Data Type (ADT), State, Strategy, Observer and Reactor Patterns were implemented in C language and each step of the implementation process was given in detail. The aim of these papers is to motivate the C programmers who do not want to be outdated by design

patterns. These implementations show that unlike the widely accepted opinion, design patterns are not necessarily OO. On the other hand, Petersen's study is just an implementation guide for five design patterns. There is no study in the literature that implements design patterns in C programming language and measures its effects. Clearly, there is a need to determine how design pattern implementations affect quality of software in C, as a non-OO programming language.

The main aim of the present study is to investigate whether the performance of real-time software is degraded with design patterns while maintainability increases. The literature largely accepts that usage of design patterns has a price in terms of system performance. Although design patterns increase one or more quality characteristic of software, increasing performance is not the aim of any of these patterns [10]. To be strictly accurate, whether the amount of degradation will be acceptable or not will be the main question here. Hence, based on a review of the relevant literature, to be presented in Chapter 2, below, the hypotheses of the present study can be stated as:

Hypothesis-1 : Using OO design patterns in C programming language, increases the software maintainability.

Hypothesis-2 : Using OO design patterns in C programming language, does not pose a threat related to performance efficiency of the system if it is used in a correct manner.

1.3 Purpose of the Study

"The problem with OO languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

(Joe Armstrong)

Since OO programming was introduced, there has been an endless debate between the developers about the subject whether OO programming is better than structured-procedural programming or not. The comparison between these groups is not the scope of this study. However, the reason why C++ is not preferred will be mentioned briefly in this section.

OO programming inherently has some features which can be very helpful, if the tasks to be carried out are not time-critical. To provide encapsulation and data abstraction, it usually sacrifices performance. Therefore, time constraint forces real-time software developers to compromise the maintainability of OO technology. On the other hand, without taking the total OO overhead, one can profit from the design principles of OO idea. [29] reports a study to show the effect of object modeling technique on non-OO language and concludes that this technique leads to fewer anomalies and needs less fixing time. This explains the reason why Bruce used object-based perspective while proposing some real time design patterns for C [12].

By implementing a similar technique, OO design patterns can be used in a non-OO way. Since the catalog is given as "Design Patterns: Elements of Reusable Object-Oriented Software", one may think that they are not applicable in non-OO programming. However, Petersen's publications show that unlike the accepted opinion, design patterns do not have to be restricted to OO software.

The primary purpose of this study is to show implementation of some OO design patterns in C programming language and to investigate the effects of these patterns on performance and maintainability

concern of the software.

1.4 Significance of the Study

While real-time software developers are searching for maintainable software components, they cannot discard the time requirements. This thesis will provide a way to have more maintainable software while satisfying the performance constraints.

One important contribution of this study is that it proposes a way to implement the OO design patterns in C programming language. Since there is no study recorded about this topic in the literature, except from Petersen's publications [21], it will help to make up for this deficiency. This thesis aims to set up a model to implement design patterns in C. The proposed model, to be explained in Chapter 3, focuses on the general class diagrams of design patterns and proposes a guide for implementation of these diagrams. The claim here is to show that many design patterns can be applied by using this guide. To proof this thesis, four design patterns, needed by software under investigation, were implemented. Then, the effects of these patterns on software maintainability and performance efficiency were investigated. These are the points that have not been answered in the literature yet. Therefore, showing that the maintainability of the software is improved and degradation of the performance is acceptable may lead real-time software developers to use design patterns.

1.5 Outline

This thesis document has been divided into five chapters and the first one is organized to underline the reason why this topic is chosen and to outline the rest of the paper.

Before going into the detailed design, a literature survey about design patterns and software quality metrics will be presented in Chapter 2. Studies that have been conducted so far will be reviewed and in which part of the literature that this study can hold a place will be underlined. Then, the proposed model will be given in Chapter 3 and the detailed information about case studies will be shared under Chapter 4. Finally, the last chapter will assess the conclusion and discussions about the study.

CHAPTER 2

LITERATURE REVIEW

"A good design is one composed of a set of design patterns applied to a piece of functional software that achieves a balanced optimization of the design criteria and incurs an acceptable cost in doing so."

(B. P. Douglass, "Design Patterns for Embedded Systems in C")

2.1 What is a Software Design Pattern?

Pattern concept was first introduced by Christopher Alexander in 1977. It was an architectural concept used for recurring design issues about buildings. Kent Beck and Ward Cunningham applied this idea to programming in 1987. After that, many attempts have been recorded in the literature. Lots of studies have been conducted, many experiences have been reported, and many groups aiming to find new patterns have been established. Some of them will be reviewed below.

Software Design Patterns are defined as "proven solutions to recurring design problems" [14]. They propose a design model for a specific purpose. Design itself is reusable and makes the software reusable with the help of "open for extension, closed for modification" principle.

Restricting the patterns with this catalogue is not right. Some architectural patterns or real time design patterns also serve the same purpose. They propose solutions to problems that occur over and over again. They provide a common vocabulary for developers. This common understanding makes cooperative working easier. Since design details are already known, the documentation process is also faster [8].

2.2 Software Design Patterns in the Literature

As software becomes large-scale, maintenance and reusability become difficult. The necessity for a well-designed code has increased in the years. When the design patterns idea was introduced, it was taken as a remedy. Many researchers started to learn and implement the design patterns and many industrial experiences were reported in the literature.

At the beginning of the years when design patterns had started to receive a lot of attention, software designers from the companies, First Class Software, AT&T, Motorola Inc., Siemens AG, Bell Northern Research and IBM Research collected their experience about design patterns in an article. Although that article does not contain measurable results, it is clear that the interest in the issue has increased

dramatically [4]. Among the companies included in another study [23], Motorola and Ericson have the first large-scale distributed system projects, adopting reuse strategy based on design pattern techniques. Dozens of design patterns are implemented in these systems. The main outcome of this study is that, reusing existing software is possible by using design patterns. After that, many industrial experience reports are added with the similar opinion. To illustrate, [25] summarizes the benefits of the design patterns usage in a speech recognition application. An OO framework is extended to develop four different applications. In order to perform the reuse of design for each application, different design patterns are used and decision mechanism for choosing the patterns is explained at each level. Adapter, Observer, Facade, Singleton, Active Object, Asynchronous Completion Token and Service Configurator Patterns are the patterns that are examined in the scope of this study. In the end, the author concludes that using design patterns enriches the design and makes the reuse of the existing software possible.

All these experience reports have a common point on the idea that design pattern smoothes the path for having reusable and easily extendable software. In the case study reported in [19], in order to see the effect of design patterns, decision tree learning system is redesigned and the effects of patterns on software flexibility and extensibility are evaluated quantitatively. Detailed analysis of different decision tree learning systems is made and hot-spots of the system are listed. Although all of them are implemented, one is chosen and its implementation process is detailed in the paper. Before applying design patterns and after the implementation, C&K metrics, given in [7], are used to measure the software flexibility and extensibility. Then, Mann-Whitny U-test, one of the non-parametric statistical hypothesis tests for assessing whether one of two samples of independent observations tends to have larger values than the other, is used with 5% difference level. According to this result, design patterns increase the software flexibility and extensibility.

Similarly, [1] evaluates effect of design pattern usage in real-time embedded software in terms of code size, reusability, extensibility and understandability. In order to make such an evaluation, a radar system which has different interfaces with different protocols is used. Whenever a new interface is introduced to the radar system, it has to be redesigned since extensibility is not applicable. In this study, pure C code implementation of this radar system is improved with five different approaches. As the sixth step, the code is converted to C++ programming language and Bridge Design Pattern is applied. Six different implementations are compared in terms of reusability, extensibility and understandability qualitatively and code size is measured. At the end of the study, the author concludes that when the code size is not critical, design pattern usage improves software reusability, extensibility and understandability more than any other method applied.

Another important research is reported in [27]. Software, which is used in ASELSAN INC. as a real-time application, is analyzed in the scope of this study. This software, named as routing software, accepts connections from other applications and routs the given message in accordance with the routing algorithm. For this study, it is redesigned by using Reactor, Acceptor Connector, Forwarder Receiver, Smart Pointer and Command Processor Patterns. At each step, one or two patterns are implemented and measurements taken before and after each step are compared. To make this comparison, twelve OO software maintainability metrics are used and detailed descriptions are given. The relationship between the metrics and their effects on maintainability are also discussed. At the end of the measurement, the author concludes that Reactor, Acceptor Connector, Forwarder Receiver Patterns improve most of the maintainability metrics while Smart Pointer and Command Processor Patterns improve all of them.

These studies and experience reports show that design patterns provide a well-structured way to satisfy design constraints. On the other hand, researchers have another common point on this issue that learning design patterns and using them effectively are difficult. [19] states that using design patterns

is helpful but it is also a hard process. Choosing the right pattern is the first step and implementation process is the second one whose difficulty level depends on the ability of the programmer. Another contribution to this idea can be found in [4] and [8]. They agree that writing good patterns is challenging and time-consuming since design patterns are quite numerous and difficult to learn and implement. For all that, author in [8] claims that the process for using design patterns uniformly can be too long for a company, but it is required.

2.3 Software Quality Metrics

Evaluation of software in a validated and widely accepted manner is important in terms of quality. Addressing this issue, ISO/IEC 9126 was published in 1991 and a revision was issued in 2001 [30]. In the first part of the revised version, software quality characteristics are given. This part of the standard includes two sections, namely internal and external quality part and quality in use part. First part introduces six main characteristics called as functionality, reliability, usability, efficiency, maintainability, and portability. Internal metrics can be applied to non-executable products while the external metrics are applicable to running software. Second part has four quality in use characteristics namely effectiveness, productivity, safety, and satisfaction. Some of these main categories also have subcategories. One can follow the sub characteristic to achieve the necessary quality requirements.

This international publication provides a framework to specify quality requirements of software and supply a clear definition of the user's requirements that can change during the development phase. On the other hand, according to the [2], ISO/IEC 9126-1 has some weaknesses. In this study, 158 final year computer science and engineering students were instructed to design two related modules of a realistic network project management tool. They were given the information about the system explained in UML diagram and conceptual data. They also had unified set of ISO/IEC 9126 quality metrics to use the evaluation of the software developed. At the end of the study, the students find the concept of usability is difficult to understand and the standard itself difficult to interpret since some of the definitions are ambiguous. With a detailed analysis, the authors conclude that some metrics are not well defined and some of them are overlapping. Some important concepts like validity and modularity are not included. Some metrics like functional understandability cannot be measured by the developers. In summary, the authors claim that ISO/IEC 9126 Quality Standard must be improved in order to achieve its purpose.

To correct these deficiencies, this standardization has been revised in 2010 and re-published with the name ISO/IEC FDIS 25010:2010 [31]. In this version, new categories are added, some of the categories are given more accurate name, and the scope of the standard is extended to computer system rather than software specific definitions. This standard includes two models as quality in use model and product quality model. Quality in use part is categorized into five characteristics as effectiveness, efficiency, satisfaction, freedom from risk, and context coverage, while product quality model has eight categories: Functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability, and portability. Definitions of all these categories are given and some of them are subcategorized in the catalog.

Quality of the software can be measured in terms of this standardization. On the other hand, keeping all quality metrics at a high level is difficult. The main requirements differ from system to system. [30] states that as a first step, one needs to determine the external quality and quality in use metrics to be met by the end product. Then, the internal metrics should be specified to achieve this goal. While doing this, a range for the satisfaction of the requirement should be determined and during the internal process, measurements should be done accordingly. To make a valid comparison, measurements

should be objective, empirical using a valid scale, and producible. These statements show that satisfying all these quality metrics is not necessary. Determining the system needs can be thought as main concern of the developers.

According to [27], maintenance is one of the most important but challenging issues for real time software. The author analyzed the effects of design patterns on maintainability in her study. On her pattern cluster, she claims that design patterns improve the software maintainability. On the other hand, performance is the most critical issue for real-time software. However, performance criterion is not among the aims of design patterns. On the contrary, design patterns have their own overhead. Clearly, there is a need to show that while gaining the maintainability, performance efficiency is not placed at a risk.

To investigate the effects of some design patterns on real-time software, a research was conducted. [3] claimed that when the design patterns are used in a correct manner, the decrease in performance due to OO design can be reduced and become acceptable. In order to prove his claim, the author chooses State, Strategy Observer, Smart Pointer, Garbage Collection, Garbage Compactor and Observer Patterns. The reason behind this choice is that these patterns are the ones expected to affect performance of real time software. After determining the patterns, he converts pure C codes to C++ in order to see the effect of OO programming on real time performance. Then, same implementation is done by using design patterns. Performances of these software implementations are examined in terms of execution time, memory consumption and memory fragmentation. At the end of the study, the author proves his claim that, using design patterns effectively decreases the loss of performance due to OO programming coming with an overhead.

In the light of [3], [10], and [27], one concludes that design patterns increase the maintainability of software and if they are used in correct manner, performance degradation is ignorable. On the other hand, all these investigations have been conducted on OO programming by using OO metrics. In the present study, the hypothesis is that design patterns can be implemented in a non-OO way, they increase the maintainability, and the effect on software performance is acceptable.

2.3.1 Maintainability

A large set of metrics used to determine the maintenance level of software can be found in [27]. When the metrics are categorized into OO and procedural way, a narrow set is gained for procedural languages. Line of code (LOC), cyclomatic complexity (CC), and knot metric are evaluated among them [28]. All these metrics can be used by analyzing their effect on maintainability one by one. Another way to determine the relative maintainability of software is combining multiple metrics into a single indicator. On this subject, the most widely used calculation was developed by Oman and Hagemeister in 1991. They developed a set of polynomial metrics at University of Idaho and validated in the Hewlett-Packard Company. To measure the maintainability of software, they combined different metrics and introduced a calculation. In 1994, this calculation was re-analyzed and final version was formulated [9]. Although different variants of it have been introduced and some of the factors have been argued over the years, this language independent calculation has been applied successfully to lots of software systems [28].

According to [9], maintainability index (MI) of software can be calculated by using the following formula:

$$MI = 171 - 5.2 * \ln(HV) - 0.23 * CC - 16.2 * \ln(LOC) + 50.0 * \sin(\sqrt{2.46 * COM})$$

MI consists of LOC, CC, Halstead volume (HV), and optionally comment on module (COM) metrics.

- *LOC*, can be described as the average number of lines of executable code in the software program or module being measured.

- *CC* metric was proposed by McCabe [20]. In his article, complexity calculation was given as:

  $V(G) = E - N + P$

  E : Number of edges in a graph

  N : Number of nodes (vertices) in a graph

  P : Number of connected components

  In software engineering, CC metric is used to count the number of linearly independent paths through the source code. To illustrate, if the source code does not have a decision point such as IF statement, the complexity is 1. It means that there is only one single path through the program. If the code has a single IF statement with a single condition, there will be two different paths to follow. One path is for IF statement is true and one path for IF statement is false. Then the CC will be 2.

- *HV* measurements were introduced by Halstead [15]. The calculations of several measures were given in his book. The volume calculation is:

  $V = N * \log(n)$

  N : Program length (N1 + N2)

  - **N1 :** The total number of operators
  - **N2 :** The total number of operands

  n : Program vocabulary (n1 + n2)

  - **n1 :** The number of distinct operators
  - **n2 :** The number of distinct operands

- *COM*, can be described as the average percent of lines of comments per module. Some implementations omit this variable in the formula.

2.3.2 Performance Efficiency

Performance efficiency is described as performance of software relative to the amount of resources and can be divided into three categories [31].

- *Time behavior*, represents the response time of program to the given task.

- *Resource utilization,* shows the usage of resources while performing the given task.

- *Capacity,* indicates the level of maximum limits that meet the system requirements.

The reason why writing code for embedded systems is more difficult than that of for a general-purpose computer system is its strictly determined design constraints. Embedded software is surrounded by a hardware which is not extendable. Software is adjusted according to integrated hardware constraints. Therefore, the main purpose of the developer is minimizing the use of hardware resources such as memory or power [12]. Due to this reason, extra memory used for design pattern implementation will be analyzed as the resource utilization and capacity usage.

In order to evaluate whether the software can meet all time constraints, the time spent to perform a given task is needed to be observed. [16] states that most crucial timing measurement for real-time systems is A-B Timing, since it allows developers to verify whether the timing objectives for a piece of code are being satisfied. A-B Timing, defined as the time spent between the two specific points of the code, will be used in this study to show that time behavior of the software does not affected severely.

2.4 Design Patterns and Non-OO Programming

[22] has the following statement: "Least Privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily this principle limits the damage that can result from an accident or error"

Many fields with the inclusion of computer science take into account this principle, known as the principle of least privilege, in the early phase of development. According to this idea, every module must be able to access only information and resources that are mandatory for its legitimate purpose. It forces giving a user account only those concessions which are essential to that user's work.

In software world, two principles, called "Low coupling" and "High cohesion", are strongly in support of this privilege. The terms coupling and cohesion were proposed by [26] for structured programming, yet the idea was applied to OO programming too. According to [26], *coupling* can be described as the measure of the strength of association from one module to another. High coupling results in strong dependency between two modules. Changing or correcting one of them ends up with alteration of the other. In addition, reusability of the module requires taking the other parts together. Therefore, reducing the coupling is important in terms of code complexity and reusability.

Cohesion, on the other hand, shows the connectivity among the elements of single module. Low cohesion is the signal of combining unrelated parts. It makes hard to comprehend and maintain the code. Reusable parts may not be reused due to the unnecessary dependency. Due to these reasons, the responsibility of the module cannot be determined easily.

Clearly, low coupling and high cohesion serve for maintainability of the code by holding the related parts together as a single module and separating the modules with necessary interfaces. This explains the reason why design patterns aim to provide low coupling and high cohesion.

A closer look reveals that the idea behind design patterns were borrowed from structured programming. The difference between the OO and non-OO programming is their way of thinking, not their goals. At this point, some explanations can clear the difference between their programming points of view.

Structured-procedural programming based on two different aspects, data structures and functions. Functions are the piece of code that perform an action and return its result by a function call. Data structures contain information needed by those functions. Any data structures can be served to any functions. They are not necessarily bounded to each other. This type of programming fully concentrates on procedures. It follows the sequence of actions in a determined flow chart.

On the other hand, OO programming can be described as collection of loosely connected agents, each of which is responsible for a specific task. Every single module combines both data and behaviors. It is based on the orthogonal paradigm. This means that one concept does not either imply or exclude the other. The support of OO programming to the least privilege principle cannot be ignored. However, this does not mean that structured programming is not able to do the same. Herein, the important point is using the language effectively. That is what the design patterns try to achieve.

CHAPTER 3

THE PROPOSED MODEL

In this study, some OO programming principles are implemented to structured programming in order to realize the design patterns. To describe this approach, this chapter is divided into five main parts. In the first part, implementations of some OO concepts in C are given. The following sections contain the proposed model for four design pattern realizations, namely, First Class ADT, Chain of Responsibility, Strategy, and Decorator.

3.1 Important Aspects of OO Programming and Their Realization in C

An OO program is structured as a community of interacting agents, called objects. An *object* can be defined as an entity in a program which includes both data and behaviors. *Class* is the template describing the details of an object. It is the repository for the states and behaviors associated with the object [13].

Object states and functions, that can manipulate these states, are kept together. They are encapsulated so that the only way to change the internal state is calling the operations. Direct access is not proposed to the other agents [14]. This is called as *encapsulation* and mostly achieved by information hiding, which means process of hiding all the secrets of an object that do not contribute to its essential characteristics. By this way, it plays an important role in terms of realization of least privilege principle.

Relations among the encapsulated objects are arranged by using association, aggregation or composition. *Association* is the weakest relationship and means that one object uses another at some time. *Aggregation* and *composition* are stronger than association and used to describe "part-of" relation. In composition, whole and part have to share the same life-cycle, while aggregation does not force this.

Another important concept is the relationships between the classes. *Inheritance* provides creating more specialized classes by using existing ones. The new class is called *subclass*, *child class* or *derived class* while the existing one is referred as *super class*, *parent class* or *base class*. When a derived class is created, it takes all attributes and definition of operations related to its base class. Then, it can add new attributes and operations. Inheritance is one of the strong features of OO programming. Generic features are implemented one time in base class and reused in subclasses. For this reason, inheritance is a form of reuse in which new class absorbs the data and behaviors of existing class and enriches it by adding new capabilities [11].

The behaviors in super class can be overwritten by subclasses. A request can be handled by different objects that have the correct interface. This means that, same message can have many forms of results. This type of diversity in OO programming is called as *polymorphism* [11].

According to [14], the choice of language is very determining for difficulty of design patterns' implementation process. They noted that, encapsulation, inheritance and polymorphism in itself are design patterns for C, since the language does not support these aspects inherently. That is to say, these three concepts must be handled first to achieve the main purpose of this study. For this reason, following subsections focus to explain implementation method of these concepts.

3.1.1 Class Representation with Some Encapsulation

Information hiding is generally referred as controlling the visibility. OO language has its own feature that a variable can be declared as private, protected, or public. Private variables cannot be accessed directly. Methods have to be presented to get or set these variables. Language protects the variables from the interruption of the other classes or clients. Since C does not force this, information hiding is the issue that deserves extra attention for this language.

In C, to provide information hiding, the public information is put into the header file and local ones are preserved in source files. All parameters of the structure presented into the header file can be accessed directly by the users including this header file. If a variable in one source file is needed in another source file, "extern" keyword is used [33].

Observing the class representation of both C and C++ together can clear the difference between the languages. Classes are thought to be structures holding the operations [12]. A simple way to implement a class in C is declaring the public variables and operations into a header file and keeping the private ones in source files. In order to implement the class shown in Listing 3.1, Listing 3.2 is generally used.

Listing 3.1 Pseudo Code for Class Representation in C++

```
class SampleClass
{
    private : int variable_1;
              int variable_2;

    Public :
         SampleClass (int variable_1, int variable_2);
         ~SampleClass(void);

         void SampleClassOperation();
}
```

Listing 3.2 Pseudo Code for Class Representation in C

```
/* SampleClass.h */

struct SampleClass
{
    int variable_1;
    int variable_2;
}

void SampleClassOperation(struct SampleClass);
```

By this way, files not including SampleClass.h can see neither the structure nor operation. By including the header file, user can only create the structure or call the operation without knowing implementation details of SampleClassOperation(). However, encapsulation means not only information hiding, but also binding data and functions. To provide this, pointers to functions are used [12]. Listing 3.3 shows encapsulated class in C. In this case, the operation is bounded to structure. Even if the header is included, calling the operation is not possible without its structure.

Listing 3.4 illustrates an example for the implementation of the operations given in header file. It is important to underline that, in C++, class members can be accessed implicitly through the "this" pointer. However, this kind of access is not possible for C. Therefore, operations need to take the calling object as an argument.

Listing 3.3 Pseudo Code for Class Representation with Some Encapsulation - Header File

```
/* SampleClass.h */
typedef struct SampleClass* SampleClass_Ptr;
typedef void (*ClassMethod)(SampleClass_Ptr spSampleClass);

struct SampleClass
{
    int variable_1;
    int variable_2;
    ClassMethod SampleClassOperation;
}

/* constructor */
SampleClass_Ptr SampleClass_New(int variable_1, int variable_2);
/* destructor */
void SampleClass_Delete(SampleClass_Ptr);
```

Listing 3.4 Pseudo Code for Class Representation with Some Encapsulation - Source File

```
/* SampleClass.c */

#include "SampleClass.h"

void SampleClassOperation(SampleClass_Ptr spSampleClass)
{
    /* perform the operation */
}
/* constructor */
SampleClass_Ptr SampleClass_New(int variable_1, int variable_2)
{
    SampleClass_Ptr spSampleClass =
         (SampleClass_Ptr)malloc(sizeof(struct SampleClass));
    spSampleClass->variable_1 = variable_1;
    spSampleClass->variable_2 = variable_2;
    spSampleClass->SampleClassOperation = SampleClassOperation;
    return spSampleClass;
}
/* destructor */
void SampleClass_Delete(SampleClass_Ptr spSampleClass)
{
    free(spSampleClass);
}
```

As can be seen from Listing 3.4, SampleClassOperation() cannot be seen from outside of the source file. When the constructor is called, the operation is determined. The only way to use this operation is calling the constructor of structure, like in C++ case. After creating the structure, the way to call its operation is shown in Listing 3.5.

Listing 3.5 Pseudo Code for Class Representation with Some Encapsulation - Class Usage

```
/* Client.c */

#include "SampleClass.h"

    ...
    ...
/* create a pointer to the struct */
SampleClass_Ptr spSampleClass = SampleClass_New(a, b);
/* call its operation */
spSampleClass->SampleClassOperation(spSampleClass);
    ...
    ...
```

Main drawback of this representation is that it reveals the structure. This makes all the attributes of class public unlike C++ representation. However, making attributes private is not always required. This can sacrifice the performance if it is used for every structure. When it is necessary, First Class ADT Pattern can be used. Hiding the structure to the source file and giving the user only a reference is what this pattern does. Although the details are not shared here, it must be underlined that this drawback can be removed by using the information given in Section 3.2.

3.1.2 Inheritance and Polymorphism

The most popular drawing among the UML diagram of design patterns is shown in Figure 3.1.

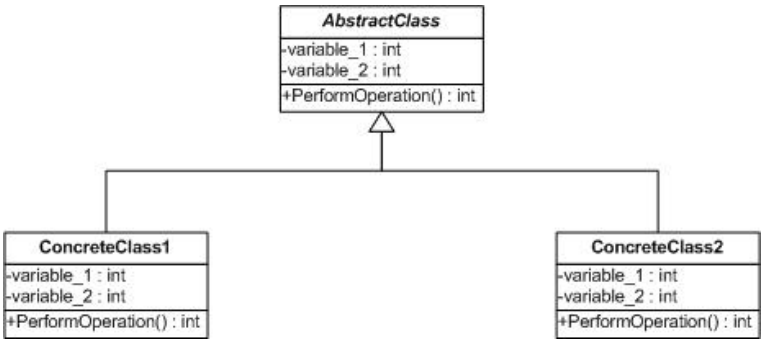

Figure 3.1: Class Diagram for Inheritance

In this representation, polymorphism is provided by allowing the subclasses to overwrite the virtual operation. Here, the idea is letting the subclasses define which operation will be performed at run-time. To achieve this goal, an interface is presented to the users and they are allowed to create a subclass and to call PerformOperation() function. By this way, implementation details of operation are kept from the user. Therefore, changing implementation of one subclass or adding new subclasses implementing the same interface can affect neither other subclasses nor user.

Definition of class and constructors of concrete classes are given in AbstractClass.h file like shown in Listing 3.6. Since all the concrete classes share the same interface and there is no added state, one structure and one destructor will be sufficient here not to make unnecessary code duplication.

Listing 3.6 Pseudo Code for Inheritance without Additional States - AbstractClass.h

```
typedef struct AbstractClass* AbstractClass_Ptr;
typedef void(*VirtualMethod)(AbstractClass_Ptr);

struct AbstractClass
{
    int variable_1;
    int variable_2;
    VirtualMethod PerformOperation;
};

/* constructors */
AbstractClass_Ptr AbstractClass_New(int variable_1, int variable_2);
AbstractClass_Ptr ConcreteClass1_New(int variable_1, int variable_2);
AbstractClass_Ptr ConcreteClass2_New(int variable_1, int variable_2);

/* destructor */
void Class_Delete(AbstractClass_Ptr);
```

Listing 3.7 shows the implementation of abstract class defined in the header file. As can be seen, AbstractClass_New() operation does not overwrite the virtual method. The method is still null since it is the responsibility of concrete classes.

Listing 3.7 Pseudo Code for Inheritance without Additional States - AbstractClass.c

```
#include "AbstractClass.h"

/* constructor */
AbstractClass_Ptr AbstractClass_New(int variable_1, int variable_2)
{
    AbstractClass_Ptr spAbstractClass =
         (AbstractClass_Ptr)malloc(sizeof(struct AbstractClass));
    spAbstractClass->variable_1 = variable_1;
    spAbstractClass->variable_2 = variable_2;
    return spAbstractClass;
}

/* destructor */
void Class_Delete(AbstractClass_Ptr spAbstractClass)
{
    free(spAbstractClass);
}
```

Since concrete classes do not add new attributes or operations for this case, there is no need to write three different structures. One structure, carrying the necessary attributes and PerformOperation() method, is sufficient for both abstract and concrete classes. Like shown in Listing 3.8 and 3.9, each subclass only overwrites the method to provide polymorphism when its constructor is called.

Listing 3.8 Pseudo Code for Inheritance without Additional States - ConcreteClass1.c

```
#include "AbstractClass.h"

void ConcreteClass1_Operation(AbstractClass_Ptr a_AbstractClass)
{
    /* do something special to ConcreteClass1 */
}

/* constructor */
AbstractClass_Ptr ConcreteClass1_New(int variable_1, int variable_2)
{
    AbstractClass_Ptr spAbstractClass = AbstractClass_New(variable_1, variable_2);
    spAbstractClass->PerformOperation = ConcreteClass1_Operation;
    return spAbstractClass;
}
```

Listing 3.9 Pseudo Code for Inheritance without Additional States - ConcreteClass2.c

```
#include "AbstractClass.h"

void ConcreteClass2_Operation(AbstractClass_Ptr a_AbstractClass)
{
    /* do something special to ConcreteClass2 */
}

/* constructor */
AbstractClass_Ptr ConcreteClass2_New(int variable_1, int variable_2)
{
    AbstractClass_Ptr spAbstractClass = AbstractClass_New(variable_1, variable_2);
    spAbstractClass->PerformOperation = ConcreteClass2_Operation;
    return spAbstractClass;
}
```

Listing 3.10 shows an example for usage of this class. While creating the class, concrete class is chosen. Virtual operation is overwritten by selected class. When the PerformOperation() is invoked, command executed according to this class. Diversity between the classes sharing the same interface is achieved by this way.

Listing 3.10 Pseudo Code for Inheritance without Additional States - Client.c

```
#include "AbstractClass.h"

int main()
{
    ...
    /* choose the concrete class */
    AbstractClass_Ptr spOperation = ConcreteClass2_New(a,b);
    /* perform the operation */
    spOperation->PerformOperation(spOperation);
     /* delete the object */
    Class_Delete(spOperation);
    ...
}
```

On the other hand, concrete classes can add new attributes or new behaviors in addition to the abstract methods and common attributes. In this case, all of the concrete classes have to create their own structures to add some new states or responsibilities. One way to solve this inheritance problem is holding a reference in the subclasses and calling its functions when needed. For [12], inheritance can be done by using Figure 3.2.

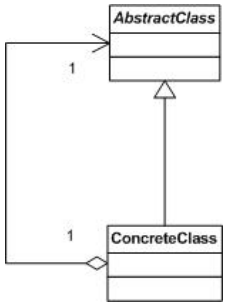

Figure 3.2: Class Diagram for Possible Inheritance Method

Implementation of this class representation can be performed by using the simple example given below. Listing 3.11 shows a concrete class, adding additional states of the existing class, and Listing 3.12 illustrates the usage of this class

Listing 3.11 Pseudo Code for Possible Inheritance Method - ConcreteClass.c

```
#include "AbstractClass.h"

struct ConcreteClass
{
    AbstractClass* spAbstractClass;
    int variable_3;
};
```

Listing 3.12 Pseudo Code for Possible Inheritance Method - Client.c

```
#include "ConcreteClass.h"

    ...
    ...
struct ConcreteClass* spConcreteClass;
spConcreteClass->spAbstractClass->PerformOperation(spConcreteClass->spAbstractClass)
    ...
    ...
```

As can be seen from the listings, clients have to know the internal structure of the concrete classes since PerformOperation() cannot be called directly anymore. Listing 3.12 clearly shows that with the inclusion of "ConcreteClass.h" file, low coupling principle is violated. Per contra, in OO implementation, user calls just PerformOperation() without knowing the details of subclasses. Due to these reasons, this method is not preferred in this study.

Another way to implement the inheritance is copying the representation of abstract class and adding components and behaviors at the end [24]. Since the beginning of a subclass is the same with that of its super class, a pointer to subclass object can be used as a pointer to super class object with up-casting. Figure 3.3 explains this idea and Listing 3.13, 3.14, and 3.15 shows a simple example.

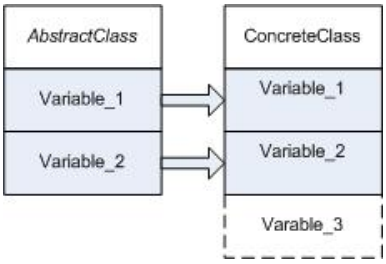

Figure 3.3: Representation of Inheritance Method

As can be seen from Listing 3.13, constructor of concrete class takes same arguments with abstract class, but returns a pointer to the concrete class. Therefore, two destructors for two pointers are needed here. Since user is not able to see the ConcreteClass structure, an operation to get variable_3 is proposed in the AbstractClass.h file. It must be underlined that this operation takes a pointer to ConcreteClass structure. In order to use this operation, user needs to call the constructor of concrete class and have a pointer to the structure.

Listing 3.13 Pseudo Code for Inheritance with Additional States - AbstractClass.h

```
typedef struct AbstractClass* AbstractClass_Ptr;
typedef struct ConcreteClass* ConcreteClass_Ptr;
typedef void(*VirtualMethod)(AbstractClass_Ptr spAbstractClass);

struct AbstractClass
{
    int variable_1;
    int variable_2;
    VirtualMethod PerformOperation;
};

/* constructors */
AbstractClass_Ptr AbstractClass_New(int variable_1, int variable_2);
ConcreteClass_Ptr ConcreteClass_New(int variable_1, int variable_2);

/* destructors */
void Class_Delete(AbstractClass_Ptr);
void ConcreteClass_Delete(ConcreteClass_Ptr);

/* added state */
int getvariable_3(ConcreteClass_Ptr);
```

As can be observed from Listing 3.14, concrete class has exactly same variables and operations in the same order with its super class. Extra parameters are listed at the end in order not to have a problem about up-casting.

Listing 3.14 Pseudo Code for Inheritance with Additional States - ConcreteClass.c

```
#include "AbstractClass.h"

struct ConcreteClass
{
    int variable_1;
    int variable_2;
    VirtualMethod Operation;
    int variable_3;
}

void ConcreteClass_Operation(ConcreteClass_Ptr spConcreteClass)
{
    /* do something special to ConcreteClass */
}

/* constructor */
ConcreteClass_Ptr ConcreteClass_New(int variable_1, int variable_2)
{
    ConcreteClass_Ptr spConcreteClass =
         (spConcreteClass)AbstractClass_New(variable_1, variable_2);
    spConcreteClass->PerformOperation = ConcreteClass_Operation;
    return spConcreteClass;
}

/* destructor */
void ConcreteClass_Delete(ConcreteClass_Ptr spConcreteClass)
{
    free(spConcreteClass);
}

/* added state */
int getvariable_3(ConcreteClass_Ptr spConcreteClass)
{
    return spConcreteClass->variable_3;
}
```

Like shown in Listing 3.15, user can call the same method both for abstract and concrete classes. With the help of this method, internal structure of concrete classes is totally hidden from its user. Only necessary information is presented in header file of abstract class.

Listing 3.15 Pseudo Code for Inheritance with Additional States - Client.c

```
#include "AbstractClass.h"

   ...
   ...
AbstractClass_Ptr spConcreteClass = (AbstractClass_Ptr)ConcreteClass_New(a,b);
spConcreteClass->PerformOperation(spConcreteClass);
x = getvariable_3((ConcreteClass_Ptr)spConcreteClass);
   ...
   ...
```

In this section, some important aspects of OO programming, necessary for design patterns, are realized. With this background, four design pattern implementations will be presented in the following sections.

3.2 First Class Abstract Data Type (ADT) Pattern

Data Type can be defined as the set of values and collection of operations working with those values [24]. To illustrate, "int" is a data type whose set contains distinct values between -32767 and 32767. Operations related to this data type are +, -, *, /.

User can define new data types and their operations by using existing ones. An example for this can be found in the below listings. Listing 3.16 represents the definition of the data type and its operations, while Listing 3.17 illustrates the implementation process. Then, Listing 3.18 shows the client code.

Listing 3.16 Pseudo Code for Definition of Data Type

```
/* RectangleDT.h */

typedef struct rectangle Rectangle;
struct rectangle
{
    float width;
    float height;
};
/* operations */
float Area(Rectangle);
float Circumference(Rectangle);
```

Listing 3.17 Pseudo Code for Implementation of Data Type

```
/* RectangleDT.c */

#include "RectangleDT.h"

float Area(Rectangle)
{
    /* return the area of the rectangle */
}
float Circumference(Rectangle)
{
    /* return the circumference of the rectangle */
}
```

Listing 3.18 Pseudo Code for Client of Data Type

```
/* Client.c */
#include "RectangleDT.h"
int main()
{
    float area, circumference;
    Rectangle rec;
    rec.height = 10;
    rec.width = 20;
    area = Area(rec);
    circumference = Circumference(rec);
    ...
    ...
}
```

A closer look reveals that structure is open for users. Struct rectangle is represented through the RectangleDT.h file. Any code including the header file can see the structure. The representation cannot be changed without changing all client programs that can access the representation directly from the interface. As a result, any change in this data type affects its users. Plainly, data type needs some rectifications in order to break off the connection between users and providers.

Hiding the representation in the implementation is possible by using abstract data type. A data type is an *Abstract Data Type*, if its internal representation is not presented to the user. Direct manipulation of the code is not allowed by this way. Operations are provided only through the interface which is a contract between the clients and providers [24].

Clients can only have the pointers rather than whole structure. Pointers permit to conceal the representation of data items and declare only possible manipulations. The reason why this pattern is called as First Class ADT Pattern is that it is the first instance among many that one can create from it.

Consequences of the pattern can be listed as follows:

- This pattern fills the missing point about encapsulation. Especially for libraries, this is very helpful.
- Implementation can change without disturbing the client.
- Program is decomposed into simpler tasks.
- Complexity is hidden from the clients.
- Readability of the code increases.

An example showing the implementation and usage of the pattern can be found in Listing 3.19, 3.20 and 3.21.

Listing 3.19 Pseudo Code for Definition of Abstract Data Type

```
/* RectangleDT.h */

typedef struct rectangle* RectanglePtr;

/* constructor */
RectanglePtr newRectangle(float, float);

/* destructor */
void deleteRectangle(RectanglePtr);

/* set operations */
void setWidth(RectanglePtr, float);
void setHeight(RectanglePtr, float);

/* get operations */
float getWidth(RectanglePtr);
float getHeight(RectanglePtr);

/* other operations */
float Area(RectanglePtr);
float Circumference(RectanglePtr);
```

Listing 3.20 Pseudo Code for Implementation of Abstract Data Type

```
/* RectangleDT.c */

#include "RectangleADT.h"

struct rectangle
{
    float width;
    float height;
};

/* constructor */
RectanglePtr newRectangle(float height, float width)
{
    RectanglePtr spRectangle = (RectanglePtr)malloc(sizeof(struct rectangle));
    spRectangle->height = height;
    spRectangle->width = width;
    return spRectangle;
}

/* destructor */
void deleteRectangle(RectanglePtr spRectangle)
{
    free(spRectangle);
}

/* set operations */
void setWidth(RectanglePtr spRectangle, float width)
{
    spRectangle->width = width;
}
void setHeight(RectanglePtr spRectangle, float height)
{
    spRectangle->height = height;
}

/* get operations */
float getWidth(RectanglePtr spRectangle)
{
    return spRectangle->width;
}
float getHeight(RectanglePtr spRectangle)
{
    return spRectangle->height;
}

/* other operations */
float Area(RectanglePtr spRectangle)
{
    /* returns the area of the rectangle */
}
float Circumference(RectanglePtr spRectangle)
{
    /* returns the circumference of the rectangle */
}
```

Listing 3.21 Pseudo Code for Client of Abstract Data Type

```
/* Client.c */

#include "RectangleADT.h"

int main()
{
    float area, circumference;
    RectanglePtr rec;
    /* create the rec pointer */
    RectanglePtr rec = newRectangle(10,20);
    /* calculate the area */
    area = Area(rec);
    /* calculate the circumference */
    circumference = Circumference(rec);
    ...
    ...
    /* get Heigth and Width values and print */
    printf("Heigth = %f\n",getHeight(rec));
    printf("Width = %f\n",getWidth(rec));
    ...
    ...
    /* change Heigth and Width values */
    setHeight(rec,30);
    setWidth(rec,50);
    /* calculate the new area */
    area = Area(rec);
    /* calculate the new circumference */
    circumference = Circumference(rec);
    ...
    ...
    return 0;
}
```

3.3 Chain of Responsibility Pattern

An agent asks for a request and there are different providers that can help for this request. To illustrate, a mail comes to the mail box. Which folder can take this mail is not known. If it is a spam, it goes to spam-box or if it is a complaint, it goes to complaint-box, etc. Bad way to solve this problem is making everything public and using reference for every object. Long if statement is used and complexity of the code increases. Low coupling principle is violated due to the dependency between sender and receivers.

To solve this problem in a more rational way, [14] proposes Chain of Responsility Pattern. This pattern provides to give more than one object a chance to handle the request without attaching requester and provider. The objects become part of a chain. They can handle the request or pass it to their successors. Request is sent across the chain until it is handled. By this way, senders can send a command without knowing what object will receive and handle it.

Advantages and drawbacks of these patterns can be found in [14]. Herein, only the implementation process regarding C programming language is presented. For this purpose, Figure 3.4 is presented to show the class diagram of the pattern.

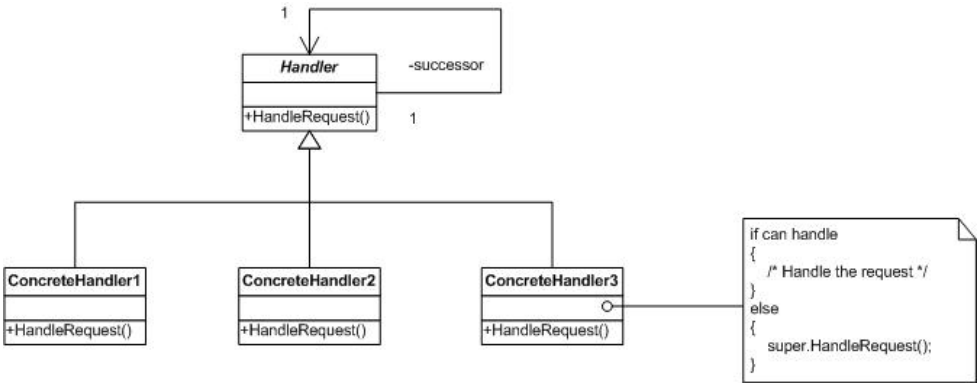

Figure 3.4: Chain of Responsibility Pattern Class Diagram [14]

As can be seen from Figure 3.4, this pattern is nothing but the inheritance with an association relation. Therefore, it can be implemented as stated in the first part of this section. Listing from 3.22 to 3.26 contain the pseudo codes to show the implementation process of the pattern while Listing 3.27 gives an example for possible usage.

As Listing 3.22 shows, Handler structure is created in ChainOfResponsibility.h file and includes a successor to pass the request in case not to handle by itself and a virtual method to be overwritten by concrete class. Since the user only see this interface, declaration of concrete class' constructor is given here. Client can create concrete handler class without knowing the implementation details. All constructors of concrete class return the same abstract type. This is preferred since the concrete classes do not have added state or behavior in this pattern representation. In order not to make code duplication, concrete classes share the same structure with abstract one.

Listing 3.22 Pseudo Code for Chain of Responsibility Pattern - ChainOfResponsibility.h File

```
typedef struct Handler* Handler_Ptr;
typedef void(*VirtualMethod)(Handler_Ptr spHandler);

/* abstract class */
struct Handler
{
    Handler_Ptr Successor;
    VirtualMethod HandleRequest;
};

/* constructors */
Handler_Ptr Handler_New();
Handler_Ptr ConcreteHandler1_New();
Handler_Ptr ConcreteHandler2_New();
Handler_Ptr ConcreteHandler3_New();

/* destructor */
Handler_Delete(Handler_Ptr spHandler);
```

Constructor and destructor implementations of abstract class are given "ChainOfResponsibility.c" file and virtual method leaves as null like shown in Listing 3.23. Each concrete class implements its constructor and overrides the virtual method as indicated in Listings 3.24, 3.25, and 3.26. When a class is created, its virtual method is determined. User can handle the request just by calling the HandleRequest() function. Special function of concrete class is used without knowing the details like shown in Listing 3.27.

Listing 3.23 Pseudo Code for Chain of Responsibility Pattern - ChainOfResponsibility.c File

```
#include "ChainOfResponsibility.h"

/* constructor */
Handler_Ptr Handler_New()
{
    Handler_Ptr spHandler = (Handler_Ptr)malloc(sizeof(struct Handler);
    return spHandler;
}

/* destructor */
Handler_Delete(Handler_Ptr sphandler)
{
    free(sphandler);
}
```

Listing 3.24 Pseudo Code for Chain of Responsibility Pattern - ConcreteHandler1.c File

```
#include "ChainOfResponsibility.h"

/* Virtual function for ConcreteHandler1 */
void ConcreteHandler1_Function(HandlerPtr spHandler)
{
    if (/* can handle */)
    {
        /* Handle */
    }
    else if (handler->Successor != NULL)
    {
        spHandler->Successor->HandleRequest(spHandler->Successor);
    }
}

/* constructor */
Handler_Ptr ConcreteHandler1_New()
{
    Handler_Ptr spConcreteHandler1 = Handler_New();
    spConcreteHandler1->HandleRequest = ConcreteHandler1_Function;
    return spConcreteHandler1;
}
```

Listing 3.25 Pseudo Code for Chain of Responsibility Pattern - ConcreteHandler2.c File

```
/* ConcreteHandler2.c */

#include "ChainOfResponsibility.h"

/* Virtual function for ConcreteHandler2 */
void ConcreteHandler2_Function(HandlerPtr spHandler)
{
    if (/* can handle */)
    {
        /* Handle */
    }
    else if (handler->Successor != NULL)
    {
        spHandler->Successor->HandleRequest(spHandler->Successor);
    }
}

/* constructor */
Handler_Ptr ConcreteHandler2_New()
{
    Handler_Ptr spConcreteHandler2 = Handler_New();
    spConcreteHandler2->HandleRequest = ConcreteHandler2_Function;
    return spConcreteHandler2;
}
```

Listing 3.26 Pseudo Code for Chain of Responsibility Pattern - ConcreteHandler3.c File

```
/* ConcreteHandler3.c */

#include "ChainOfResponsibility.h"

/* Virtual function for ConcreteHandler3 */
void ConcreteHandler3_Function(HandlerPtr spHandler)
{
    if (/* can handle */)
    {
        /* Handle */
    }
    else if (handler->Successor != NULL)
    {
        spHandler->Successor->HandleRequest(spHandler->Successor);
    }
}

/* constructor */
Handler_Ptr ConcreteHandler3_New()
{
    Handler_Ptr spConcreteHandler3 = Handler_New();
    spConcreteHandler3->HandleRequest = ConcreteHandler3_Function;
    return spConcreteHandler3;
}
```

Listing 3.27 Pseudo Code for Chain of Responsibility Pattern - Client.c File

```
/* Client.c */
#include "ChainOfResponsibility.h"
int main()
{
    /* create the part of chain */
    HandlerPtr spHandler1 = (HandlerPtr)ConcreteHandler1_New();
    HandlerPtr spHandler2 = (HandlerPtr)ConcreteHandler2_New();
    HandlerPtr spHandler3 = (HandlerPtr)ConcreteHandler3_New();

    /* set the chain */
    spHandler1->Successor = spHandler2;
    SpHandler2->Successor = spHandler3;

    /* perform the action */
    spHandler1->HandleRequest(spHandler1);

    /* delete the chain */
    Handler_Delete(spHandler1);
    Handler_Delete(spHandler1);
    Handler_Delete(spHandler1);
}
```

3.4 Strategy Pattern

Strategy pattern is used when one request can be performed by different ways. Algorithm can vary independently from the user. Figure 3.5 shows the class diagram of the pattern.

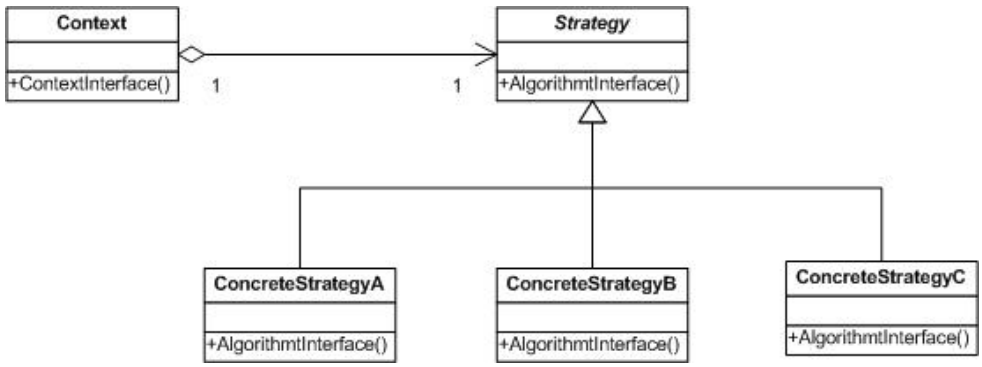

Figure 3.5: Strategy Pattern Class Diagram [14]

In this diagram, Context receives requests from the client and sends them to its Strategy object. Client creates the concrete class, preferred to use, and pass it to the Context. Context contains a reference to concrete strategy so that interaction between the Strategy and client is performed across the context. Strategy provides an interface to the concrete class performing the same task by using different algorithms. ConcreteStrategy implements the algorithm presented in Strategy interface.

Advantages and drawbacks of these pattern can be found in [14]. Herein, only the implementation process regarding C programming language is presented. Like Chain of Responsibility, this pattern can be implemented as stated in the first part of this section. Following the idea emphasized there, this

can be realized easily. Listings from 3.28 to 3.34 contain the pseudo codes to show the implementation process of the pattern while Listing 3.35 gives an example for possible usage.

As can be seen from Listing 3.28, aggregation relation between the Context and Strategy classes is provided by giving a pointer to the Context structure. This pointer shows that Context structure "has-a" Strategy structure in it. Constructor of this class takes a pointer of Strategy structure and sets its Strategy attribute as can be shown in Listing 3.29. By this way, when its ContextInterface() operation is called, it invokes AlgorithmInterface() of its Strategy attribute.

Listing 3.28 Pseudo Code for Strategy Pattern - Context.h File

```
#include "Strategy.h"

typedef struct Context* ContextPtr;
typedef void (*ContextMethod)(ContextPtr spContext);

struct Context
{
    ContextMethod ContextInterface;
    StrategyPtr Strategy; /* aggregation */
};

/* constructor */
struct Context* Context_New(StrategyPtr spStrategy);

/* destructor */
void Context_Delete(ContextPtr context);
```

Listing 3.29 Pseudo Code for Strategy Pattern - Context.c File

```
#include "Context.h"

void ContextInterface(ContextPtr spContext)
{
    /* let the concrete strategy perform the task */
    spContext->Strategy->AlgorithmInterface();
}

/* constructor */
ContextPtr Context_New(StrategyPtr spStrategy)
{
    ContextPtr spContext = (ContextPtr)malloc(sizeof(struct Context));
    spContext->Strategy = spStrategy;
    spContext->ContextInterface = ContextInterface;
    return spContext;
}

/* destructor */
void Context_Delete(ContextPtr spContext)
{
    free(spContext);
}
```

Implementation of the Strategy class and its concrete classes are very similar to that of Chain of Responsibility Pattern since they share the same class diagram with different intent. Abstract and concrete classes use the same structure and different virtual functions provide polymorphism.

Listing 3.30 Pseudo Code for Strategy Pattern - Strategy.h File

```
typedef struct Strategy* StrategyPtr;
typedef void (*VirtualMethod)();

struct Strategy
{
    VirtualMethod AlgorithmInterface;
};

/* constructors */
StrategyPtr Strategy_New();
StrategyPtr ConcreteStrategyA_New();
StrategyPtr ConcreteStrategyB_New();
StrategyPtr ConcreteStrategyC_New();

/* destructor */
void Strategy_Delete(StrategyPtr spStrategy);
```

Listing 3.31 Pseudo Code for Strategy Pattern - Strategy.c File

```
#include "Strategy.h"

/* constructor */
StrategyPtr Strategy_New()
{
    StrategyPtr spStrategy = (StrategyPtr)malloc(sizeof(struct Strategy));
    return spStrategy;
}
/* destructor */
void Strategy_Delete(StrategyPtr spStrategy)
{
   free(spStrategy);
}
```

Listing 3.32 Pseudo Code for Strategy Pattern - ConcreteStrategyA.c File

```
#include "Strategy.h"

/* special method for ConcreteStrategyA */
void AlgorithmA()
{
    /* perform the task by using algortihm A*/
}

/* constructor */
StrategyPtr ConcreteStrategyA_New()
{
    StrategyPtr spConcreteStrategyA = Strategy_New();
    spConcreteStrategyA->AlgorithmInterface = AlgorithmA;
    return spConcreteStrategyA;
}
```

Listing 3.33 Pseudo Code for Strategy Pattern - ConcreteStrategyB.c File

```
#include "Strategy.h"

/* special method for ConcreteStrategyB */
void AlgorithmB()
{
     /* perform the task by using algortihm B*/
}
/* constructor */
StrategyPtr ConcreteStrategyB_New()
{
     StrategyPtr spConcreteStrategyB = Strategy_New();
     spConcreteStrategyB->AlgorithmInterface = AlgorithmB;
     return spConcreteStrategyB;
}
```

Listing 3.34 Pseudo Code for Strategy Pattern - ConcreteStrategyC.c File

```
#include "Strategy.h"

/* special method for ConcreteStrategyC */
void AlgorithmC()
{
    /* perform the task by using algortihm C*/
}
/* constructor */
StrategyPtr ConcreteStrategyC_New()
{
    StrategyPtr spConcreteStrategyC = Strategy_New();
    spConcreteStrategyC->AlgorithmInterface = AlgorithmC;
    return spConcreteStrategyC;
}
```

Listing 3.35 Pseudo Code for Strategy Pattern - Client.c File

```
/* Client.c */
#include "Context.h"
int main()
{
    /* Perform the given task by using ConcreteStrategyA */
    StrategyPtr spConcreteStrategyA = ConcreteStrategyA_New();
    ContextPtr spContext = Context_New(spConcreteStrategyA);
    spContext->ContextInterface(spContext);
    Strategy_Delete(spConcreteStrategyA);

    /* Change the strategy and perform the same task by using ConcreteStrategyC */
    StrategyPtr spConcreteStrategyC = ConcreteStrategyC_New();
    spContext->Strategy = spConcreteStrategyC;
    spContext->ContextInterface(spContext);
    Strategy_Delete(spConcreteStrategyC);

    /* Delete spContext*/
    Context_Delete(spContext);
}
```

3.5 Decorator Pattern

Decorator Pattern is designed to add additional responsibility dynamically to an object. Subclassing would solve the problem when the number of subclasses is acceptable. However, when this number is impractical, it would be hard to handle. Instead of creating a subclass, including entire operations, adding additional responsibilities at run time is the main motivation of this pattern.

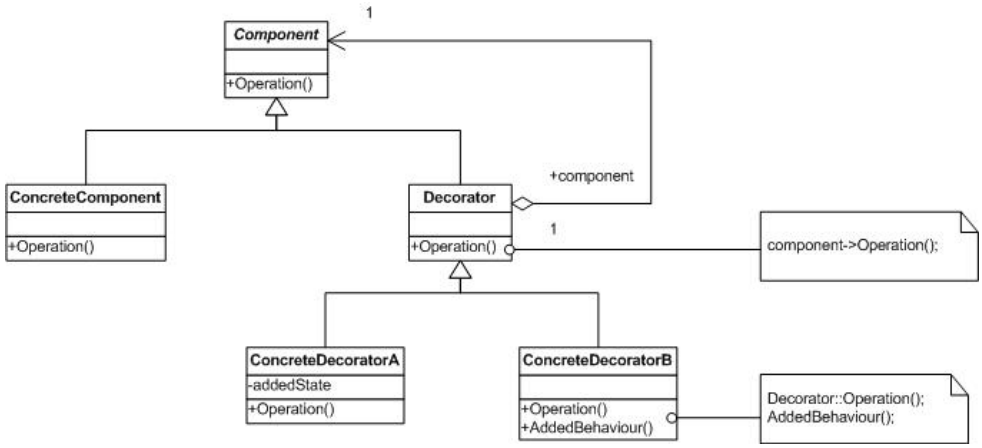

Figure 3.6: Decorator Pattern Class Diagram [14]

Figure 3.6 shows the class diagram of the pattern. In this diagram, Component provides an interface for objects that can have additional responsibilities at run-time. ConcreteComponent implements the default behavior of the class. ConcreteDecorator adds new behaviors or states to default ones and Decorator presents an interface for ConcreteDecorator.

Advantages and drawbacks of this pattern are discussed in [14]. Herein, only the implementation process regarding C programming language is presented. To implement this pattern, necessary information can be found in the first part of this section. In this case, additional responsibilities are given to the concrete classes. Therefore, each has to create their special structure. One common point between them is that they have to connect Component class with an aggregation relation.

Listings from 3.36 to 3.41 contain the pseudo codes to show the implementation process of the pattern while Listing 3.42 gives an example for possible usage. As can be seen from the listings, "inheritance without additional states" method is used to set up the relation between the Component and ConcreteComponent classes. Decorator and ConcreteDecorator classes, on the other hand, require using "inheritance with additional states" method. While implementing this method, Decorator.c file may not be used in order not to make code duplication. Then, the declarations of concrete decorators' constructors and destructors are given in Component.h file.

Listing 3.36 Pseudo Code for Decorator Pattern - Component.h File

```
typedef struct Component* ComponentPtr;
typedef void (*VirtualMethod)(ComponentPtr spComponent);

struct Component
{
    int variable_1;
    int variable_2;
    VirtualMethod Operation;
};

/* constructor */
ComponentPtr Component_New();
ComponentPtr ConcreteComponent_New();

/* destructor */
void Component_Delete(ComponentPtr spComponent);
```

Listing 3.37 Pseudo Code for Decorator Pattern - Component.c File

```
#include "Component.h"

/* constructor */
ComponentPtr Component_New()
{
    ComponentPtr spComponent = (ComponentPtr)malloc(sizeof(struct Component));
    return spComponent;
}

/* destructor */
void Component_Delete(ComponentPtr spComponent)
{
    free(spComponent);
}
```

Listing 3.38 Pseudo Code for Decorator Pattern - ConcreteComponent.c File

```
#include "Component.h"

/* base operation */
void FundamentalOperation(ComponentPtr spComponent)
{
    /* perform fundemantal operation */
}

/* constructor */
ComponentPtr ConcreteComponent_New()
{
    ComponentPtr spComponent = Component_New();
    /* initialization */
    spComponent->variable_1 = 0;
    spComponent->variable_2 = 0;
    spComponent->Operation = FundamentalOperation;
    return spComponent;
}
```

Listing 3.39 Pseudo Code for Decorator Pattern - Decorator.h File

```
#include "Component.h"

typedef struct Decorator* DecoratorPtr;
typedef struct ConcereteDecorator* ConcereteDecoratorPtr;

struct Decorator
{
    int variable_1;
    int variable_2;
    VirtualMethod Operation;
    ComponentPtr spComponent; /* aggregation */
};

/* constructors */
DecoratorPtr Decorator_New();
ConcereteDecoratorPtr ConcereteDecorator_New();

/* destructors */
void Decorator_Delete(DecoratorPtr spDecorator);
void ConcereteDecorator_Delete(ConcereteDecoratorPtr spConcereteDecorator);

/* added state */
int getVariable_3(ConcereteDecoratorPtr spConcereteDecorator);
```

Listing 3.40 Pseudo Code for Decorator Pattern - Decorator.c File

```
#include "Decorator.h"

/* constructor */
DecoratorPtr Decorator_New()
{
    DecoratorPtr spDecorator = (DecoratorPtr)malloc(sizeof(struct Decorator));
    spDecorator->spComponent = ConcreteComponent_New();
    return spDecorator;
}

/* destructor */
void Decorator_Delete(DecoratorPtr spDecorator)
{
    Component_Delete(spDecorator->spComponent);
    free(spDecorator);
}
```

As can be observed from Listing 3.38, Component structure has two variables and they are initialized when the structure is created. If the user calls Operation() method, these attributes are calculated according to the FundamentalOperation() function. On the other hand, ConcreteDecorator has additional one parameter as can be seen from Listing 3.41. It uses the same base operation, but performs extra calculations for variable_3. To do this, fundamental operation, defined in ConcreteComponent.c file, is used with the help of the aggregation relation between the Component and Decorator interfaces.

Listing 3.41 Pseudo Code for Decorator Pattern - ConcereteDecorator.c File

```
#include "Decorator.h"

struct ConcereteDecorator
{
    int variable_1;
    int variable_2;
    VirtualMethod Operation;
    ComponentPtr spComponent; /* aggregation */
    int variable_3; /* added state */
};

/* added behavior */
int AddedBehavior()
{
    /* perform some added operation */
}

/* virtualMethod for ConcreteComponent */
void ConcreteDecoratorOperation(ConcereteDecoratorPtr spConcereteDecorator)
{
    /* base operation */
    spConcereteDecorator->spComponent->Operation((ComponentPtr)spConcereteDecorator);
    /* added behavior */
    spConcereteDecorator->variable_3 = AddedBehavior();
}

/* constructor */
ConcereteDecoratorPtr ConcereteDecorator_New()
{
    ConcereteDecoratorPtr spConcereteDecorator =
         (ConcereteDecoratorPtr)malloc(sizeof(struct ConcereteDecorator));
    spConcereteDecorator->spComponent = (ComponentPtr)ConcreteComponent_New();
    spConcereteDecorator->variable_3 = 0;
    spConcereteDecorator->Operation = (VirtualMethod)ConcreteDecoratorOperation;
    return spConcereteDecorator;
}

/* destructor */
void ConcereteDecorator_Delete(ConcereteDecoratorPtr spConcereteDecorator)
{
    free(spConcereteDecorator);
}

/* return added state */
int getVariable_3(ConcereteDecoratorPtr spConcereteDecorator)
{
    return spConcereteDecorator->variable_3;
}
```

Providing same interface for Component and Decorator classes requires extra attention here. Like shown in Listing 3.41, ConcreteDecoratorOperation() gives spConcreteDecorator as an input to the base function so that the results are written into spConcreteDecorator, not (spConcreteDecorator -> spComponent), although it uses the (spConcreteDecorator->spComponent)'s operation actually. By this way, user does not need to know the internal structure of concrete classes, as can be seen from Listing 3.42

Listing 3.42 Pseudo Code for Decorator Pattern - Client.c File

```
/* Client.c */
#include "Decorator.h"
int main()
{
    int var1, var2, var3;

    /* Perform the given task by using ConcreteDecorator */
    DecoratorPtr spConcereteDecorator = (DecoratorPtr)ConcereteDecorator_New();
    spConcereteDecorator->Operation((ComponentPtr)spConcereteDecorator);

    /* get the results */
    var1 = spConcereteDecorator->variable_1;
    var2 = spConcereteDecorator->variable_2;
    var3 = getVariable_3((ConcereteDecoratorPtr)spConcereteDecorator);
    ConcereteDecorator_Delete(spConcereteDecorator);
       ...
       ...
    return 0;
}
```

CHAPTER 4

EXPERIMENTAL WORK

The purpose of the study is to show implementation of some design patterns in C programming language and to investigate the effects of these design patterns on performance and maintainability concern of the software. In accordance with the purpose of this study, the following questions are investigated:

1. How can design patterns be implemented by using C programming language?
2. Does the usage of design patterns in C programming language increase the software maintainability?
3. Does the usage of design patterns in C programming language decrease the software performance efficiency critically?

Answer for the first question was presented in Chapter 3. To give the explicit answers for the rest of the research questions, a quantitative research model is performed. This study focuses on some real-time embedded software codes prepared by ASELSAN INC. The results are expected to be generalizable to all C codes.

The research design, procedures, data collection, and analysis techniques used in the study will be explained in this section. After the descriptions of the software will be given, methodology of the experiments will be stated. In that section, validity and reliability issue of the research will be underlined. Then, in subsequent section, the instruments used in the study will be stated. Finally, experimental process and results will be discussed in the Experimental Process section.

4.1 Description of the Software

In computer science, a *library* is a compilation of an application performing a specific task. It has a well-defined interface so that can be invoked by independent programs. Main difference between a piece of code and a library is that libraries are organized in order to be reused by different programs. The user only knows the interface, not the details of the behavior [37].

ASELSAN INC has its own libraries. Each library is given a responsibility needed for more than one project. Project can be thought as the collection of libraries and control tasks connecting the libraries and sharing the responsibilities.

In the scope of this thesis, a study on two libraries of ASELSAN INC, written in C, is conducted. Due

to the company secrecy, the details of the libraries will not be given here. The applications used for this purpose will be referred as LibraryA and LibraryB.

4.2 Experimental Methodology

Applying the method given in Chapter 3, majority of design patterns can be implemented easily. Four of them are given a place in this study. The reason behind this choice will be stated here. Then the steps followed during the experimental work will be shared.

Of the first consideration, libraries need well-defined interfaces independent from their behaviors. Information special to the library must be hidden from users. According to the background given in Chapter 3, this need addresses the First Class ADT Pattern. This pattern improves information hiding and helps to create interfaces independent from their users. Structures specific to the library can be hidden from the user with the help of this pattern. Therefore, in order to create reusable libraries, using First Class ADT Pattern for both applications is not coincident.

Another common necessity of the libraries can be thought as solving the additional responsibilities problem. To illustrate, Project X expects from two variables after the execution of the library, while Project Y waits for three variables. In ASELSAN INC, to solve this problem, biggest set of the variables are calculated without taking into account the requirements of the users. It is true in one aspect that this makes the library independent from the user and avoids from putting if statements for each project. On the other hand, unnecessary calculations can be performed for some projects. In this study, Decorator Pattern is proposed as a solution to this issue. As stated in Chapter 3, this pattern is used to add extra responsibilities at run-time. That is to say, instead of preparing an output structure with biggest set of variables, smallest set with essential variables for all projects, can be used. Then, with the help of Decorator Pattern, additional states or behaviors can be added at run-time.

Chain of Responsibility and Strategy Patterns are very helpful in terms of reducing the conditional statements. Degradation of code complexity can be achieved by using these patterns in accordance with their purposes stated in Chapter 3.

Among the libraries in ASELSAN INC, two are selected by taking into account their needs to design patterns. Same patterns are preferred to be used so that second library can echo the confirmation of the claim.

In order to build the assertion of the study on firm ground, exactly same procedures are used on two libraries. First of all, existing version of the codes are analyzed in terms of MI, A-B timing, and memory utilization. Then, each library is subjected to four design patterns one by one. For each implementation, original code is taken as a base. MI, A-B timing, and memory utilization of the pattern applied version of the codes are analyzed again. As a final step, all patterns are applied together to observe the effects of design patterns on the software under investigation.

Validity and Reliability Issues of the Study In this study, a quantitative research model is used. Therefore, validity and reliability issue of a quantitative research is taken into account.

For this type of study, internal validity is the ability to control other variables [40]. It means that observations are actually the result of the experimental treatment. In this study, internal validity is improved by providing that same procedure is followed both before and after design pattern implementations. There is no difference between the instruments and experimental set up for two cases. The difference

between the metrics gives only the effect of design patterns.

Together with its strong internal validity, external validity of the study is also high. External validity refers to generalizatibility of the experiement [40]. The procedure followed in this study is not machine dependent. The same results can be observed by using different real-time operating systems, or code can be developed with different development tools. Moreover, changing the static analysis tool would not create any difference between the results.

For the MI calculation and memory usage, data is the code itself. These are the measurements that are not hardware dependent. The only measurement that is affected by environmental change is A-B timing. Therefore, this analysis deserves extra attention for this study. In order to minimize the chances of machine malfunction and maintain validity and reliability of the study, the same measurements are taken many times until the confident stopping rule is satisfied [36]. As stopping rule, achieving 95% confidence level with ±0.1 confidence intervals is used. Then, average of the results is noted as the performance of the software.

For the performance measurement, the data given to the code is selected randomly among the test vectors. These test vectors are prepared by using data collected on real platform and used for the library test in ASELSAN INC. That is to say, random sampling is used for this study in order to show that the results do not depend on the data supplied to the code. Moreover, the same data is used both before and after implementation phase to control the effect of data on the performance measurements.

Another important point that must be stated here is that since the codes are library applications, there are different projects using these codes. Projects set the input parameters of the library in accordance with their system specifications. It means that, with the same data set, each project can get different results from the library due to the constraints of the projects. In this study, same data sets are run for three different projects. For Chain of Responsibility, Strategy and First Class ADT Patterns, changing the limit of the library is nothing but feeding the algorithm with different data sets. However, Decorator Pattern changes its behavior according to the project, since projects can add additional responsibilities at run-time. Therefore, in this pattern representation and the case for all patterns applied, graphs and tables are divided into three sub parts. By this way, the effects of Decorator Pattern will be elaborated.

Despite all these precautions, the validity of this study will still be limited by the reliability of the instruments. Instruments used in the study will be described in following section.

4.3 Instruments

In this study, code is developed on Wind River Workbench development tool [34], compiled by using Vxworks operating system and run on the CPU-7448 processor card developed by ASELSAN INC. By using the run-time analysis tools of Wind River Workbench, information is collected for performance analysis. The obtained results are analyzed by using Mathworks Matlab. Then, maintainability analysis of developed code is done by using Understand for C/C++ tool. The general description and features of these tools will be given in this section.

VxWorks is a real time operating system which is developed for use in embedded systems. *Workbench* is an integrated development environment (IDE) developed by WindRiver Company and facilitates managing and building projects, running, debugging, and monitoring VxWorks applications [39]. Workbench consists of the following components: [34]

- ✓ Eclipse
  - Eclipse platform 3.3.1
  - C/C++ Development Tooling
  - Target Management/Remote System Explorer
  - Device Debugging
- ✓ Project System
- ✓ Build System
- ✓ Index-based global text search-and-replace
- ✓ Wind River compilers
  - Wind River Compiler for VxWorks
  - Wind River GNU Compiler
- ✓ Debugger
  - Target debugging agents for Wind River Linux
  - Target debugging agent for VxWorks
- ✓ Shell environments
- ✓ Simulators
  - VxWorks Simulator
  - QEMU open source emulator
- ✓ Configuration tools
  - VxWorks Kernel Configurator
  - Linux Kernel and User Space Configuration Tools
  - Linux File System Configurator
- ✓ Run-time visualization and analysis tools
  - System Viewer
  - Performance Profiler
  - Memory Analyzer
  - Data Monitor
  - Code Coverage Analyzer
  - Function Tracer

In this study, System Viewer and Memory Analyzer tools are used. In order to measure the A-B timing, wvEvent() function of VxWorks library is used and the difference between the two wvEvent() function calls is observed on Wind River System Viewer Event Graph. The data is collected by using CPU7448 processor card with 166.66 MHz clock rate. The results, gathered by System Viewer, is plotted by

using *Mathworks Matlab* which is a high level language and an interactive development environment (IDE) for programming, numeric calculations and visualizations [38].

Memory usage is observed by using Memory Analyzer tool. For all design pattern implementation given in Chapter 3, dynamic memory allocation is used. On the other hand, dynamic memory allocation is not preferred since it can fragment the memory. In ASELSAN INC, as a principle, memory needed during the execution of the library is allocated by the user at one time and given to the library. In order to hold by this principle, memory, feeding the library at the outset, is used for the design pattern implementation. This ensures that memory fragmentation is not allowed during the experimental process. Therefore, in the scope of memory utilization, only maximum memory need during the execution of the libraries is considered.

For MI calculation, Understand for C/C++ tool is used. *Understand* is a static code analysis software tool produced by SciTools. It is used for reverse engineering, documentation and metrics measurement purposes. *Understand for C/C++* is an IDE designed to help, maintain and understand large amounts of C and C++ source code [32]. For this study, basic features of the tool are not sufficient, so a plug-in is used to measure MI of the examined codes [35].

4.4 Experimental Process

In this study, experimental process focuses on two software quality metrics, maintainability and performance efficiency. For performance efficiency, software is analyzed in terms of both time behavior and resource utilization.

Explanation of MI Tables To determine maintainability level of the software, following MI calculation, proposed in [9], is used:

$$MI = 171 - 5.2 * \ln(HV) - 0.23 * CC - 16.2 * \ln(LOC) + 50.0 * \sin(\sqrt{2.46 * COM})$$

In this section, while presenting MI, following abbreviations are used in the tables:

HV : $5.2 * \ln(HV)$
CC : $0.23 * CC$
LOC : $16.2 * \ln(LOC)$
COM : $50.0 * \sin(\sqrt{2.46 * COM})$

Among these factors, HV and CC simply indicate the difficulty of the program to write or understand as stated in Chapter 2. Any decrease in these factors is good in terms of maintainability. Another factor whose decrease results in an increase for maintainability is LOC. Programs with larger LOC values need more effort to develop. High LOC is used as a sign of low cohesion for this formula, although it is not always the case. That is to say, as LOC increases, MI decreases. Therefore, COM is only factor contributes to the MI here. All tables showing the maintainability analysis are interpreted according to these opinions.

In the tables showing the maintainability analysis, right most columns indicate the change as a percentage of the original code's MI. The leftmost column contains the short version of the following statements:

LibraryX : Original Code
LibraryX+CoR : Chain of Responsibility Pattern applied version
LibraryX+Strategy : Strategy Pattern applied version
LibraryX+Dec : Decorator Pattern applied version
LibraryX+FCADT : First Class ADT Pattern applied version
LibraryX+CoR+Strategy : Chain of Responsibility and Strategy Pattern applied version
LibraryX+CoR+Strategy+Dec: Chain of Responsibility, Strategy and Decorator applied version
LibraryX+AllPatterns : Four patterns applied together version

Explanation of ABTM Figures The results of A-B timing are introduced as a graph showing both the values of original code and pattern applied code. Each graph, named as "ABTM Measurement of ... Pattern", has the following common features unless otherwise indicated.

- Test vectors, mature data sets for the test of the library, are numbered starting from 1. X axis shows the index of these test vectors.
- Y axis indicates the time passed to execute the given test vector. Declared time is the average time of iterations that are repeated until the result falls into the confident interval range.
- Time measurement unit is given as milliseconds.
- "o" marked-red points denote the execution time of original code. "Original Code" legend is used to refer the result of the code that is not applied any design pattern.
- "x" marked-blue points are used to indicate the execution time of the code subjected to specified pattern. "Pattern Applied Code" legend is used to cite this meaning.
- A-B Timing graphs contain three parts. First one-third indicates that library is used according to the Project-1 limits. Middle of the graph illustrates the results of the library usage in Project-2 format while the last-third is arranged to show the usage format of Project-3. For Chain of Responsibility, Strategy and First Class ADT Patterns, division of the graphs does not make any sense. However, graphs of Decorator Pattern and the case when all patterns are applied together are divided into three parts visually to show the effects.

Explanation of Memory Utilization Tables In the scope of resource utilization, memory usage is analyzed. Amount of memory retained during the execution is noted and presented as a table. For each pattern, original code and pattern applied version of it are shared together. Same abbreviations with the MI tables are used. To show the effect of project constraints, table of Decorator Pattern has also following notations:

LibraryX+Dec(Project-1) : Decorator Pattern applied version used with Project-1 parameters
LibraryX+Dec(Project-2) : Decorator Pattern applied version used with Project-2 parameters
LibraryX+Dec(Project-3) : Decorator Pattern applied version used with Project-3 parameters

Similarly, when all patterns are applied together, extra abbreviations needed are as follows:

LibraryX+AllPatterns(Project-1): Four patterns applied version used with Project-1 parameters
LibraryX+AllPatterns(Project-2): Four patterns applied version used with Project-2 parameters
LibraryX+AllPatterns(Project-3): Four patterns applied version used with Project-3 parameters

In these tables, difference is indicated in the right most columns as percentage of the extra memory needed for design pattern. In this representation, if the memory demand of the code decreases with design pattern, negative sign is used.

To show the results, this section is divided into two main parts as LibraryA Tests and LibraryB Tests. Each main part includes the results of four design pattern implementations. These results are presented according to the remarks stated above.

4.4.1 LibraryA Tests

This section includes the tests conducted on LibraryA and contains four parts for each design pattern and one for all of them. For these tests, 60 test vectors are selected randomly among the data sets. However, ABTM figures show 180 indexes since it covers three different projects.

4.4.1.1 Chain of Responsibility Pattern

In this test, a code segment containing an if-statement is rearranged by using Chain of Responsibility Pattern. The aim of these statements is to determine the analysis method of the variables given as an input and set necessary parameters at the end. Starting from the first case, when the condition is not satisfied, it is given to next statement to check the variables according to its prerequisite. Chain of Responsibility Pattern is used to allocate these conditional statements and wrap the calculations on different functions.

Maintainability Analysis Table 4.1 shows the maintainability analysis of Chain of Responsibility Pattern as a part of LibraryA tests. As can be observed from the table, pattern yields to decrease in average HV, average CC, average LOC, and average COM ratio. Among these parameters only decrease in COM affects maintainability in a worse way. Other decline strengthens the maintainability. As a result of these degradations, MI increases.

The reason can be explained as follows: Chain of Responsibility Pattern breaks the conditional statements and creates a new function for each of these statements. As a consequence, complexity and HV of the code decrease. In addition, increase in the number of functions leads average LOC decreases although LOC increases actually. Consequently, despite the small decrease in COM, maintainability of LibraryA rises at a rate of 4.76 percent with this pattern implementation.

Table 4.1: LibraryA Tests - MI of Chain of Responsibility Pattern

	MI	HV	CC	LOC	COM	Change(%)
LibraryA	100.26	40.45	2.45	70.34	42.50	-
LibraryA+CoR	105.03	39.30	1.96	66.88	42.18	4.76

A-B Timing Figure 4.1 illustrates the execution time of the test vectors prepared for the algorithm. As can be shown from the figure, Chain of Responsibility Pattern affects the timing of LibraryA very slightly. Maximum overhead is recorded as $5.5681 * 10^{-5}$ msec. The percentage of this duration to the execution time of the original code is 2.3690%. Small difference between the original code and pattern applied version can be thought as the result of the function call overhead.

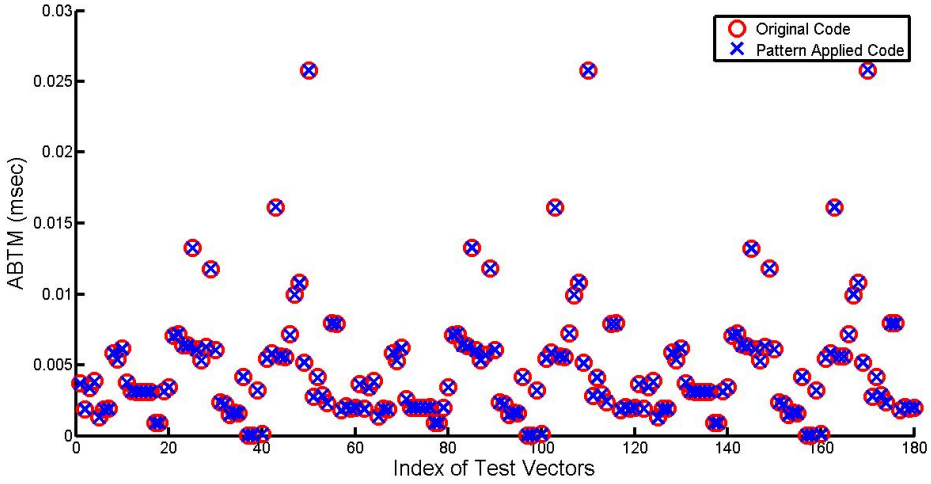

Figure 4.1: LibraryA Tests - ABTM Measurement of Chain of Responsibility Pattern

Memory Utilization Table 4.2 shows the memory utilization of the Chain of Responsibility Pattern. As stated in class diagram given in Chapter 3, each handler in the chain contains a successor and a function pointer to be overwritten. 8 bytes are used for these pointers. This makes 24 bytes for three handlers. One chain is located in one function and one for the other for this test. As a result, extra 48 bytes are consumed.

Even if design pattern is not used, client has to have 8804072 bytes in order to use this library. 48 bytes are 0.000545% of this amount. Therefore, it does not hurt to say that Chain of Responsibility Pattern does not pose a threat in terms of memory utilization for LibraryA application.

Table 4.2: LibraryA Tests - Memory Utilization of Chain for Responsibility Pattern

	Memory Usage(byte)	Additional Memory(%)
LibraryA	8804072	-
LibraryA+CoR	8804120	0.000545

4.4.1.2 Strategy Pattern

In this test, Strategy Pattern is used to separate the switch cases in the function whose responsibility is sorting. One concrete strategy class per each switch case is created. Instead of calling the sorting function with a parameter indicating the sorting method, a concrete strategy class carrying related sorting strategy is invoked.

Maintainability Analysis Table 4.3 shows that Strategy Pattern adds up the MI of LibraryA. This increase is the result of the decrease in HV, CC, and LOC. The reason is similar to that of Chain of Responsibility Pattern. Removing the conditional statement leads a fall in complexity of code. Increasing number of files and functions not only facilitates to this fall, but also decreases average LOC. Combination of small degradation in HV and CC with the high one in LOC dominates the small decrease of COM. As a result, MI of the LibraryA increases with the addition of Strategy Pattern at a rate of 4.17 percent.

Table 4.3: LibraryA Tests - MI of Strategy Pattern

	MI	HV	CC	LOC	COM	Change(%)
LibraryA	100.26	40.45	2.45	70.34	42.50	-
LibraryA+Strategy	104.44	39.43	2.00	67.31	42.18	4.17

A-B Timing Instead of calling a function with a parameter indicating the sorting method, calling the concrete strategy class operation is what this pattern offers. Timing overhead related to the usage of design pattern is expected to be small since both methods are predicated on function call. Figure 4.2 illustrates the execution time of the test vectors prepared for the algorithm and validates this expectation. As can be seen from the figure, there is no significant difference between the original code and pattern applied one. Maximum difference reported on graph is $1.8252 * 10^{-5}$ msec and it is 1.9846% of the time passed to execute the original code.

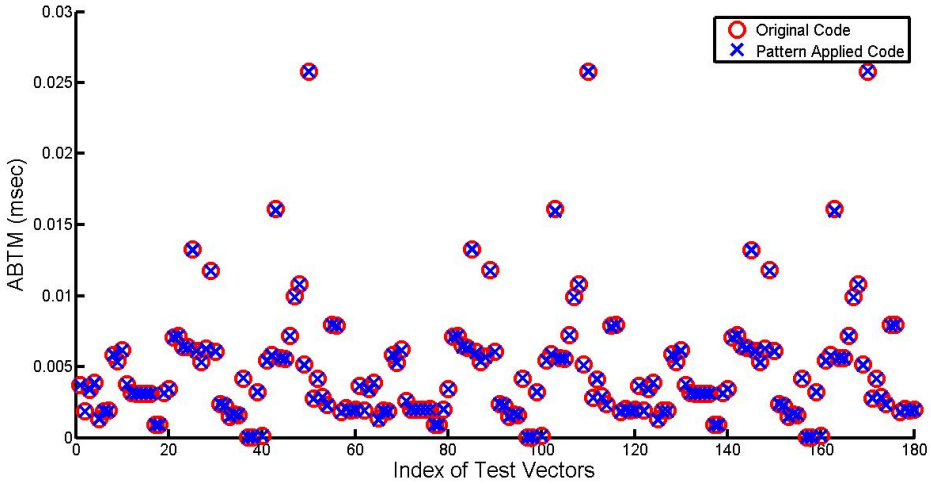

Figure 4.2: LibraryA Tests - ABTM Measurement of Strategy Pattern

Memory Utilization Table 4.4 shows the memory utilization before and after Strategy Pattern implementation. As can be seen, Strategy Pattern needs additional 24 bytes for LibraryA tests. Strategy structure only holds the pointer to the AlgorithmInterface() virtual function and uses only 4 bytes. Context structure includes a pointer for sorting structure and a pointer to ContextInterface() function. 8 bytes are reserved for these pointers. Additional 12 bytes are used as input parameters of the sorting methods. 24 bytes are very small compared to the original need of the library and only 0.000273% of it. This result makes Strategy Pattern secure in terms of memory utilization.

Table 4.4: LibraryA Tests - Memory Utilization of Strategy Pattern

	Memory Usage(byte)	Additional Memory(%)
LibraryA	8804072	-
LibraryA+Strategy	8804096	0.000273

4.4.1.3 First Class ADT Pattern

This pattern is used to hide the structures of the library from its users so that coupling between them decreases. Input and output structures are preserved and get-set operations are presented to the user.

Maintainability Analysis As stated in the Experimental Methodology part, First Class ADT Pattern is just the right thing for library applications. The result shown in Table 4.5 confirms this statement. High increase in MI is due to the decrease in HV, CC and LOC. The reason behind these declines is the increase in number of functions. Output and input structures of the library are covered by using this pattern. As a result, getting or setting parameters necessitate creating new functions. Direct access is not allowed anymore. Coupling between the library and its users is decreased. All these improvements reflect positively on MI.

Table 4.5: LibraryA Tests - MI of First Class ADT Pattern

	MI	HV	CC	LOC	COM	Change(%)
LibraryA	100.26	40.45	2.45	70.34	42.50	-
LibraryA+FCADT	112.39	37.49	1.46	61.52	41.86	12.1

A-B Timing Internal design of the code is not changed during this pattern implementation. Makeup of the library is redesigned so that low coupling is established between the requesters and provider. The only overhead here is setting the input and getting the output parameters through function calls. Test results shown in Figure 4.3 include the preparation time of the input structure and copying time of the output structure. Maximum lost on the graph is $4.7122 * 10^{-5}$ msec and it is 2.4744% of the execution time of the original code. Therefore, it can be said that performance of the LibraryA does not place at risk with First Class ADT Pattern implementation.

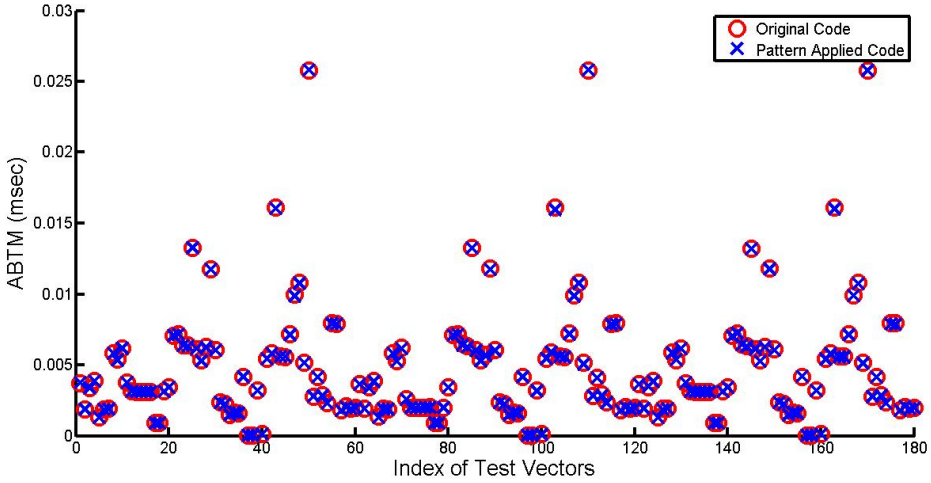

Figure 4.3: LibraryA Tests - ABTM Measurement of First Class ADT Pattern

Memory Utilization This pattern does not need additional memory since it does not affect the internal structure of the library.

Table 4.6: LibraryA Tests - Memory Utilization of First Class ADT Pattern

	Memory Usage(byte)	Additional Memory(%)
LibraryA	8804072	-
LibraryA+FCADT	8804072	0

4.4.1.4 Decorator Pattern

Decorator Pattern is used to add additional responsibility at run-time. Output structure is rearranged as the smallest set of variables needed for all projects. Calculations and variables which are special to one project are excluded from the main function of the program. Instead of this, concrete classes are created for those extra features. Projects call one of the concrete constructors depending on their demand from the library. Extra parameters are gotten with a function call since the internal structures of the concrete classes are abstracted from the user.

In this test, Project-1 and 3 use different concrete classes each of which has one extra parameter to the default output structure. Extra calculation is performed for both of them. Project-2 uses only default set and does not need additional behavior.

Maintainability Analysis According to the result shown in Table 4.7, all changes implemented in the scope of Decorator Pattern eventuate in increase of MI. In addition to the small decrease in HV and CC, LOC decrease with the additional functions improves the maintainability about 8.69%.

Table 4.7: LibraryA Tests - MI of Decorator Pattern

	MI	HV	CC	LOC	COM	Change(%)
LibraryA	100.26	40.45	2.45	70.34	42.50	-
LibraryA+Dec	108.97	38.36	1.73	64.12	42.18	8.69

A-B Timing Figure 4.4 shows the execution time of the library before and after Decorator Pattern implementation. A closer look reveals that at some points, blue points have smaller time than red ones. This means that, excluding the calculation not needed results small improvement in the performance. If data set does not already pass through excluded lines, extra function call can affect the performance a little bit. At worse case, max overhead on the graph is $4.7850 * 10^{-5}$ msec and it is the 2.5445% of the execution time of the original code with the same data set.

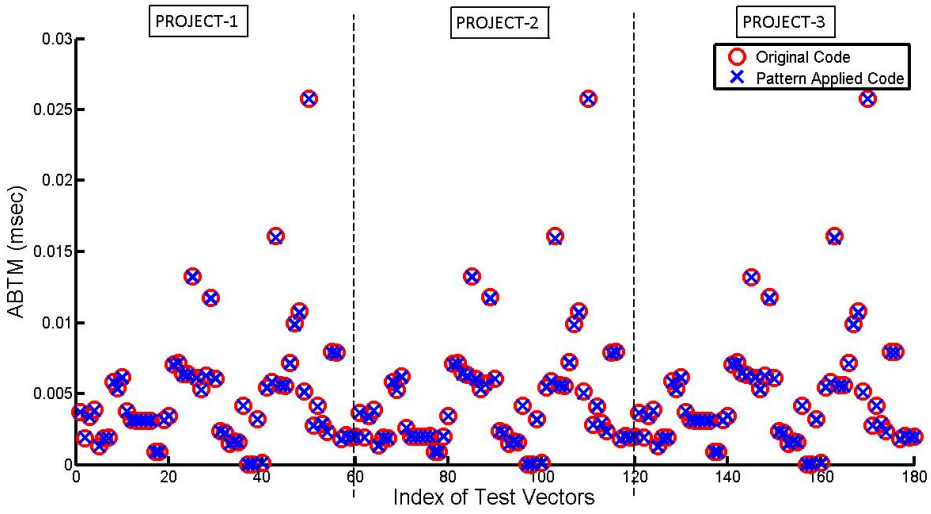

Figure 4.4: LibraryA Tests - ABTM Measurement of Decorator Pattern

Memory Utilization With the exclusion of the unnecessary behaviors, some memory gain is recorded. Project-1 and 3 gain 1552 bytes while Project-2 saves 4000 bytes. Percentages of these amounts are not too high, as can be shown from Table 4.8. However, main concern of this study is showing that design patterns do not demand considerable memory rather than providing a remarkable profit. By looking at the Table 4.8, it can be said that this claim is satisfied.

Table 4.8: LibraryA Tests - Memory Utilization of Decorator Pattern

	Memory Usage(byte)	Additional Memory(%)
LibraryA	8804072	-
LibraryA+Dec(Project-1)	8802520	-0.0176
LibraryA+Dec(Project-2)	8800072	-0.0454
LibraryA+Dec(Project-3)	8802520	-0.0176

4.4.1.5 All Patterns

This test is prepared to observe the consequences of the design patterns when they are applied together.

Maintainability Analysis Table 4.9 shows the effect of the MI when the patterns are applied consecutively. The result shows that applying patterns together increase the maintainability. Since all of the patterns are reported as maintainability increase pattern in the previous sections, result is not surprising. However, as expected from the MI formula, the increase is neither linear nor the cumulative of all improvements.

Chain of Responsibility and Strategy Patterns break the conditional statements in accordance with their purposes and share the responsibilities among different functions rather than simple switch cases. Decorator Pattern cleans the project special calculations from the main function of the library and keeps them into concrete classes which do not have interface with the user. Finally, First Class ADT gives a new form to the framework of the library and decreases coupling to its users. At the end of these efforts, as can be seen from Table 4.9, very appreciable increase is recorded as 18.04% of the MI of the original code.

Table 4.9: LibraryA Tests - MI of All Patterns

	MI	HV	CC	LOC	COM	Change(%)
LibraryA	100.26	40.45	2.45	70.34	42.50	-
LibraryA+CoR	105.03	39.30	1.96	66.88	42.18	4.76
LibraryA+CoR+Strategy	108.13	38.50	1.68	64.54	41.86	7.85
LibraryA+CoR+Strategy+Dec	113.48	37.05	1.33	60.31	41.17	13.19
LibraryA+AllPatterns	118.35	35.78	1.08	56.61	40.82	18.04

A-B Timing When the slight increases in execution time are taken into account, two questions arise. One of them is that whether these small changes cumulate when all of the patterns are applied together. Regarding to the answer of this question, second one is whether the consequence is acceptable or not.

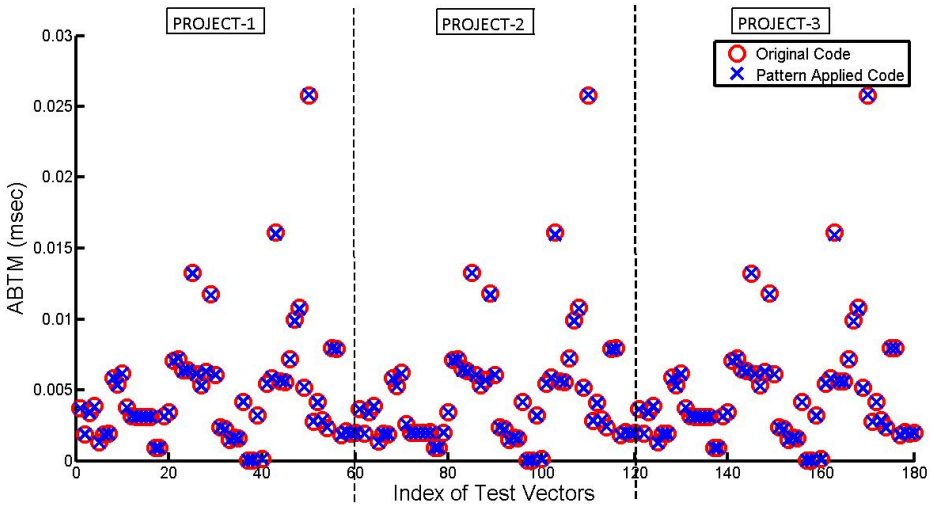

Figure 4.5: LibraryA Tests - ABTM Measurement of All Patterns

Figure 4.5 gives an explicit answer to these questions. Maximum difference in this measurement is reported as $6.7632 * 10^{-5}$ msec and it is 2.8775% of the execution time of the original code. This means that, the increase is neither cumulative nor threatening.

Table 4.10 presents a summary about the maximum overhead of the patterns. Since each pattern affects different part of the code, the result highly depends on the characteristic of the data given to the code. That is to say, maximum overhead of all patterns is not the sum of maximum overhead of each pattern. Among these patterns Strategy has the smallest overhead and Decorator has the biggest. However, neither of them puts the performance efficiency at a risk. Even if they are applied together, the overhead is not too high to be unacceptable.

Table 4.10: LibraryA Tests - Maximum Overhead of All Patterns

	Max Difference(msec)	Max Difference(%)
LibraryA+Strategy	$1.8252 * 10^{-5}$	1.9846
LibraryA+CoR	$5.5681 * 10^{-5}$	2.3690
LibraryA+FCADT	$4.7122 * 10^{-5}$	2.4744
LibraryA+Dec	$4.7850 * 10^{-5}$	2.5445
LibraryA+AllPatterns	$6.7632 * 10^{-5}$	2.8775

Memory Utilization Same questions are valid in terms of memory utilization too. Table 4.11 shows that only extra memory consumption is demanded by Chain of Responsibility and Strategy Patterns. Percentages of these utilizations are very small compared to the need of original code. First Class ADT Pattern does not require additional memory and Decorator makes an improvement for this tests. At the end, in spite of other patterns, users still have the memory gain due to the Decorator Pattern.

It must be emphasized that the purpose of the Decorator Pattern is not recording gains from memory or timing. This decrease in memory consumption of the library is not always the case. The aim here is showing that performance efficiency is not affected severely due to design patterns. Table 4.11 corrects this claim for this case study conducted on LibraryA.

Table 4.11: LibraryA Tests - Memory Utilization of All Patterns

	Memory Usage(byte)	Additional Memory(%)
LibraryA	8804072	-
LibraryA+Dec(Project-1)	8802520	-0.0176
LibraryA+Dec(Project-3)	8802520	-0.0176
LibraryA+Dec(Project-2)	8800072	-0.0454
LibraryA+FCADT	8804072	0
LibraryA+Strategy	8804096	0.000273
LibraryA+CoR	8804120	0.000545
LibraryA+AllPatterns(Project-1)	8802592	-0.0168
LibraryA+AllPatterns(Project-3)	8802592	-0.0168
LibraryA+AllPatterns(Project-2)	8800144	-0.0446

4.4.2 LibraryB Tests

This section is prepared to share the results of the experiments conducted on LibraryB and contains four parts for each design pattern and one for all of them. For these tests, 25 test vectors are selected randomly among the data sets. However, ABTM figures show 75 indexes since it covers three different projects.

4.4.2.1 Chain of Responsibility Pattern

In LibraryB application, when the data is given to the algorithm, a calculation is performed first. Then the data is send to an analysis function according to the result of this calculation. If the prerequisite of first analysis is not satisfied, it is forwarded to the next one. This mechanism is provided by using if-statement. In this test, Chain of Responsibility Pattern is offered instead of conditional statement.

Maintainability Analysis The effect of Chain of Responsibility Pattern on MI of LibraryB is recorded as 3.02%, as can be seen from Table 4.12. The reason of this increase is similar to that of LibraryA. Separating if statement into different files leads to decreases in HV, CC and LOC. Despite small decrease in COM, MI increases.

Table 4.12: LibraryB Tests - MI of Chain of Responsibility Pattern

	MI	HV	CC	LOC	COM	Change(%)
LibraryB	98.97	39.33	2.29	69.26	38.85	-
LibraryB+CoR	101.96	38.57	1.98	66.92	38.42	3.02

A-B Timing Figure 4.6 shows the effect of Chain of Responsibility Pattern on A-B timing. As can be seen from the figure, the difference is not distinguishable. Maximum overhead is $1.4270 * 10^{-5}$ msec and it is 0.0607% of the A-B timing of the original code. When function call overhead is considered, this slight change seems logical.

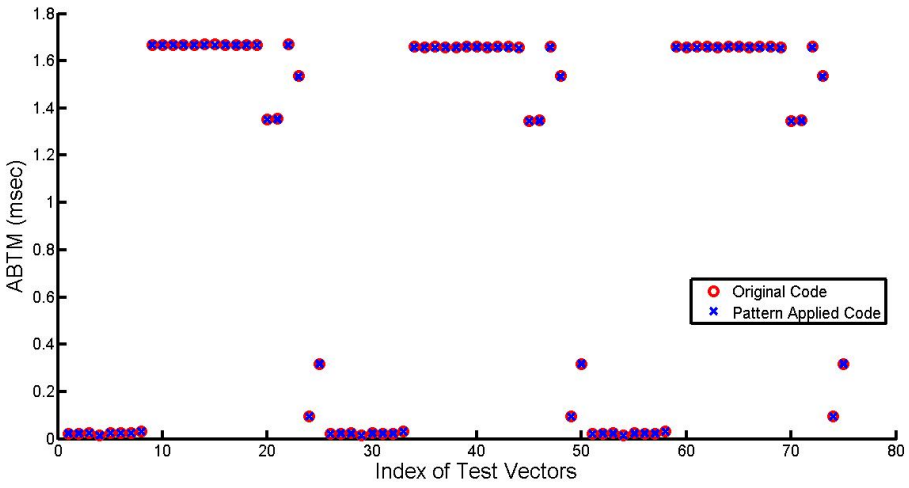

Figure 4.6: LibraryB Tests - ABTM Measurement of Chain of Responsibility Pattern

Memory Utilization This pattern needs extra 24 bytes for LibraryB implementation. Each handler holds a virtual function pointer and a pointer for its successor. This makes 8 bytes for each handler. Three handlers are used for this chain. As a result, 24 bytes are consumed. As stated on the table, the ratio between the memory usage of the library and this consumption is 0.00243%. Therefore, there is no hesitation to say that Chain of Responsibility Pattern does not create a threat in terms of memory utilization for this case study.

Table 4.13: LibraryB Tests - Memory Utilization of Chain for Responsibility Pattern

	Memory Usage(byte)	Additional Memory(%)
LibraryB	988224	-
LibraryB+CoR	988248	0.00243

4.4.2.2 Strategy Pattern

Similar to LibraryA, Strategy Pattern is used to reorganize a sorting function in LibraryB. This function decides which sorting method is performed according to the number given as input. Four if statements carry four different sorting types. For Strategy Pattern implementation, each of these methods is defined as a concrete class inherited from the abstract Strategy interface. Depending on the chosen concrete class, different methods are performed.

Maintainability Analysis Removing conditional statement and creating functions, whose responsibilities are well-defined, decrements HV, CC and LOC factors on the formula. Despite the slight decrease in COM, 3.63% gain is recorded with the help of Strategy Pattern as can be seen from the Table 4.14.

Table 4.14: LibraryB Tests - MI of Strategy Pattern

	MI	HV	CC	LOC	COM	Change(%)
LibraryB	98.97	39.33	2.29	69.26	38.85	-
LibraryB+Strategy	102.56	38.33	1.88	66.20	37.98	3.63

A-B Timing In the A-B Timing measurement of Strategy Pattern, no critical overhead is recorded. As can be seen from Figure 4.7, execution time of the original code and pattern applied code is very close to each other. Maximum difference between blue and red points on the graph is $3.4183 * 10^{-6}$ msec. This time is the 0.0036% of the execution time of the original code. This small increase can be explained as the overhead of function call rather than performing the sorting in a conditional statement.

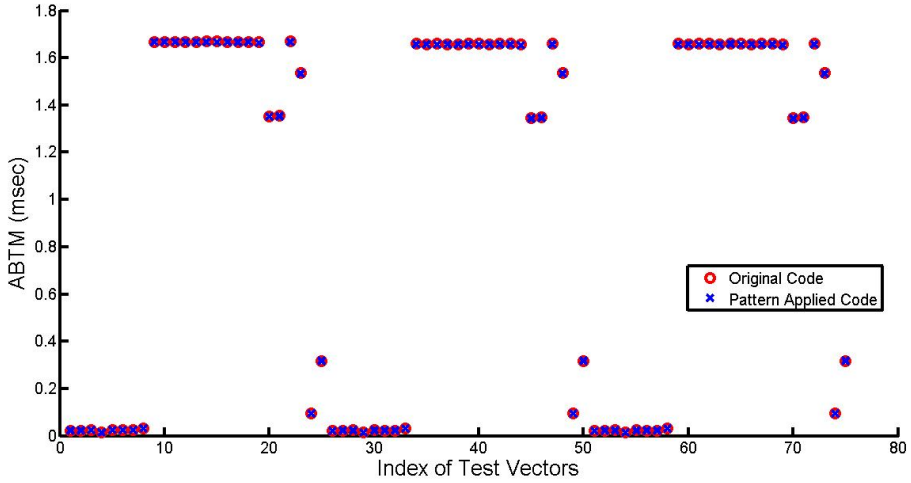

Figure 4.7: LibraryB Tests - ABTM Measurement of Strategy Pattern

Memory Utilization For this pattern implementation, four concrete classes are created. They are all given to the same pointer since they are not used together. One Strategy class needs only 4 bytes to hold a pointer to its virtual function. Context class, on the other hand, requires two pointers one of which is for ContextInterface() function and the other one is for Strategy class. This means that 12 bytes are consumed by the essential features of the pattern. Additional 20 bytes are needed by Context structure to give to the sorting methods. Like shown in Table 4.15, extra 32 bytes is only 0.0032% of the memory utilization of the original code.

Table 4.15: LibraryB Tests - Memory Utilization of Strategy Pattern

	Memory Usage(byte)	Additional Memory(%)
LibraryB	988224	-
LibraryB+Strategy	988256	0.0032

4.4.2.3 First Class ADT Pattern

First Class ADT Pattern is used to hide the structures of the library from its users. Only necessary functions are presented to set input and get output structures.

Maintainability Analysis This pattern confirms its necessity for library applications one more time for this case study. As can be seen from Table 4.16, maintainability shows an increase at the rate of 11.56%. This increase is the result of decrease in HV, CC and LOC. Saving the structure inside requires presenting get and set options to the client for essential parameters. Small functions, whose responsibility is only giving an access to the user, decrease the average HV, CC and LOC of the code.

Table 4.16: LibraryB Tests - MI of First Class ADT Pattern

	MI	HV	CC	LOC	COM	Change(%)
LibraryB	98.97	39.33	2.29	69.26	38.85	-
LibraryB+FCADT	110.41	36.46	1.39	60.71	37.98	11.56

A-B Timing Figure 4.8 shows that First Class ADT Pattern does not create an overhead that cannot be handled. Maximum of these overheads is $1.1284 * 10^{-5}$ msec. This is the 0.0363% of the execution time of the original code.

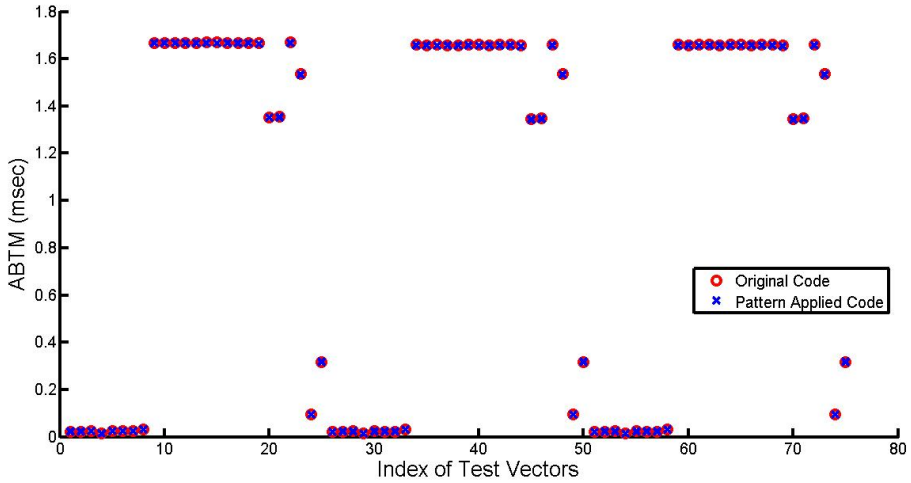

Figure 4.8: LibraryB Tests - ABTM Measurement of First Class ADT Pattern

Memory Utilization This pattern does not need additional memory like shown in Table 4.17.

Table 4.17: LibraryB Tests - Memory Utilization of First Class ADT Pattern

	Memory Usage(byte)	Additional Memory(%)
LibraryB	988224	-
LibraryB+FCADT	988224	0

4.4.2.4 Decorator Pattern

In this test, the main function of the library is minimized to operate basic operations. Concrete classes are given to the responsibility of executing extra features. Among the projects considered in this test, Project-1 uses only default behaviors of the library. There is no additional state or function is required for this project. Project-2 expects two additional states and calculations from the library whereas Project-3 uses four extra behaviors.

Maintainability Analysis Decorator Pattern increases MI of LibraryB due to the similar reasons with LibraryA. Main function of the code is separated into different parts. Pieces of code, not necessary for all projects, are given to the concrete classes. By doing so, the improvement of the code complexity can be observed with HV and CC factors of the Table 4.18. Main contribution comes from the LOC factor. Since a long function performing too many responsibilities is separated into different functions, average LOC highly decreases.

Table 4.18: LibraryB Tests - MI of Decorator Pattern

	MI	HV	CC	LOC	COM	Change(%)
LibraryB	98.97	39.33	2.29	69.26	38.85	-
LibraryB+Dec	105.06	37.69	1.66	64.11	37.52	6.15

A-B Timing As Figure 4.9 shows, very small improvement is recorded at some points for Project-1 portion of the graph. However, since Project-2 expects two additional states and calculations from the library, its timing includes a little bit overhead due to function calls. As can be seen from the figure, difference increases on Project-3 part of the graph since it uses four additional behaviors. At worse case, project suffers 0.0110 msec delay. This is 0.8172% of the original waiting time of user for library execution.

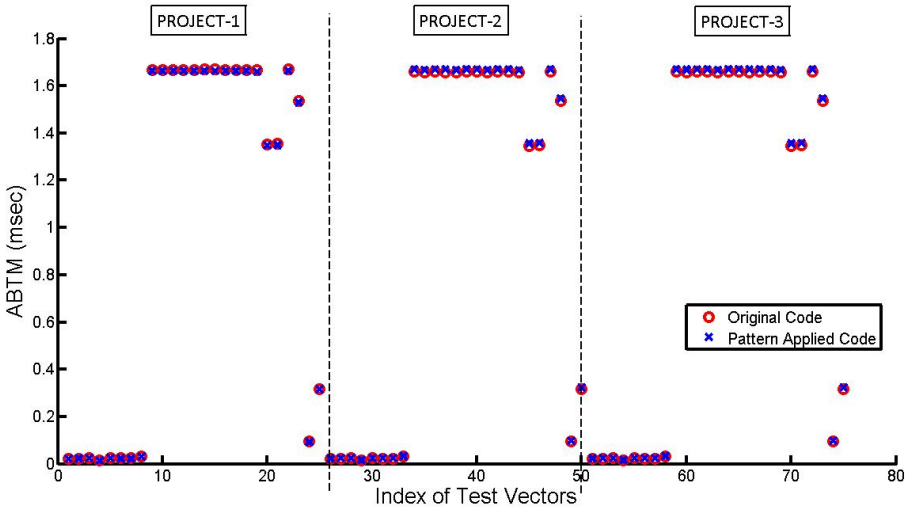

Figure 4.9: LibraryB Tests - ABTM Measurement of Decorator Pattern

Memory Utilization As stated in Chapter 3, in this pattern implementation, concrete classes start with the same structure with the abstract class, follow with a pointer to the concrete component and add extra parameters to the end. To call the concrete component class operation in order to perform default behavior, pointer of concrete component is needed to be given a memory. This means that, smallest set of the output structure has to be created in addition to that of concrete classes. This is the reason behind the increase of memory utilization of Project-2 and 3. Project-1 only uses default behavior and slight improvement is recorded for this project. Even in the worse case, lost is 0.0045% of the original memory requirement of the library.

Table 4.19: LibraryB Tests - Memory Utilization of Decorator Pattern

	Memory Usage(byte)	Additional Memory(%)
LibraryB	988224	-
LibraryB+Dec(Project-1)	988212	-0.0012
LibraryB+Dec(Project-2)	988260	0.0036
LibraryB+Dec(Project-3)	988268	0.0045

4.4.2.5 All Patterns

This test is prepared to observe the consequences of the design patterns when they are applied together. As a first step, Chain of Responsibility Pattern applied code is subjected to the Strategy Pattern. After the MI analysis of this version, Decorator Pattern is implemented. MI of the code is measured again. Finally, First Class ADT Pattern is applied and measurement is repeated. Then, all patterns applied version of the code is analyzed in terms of A-B timing and memory utilization.

Maintainability Analysis When the patterns are applied to the base code, major improvements are gained after Decorator and First Class ADT Pattern implementations. Decorator increases maintainability 6.15%, while First Class ADT makes 11.56% improvements. Table 4.20 shows the results when these patterns applied together. After each pattern implementation, maintainability is analyzed. Table shows an incremental graphic during the pattern implementation. Biggest progress is still due to the Decorator and First Class ADT Patterns.

Table 4.20: LibraryB Tests - MI of All Patterns

	MI	HV	CC	LOC	COM	Change(%)
LibraryB	98.97	39.33	2.29	69.26	38.85	-
LibraryB+CoR	101.96	38.57	1.98	66.92	38.42	3.02
LibraryB+CoR+Strategy	104.77	37.73	1.68	64.34	37.52	5.86
LibraryB+CoR+Strategy+Dec	108.89	36.58	1.34	60.74	36.55	10.02
LibraryB+AllPatterns	115.02	35.09	1.05	56.38	36.55	16.22

A-B Timing Figure 4.10 shows the ABTM measurement of the LibraryB when all the patterns applied together. In this figure, maximum overhead is 0.0247 msec. This amount seems high compared to the other test results and the difference can be observed from the graph for this case, unlike the others. However, this amount is only the 1.6085% of the execution time of the original code. This means that, the percentage of this overhead is smaller than that of LibraryA. Therefore, it can be concluded that the reason behind the visibility of the difference is not the amount, but the scale of the graphing.

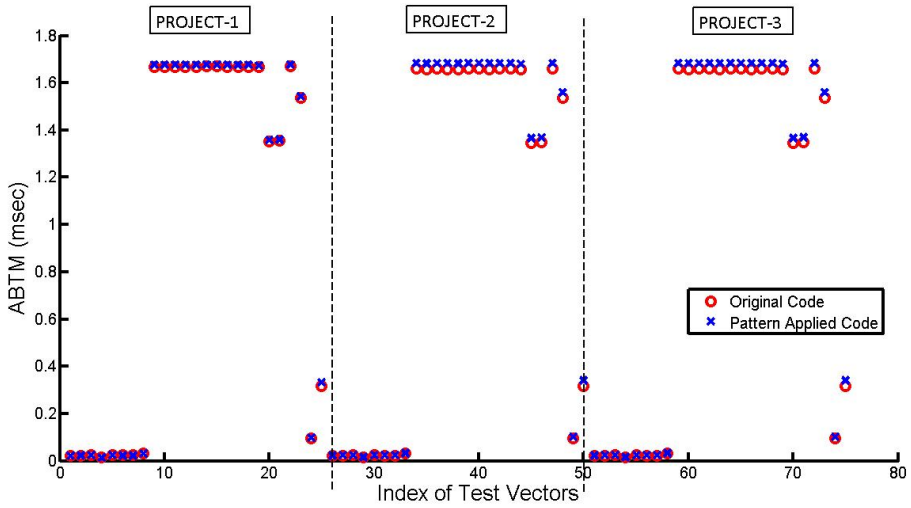

Figure 4.10: LibraryB Tests - ABTM Measurement of All Patterns

Table 4.21 summarizes the results given above sections combining with this one. According to the results, minimum overhead belongs to Strategy Pattern. The reason behind this small change is explained in Strategy Pattern section. For this case study, Decorator Pattern has maximum overhead among these patterns. These outcomes are similar to that of LibraryA application except from Chain of Responsibility Pattern swops places with First Class ADT Pattern. Apart from small differences between the maximum overhead of the patterns, percentages of all overheads are very slight to be dangerous.

Table 4.21: LibraryB Tests - Maximum Overhead of All Patterns

	Max Difference(msec)	Max Difference(%)
LibraryB+Strategy	$3.4183 * 10^{-6}$	0.0036
LibraryB+FCADT	$1.1284 * 10^{-5}$	0.0363
LibraryB+CoR	$1.4270 * 10^{-5}$	0.0607
LibraryB+Dec	0.0110	0.8172
LibraryB+AllPatterns	0.0247	1.6085

Memory Utilization Memory utilization of each pattern is given in previous subsections. In this section, a general look is presented with Table 4.22. As can be observed from the table, design patterns need extra memory expect from the Decorator Pattern working with Project-1. Reducing the memory usage with Decorator Pattern is not the intent here. This is a consequence of the making output structure smaller. Other patterns have small addition regarding to the original need of the library. LibraryB requires 988224 bytes like shown in the table. This amount does not affected by First Class ADT Pattern usage. Chain of Responsibility and Strategy Patterns need very small additional memory compared to that of original code. When the Decorator Pattern is used by Project-2, the highest increase is achieved as 0.0045%.

In the case when all patterns are applied together, the minimum additional memory is 44 bytes. In the worse case, the amount rises to 100 bytes. This is 0.0101% of the original memory requirement. Therefore, it can be concluded that memory usage is not threatening in terms of resource utilization.

Table 4.22: LibraryB Tests - Memory Utilization of All Patterns

	Memory Usage(byte)	Additional Memory(%)
LibraryB	988224	-
LibraryB+Dec(Project-1)	988212	- 0.0012
LibraryB+FCADT	988224	0
LibraryB+CoR	988248	0.00243
LibraryB+Strategy	988256	0.0032
LibraryB+Dec(Project-2)	988260	0.0036
LibraryB+Dec(Project-3)	988268	0.0045
LibraryB+AllPatterns(Project-1)	988268	0.0045
LibraryB+AllPatterns(Project-2)	988316	0.0093
LibraryB+AllPatterns(Project-3)	988324	0.0101

CHAPTER 5

DISCUSSION AND CONCLUSIONS

This study was performed in order to observe the effect of design patterns on non-OO real-time software. Main motivation of this study is incontestable effects of design patterns on OO language. The aim here is not to show how OO programming is implemented by using C. Like [29] or [24], there are many books and articles written for this purpose. When using OO programming is not possible for some development environment or systems, these techniques can be preferred. However, the reason here is not due to the constraints of the systems. For software used in this study, OO programming is consciously not used for specific reasons such as preventing OO overhead that is not acceptable for time-focused software. Therefore, target audience of this study is real-time software developers who still choose a procedural language even if their environments do not force them.

Herein, after previous studies about the topic were reviewed, a guide for implementing the design patterns in C language was proposed. Practices of First Class ADT, Strategy, Chain of Responsibility and Decorator Patterns were detailed. These patterns were implemented on two library applications developed in ASELSAN INC. The effects on software maintainability were analyzed by using MI calculation proposed by [9]. For performance efficiency, A-B timing [16], and maximum memory usage were investigated.

The results of the experimental work showed that First Class ADT and Decorator Patterns increased MI significantly, whereas Chain of Responsibility and Strategy Patterns caused less obvious improvement. In addition, effect of the patterns on performance efficiency is so marginal that neither of them creates a risk in terms of timing or memory utilization.

The most important point that must be stated here is that design patterns are useful only if they are used correctly, like [3] and [10] stated. The present study does not claim that implementing design patterns always solves the problems and improves maintainability. Here, the results showed that patterns were implemented in code which really needed design patterns. However, as stated in [4],[8], and [19], using design patterns effectively requires a lot of practices.

This thesis has some similarities with previous work carried out by T. Turk and M.Ayata. Turk [27] investigated the effects of some design patterns on maintainability and Ayata [3] studied the real-time performance effects of design patterns. However, those studies analyzed the design patterns applied in OO environments and used OO metrics. In the present study, both maintainability and performance issues of different pattern cluster were investigated together, but in a non-OO environment.

This study can be considered as a deep analysis of Petersen's series [21]. In those series, implementation of First Class ADT, State, Strategy, Observer and Reactor Patterns were given. On the other hand, experimental analysis was not conducted on those implementations. This study presented new pattern implementations and most importantly measured their effects on some software quality metrics.

At the end of the case study, the research hypotheses stated in Chapter 1 were satisfied on two library applications. To make general statements on these claims, further implementations on different software components must be studied. This study opens a door for future works. Implementation of different design patterns in C can be studied. The effects of design patterns on different software quality metrics can be analyzed. Other than all of these, metrics used in this study can be improved.

REFERENCES

[1] Akdur, D. "Comparative Evaluation of Design Pattern Usage in Real-Time Embedded Software Development", Technical Report, Informatics Institute, METU (2011)

[2] Al-Kilidar, H., Cox, K.& Kitchenham, B. "The Use and Usefulness of the ISO/IEC 9126 Quality Standard", International Symposium on Empirical Software Engineering (2005)

[3] Ayata, M. "Effect of Some Software Design Patterns on Real Time Software Performance", Ms Thesis, Information System Department, METU (2010)

[4] Beck, K. et. all. "Industrial Experience with Design Patterns", Proceedings of the 18th International Conference on Software Engineering, 103-114 (1996)

[5] Billard, E.A. "Language-Dependent Performance of Design Patterns", ACM SIGSOFT Software Engineering Notes, 28(3) (2003)

[6] Booch, G. "Object Oriented Design with Applications", Redwood City, CA : Benjamin/Cummings Publishing Company Inc. (1994)

[7] Chidamber, S. & Kemerer, C. "A Metrics Suite for Object-Oriented Design", IEEE Transactions on Software Engineering, 20(6) (1994)

[8] Cline, M.P. "The Pros and Cons of Adopting and Applying Design Patterns in the Real world", Communications of the ACM, 39(10) (1996)

[9] Coleman, D. M.,Ash D., Lowther, B. & Oman, P. W. "Using Metrics to Evaluate Software System Maintainability", IEEE Computer, 27(8), 44-49 (1994)

[10] Constantinescu, I. "Performance of Object Oriented Design with Patterns", MS Thesis, ECE Department, Carleton University (1999)

[11] Deitel, P. C++ How to Program, 5th Edition, Prentice Hall (2005)

[12] Douglas, B.P. Design Patterns for Embedded Systems in C, 1st ed., Oxford : Elsevier Inc (2011)

[13] Farrell, J. Object Oriented Programming Using C++, 4th Edition, Course Technology (2008)

[14] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, 1st ed., Addison-Wesley, (1995)

[15] Halstead, M. H. Elements of Software Science (Operating and Programming Systems Series), New York, NY (1977)

[16] Hillary, N. "Measuring Performance for Real-Time Systems", Freescale Semiconductor, November (2005)

[17] Huaxin, M. & Shuai, J. "Design Patterns in Software Development", IEEE 2nd International Conference on Software Engineering and Service Science, 322-325 (2011)

[18] Huaxin, M. & Shuai, J. "Design Patterns in Object Oriented Analysis and Design", IEEE 2nd International Conference on Software Engineering and Service Science, 326-329 (2011)

[19] Masuda, G., Sakamoto, N. & Ushijima, K. "Redesigning of an Existing Software Using Design Patterns", Proceedings of the International Symposium on Principles of Software Evaluation (2000)

[20] McCabe, T. J. "A Complexity Measure", IEEE Trans. Software Eng, 2(4), 308-320 (1976)

[21] Petersen, A. "Patterns in C", C Vu, Journal of the ACCU (Association of C and C++ Users), 17(1, 2, 3, 4, 5) (2005)

[22] Saltzer, J.H. & Schroeder, M.D. "The Protection of Information in Computer Systems", Proceeding of IEEE, 63(9), 1278-1308 (1975)

[23] Schmidt, D.C. "Using Design Patterns to Develop Reusable Object-Oriented Communication Software", Communications of the ACM - Special Issue on Object-Oriented Experiences and Future Trends, 38, 10 (1995)

[24] Schreiner, A.T., Object Oriented Programming with ANSI-C (1993)

[25] Srinivasan, S. "Design Patterns in Object-Oriented Frameworks", IEEE Journal, 32(2), 24-32 (1999)

[26] Stevens, W., Myers, G. & Constantine, L. "Structured Design", IBM Systems Journal, 13(2), 115-139 (1974)

[27] Turk, T. "The Effect of Software Design Patterns on Object-Oriented Software Quality and Maintainability", MS Thesis, EEE Department, METU (2009)

[28] Welker, K. "The Software Maintainability Index Revisited", CROSSTALK, 18-21 (2001)

[29] Williams, K. "Using Object Oriented Analysis and Design in a Non-Object Oriented Environment Experience Report", International Conference on Software Maintenance, 109-114 (1995)

[30] International Standard ISO/IEC 9126-1:2001, Software Engineering - Product Quality - Part 1: Quality Model (2001)

[31] International Standard ISO/IEC FDIS 25010:2010, Systems and Software Quality Requirements and Evaluation - System and Software Quality Models (2010)

[32] Scientific Toolworks, Inc., "Understand 3.0 User Guide and Reference Manual" (2012)

[33] SEL-94-003, "The C Style Guide", NASA Software Engineering Laboratory Series, Greenbelt, Maryland (1994)

[34] Wind River Systems, Inc., "Wind River Workbench 3.0" (2007)

[35] "acjf_maint_index_halstead.pl", http://www.scitools.com/plugins/perl_scripts.php, Access date: 02/01/2013

[36] "Benzetim Sonuçlarının Güvenilirliği", http://www.eee.metu.edu.tr/ bilgen/Guvenilirlik.pdf, Access date: 07/01/2013

[37] "Library (computing)", http://en.wikipedia.org/wiki/Library_(computing), Access date: 01/01/2013

[38] "MATLAB-The Language of Technical Computing", http://www.mathworks.com/products/matlab, Access date: 02/01/2013

[39] "Wind River VxWorks", http://www.windriver.com/products/vxworks, Access date: 02/01/2013

[40] "Validity and Reliability", http://hsc.uwe.ac.uk/dataanalysis/quantissuesvalid.asp, Access date: 04/01/2013