EFFECT OF SOME SOFTWARE DESIGN PATTERNS ON REAL TIME SOFTWARE PERFORMANCE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MESUT AYATA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

JUNE 2010

Approval of the Graduate School of Informatics,

_____

Prof. Dr. Nazife Baykal
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____

Assist. Prof. Dr. Tuğba Taşkaya Temizel
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____

Prof. Dr. Semih Bilgen
Supervisor

Examining Committee Members

Assist. Prof. Dr. Altan Koçyiğit	(METU)	_____
Prof. Dr. Semih Bilgen	(METU)	_____
Tanın Afacan (M.Sc.)	(ASELSAN)	_____
Assist. Prof. Dr. Aysu Betin Can	(METU)	_____
Assist. Prof. Dr. Erhan Eren	(METU)	_____

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 : Mesut Ayata

Signature :

ABSTRACT

EFFECT OF SOME SOFTWARE DESIGN PATTERNS ON
REAL TIME SOFTWARE PERFORMANCE

Ayata, Mesut
M.Sc., Department of Information Systems
Supervisor: Prof. Dr. Semih Bilgen

June 2010, 86 pages

In this thesis, effects of some software design patterns on real time software performance will be investigated. In real time systems, performance requirements are critical. Real time system developers usually use functional languages to meet the requirements. Using an object oriented language may be expected to reduce performance. However, if suitable software design patterns are applied carefully, the reduction in performance can be avoided. In this thesis, appropriate real time software performance metrics are selected and used to measure the performance of real time software systems.
Keywords: Real Time Software Performance, Design Patterns, Real Time Design Patterns.

ÖZ

BAZI YAZILIM TASARIM ÖRÜNTÜLERİNİN
GERÇEK ZAMANLI YAZILIM BAŞARIMINA ETKİLERİ

Ayata, Mesut
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Prof. Dr. Semih Bilgen

Haziran 2010, 86 sayfa

Bu tezde bazı yazılım tasarım örüntülerinin gerçek zamanlı yazılımın başarımı üzerindeki etkileri araştırılacaktır. Gerçek zamanlı sistemlerde, başarım gereksinimleri kritiktir. Nesne yönelimli bir dil kullanmanın bu sistemlerde başarım düşüşüne yol açması beklenebilir. Ancak, uygun tasarım örüntüleri doğru bir şekilde uygulandığında, başarım kaybı önlenebilir. Bu tezde, uygun gerçek zamanlı başarım metrikleri seçilmiş ve gerçek zamanlı yazılım sistemlerinin başarımının ölçümünde kullanılmıştır.

Anahtar Kelimeler: Gerçek Zamanlı Yazılım Başarımı, Tasarım Örüntüleri, Gerçek Zamanlı Tasarım Örüntüleri

Sevgili Aileme
ve
Zeliha'ya ...

ACKNOWLEDGEMENTS

I would like to present my deepest gratitude to Prof. Dr. Semih Bilgen for his guidance, advice, understanding and supervision throughout the development of this thesis study.

I would like to thank to the committee members for their valuable comments and discussions. I would also like to thank to ASELSAN Inc. for the support on academic studies and letting me involve in this study.

I would like to thank to Zeliha Bozkurt and I will never forget her understanding, trust and great support through both my undergraduate and graduate years.

Finally, I would like to express my special thanks to my parents Nehiye and Bedri Ayata, my brothers Değer and Hakan, my sisters Güllü, Fatma, Selda and Duygu, for their love, trust, understanding and every kind of support not only throughout my thesis but also throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

TABLE

LIST OF FIGURES

FIGURE

LIST OF CODES

LISTING

LIST OF ABBREVIATIONS AND ACRONYMS

ABTM	A-B Timing Metric
CBO	Coupling Between Objects
COF	Coupling Factor
DIT	Depth of Inheritance Tree
DTS	Data Transmission System
EMM	Execution Memory Metric
GCLP	Garbage Collection Pattern
GCMP	Garbage Compactor Pattern
GOF	Gang of Four
GRASP	General Responsibility Assignment Software Patterns or Principles
IMM	Initialization Memory Metric
MCM	Memory Consumption Metrics
MFM	Memory Fragmentation Metrics
MHF	Method Hiding Factor
OFSM	Object File Size Metric
OO	Object Oriented
OOAD	Object Oriented Analysis and Design
OOL	Object Oriented Language
RCA	Reference Counting Algorithm
RFC	Response for a Class
RTOS	Real Time Operating System
RTS	Real Time System(s)
SPP	Smart Pointer Pattern

TMM	Task Memory Metric
UML	Unified Modeling Language
WDP	With Design Pattern
WMC	Weighted Methods per Class
WODP	Without Design Pattern

CHAPTER 1

INTRODUCTION

1.1 Real Time Software Performance and Design Patterns

The main difference between a non-real time and real time software system is the performance criteria. Timing is especially important in a real time system (RTS). This does not mean that an RTS should be very fast. It means that the RTS should be as fast as required. [6]. That is, performance is critical for an RTS.

Design patterns have a great importance in software engineering. Each design pattern deals with a problem that is seen over and over again. It provides a general solution to the problem so that the problem is not solved from scratch each time it is encountered. Instead, the established solution is applied to the problem. This increases efficiency of developing code. Moreover, design patterns increase maintainability, reusability and understandability of the system [3]. Design patterns promote Open-Closed principle. This principle states that software should be open for extension and closed for modification. [10]. Design patterns are also consistent with object oriented analysis and design concepts. Especially Gang of Four (GOF) design patterns [3] have emerged directly from object oriented analysis and design considerations. Any software system that is analyzed and designed in the object oriented way can usually not be considered without design patterns. Many software engineers use software design patterns even when they are unaware of their names while applying object oriented analysis and design

(OOAD). However, this does not mean that design patterns may not be used explicitly during OOAD. To the contrary, design patterns are widely used in the object oriented (OO) world.

Design patterns are not ready codes to deploy in a software program. Rather, they give a general solution to the problem. There may be similar problems in different environments, or software systems. The same design pattern may be applied to many different systems, but it is the reason to use that design pattern that is common. However, the code will most probably be very different. Therefore, they should not be used anytime anywhere. They are useful if they are really necessary. If the right design pattern(s) are applied to the problem, the reusability, readability, maintainability of the software system will increase. To sum up, design patterns aim to increase the quality of the system.

1.2 The Purpose and Scope of the Study

The purpose of this study is to investigate the effects of some design patterns on RTS performance. There are various motivations for this purpose.

There are many studies about the effects of design patterns on software systems. Usually, the effects of design patterns on the maintainability, reusability and flexibility of the software are investigated. This study is focused on the performance effects of design patterns.

Moreover, in the literature the measurements of the metrics are usually done as a prediction and/or post-execution calculation. However, in this study, all the measurements are done in a real time manner. That is, the results are not a prediction and/or post-execution calculation; they are measured during the run time of the RTS, with the aim of providing direct, rather than indirect assessment of the performance effects of design patterns.

Lastly, this study investigates the effects of GOF and RT design patterns separately on RTS performance. Furthermore, a GOF and an RT design pattern are applied together to an RTS to see the effects of their combination on performance.

The maintainability, reusability, safety etc. of the software systems are out of the scope of this thesis. Moreover, the focus is specifically on RTS, not general software systems. Since each design pattern deals with a specific kind of a problem; an applicable platform, software and a system should be found to apply each one. However, there are practical limitations on the number of available real life professional projects to apply design patterns and investigate their effects. These facts determine the scope of this study. That is, only the GOF and RT design patterns that could possibly be applied on available projects have been studied. Furthermore, only those design patterns that are known to effect execution time performance are investigated.

1.3 Outline

This thesis is composed of the following parts.

In chapter 2, in order to determine the real time performance metrics and design patterns that will be used in this thesis, a literature review has been performed. Both RT and GOF design patterns are examined in conjunction with real time performance metrics.

In chapter 3, experimental work that has been performed to see the effect of design patterns on RTS is explained. The term “experiment” is used throughout this report to indicate the controlled setting whereby existing software from real life projects has been copied and adapted for measurement of the designated metrics, with and without the application of selected design patterns. As such, the experiments are controlled and repeatable, as well as being representative of real-life projects. Description of the projects, the experimental methodology and the tools used

during experiments are explained. Selection of design patterns and RT performance metrics are described. The experimental process consists of 7 steps: Each selected design pattern is applied to a different part of RTS. The RT performance metrics are measured in RTS implementations with and without design patterns. Both results are compared and discussed for each design pattern implementation. As the last case, a GOF design pattern is applied together with an RT design pattern to obtain an idea about the effects of combination of GOF and RT design patterns. The effects of design patterns on the performance of the RTS are stated at the end of each subsection.

In chapter 4, discussion and conclusions are presented. Some future work topics are suggested.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

In the embedded systems community, many believe that a functional language such as C is more powerful than C++. However, the problem is not C++ which is a powerful language. To use the power of C++ efficiently is the responsibility of the developer. Bad programming may result in insufficiency and/or inefficiency. C++ language includes C and more. C++ provides many features such as encapsulation, inheritance, polymorphism, exceptions, templates and the standard library [5].

Encapsulation, inheritance and polymorphism are essential properties of any object oriented language [18]. C++ is useful when it is used as an object oriented language (OOL). For example, if you use many switch-cases, if-else-if combinations to implement variation of objects, then using C++ will probably make your code even worse. On the other hand, if C++ is used as an OOL, if the system is analyzed and designed in an object oriented manner, the system will be much better than the previous system.

RTS software can be improved more by using design patterns. A design pattern describes a problem which is encountered many times, and finds a reusable solution to that problem [3]. Famous software design patterns are GOF patterns

and General Responsibility Assignment Software Patterns or Principles (GRASP). There are also RT design patterns specialized for RTS.

2.2 Software Design Patterns

In this section, the effects of some GOF and Real-Time (RT) design patterns on performance of RTS are discussed. Each design pattern is taken from [3] or [2]. The relation between and the effects of GRASP from [8] are also considered in the GOF and RT design patterns. Some similarities and relations between GRASP, GOF and RT design patterns are discussed in each design pattern section if exist. GOF design patterns can be thought to be object oriented software design patterns and RT design patterns can be thought to be prepared especially for RTS.

GOF patterns are classified in three groups. These are creational, structural and behavioral patterns. Creational patterns are related to the process of object creation. They are not examined in this thesis because there is no applicable area for applying these patterns in the available projects. Structural patterns mainly address the composition of classes or objects. To form larger structures, classes and objects are composed. Like creational patterns, structural patterns will not be investigated in this thesis with the similar reason in creational patterns. Behavioral patterns deal with the algorithms and the assignment of responsibilities between objects. They also describe the patterns of communication between objects. These are the patterns expected to affect performance significantly. Strategy, State and Observer patterns will be studied in the scope of this thesis. These patterns are classified as behavioral patterns. Each pattern is discussed with its intent, applicability, and structure. The aim of the pattern, the applicable areas/situations and/or problems and the overall Unified Modeling Language (UML) description of the patterns are given.

RT design patterns are prepared especially for RTS. [2] introduces about 50 RT design patterns for RTS. These patterns are designated as architectural design

patterns and classified in 6 groups. These groups are subsystem and component architecture patterns, concurrency patterns, memory patterns, resource patterns, distribution patterns and safety and reliability patterns. These patterns deal with problems which are essential for RTS.

There are hundreds of software design patterns. It is impossible to use and see the effects of every design pattern by applying them. This is not feasible for a thesis study. Moreover, most of them are not known widely by the software engineers because they address very specific problems in a narrow scope. In fact, the design patterns that will be used in this thesis are very famous and very common design patterns. They solve very common and often encountered problems in software engineering and RTS. They are widely used in object oriented analysis and design phases. It is also important to find a problem for the applied design pattern. An essential step of the study has been to find a suitable design pattern for a specific problem because design patterns are only useful if they are used correctly. The limitations about the number of available projects limit the number of available problems for applying design patterns. This is another important reason to select the design patterns to be used in this thesis.

Subsystem and component architecture patterns are about layering the RTS. They will not be discussed in this thesis because they are more about the architecture of the system and this subject is not in the scope of this thesis.

Concurrency patterns deal with the concurrency problems which are very important in RTS. Some of them deal with the priorities of tasks, some deal with message queuing, some with interrupts etc. However these patterns are not applicable for this thesis because the projects available for study within the scope of this study are already built in a real time operating system (RTOS). The RTOS already gives many services that are addressed with concurrency patterns. Therefore using these patterns is not meaningful.

Resource patterns are about preventing the system from crash by avoiding the system from deadlocks, limiting priority inversion and locking of resources. These are more about the safety and reliability of the system. There is nothing to do with execution time and/or memory performance. Therefore these patterns are also left outside the scope of this study.

Safety and reliability patterns, as the name indicates, deal with the safety and reliability of the RTS. They can be very useful in safety critical applications. However, they do not affect the performance of the system. Therefore, they will also be left outside the scope of the study.

Memory patterns deal with the memory problems in RTS. Usually memory is very limited in RTS. Consequently, this limitation increases the importance of these patterns for RTS. Memory patterns make the use of the memory more efficient. They deal with allocation, de-allocation and fragmentation of memory. They suggest efficient way to handle these problems. Smart pointer pattern (SPP), garbage collection pattern (GCLP) and garbage compactor pattern (GCMP) will be discussed from this group in this thesis. Other memory patterns are not expected to have any effect on performance.

Distribution patterns deal with the distribution of resources among multiple processors and/or systems. Most of these patterns are not applicable for this thesis because there are no available projects suitable for these kinds of patterns.

The observer pattern has also been studied in this thesis. It can be thought as publish and subscribe model. It is very similar to the observer pattern stated by [3].

Below, the design patterns to be studied are defined in detail.

2.2.1 State Pattern [3]

State pattern allows an object to change its behavior when its internal state changes. The object will behave as if it is changing its class. This pattern is used especially when there is a state dependency of an object. If an object is required to change its behavior at runtime depending on its state this pattern is very suitable. Moreover, if there are large numbers of conditional statements which are about the states of an object, this pattern should be used. State pattern solves these problems by implementing each state as a separate class. (See Figure 1)

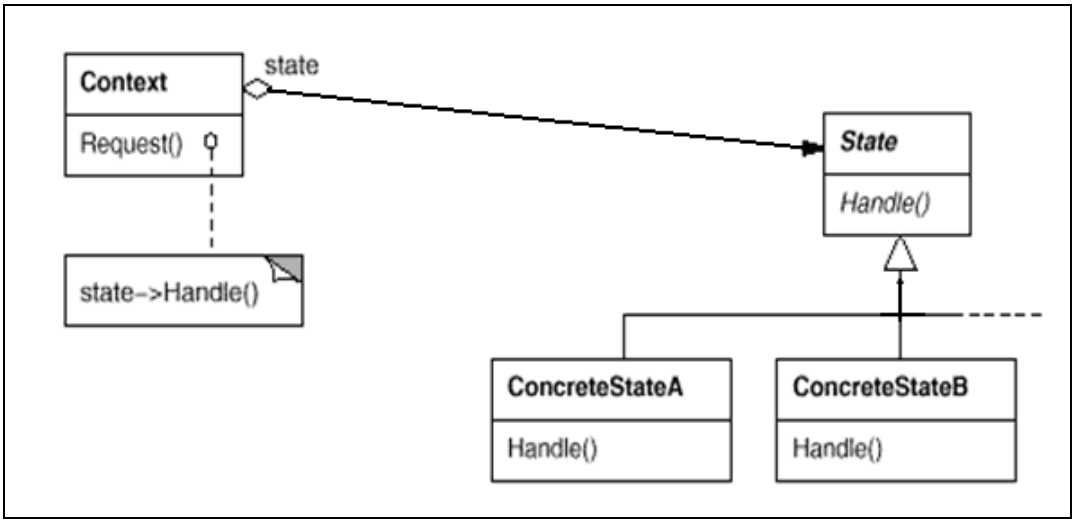

Figure 1: State Pattern (Adapted From [3])

[3] gives an example on TCP connection. TCP connection has states like listening, established and closed. In fact, the TCP connection should respond to events depending on its state.

State pattern promotes High Cohesion, Polymorphism and Protected Variations Principles stated by [8] just like Strategy pattern. The main difference between strategy and state pattern is that, state pattern focuses on the state changes of an object whereas; strategy pattern focuses on the type variations of an object.

In RTS, there are many state dependent objects. For instance in communication, there are states and at each state, different things are done for the same behavior.
To summarize, when there is a state dependency in an algorithm, state pattern should be used instead of conditional statements in order not to check every state every time. Since excess use of conditional statements decreases the performance of the system, it is a good way of increasing the performance of the system also. Finally, in RTS where performance is the most important criteria, this pattern becomes very important.

If a class has different states and behaviors, state pattern decreases the complexity of that class. As a result, weighted method per class (WMC) decreases. However, this pattern adds new associations to the objects, therefore coupling between objects (CBO) slightly increases. [1]

2.2.2 Strategy Pattern [3]

Strategy pattern is useful when there are different algorithms of the same behavior. The clients are not aware of the algorithm, they just know the behavior. By this way, algorithms are not coupled to the client and may be implemented independently. (See Figure 2)

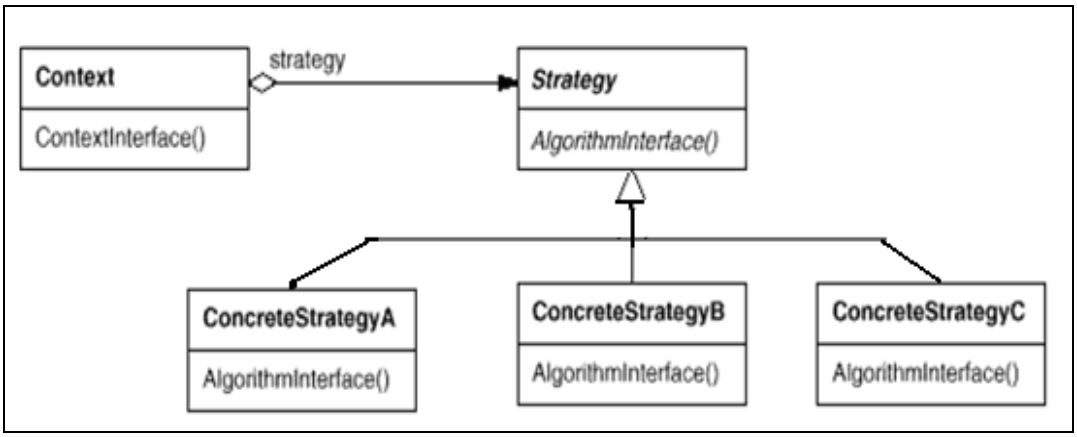

Figure 2: Strategy Pattern (Adapted From [3])

Strategy pattern has an emphasis on the same behavior with differences. For example, you can get a print on an A4 paper, also A2 paper, A1, A3, A5 and so on. Here the behavior is printing the page. All the printers will print the page, but there are differences in size. Therefore the printing stage will be different for different paper.

Moreover, the user is responsible for selecting the right paper. It is up to the user on which paper he/she will get a print. This example clearly demonstrates the use of strategy pattern.

- Strategy pattern eliminates conditional logic algorithm.
- It provides a way to configure a class with one of many behaviors.

Strategy pattern mainly promotes Polymorphism principle stated by [8]. Polymorphism deals with the alternatives of objects based on type. For instance, there are many conditional variations in many software programs. Most of them are about they type of the behavior of the object. Polymorphism principle states that, these variations should be handled as different subclasses so that these conditional statements are avoided.

Strategy pattern also have emphasis on Protected Variations Principle stated by [8]. Protected Variations decouples clients from the changes in the system being used. Strategy pattern also promotes OPC (Open Closed Principle). This principle states that, any software should be open for extension but closed for modification [10]. It is a very important principle in object oriented analysis and design.

Each created subclass is focused on its own job which in turn increases cohesion. Strategy Pattern promotes the High cohesion principle. High Cohesion principle is about the boundaries of duties of objects. The problem is how to keep objects focused on their duties, so that the code will be more maintainable. [8]. In RTS, usually there exist long switch-cases, if-else if else-if statements. These can emerge from the hardware or software dependency, or even from the customer requirements. For instance, customer may want a new behavior without eliminating the previous behavior about a model. Using extra conditional logic to solve the problem will be hard to maintain and decrease the performance. Since strategy pattern eliminates conditional logic, it increases the performance of the system in terms of execution times. Therefore, this pattern becomes more important in RTS where performance is the most important criteria.

Strategy pattern reduces the WMC and Response for a Class (RFC) metrics. On the other hand, it increases CBO. However, this pattern has improvement on software, such that it reduces complexity and inheritance related OO metrics. [1]. This is expected, because strategy pattern is an alternative way to subclassing.

2.2.3 Observer Pattern [3] [2]

This pattern is defined in both [3] and [2] with the almost same concepts and rules. It defines a way for one to many dependencies. When one object changes state, all the dependent objects notified and updated automatically. Both [3] and [2] states that this pattern is also known as “publish & subscribe”. (See Figure 3 & Figure 4)

Observer pattern can be used when:

- There are dependent objects and it is not known how many objects will be changed when an event or a change occurs on an observed object.
- When the object that notifies other objects has no information about what and how many are other objects.

Observer Pattern decreases coupling between objects. This property promotes Low Coupling Principle stated by [8]. This principle states that, the dependency between objects should be minimized so that any change in an object should not affect other related objects very much. It can be understood that, low coupling is very close to high cohesion. Usually, coupling and cohesion are inversely proportional properties. [1] informs that, observer pattern increases reusability and analyzability of the software, whereas, it affects software error-proneness in a negative way.

In real time operating systems, there are some services for the system such as using message queues. This service can be used together with this service and a fast and well designed communication between objects can be established. This pattern should be used for performance considerations also, because, it enables to decouple subscriber form publisher and it will allow using only the needed system resources.

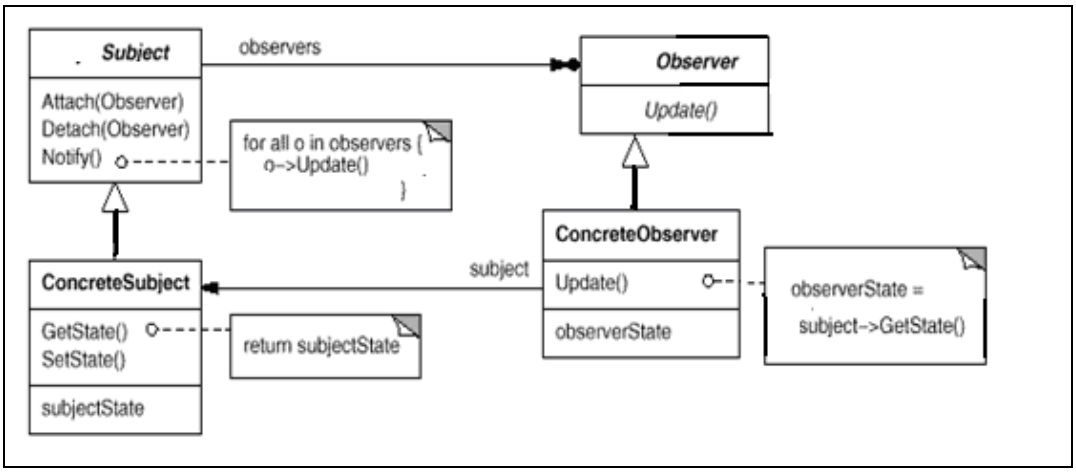

Figure 3: Observer Pattern (Adapted From [3])

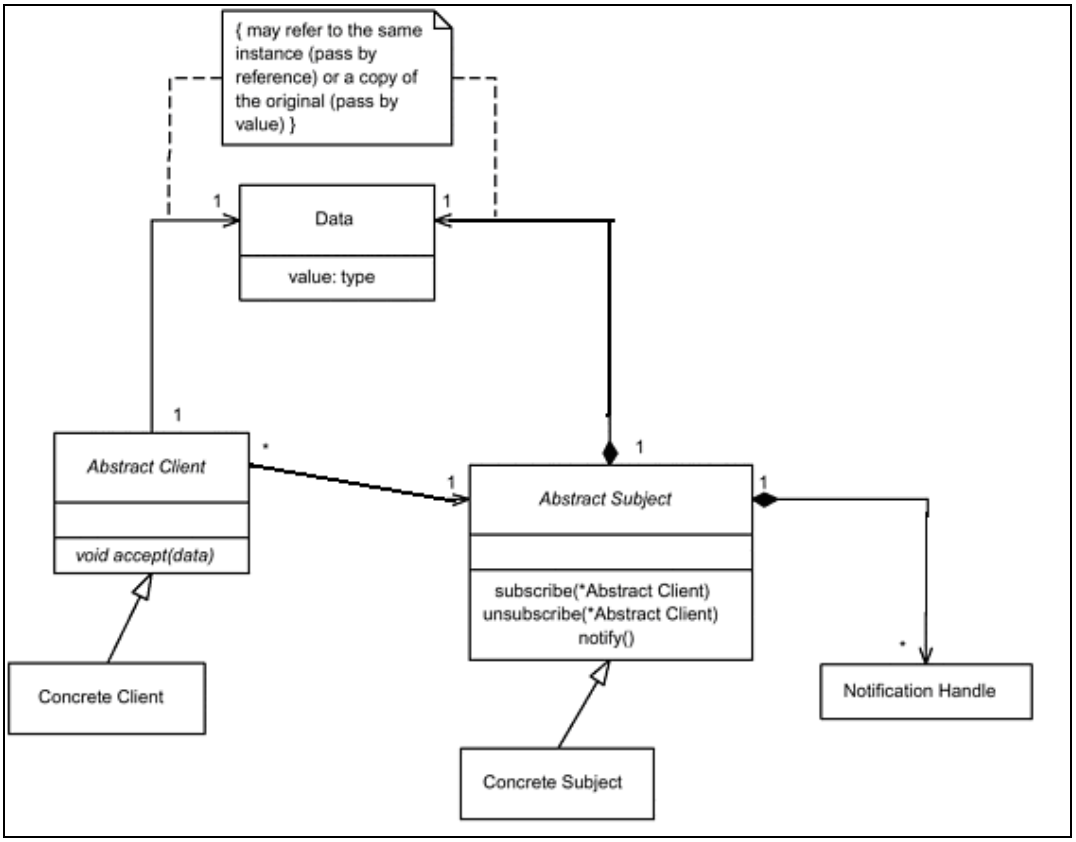

Figure 4: Observer Pattern (Adapted From [2])

2.2.4 Smart Pointer Pattern [2]

SPP is a RT design pattern presented in [2] in the group of memory patterns as stated before. Using pointers in the development of software is usually very useful and necessary. However, there may be many problems about pointers in programming. SPP mainly deals with the pointer problems in RTS. These problems can be listed as memory leaks, uninitialized pointer, dangling pointer and pointer arithmetic defects. Memory leaks are the problems occur when the pointer is destroyed before the memory is released. There is no way to de-allocate the memory referenced by the pointer after the pointer is destroyed. This problem results in rejecting of memory requests because the memory is wasted in time. Uninitialized pointer problem occurs when the pointer is pointing an object where the object is not allocated properly. Dangling pointer is a pointer which points to a de-allocated memory. Pointer arithmetic defects occur when there is an inappropriate iteration on an array. For instance, pointing 3.element of an array as if it is the 10. element.

SPP solves these problems by creating an object from the pointer. In other words, the pointer is changed to an object containing previous information and more. By this way, the pointer becomes a smart pointer and can be useful to avoid the problems stated above. [17] states the benefits of SPP in terms of maintenance. WMC, depth of inheritance tree (DIT), CBO, RFC, method hiding factor (MHF) and coupling factor (COF) are improved by this pattern. SPP makes the system more maintainable. [17].

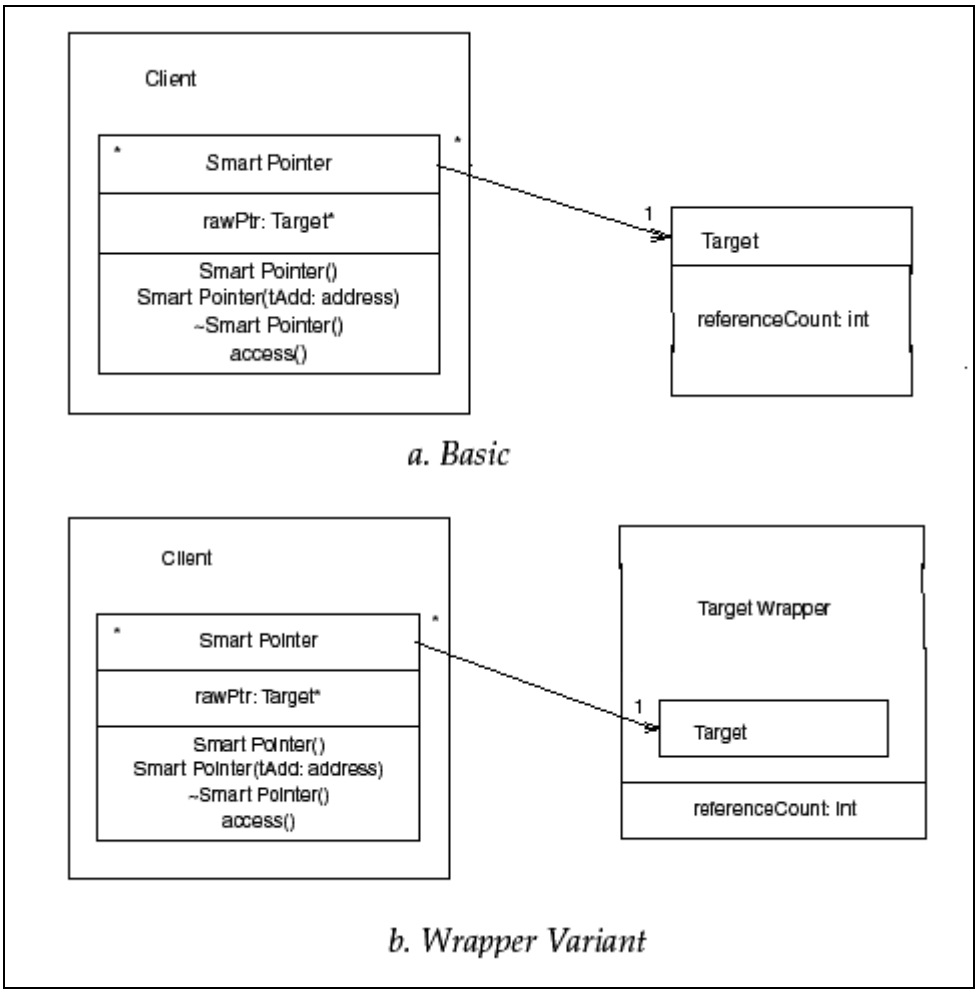

Figure 5: Smart Pointer Pattern (Adapted From [2])

In Figure 5 that depicts the Smart Pointer Pattern structure, the client is the user of the pointer. The actual pointer is stored in the smart pointer as rawPtr. There are also constructor, destructor, and access operations for smart pointer. The Target is the service provider for the client where has a reference count attribute. This attribute is used to track the number of smart pointers that reference the target. Each construction of a smart pointer increases this attribute and vice-versa. When there is no referencing smart pointer, the target is destroyed. This can be found easily from the reference count.

2.2.5 Garbage Collection Pattern [2]

Sometimes the programmers forget to de-allocate memory which is previously allocated. This pattern takes the responsibility of de-allocating the previously allocated memory and saves the programmer from explicitly de-allocating the memory. There are different kinds of algorithms to achieve garbage collection. The common properties of all algorithms are to detect the garbage objects and to make them usable again, so that the memory is used more efficiently.

[20] states seven kinds of collection algorithms. Reference counting is the first and basic one. In this algorithm, a reference counter for the object is used to track the number of references to the object. When the total number of references is zero, this indicates that the object is no longer used. This algorithm is used previously in SPP section by [2] and also stated in the proposal of [12]. It is famous with its simplicity and is one of the earliest algorithms. The second algorithm is tracing collectors. It is also known as mark & sweep algorithm. The living objects are marked and all the unmarked objects are thought to be dead, so they are moved to the main memory to be used again when necessary. [2] uses this algorithm for garbage collection. Again it is also stated by [12]. The third one is copying constructor algorithm which is also stated by [12]. There are also compacting, generational, adaptive and train algorithms. There may be found more collector algorithms in the literature. However, a complex and detailed algorithm is not in the scope of this thesis, Therefore, garbage collection pattern will be implemented using reference counting algorithm (RCA). Note that, the efficiency and requirements change, so there are different algorithms but they all have common important properties as stated above.

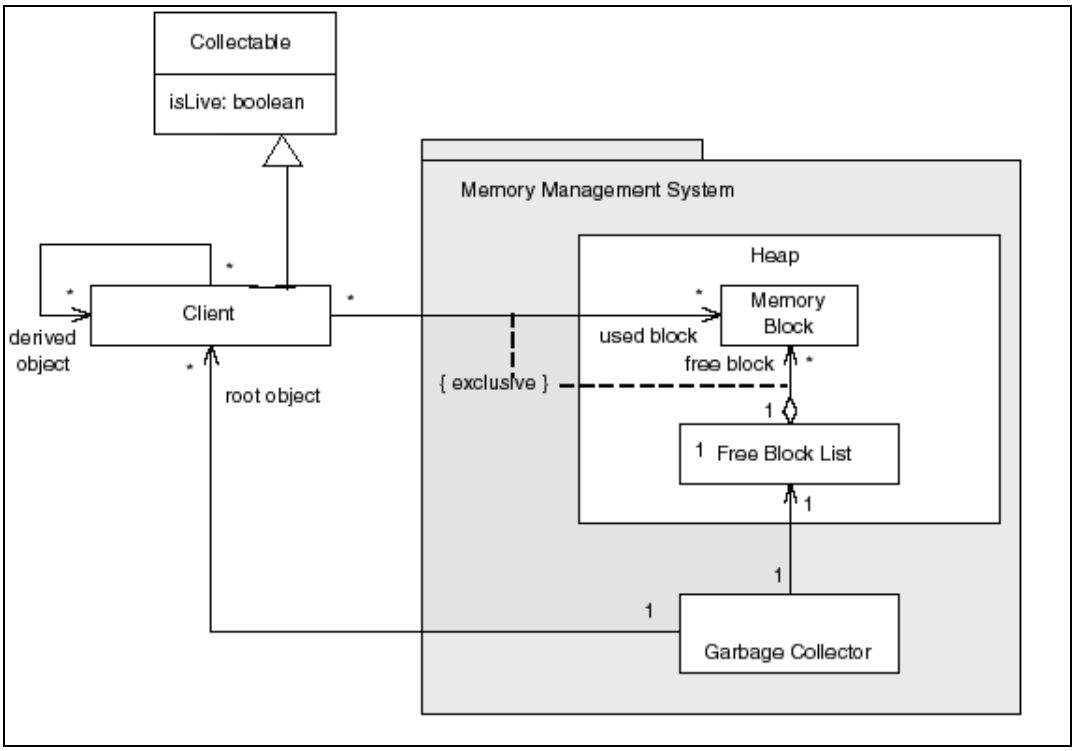

Figure 6: Garbage Collection Pattern (Adapted From [2])

Figure 6 shows a typical block diagram for GCLP. In the designed memory management system (MMS), all objects are created as live objects. When there is low memory problem or an explicit for GCLP, the collector starts to look for the objects which are no longer used. It adds these objects to the free list using the RCA. For instance, if the object has no pointers referencing itself, then it is marked as free and is moved to the free list. The collection process checks every object for its activeness. Then, the required memory (or the object), is found through these objects in the free list if possible.

2.2.6 Garbage Compactor Pattern [2]

GCLP does well about the de-allocation problem. However, there still exists a major problem about memory. This is fragmentation. Memory fragmentation does not have an effect on memory utilization but it becomes important when there is a need for an amount of memory which is larger than the biggest free memory block but smaller than the total amount of free blocks. In this case, the free memory can not be used [7]. In fact, GCLP does nothing about fragmentation. The allocated memory is not forgotten, always de-allocated if unused. It is given to the free memory list, but the memory is fragmented.

GCMP solves this problem also. This pattern is similar to GCLP but achieves more. The major difference between the GCLP and GCMP is that this pattern also helps to de-fragment the de-allocated memory. However, GCLP just deals with the de-allocation problem. In fact, fragmentation of memory may result in the rejection of memory requests even though the total amount of available memory is sufficient for the request. Considering the fragmentation problem in real time systems, this pattern becomes more important. In this thesis, GCMP will be based on the GCLP stated in section 2.2.6.

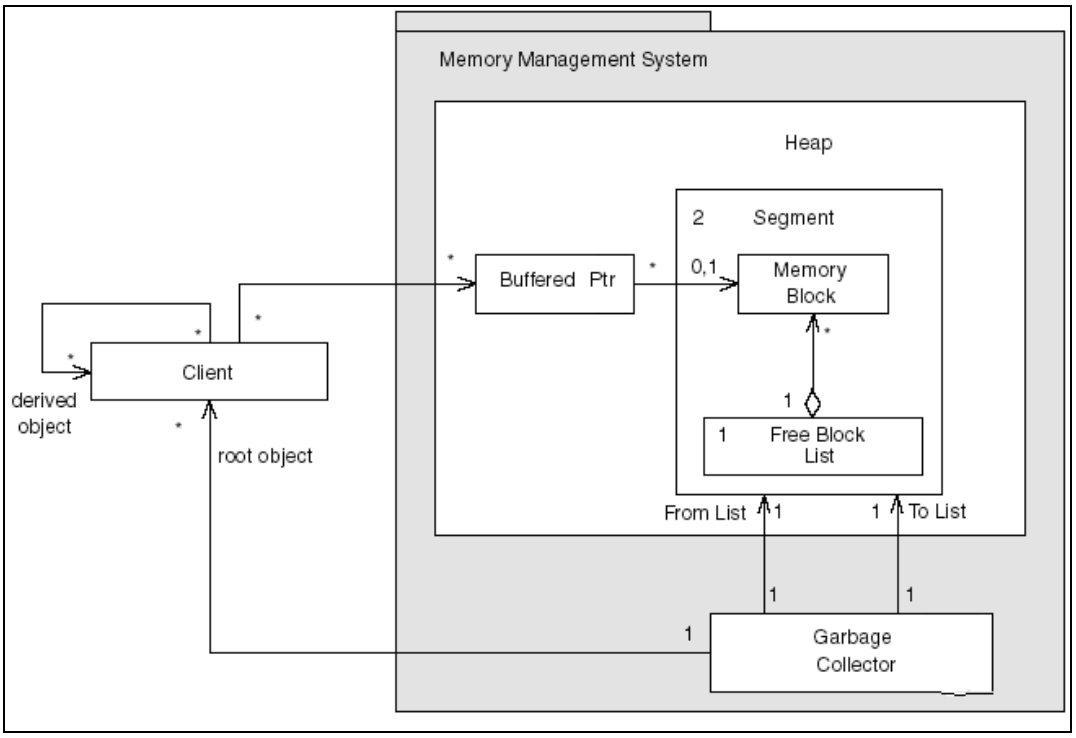

Figure 7: Garbage Compactor Pattern (Adapted From [2])

In Figure 7, the client is the object that allocates memory defined by the user. There is a bufferedPtr between the memory block and the client. Segments and bufferedPtrs are included in the heap. Garbage compactor uses two segments, one for providing memory and the other for compacting the de-allocated memory blocks.

2.3 Performance Measurement of Real Time Systems

2.3.1 Introduction

One of the major factors that makes RTS different than other systems is the performance criteria. A RTS should conform strictly to performance requirements. Otherwise, it fails. However, this does not mean that a RTS should be very fast.

Being fast or slow is a relative concept and depends on the requirements and designed system. This means that it should be as fast as required, no more or no less. The important point is that, it should be reliable on time critical events. Timing is very critical in every RTS [6]. It can be easily seen that performance of a RTS can be considered as the heart of the system. Consequently, measuring the performance of the system becomes essential in RTS. As a result, knowing how to measure the performance of a RTS becomes important.

2.3.2 Performance Testing

Performance of a software system can be determined through performance testing. Performance testing is to find out the speed, scalability and or stability of a system. There are also load test, stress test and capacity tests which can be considered as categories of performance testing. Load test is done to see the behavior of the system under normal and peak load conditions. Sometimes the load conditions are beyond peak load conditions. This test can be considered as stress test. It can be used to find out bugs that appears when there is high load conditions. Finally, capacity tests are done to determine the number of users that a system can support and still achieve its performance criteria [9].

Another definition for performance testing is made by [4] which emphasizes the validation of the system according to its performance and capacity. Software performance testing and evaluation can be handled by measuring processing speed, latency, response time, throughput, availability, reliability, scalability and utilization of the system. The first four are prepared to show the speed of the system and they are very close and dependent to each other. Processing speed is the time elapsed for performing certain kind of a job, which is directly related to response time. Latency is the delay between processing events which is also measured with time. Throughput is also the time to process an action. Utilization shows the consumption of various system resources such as CPU and memory.

Availability shows how much a system is available in terms of percentage of time. For instance, %99 availability means that, the system is performing its work 99 hours in every 100 hours. Reliability is a measure that shows how much an output can be trusted from a system. Scalability shows the boundaries of the system, that is, how large the system can be constructed, or how widely system parameters may vary while all its principles and characteristics remain valid. . [4]

Although these tests are prepared for web applications they can also be applied for some RTS. For instance, number of users in a real time mobile communications is an important factor that affects performance. Load, stress and capacity testing may give considerably useful information about the performance behavior of these kinds of RTS.

2.3.3 Performance Metrics

In general a metric is a property to measure in any system. In this study, the focus is on metrics for software systems. Metrics should be simple and precisely definable. It should be clear how the metric is to be evaluated. Performance of a system is one of the fundamental qualities of the system. Performance of a software system includes mainly response time, throughput and speed of the system [11]. If identified and used correctly, performance metrics are very useful for providing information about how the system is performing its performance requirements. Processor utilization, memory consumption, memory available, memory utilization and disk utilization are some other metrics suggested for performance assessment [9]. These metrics can be used for RTS.

For mobile devices, throughput and latencies in the system are also considered as performance metrics [13]. System-User response time is given as a typical performance metric in [4]. Moreover, task throughput is another common

throughput metric. Memory, cache, CPU and disc utilization are included in the Utilization Metrics.

2.3.4 Real Time Software Performance Metrics

[6] states four fundamental performance metrics for RTS. These are performance profiling, A-B timing, response to external events and RTOS task performance. Memory fragmentation is also considered as an issue effecting performance of an RTS. Memory consumption, memory available and memory utilization are also stated as performance metrics in [9].

Performance profiling metric gives an idea about functions in a RTS. It indicates how much time is spent on each function during run-time. Hence, the developers can see the problems of functions that are out of performance goals (i.e. slow compared to the performance requirements) and fix the problems. The aim is not to speed up all functions because the effect of the functions on the overall performance of the system changes from one to another. It is to find out the functions which are slower than required and affect the system more than other functions [6]. This metric can be considered as a special case for measuring execution time. It is specialized for measuring just the execution time for functions. Therefore, this metric can also be considered as a kind of response time and throughput metric as stated by [4]. Response to external events metric measures the time between an occurrence of external event and the starting instance for a response of RTS (e.g., interrupt latency periods).

RTOS task performance metrics can be divided into two categories: Firstly, task deadline performance measurements give the time for each task in a real time multi tasking environment to reach its deadline after a triggering event occurs. Secondly, task profiling performance measurements are similar to performance profiling but this time the metric is not function based but it is task based. It finds

out the tasks where the system spends much of its time [6]. These metrics are again kinds of execution time metrics but unlike performance profiling, these are specific for task performance. They measure task execution time. In fact, they are subsets of response time and throughput metrics stated by [4]. Moreover, they are also related with process utilization metrics stated by [9] since the response time of a task depends on the process utilization of a processor.

A-B timing metric (ABTM) is very important in measuring performance of RTS because it gives the time to go from one point in code to another in runtime. It helps to verify the required timings of code pieces in a system [6]. It can be directly considered as response time of some block of code. Therefore, it is a kind of response time and throughput metrics stated by [4]. It is also related with process utilization metrics stated by [9] since the response time of a block of code depends on the process utilization of a processor. This metric is not bounded to a function or a task, it is more general. Therefore, it can be more useful.

Memory fragmentation affects the system performance very much. As the memory becomes fragmented, it takes more time to find a memory block in case of a memory allocation attempt [6]. Memory fragmentation will be measured with the memory fragmentation metrics (MFM). [15] states different kinds of MFM such that smallest-biggest block metric (SBBM), free block metric - average size (FBM-AS), internal fragmentation (IF) metrics. SBBM gives the maximum amount of memory that will always be successful upon a memory request and FBM-AS gives the average memory size for available blocks in memory. IF gives an idea about the amount of memory which is wasted when a memory request is served by a free block larger than requested.

Memory consumption, memory available and memory utilization all give an idea about the performance of the system [9]. For example, memory usages of each task can be used as a metric for performance in an RTS. Memory consumption is about task memories, the generated object file sizes, initialization and execution of

systems. Task memories directly effect the consumption in RAM and object file sizes of the files directly affect the flash memory consumption. Both are important in RTS. Task memory metric (TMM) and object file size metric (OFSM) can be considered as subcomponents of memory consumption, thus they are important also. Moreover, memory is consumed at initialization and execution of a system. Investigation of a detailed memory consumption work includes also these states since there maybe quite different amount of memory consumption between initialization and execution. Therefore, initialization memory metric (IMM) and execution memory metric (EMM) should also be considered as subcomponents of memory consumption metrics (MCM)

2.3.5 Difficulties of Measuring Software Performance

There may be several kinds of metrics that measure the speed of the system. Applying the right metrics to the system is the choice of the engineer. [4]

[13] states that both hardware and software architecture of the system determine the overall performance of the system. Hardware configuration which includes the processor speed, bus speed, cache configuration, number of processors and type of processors etc., affect the system performance. Software complexity, coupling between software components, task structures etc. are other factors that affect the performance of software. It can be seen from this statement that, measuring software performance is not so easy. There are many tasks using the same resources as the measured task and/or function. Moreover, there are other modules in the system which may be in interaction with the measured part of the system. These effects can be thought as noise and they are inevitable. However, this noise can be minimized by suspending, if possible, all the tasks not related to the measured task and/or function. This will give more accurate results about the performance of the task/function or code block being measured.

2.4 Conclusions

Performance is more meaningful for RTS than other usual systems, because the most important objective of RTS is to meet performance criteria. Therefore, measuring performance of RTS becomes crucial. It is important for the developers to see whether the performance requirements are being met or not. As a result, choosing the right performance tests and right metrics are the key to have a correct result.

Performance profiling, ABTM, response to external events, RTOS task performance (if exists), memory fragmentation, memory consumption, memory available and memory utilization can be used to measure the performance of the real time software system.

In this study, different RTS will be evaluated in terms of performance using ABTM, memory consumption metrics. Memory fragmentation will be also used as a performance metric where applicable. The reasons for using these metrics are explained in section 3.2. Comparing performance of three systems should not be confused with measuring the performance of these systems absolutely. The three systems will use the same platforms, same hardware and same configurations. There will be only difference in software implementation, but all systems will behave identically, because they will realize the same functionality. Since the aim is to compare the systems in terms of performance, it is more important to assess relative performances.

The aim of this thesis has been stated as finding out the effects of design patterns on real time software performance. Before starting the discussion of the experiments performed, it will be better to summarize our predictions of the effects of software design patterns here. Table 1 shows the expected effect of design patterns on software performance. These predictions are all based on the literature review presented in the previous sections.

Table 1: Expectation for Effect of GOF and RT Design Patterns on Performance

Pattern Name	Expected Effect On Performance Metrics		
	ABTM	Memory consumption	Memory fragmentation
1. Strategy	Decreases	NC	NC
2. State	Decreases	NC	NC
3. Observer	Increases	NC	NC
4. Smart Pointer Pattern	Increases	Decreases	Increases
5. Garbage Collector Pattern	Increases	Decreases	Increases
6. Garbage Compactor Pattern	Increases	Decreases	Decreases

Increases: This term indicates that, the related metric will increase in terms of quantity.

Decreases: This term indicates that, the related metric will decrease in terms of quantity.

NC (No Change): This term indicates that, the related metric will not change in terms of quantity.

CHAPTER 3

EXPERIMENTAL WORK

3.1 Description of the Projects

Two different real time communication systems developed in ASELSAN Inc. are used as platforms for experiments during this thesis. The existing projects are mostly projects on which staff is currently working on; as such, a "snapshot" is taken and used for these experiments. Thus, the code used for these experiments has been "inspired" from actual projects. Some code was written from scratch to apply RT design patterns. These codes were used to apply design patterns and measure the RT software performance.

3.2 Experimental Methodology

In this study, three systems for each GOF design pattern and two systems for each RT design pattern are compared in terms of performance using ABTM, memory consumption and memory fragmentation metrics, wherever applicable. The aim is to reveal whether or not severe performance degradation in comparison to C code results when the object oriented language C++ is used together with design patterns. The programming languages were different but the responsibilities of the software were the same in these systems. What is measured is the performance

difference between these systems. These systems realize the same functionality but the functional structure may be different, because each system is constructed by a different programming language and/or design. In fact, one system is in C programming language and the others are in C++. The design patterns investigated are object oriented and non-object oriented real-time design patterns. In this context, some GOF design patterns and some RT design patterns are applied to the system programmed in C++. Then all the systems are compared in terms of performance.

As stated above, in all comparisons, three implementations are involved: the first one is the original system, the second is the system implemented with OOL but without design patterns and the third is the system with OOL on which design patterns are applied. Note that RT design patterns are only investigated in two systems which are OOL with RT design patterns and a system without design patterns. It is an important fact that the measurements are made to compare the systems; that is, relative rather than absolute measurements are the subject of study. Since we deal with two or three systems which have the same hardware, the same compiling and building environments, the same real time operating systems, the same tasks and processes running, then the only difference in these systems is whether or not design patterns are applied. Consequently, the relative performances give meaningful information about the effect of applied design patterns.

The structure of the code also affects the performance metrics. For instance, the number of conditional statements before applying state pattern directly affects the execution time and memory consumption. The relative differences of performance metrics between the cases with and without design pattern are affected by the number of conditional statements and how the system is used by the client. For instance, if the client sends a signal that will be consumed in the system in the first conditional statement, it will have a better execution time relative to the system with consuming a signal in the last conditional statement. In fact, the arrangement

of conditional statements and the arrangement of cases that will use these conditional statements affect directly the execution time. Therefore, the execution time is also client code dependent. For instance, as the load on the client code increases, the execution time also increases. In the experiments, for a fair comparison, the client codes in the comparison of with and without design pattern cases are the same. However, performance metrics may be measured differently in another system with another client code. The compiler and compilation arguments such as optimization flags are other factors on the performance metrics. For example, optimization flags may improve the memory consumption metrics at the expense of execution time and vice versa. In this work, GNU compiler is used for compilation and no optimization flags are used. To conclude, the performance metrics would be affected by factors such as the design of the system, the client code and the compiler.

During performance metrics measurement, in each case (software with and without design patterns) the measurement is handled after a restart of the system to clear the effect of previous tests and make them independent. Moreover, there are many services in the system, but most of them are not used during the experiments so that the RTS is just working for the experiments and some critical services needed for the system.

In the following subsections, the metrics and patterns used in this study will be discussed.

3.2.1 Selection of Design Patterns

There are many conditional statements that are encountered in real time software development. If the conditional statements are about types of objects or elements, they will probably be used in every step of a layered architecture. Among GOF design patterns, the strategy pattern is used to eliminate the conditional statements

based on type. The state pattern is used to eliminate the state dependent conditional statements just like strategy pattern. The observer pattern is used to implement publish and subscribe rule. This pattern is applied together with real time operating system services such as message queues. Note that, observer pattern is also used as a RT design pattern. Among other RT design patterns, SPP is used to handle the problems about pointers. This pattern is expected to slightly decrease the execution time but increase the performance of memory since it alleviates the problems about pointers. GCLP and GCMP are used to collect the memory blocks that are allocated but will no longer be used. GCMP is also used to prevent memory fragmentation. It is expected to bring some execution overhead to the CPU while increasing memory performance.

These patterns are selected according to their applicability for the available projects. Moreover, the selected patterns are used frequently in software developing and they can provide solutions to most of the problems in real time software systems.

3.2.2 Selection of Performance Metrics

In this part, the metrics that will be investigated will be discussed. As discussed before, performance profiling, ABTM, response to external events, RTOS task performance, memory fragmentation, memory consumption, memory available, memory utilization are some performance metrics that can be used in measuring performance of RTS. All the detail about these metrics were discussed in the Performance Measurement of RTS section (Sec 2.3)

Most critical metrics for performance measurement are related to execution timing. The metrics related to execution time mentioned above are performance profiling, A – B timing, response to external events and RTOS task performance. All of these metrics give an idea about execution time of some block of code. However,

since we deal with two systems with different programming languages, it is not suitable to use performance profiling. Instead, it is more useful to use ABTM, because it gives the elapsed time for realizing a specific functionality. It is a more meaningful metric for the present study because it will be easy to compare systems by this metric in terms of execution time. Response to external events metric is not related to the programming language but it is related to the performance of RTOS. Since the platforms will be the same, there is no need to measure this metric. RTOS task performance metric is also more specific than ABTM and is less meaningful for comparing systems with just language differences.

Another important issue in RTS is memory. The metrics related to memory are memory fragmentation, memory consumption, memory available, and memory utilization. All of these metrics give an idea about the memory performance of the system. Memory consumption, memory available and memory utilization metrics all can be used to compare systems in terms of memory performance. In fact, memory consumption and memory available are almost giving the same idea; they are complement of each other. Moreover, memory utilization is a function related to memory consumption and memory available. Therefore, it is also not necessary to use this metric. Choosing memory consumption fits well with the aim of comparing systems. TMM, OFSM, IMM and EMM will be used to measure memory consumption. These four metrics will give quite a good idea about memory consumption.

MFM are not related to GOF design patterns, but are strictly related with RT memory design patterns. The major difference between the two RT design patterns GCLP and GCMP is that, GCMP solves the memory fragmentation problem whereas GCLP does not. It should be realized that, memory fragmentation is not related to whether the program is written in C or C++. Memory fragmentation is about the method used in allocation and de-allocation of memory blocks. While allocating and freeing the memory, sometimes the used memory can not be used

again because of fragmentation even if it is de-allocated. This problem can be handled by controlling the memory allocation/de-allocation and this problem is the responsibility of RT memory design pattern GCMP. In general, RT memory design patterns affect memory fragmentation in positive or negative ways, whereas GOF design patterns do not. Therefore, MFM will be used in GCLP and GCMP implementations. SBBM and FBM-AS will be used to measure the memory fragmentation as a type of MFM. These two are enough to give detailed information about the fragmentation of memory, because the maximum amount of free block size together with the average free block size fit well with the measure of how much the memory is fragmented. IF is a much more detailed metric to measure and it is not considered in the scope of this thesis.

As a result seven metrics will be investigated to measure the performance of systems. These are:

- ABTM for execution time,
- TMM, OFSM, EMM and IMM for memory consumption,
- SBBM and FBM-AS as MFM for memory fragmentation.

ABTM metric will be measured by using a shell connected to the running real time software system. The shell is a service provided by Wind River [19] and Tornado [16] environments. At the starting point of a job to do (may be a number of steps), a timer will be started and at the end the timer will be stopped. The elapsed time will be found by the value of the timer and this result will be stored in an array on each trial. The timer is incremented by a clock. However, usually the clock resolution is not sufficient for a precise measurement in a single execution. Therefore, the job is processed several times in a loop and the number of entries to the loop is saved. The number of entries to the loop is determined according to the linearity of the results. For instance, if the number of entries is 50 and the elapsed time is 405ms (milliseconds) and if it is 100, corresponding is about 910 ms, and then the measurement becomes linear (see Fig. 8). This number

of entries is enough but usually a higher value is selected to achieve a high resolution of the execution time. Moreover, the experiment will be done several times to obtain reliable information on the metric (i.e. 1000 times). The maximum, minimum and the average values of the metric will be saved. The standard deviation of ABTM will also be calculated and used to have an idea about the stability of the execution time measurement.

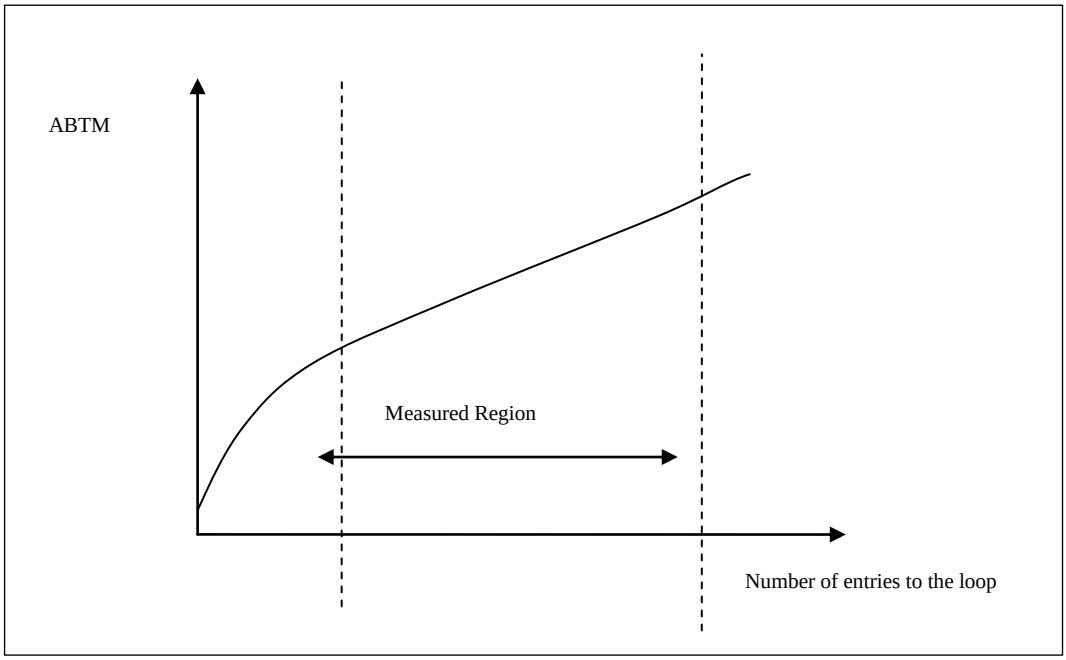

Figure 8: ABTM Measurement Linearity

Like ABTM, one of the memory consumption metrics will also be measured by using a shell connected to the running RTS. The values of the tasks' memories can already be seen using some properties of the RTOS. This will give the TMM. Object file sizes will also be measured as a kind of memory consumption which can already be found by just obtaining the size of the corresponding object file from its properties, provided by the OS. This will give OFSM. Moreover, EMM and IMM will also be measured using a shell connected to the system. The free

RAM size will be saved before and after an initialization and execution because the free memory size can also be affected by other modules running on the system. Therefore, EMM and IMM can only be measured when there is a great effect on memory consumption. As a result, these metrics will be used in the implementation of memory patterns only. Otherwise, it will not give healthy results. These values will give valuable information about the memory consumption. Note that some RTS may not have flash memories. The selection of metrics changes from one system to another. As [4] states, it is the role of the engineer to select the right metrics for the system because each system may require different kind of metrics.

MFM will also be measured by an interface implemented for the statistics of fragmentation. The outputs will give an idea about memory fragmentation. SBBM and FBM-AS metrics will be measured using this interface. These measurements will give detailed information about fragmentation. From this information, for instance, when the number of memory blocks is constant and the average size of memory blocks is decreased, then it can be understood that the memory is more fragmented. This kind of measurement will be used to see whether applied patterns help to de-fragment the memory or not.

3.3 Tools Used In the Experiments

As development tools, Wind River Workbench [19] and Tornado [16] environments are used to build the systems. These tools are famous among the RTS community and have C/C++ compiler and linker. All the software that will be downloaded to the target RTS will be compiled and built in these environments. The IBM Rational Rhapsody [14] environment will be used to design the projects when applying design patterns. However, all the code will be written in Wind River and Tornado environments.

To compare the systems, the selected metrics will be used within the same environment for the same projects to see the effect of each design pattern that is applied. All time measurements are given in microseconds and all memory measurements are given in bytes, unless explicitly specified otherwise.

3.4 Experimental Process

3.4.1 Step 1: State Pattern

3.4.1.1 Introduction

State pattern is applied to a call manager system. There are different states of the system and the system behaves differently according to its state for the same operations. For instance, the system can be in idle state, calling state, active state, ringing state or releasing state. In fact, the system can process an establish call request in idle state, but it is not possible at active state, because there is already an ongoing call. This information shows that state pattern is very suitable for call manager system.

3.4.1.2 Before Applying State Pattern

In the current system, the states are checked in every operation. This results in using conditional statements, as shown in Listing 1.

```
/* signals to call manager */
case CM_EstablishCallRequest:
{
            if  ( callState == ACTIVE )
            {/* do nothing, return error */
            }
            else if ( callState == RINGING )
            {/* do smt for ringing state */
            }
            else if ( callState == IDLE )
            { /* establish call for idle state  */

                        /*
                         * Other jops to do
                         * ....
                         */
            }
            else if( callState == RELEASE )
            {/* *****  */}
                  /*
                   * other states..
                   */
            break;
}
case CM_ReleaseCallRequest:
{
      if  ( callState == ACTIVE )
      { /* end call for active state  */

                  /*
                   * Other jops to do
                   */
      }
      else if ( callState == RINGING )
      {/* do smt for ringing state */
      }
      else if ( callState == IDLE )

      {/* do nothing, return error */

      }

      else if( callState == RELEASE )

      {/* *****  */

      }

                  /*

                   * other states..

                   */

      break;

}
```

Listing 1: Pseudo Code Before Applying State Pattern

There are signals coming to the call manager system and at each signal, the state is checked because the operations are changing from one state to another. (See Listing 1)

3.4.1.3 After Applying State Pattern

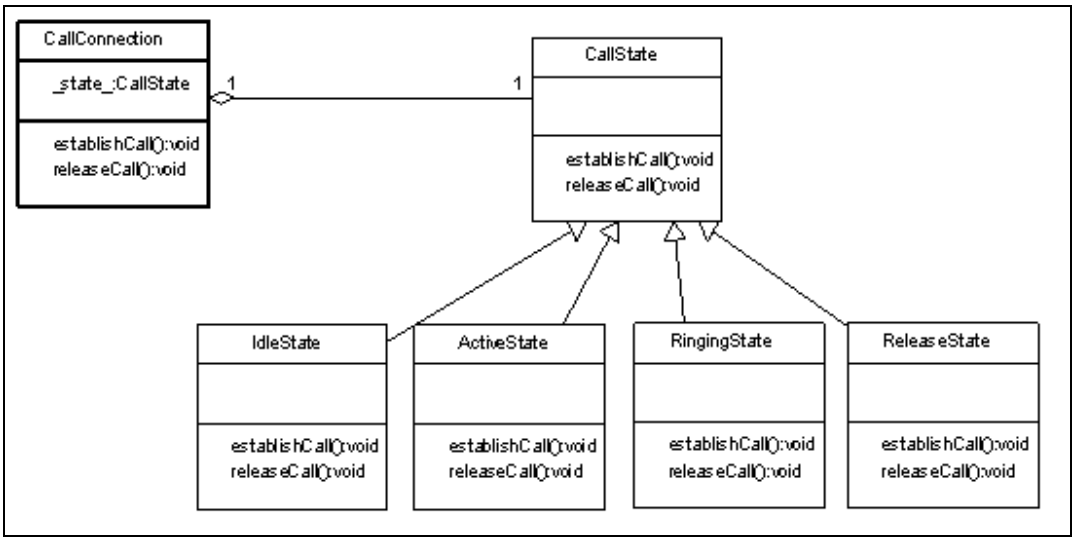

Figure 9: Pseudo UML Diagram After Applying State Pattern

```
/* signals to call manager */
case CM_EstablishCallRequest: {
   callConnection->establishCall(..);
   break; }
case CM_ReleaseCallRequest: {
   callConnection->releaseCall(..);
   break; }
```

Listing 2: Pseudo Code After Applying State Pattern

State pattern applied as shown in Figure 9 solves the if, else-if problem. It eliminates the conditional logic depending on the state of the call. The client is not aware of the actual state, the current state is stored in the CallConnection class, but it is changed by the call state objects.

3.4.1.4 Results, Comparison and Discussion

Table 2: ABTM for State Pattern

Cases	Number of Tests	Maximum	Minimum	Average	Change (%)		Std. Dev.	Change (%)
C	200	408,3	391,7	397,3	0		3,2	0
CPP WODP	200	427,1	406,3	413,2	4,014		3,9	21,927
CPP WDP	200	408,3	389,6	395,8	-0,361		3,8	18,902

In Table 2, C case is the original project programmed in C language. CPP WODP is the project with C++ language with out applied any design pattern, where WODP stands for “without design pattern”. CPP WDP is the project with applied design pattern where WDP stands for “with design pattern”. Note that the maximum, minimum and average values are in microseconds.

It can be seen that, CPP WODP has increased ABTM metric by %4, whereas CPP WDP has slightly decreased ABTM. Coming to standard deviations (std. dev.), CPP WDP can be seen more stable than CPP WODP but less than C case. Since, ABTM is a metric for execution time, this result shows that the state pattern can overcome the execution time increase with CPP language.

Table 3: TMM for State Pattern

Cases	Size (bytes)	Change (%)
C	440	-
CPP WODP	416	-5,45
CPP WDP	412	-5,46

Table 3 shows that, again, CPP WDP has the smallest TMM. This shows that, CPP WDP has decreased the TMM by around %5 compared to C case. It is also slightly smaller than CPP WODP case.

Table 4: OFSM for State Pattern

Cases	Size (bytes)	Change (%)
C	30656	-
CPP WODP	22016	-28,18
CPP WDP	22692	-25,98

Table 4 shows that CPP WDP decreased the OFSM by around 26%. It is slightly worse than CPP WODP case, but they are very close.

Table 5: ABTM for Initialization for State Pattern

	# of Tests	Maximum	Minimum	Average	Std. Dev.
CPP WDP	20	64,6	54,2	59,1	4,0

Initialization overhead is the overhead for CPP cases. The objects for state pattern should be created once, most probably this done at the initialization of the system.

There is some execution overhead for creation of objects. The required objects creation time is recorded in Table 5 above.

To summarize, the state pattern has decreased ABTM, TMM and OFSM. It has no effect on MFM, either increase or decrease; therefore the MFM has not been recorded for this pattern. In CPP WODP case, ABTM is higher than C and CPP WDP cases, whereas, CPP WODP and CPP WDP are very close to each other in TMM and OFSM. This information shows that, state pattern has a more effect on ABTM, but it has no essential effect on TMM and OFSM. It can be understood that, state pattern has increased execution time performance, since ABTM is an execution time metric. It is also obvious that, the reduction in TMM and OFSM are because of programming in CPP language. Coming to the initialization overhead shown in Table 5, the time required to create objects is much smaller than the test done for state pattern (it was around 400 micro seconds). Therefore, depending on the requirements, this initialization overhead can be ignored in most of the RTS.

From the performance aspect, the state pattern has almost a negligible improvement on ABTM relative to C case. However, together with CPP, it has increased the memory performance very much relative to C case. As a result, instead of performance degradation, there is some improvement in overall performance of the RTS by applying this pattern. Moreover, according to [1], this pattern positively affects the design and reduces complexity from the quality view of the system.

Note that, as stated before, the number and arrangement of conditional statement directly affect performance metrics. The performance metrics measured here are dependent to the client code, design of the system, the compiler used etc. However, comparison of the systems is meaningful since they have the same client code and the compiler.

3.4.2 Step 2: Strategy Pattern

3.4.2.1 Introduction

The strategy pattern is applied to Data Transmission System (DTS) that is based on the RS232 protocol. DTS has many different transmitting rates. The link is established between two ends of transmission according to their bit rates. The rate ranges from hundreds of bps (bits per second) to ten thousands of bps. There are about 10 different transmission rates used in the system. When establishing a link between two ends, a conditional logic is performed based on the bit rate of the transmission. In other words, the transmission is based on the rate of the link. This is very suitable for the strategy pattern, because the same behavior, that is transmission, is based on the selected rate, which is rate strategy.

3.4.2.2 Before Applying Strategy Pattern

In the current system, the rates are checked in every operation. This results in using conditional statements, as shown in Listing 3.

```
/* constructing a link */
switch (linkRate)
{
      case 300:
      /*
       * Construct a 300 bps link
       */
      break;

      case 600:
      /*
       * Construct a 600 bps link
       */
      break;

      case 1200:
      /*
       * ....
       */
}
```

Listing 3: Pseudo Code Before Applying Strategy Pattern

There are link establish requests coming to DTS and at each request, the rate is checked because the operations are changing from one rate to another.

3.4.2.3 After Applying Strategy Pattern

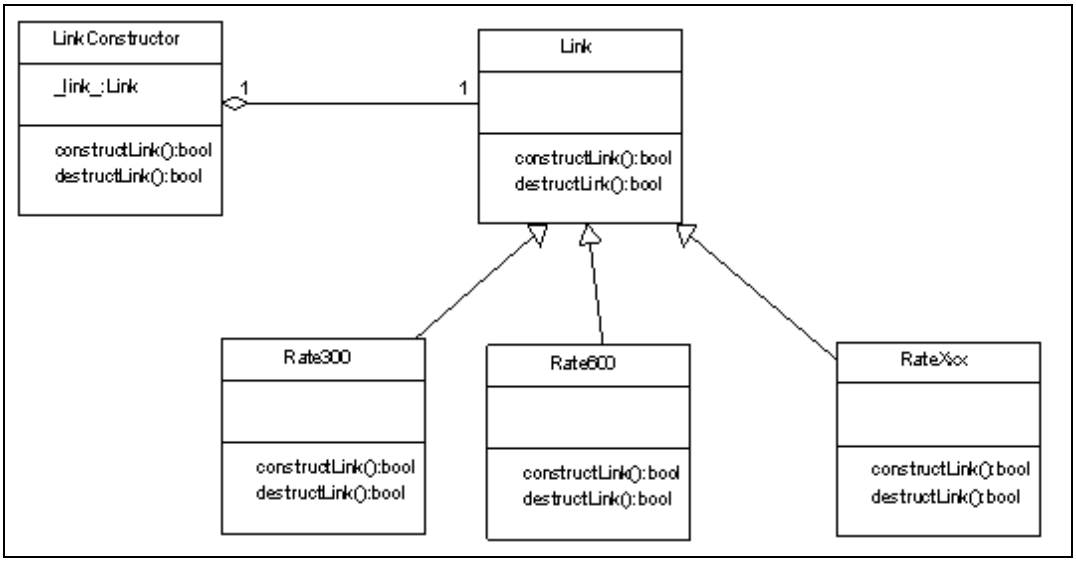

Figure 10: Pseudo UML Diagram After Applying Strategy Pattern

```
/* constructing a link */

        linkConstructor->constructLink(..);
```

Listing 4: Pseudo Code After Applying Strategy Pattern

As can be seen from Figure 10 and Listing 4, the strategy pattern removes the conditional logic. It eliminates the conditional logic depending on the rate of the call. The rates can be thought to be strategies. Each class constructs a different link depending on the link rate with the same interface for constructing a link request.

3.4.2.4 Results, Comparison and Discussion

Table 6: ABTM for Strategy Pattern

Cases	Number of Tests	Maximum	Minimum	Average	Change (%)		Std. Dev.	Change (%)
C	200	37,71	36,04	36,39	-		0,37	-
CPP WODP	200	55,63	54,38	54,76	50,47		0,34	-6,78
CPP WDP	200	42,71	41,46	41,72	14,65		0,30	-17,66

It can be seen from Table 6 that ABTM is about %15 larger at CPP WDP relative to C case. However, CPP WODP case has a much higher ABTM. There is a great effect of strategy pattern on performance. CPP language is seen to increase the ABTM by around %50, whereas, strategy pattern decreases this value to the %15 values.

Table 7: OFSM for Strategy Pattern

Cases	Size (bytes)	Change (%)
C	33888	-
CPP WODP	16748	-50.58
CPP WDP	14660	-56.74

Table shows that CPP WDP decreased the OFSM by around 56%. It is slightly better than the CPP WODP case, but it is far better than the C case.

Table 8: Initialization Overhead for Strategy Pattern

	# of Tests	Maximum	Minimum	Average	Std. Dev.
CPP WDP	100	168.8	158.3	161.2	2.0

There is also some initialization overhead for strategy pattern because there are some objects to be created. These objects should be created once, most probably this done at the initialization of the system. The required objects creation execution time is recorded in Table 8 above. This required time is around 3-5 times higher than the time recorded in Table 6. This can be considered as a very large overhead. However, it should be kept in mind that, this is done only once.

As a property of the project, the applied part has no task; it is just working as a relation of function and/or layers without any task. Therefore, TMM is not applicable for strategy pattern in this project. Likewise, MFM are not recorded for this pattern because all cases have no effect on MFM.

The results of the strategy pattern tests show that the ABTM is increased by about %15. However, it is much better than the CPP WODP case. OFSM is the best in

CPP WDP. Note that these values may change form one system to another since the number of subclasses, operations, methods differ from one to the other. However, ABTM is very close in C and CPP WDP cases. The effect of DPs on ABTM should not be underestimated since it decreased ABTM a lot relative to CPP WODP. [1] states that the strategy pattern decreases the complexity and subclassing. It is stated that the strategy pattern decreases the WMC and RFC values with a side effect of increasing the CBO metric. Therefore, the strategy pattern can be used in many systems provided that the 15% increase in ABTM is not considered very important, because it also improves the software. Moreover, it is seen that, the memory consumption is also decreased by the strategy pattern.

Note that, as stated before, the number and arrangement of conditional statements directly affect performance metrics. The performance metrics measured here are dependent to the client code, design of the system, the compiler used etc. However, comparison of the systems is meaningful since they have the same client code and the compiler.

3.4.3 Step 3: Observer Pattern

3.4.3.1 Introduction

The observer pattern is applied to a system which has the duty of shutting down the system in a safe manner. When a shut down occurs, a RT interrupt is generated and many parts of the system are invoked by a notifier. The interrupt can be considered as an event for starting the invoking phase and the notification is done using the message queue utilities of RTOS. The event is an interrupt generated by RTOS and publish & subscribe rule can be implemented using both languages. Therefore, the observer pattern can be implemented in both a functional language and an OOL. Both C and C++ languages are applicable for this pattern. In this thesis, this pattern is implemented in C++ language.

3.4.3.2 Before Applying Observer Pattern

In the system without the observer pattern, shown in Listing 5, no subscription is done to a publisher. It is not possible to dynamically subscribe to the publisher. Instead, the modules to be notified should be known at compile time.

```
/* Interrupt service routine */
void intService(void){
      semGive (shutDownSem); }

void shutDownTask(void){
      semTake(shutDownSem, WAIT_FOR_EVER);
      msgQSend(msgID1, SHUT_DOWN_MSG, .., NO_WAIT, .. );
      msgQSend(msgID2, SHUT_DOWN_MSG, .., NO_WAIT, .. );
      msgQSend(msgID3, SHUT_DOWN_MSG, .., NO_WAIT, .. );
      /*
       *other observers...
       */
}
```

Listing 5: Pseudo Code Before Applying Observer Pattern

The intService function is connected to the shut down interrupt. When a shut down event occurs, an interrupt is generated and it calls the intService function above. All modules that are required to be aware of this event should be coded statically in the shutDownTask function. Note that, a simple semaphore is a very simple and efficient method for handling interrupts. Since an interrupt is locking the system for a moment, its responsibility, that is the intService function, should be completed as soon as possible. Creating a semaphore is just a single operation for the system. shutDownTask is a task waiting for the shutDownSem. When the event occurs, the semaphore is released by the intService function and taken by the shutDownTask function. Then all the modules are invoked by the message queues.

3.4.3.3 After Applying Observer Pattern

```
/* Interrupt service routine */
void intService(void){
      semGive (shutDownSem);
}
void shutDownTask(void){

      semTake(shutDownSem, WAIT_FOR_EVER);
      publisher->notifyAll();
}
/* subscription */
void subscribe(MSG_Q_ID msgID){
      publisher->addToList(msgID);
}
/* sample client code */
  subcribe(myMgID);
```

Listing 6: Pseudo Code After Applying Observer Pattern

Like the section 3.4.3.3, same RTOS utilities, interrupt, semaphore and message queues are used here as shown in Listing 6. However, there are some differences. First of all, the subscription method gives the opportunity to be subscribed dynamically. In fact, the publisher does not need to know the modules to be notified. They are subscribing themselves at run-time. When the event occurs, all the subscribers in the list will be invoked.

3.4.3.4 Results, Comparison and Discussion

Table 9: ABTM for Observer Pattern

Cases	Number of Tests	Maximum	Minimum	Average	Change (%)		Std. Dev.	Change (%)
C	200	142.2	131.8	137.0	-		2.2	-
CPP WODP	200	134.4	127.6	130.8	-4.481		1.5	-29.235
CPP WDP	200	141.7	129.7	135.8	-0.871		2.1	-4.883

Table 9 indicates different results than the strategy and state patterns. The ABTM for all cases are quite close to each other. Moreover, CPP WODP has the least ABTM. CPP WDP is between C and CPP WODP case. However, they are all very close to each other.

Table 10: OFSM for Observer Pattern

Cases	Size	Change (%)
C	24220	-
CPP WODP	9736	-59.80
CPP WDP	8868	-63.39

Table 10 shows that the OFSM, CPP WDP has the least value whereas C case has a much more OFSM. CPP WDP decreased the file size by around 63% which is a great opportunity for decreasing memory consumption.

Table 11: Initialization Overhead for Observer Pattern

	# of Tests	Maximum	Minimum	Average
CPP WDP	20	38.1	35.8	36.2

As in the earlier cases, there is some initialization overhead for this pattern also. The required time for creating objects is stated above. The 36 microsecond overhead seen in Table 11 may or may not be acceptable in comparison with the ABTM in Table 9, according to system requirements; hence the usage of the observer pattern would be a matter of decision for the system designer.

Again as a property of the project, the applied part has no task; it is just working as a relation of interrupts and message queues without any task. Therefore, TMM is not applicable for observer pattern in this project. Likewise, MFM are not recorded for this pattern because all cases has no effect on MFM.

To summarize, the observer pattern has decreased ABTM slightly compared to the C case, but increased a bit compared to CPP WODP case. It also decreased the OFSM most. The observer pattern has no effect on MFM, either increase or decrease; therefore the MFM has not been recorded for this pattern. Coming to the initialization overhead shown in Table 11, the time required to create objects is much smaller than the test done for observer pattern. Therefore, depending on the requirements, this initialization overhead can be ignored in most of the RTS.

From the performance aspect, the observer pattern has a slight improvement on ABTM relative to the C case. As a result, instead of performance degradation, it is seen that there is some improvement in overall performance of the RTS by applying this pattern. In fact, this pattern has a more effect on the quality of the system. It is known as a “publish & subscribe” rule. It provides a way for other modules to subscribe to the publisher so that they will be aware of an event. This mainly decreases coupling between objects and/or modules in a system. [1] states

the benefits of this pattern on the high coupling problem and recommends using observer together with the mediator pattern to obtain more substantial benefits.

Note that, as stated before, the performance metrics measured here are dependent to the client code, design of the original system and the compiler used etc. However, comparison of the systems is meaningful since they have the same client code and the compiler.

3.4.4 Step 4: Smart Pointer Pattern Using Reference Counting Algorithm

3.4.4.1 Introduction

As an RT memory pattern, SPP is a pattern that is used to delete objects which are not necessary any more. Sometimes it is forgotten to delete an object which will not be used again. This will result in low memory in time. Therefore, SPP is not applied to a specific case, but it is used as a general pattern for de-allocation problems in RTS and tested on a RT communication system.

3.4.4.2 Before Applying SPP

```
  mySPPTestFuncWOP()
{
Person * p = new Person(" emir kustarika ", 50);
      p->Display();
  // and just the person pointer p will be deleted
}
```

Listing 7: Pseudo Code Before SPP

An object is created in the function shown in Listing 7. However, it is forgotten to delete the created object when exiting the function. Using the above function will result in memory leaks, because the reference `p` is destructed when leaving the function but the actual object is still in the memory in an unreachable manner.

3.4.4.3 After Applying SPP Using RCA

```
mySPPTestFuncWP()
{
        SmartPointer <Person>  p (new Person(" emir kustarika ", 50));
    p->Display();

    // both person pointer p and the actual Person object will be deleted
}
```

Listing 8: Pseudo Code After SPP Using RCA

Like the previous case, an object is created and it is forgotten to delete the created object when exiting the function. However, there is no de-allocation problem in the code shown in Listing 8. Since the smart pointer reference object will be destructed when leaving the function, it will also call the destructor of the actual object if there is no any other reference to it. Smart pointers use a reference counter to track the numbers of references to the actual object and to know whether it is being used or not. To summarize, using the above function will not be resulted in memory leaks, because both the smart pointer and the actual objects are destructed when exiting the function.

3.4.4.4 Results, Comparison and Discussion

Table 12: ABTM for SPP Using RCA

Cases	Number of Tests	Maximum	Minimum	Average	Change (%)		Std. Dev.	Change (%)
CPP WODP	100	14.2	11.9	13.2	-		0.25	
CPP WDP	100	50.4	49.2	49.5	275.047		0.35	39.206

As expected, Table 12 shows that SPP brought execution overhead to the CPU. The ABTM for SPP is much higher than the regular object creation. CPP WDP has almost 2 times more ABTM than CPP WODP case.

Table 13: OFSM for SPP Using RCA

Cases	Size	Change (%)
CPP WODP	1288	-
CPP WDP	1724	33.85

Table 13 shows that CPP WDP has also brought some OFSM overhead. It is about 500 bytes higher than the CPP WODP case. This is because of the objects created for SPP. When a smart pointer is created, it is created with a reference counter object and an actual referenced object. This makes the OFSM a bit larger.

Table 14: EMM for SPP Using RCA

Cases	Before	After	Consumption
CPP WODP	9208104	7608104	1600000
CPP WDP	7608104	7608040	64

Table 14 shows the benefits of SPP. CPP WODP case consumes the free memory more and more in time whereas, CPP WDP has just a negligible effect on the free memory size. There is an incomparable difference between CPP WDOP and CPP WDP cases.

Note that since there is no task running for SPP, TMM is not applicable for SPP in this project. Moreover, for SPP, the initialization overhead is already effected the ABTM since SPP deals with the creation and destruction of objects. Therefore, it can not be recorded as a different measurement for this pattern. Since SPP allocates and de-allocates memory for object creation and destruction, it increases memory fragmentation. However, it is not possible to measure MFM. This is because it is not known how much the memory is fragmented in SPP implementation. Therefore, MFM is not recorded, but it increases MFM.

In a regular object creation, the memory is consumed more and more. This will continue up to a memory low error of the system. Of course, it is supposed that the created objects are forgotten to be destructed so that these problems occur. However, SPP removes the necessity of deleting an object when leaving a function and/or block. The memory is not consumed like the ordinary case and memory leaks do not occur. This will also make the system independent from programming mistakes more. It can be concluded that, SPP is solving most of the memory leak problems in the expense of increasing ABTM. Moreover, [17] states that SPP increases maintainability of the software. SPP improves maintainability metrics such as WMC, DIT, CBO, RFC and COF.

3.4.5 Step 5: Garbage Collection Pattern Using Reference Counting Algorithm [2]

3.4.5.1 Introduction

In this step, an MMS is designed using GCLP. GCLP is implemented using reference counting as stated before in section 2.2.5. As an RT memory pattern, GCLP is a pattern that is used to de-allocate memories which are not necessary any more. Likewise SPP, GCLP removes the developers from the responsibility of de-allocating memories. However, there is a major difference with SPP such that, GCLP is implemented in an MMS here. Therefore, the interface of the system is quite different as will be seen later.

3.4.5.2 Before Applying GCLP

```
void myGCLPTestFuncWOP(void)
{
      . . .

      memC1 = malloc(i);
      memset ( memC1 ,'1',i);

      memC4 = malloc(i);
      memset ( memC4 ,'4',i);

      . . .

      memcpy( memC4, memC3, size );
}
```

Listing 9: Pseudo Code Before GCLP

In Listing 9, memory is allocated using standard C malloc() function. If it is forgotten to free the allocated memory before exiting the myGCLPTestFuncWOP() function, then the allocated memory will no longer be reachable. This will result in memory leaks. GCLP will solve these kinds of problems.

3.4.5.3 After Applying GCLP Using RCA

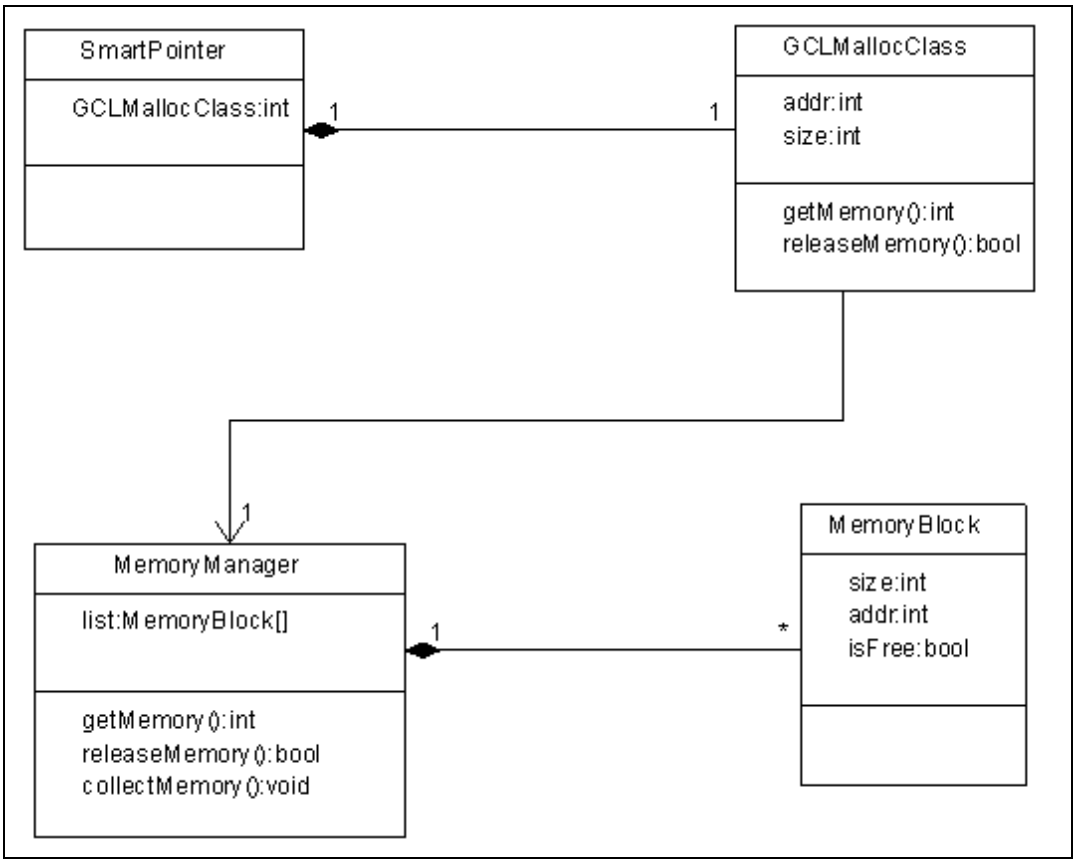

Figure 11: Implemented GCLP Class Diagram Using RCA

```
void myGCLPTestFuncWP(void)
{       . . .

       SmartPointer <GCLMallocClass>
       memCpp3 (new GCLMallocClass(i));
       memset ( (memCpp3->addr) ,'3',i);

       i +=increment;

       SmartPointer <GCLMallocClass>
       memCpp4 (new GCLMallocClass(i));
       memset ( (memCpp4->addr) ,'4',i);

       . . .

       memcpy((memCpp4->addr), (memCpp3->addr), size );
```

Listing 10: Pseudo Code After GCLP Using RCA

Figure 11 shows the class diagram for implementing GCLP. Memory manager keeps the memory blocks. Each memory block is free or used. The smart pointer keeps a reference counter to the GCLMallocClass which is an interface between the smart pointer and memory manager. This is done for decreasing coupling and increasing cohesion stated by [8]. The memory block is flagged as free or used by the information coming form the smart pointer. Listing 10 shows the client's pseudo block of code after using GCLP. Once the memory object is created, it can be used as a memory allocated with malloc() function.

3.4.5.4 Results, Comparison and Discussion

Table 15: ABTM for GCLP Using RCA

	# of Tests	Maximum	Minimum	Average	Change (%)	Std. Dev.	Change (%)
C	200	749.99	625.0	670.14	-	13.54	-
CPP WDP	200	21354.13	2760.41	10061.44	1401.39	5471.21	40307,75

As expected before, Table 15 shows that GCLP brought an execution overhead to the CPU. The ABTM for GCLP is much higher than the regular memory allocation. On average, C case is about 14 times faster than CPP WDP case.

Table 16: OFSM for GCLP Using RCA

Cases	Size	Change (%)
C	9204	-
CPP WDP	16611	80.5

Table 16 shows that CPP WDP has also brought some OFSM overhead. It is about 80% higher than the C case. This is mainly because GCLP is applied together with an MMS. Therefore, there are several files to implement GCLP. This has resulted in higher OFSM.

Table 17: EMM for GCLP Using RCA

Cases	Before	After	Consumption
C	8047624	936	8046688
CPP WDP	8047816	7976904	70912

Table 17 shows the most important benefit of GCLP. C case decreases the memory in time whereas CPP WDP has just a very small effect on the free memory size. There is an incomparable difference between CPP WDOP and CPP WDP cases.

Table 18: ABTM for Initialization of GCLP Using RCA

	# of Tests	Maximum	Minimum	Average
CPP WDP	20	16.25	14.17	14.56

As in the earlier cases, there is some initialization overhead for this pattern also. The required time for creating objects is stated above. The 14.56 microsecond overhead seen in Table 18 is quite small in comparison with the ABTM in Table 15. However according to system requirements; the usage of the GCLP would be a matter of decision for the system designer.

Table 19: IMM for GCLP Using RCA

Cases	Before	After	Consumption
CPP WDP	9358576	8047816	1310760

Table 19 shows the amount of memory consumed in initialization of MMS. This memory includes a block of memory with size 1 MB dedicated for MMS used by CPP WDP case. This memory is the heap and constant for this system. Remaining part is consumed for variables and objects required for initialization of MMS. However, C case consumes memory more and more in time indefinitely. When the memory is consumed totally, the system becomes open to crash. In fact, the size required for CPP WDP can be changed according to system requirements. For instance, a 1MB choice for an embedded system can be considered huge, but it is

very small relative to a web server. Of course, it is the choice of the designer of that system.

Table 20: SBBM for GCLP Using RCA

Cases	SBBM	Change (%)
C	1024	-
CPP WDP	1024	0

GCLP has no effect on fragmentation. As seen in Table 20, after all memory in MMS is consumed, the SBBM becomes 1024 bytes, which is one of block sizes allocated before, which is much smaller then the capacity of MMS.

Table 21: FBM-AS for GCLP Using RCA

Cases	FBM-AS	Change (%)
C	542.4	-
CPP WDP	542.4	0

Likewise, Table 21 shows that after all memory in MMS is consumed, the FBM-AS becomes 542.4 bytes, which is much smaller than the capacity of MMS. Note that, during the measurement of MFM, it is assumed in C case that, all memory is consumed with the same function as in CPP WDP case. This assumption arises from the fact that, CPP WDP uses MMS, which has a dedicated heap memory. However, C case uses the system memory directly, which can be used by other modules in the system also.

Being a memory pattern, it is seen that, GCLP solves the memory de-allocation problem with the expense of increasing ABTM. GCLP removes the responsibility

of the programmer for de-allocating the previously allocated memory in a designed MMS. However, there is some initialization overhead in terms of memory and execution time. Moreover, the amount of dedicated memory for MMS should be considered and decided carefully by the designer of the system. Note that since there is no task running for GCLP, TMM is not applicable for GCLP in this project, so it is not recorded.

Note that, there exist different algorithms for GCLP implementation. Of course, the performance metrics for GCLP are also affected by the algorithm used for collection. Therefore, these metrics should be seen as the performance of GCLP with reference algorithm implementation. Moreover, the client code that used this system also affects the performance metrics. Different client codes will result in different performance metrics.

3.4.6 Step 6: Garbage Compactor Pattern Using Reference Counting Algorithm

3.4.6.1 Introduction

As stated before in section 2.2.6, GCMP is GCLP plus compacting property. GCMP does everything which GCLP does, but it is also able to compact the fragmented memory. A compacting property will be added to the GCLP with RCA. When the compaction will be done is the choice of the software designer of the system. In fact, every system has different requirements. Compaction can be done internally, as such when a memory requirement fails. It can be done externally or it can be done periodically. There may be other choices for when to start compaction. In the present study, compaction is done automatically when a memory requirement fails. Note that, GCMP is also deployed in a memory management system.

3.4.6.2 Before Applying GCMP

```
void myGCMPTestFuncWOP(void)
{     memC1 = malloc(i); memset ( memC1 ,'1',i);
      memC4 = malloc(i); memset ( memC4 ,'4',i);
      memcpy( memC4, memC3, size );}
```

Listing 11: Pseudo Code Before GCMP

In Listing 11, memory is allocated using standard C malloc() function. If it is forgotten to free the allocated memory before exiting the myGCMPTestFuncWOP() function, then the allocated memory will no longer be reachable. This will result in memory leaks. GCMP will solve these kinds of problems. Note that, this is the same code used in GCLP tests in section 3.4.5.2.

3.4.6.3 After Applying GCMP Using RCA

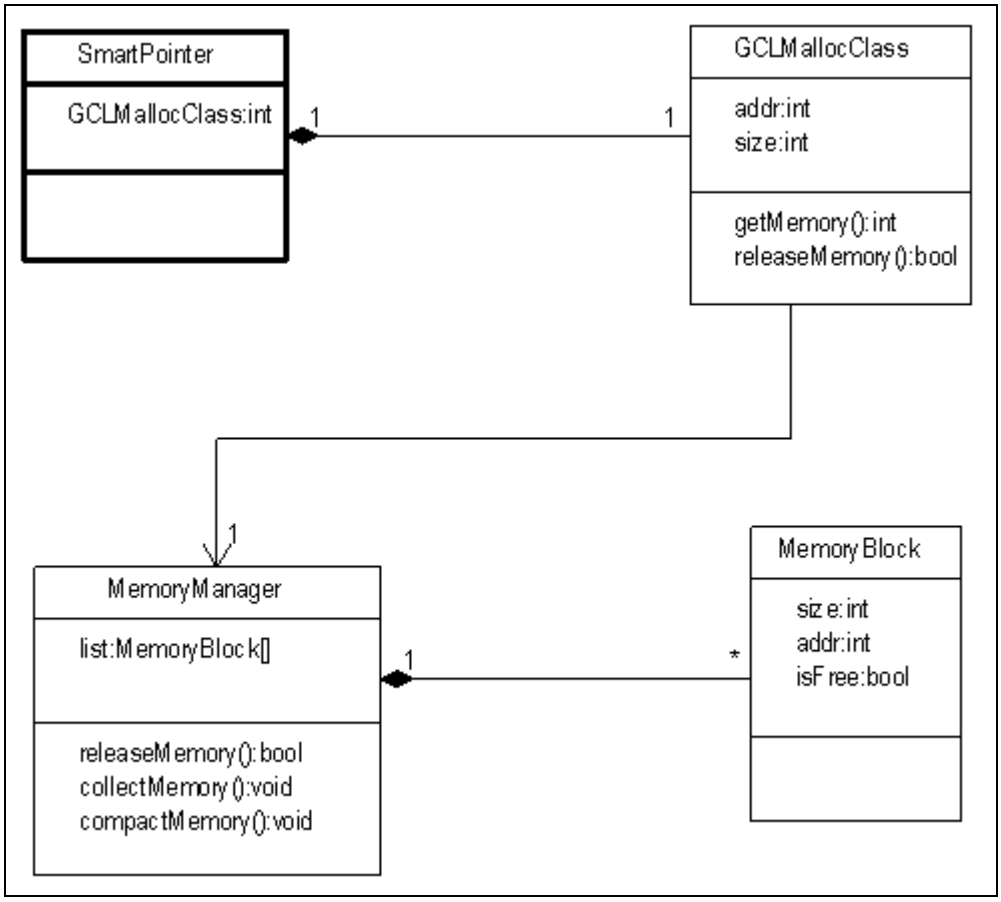

Figure 12: Implemented GCMP Class Diagram Using RCA

It can be noticed easily that Figure 12 is very similar to Figure 11 in section 3.4.5.3. This is expected because the main difference between GCLM and GCMP is compacting property. Memory manager keeps the memory blocks. Each memory block is free or used. The smart pointer keeps a reference counter to the GCLMallocClass which is an interface between the smart pointer and memory manager. This is done for decreasing coupling and increasing cohesion stated by [8]. The memory block is flagged as free or used according to the information coming form the smart pointer. When a memory requirement fails, first a

collection is done. If there is still a fail for memory service, then the compaction of memory is started.

```
void myGCMPTestFuncWP(void)
{
      . . .

      SmartPointer <GCLMallocClass>
      memCpp3 (new GCLMallocClass(i));
      memset ( (memCpp3->addr) ,'3',i);

      i +=increment;

      SmartPointer <GCLMallocClass>
      memCpp4 (new GCLMallocClass(i));
      memset ( (memCpp4->addr) ,'4',i);

      . . .

      memcpy((memCpp4->addr), (memCpp3->addr), size );
}
```

Listing 12: Pseudo Code After GCMP Using RCA

Listing 12 shows the client's pseudo code after using GCMP. Once the memory object is created, it can be used as a memory allocated with malloc() function. Note that, GCMP has the same interface for clients as GCLP.

3.4.6.4 Results, Comparison and Discussion

Table 22: ABTM for GCMP Using RCA

	# of Tests	Maximum	Minimum	Average	Change (%)
C	200	708.33	645.83	652.02	-
CPP WDP	200	23958.30	2812.50	12820.98	1866.35

As expected and noted before, Table 22 shows that GCMP brought a huge amount of execution overhead to the CPU. The ABTM for GCMP is much higher than the regular memory allocation. On average, C case is about 19 times faster than CPP WDP case.

Table 23: OFSM for GCMP Using RCA

Cases	Size	Change (%)
C	9200	-
CPP WDP	18051	96.21

Table 23 shows that CPP WDP has also brought some OFSM overhead. It is about double amount of the C case. This is mainly because GCMP is applied together with an MMS. Therefore, there are several files to implement GCMP. This is resulted in higher OFSM.

Table 24: EMM for GCMP Using RCA

Cases	Before	After	Consumption
C	5768544	3304	5765240
CPP WDP	8047800	7954144	93656

Table 24 shows one of the most important benefits of GCMP. C case decreases the memory in time whereas CPP WDP has just a very smaller effect on the free memory size. There is an incomparable difference between CPP WDOP and CPP WDP cases.

Table 25: ABTM for Initialization of GCMP Using RCA

	# of Tests	Maximum	Minimum	Average
CPP WDP	50	16.25	14.17	14.78

As in the earlier cases, there is some initialization overhead for this pattern also. The required time for creating objects is stated above. The 14.78 microsecond overhead seen in Table 25 is quite small in comparison with the ABTM in Table 22. However, according to system requirements; the usage of the GCMP would be a matter of decision for the system designer likewise GCLP.

Table 26: IMM for GCMP Using RCA

Cases	Consumption
CPP WDP	1310860

Table 26 shows the amount of memory consumed in initialization of MMS. This memory includes a block of memory with size 1 MB dedicated for MMS used by CPP WDP case. This memory is the heap and constant for this system. Remaining part is consumed for variables and objects required for initialization of MMS. However, C case consumes memory more and more in time with no any limit. When the memory is consumed totally, the system becomes open to crash. In fact,

the size required for CPP WDP can be changed according to system requirements. For instance, a 1MB choice for an embedded system can be thought to be huge, but it is very small relative to a web server. Of course, it is the choice of the designer of that system.

Table 27: SBBM for GCMP Using RCA

Cases	SBBM	Change (%)
C	1024	-
CPP WDP	1045276	101977.73

GCMP has a significant effect on fragmentation. As seen in Table 27, using the regular memory allocation, after all memory in MMS is consumed, the SBBM becomes 1024 bytes, which is much smaller then the capacity of MMS. However, GCMP applies a compaction algorithm to compact almost all of the free memory.

Table 28: FBM-AS for GCMP Using RCA

Cases	FBM-AS	Change (%)
C	542.4	-
CPP WDP	87397	16013.02

Similar to Table 27, Table 28 shows that, after all memory in MMS is consumed, the FBM-AS becomes 542.4 bytes using regular memory allocation, which is much smaller than the capacity of MMS. However, GCMP makes the average block size much higher than C case. This is the main property of GCMP. Note that, during the measurement of MFM, it is assumed in C case that, all memory is consumed with the same function as in CPP WDP case. This assumption arises from the fact that, CPP WDP uses MMS, which has a dedicated heap memory.

However, C case uses the system memory directly, which can be used by other modules in the system also.

Table 29: ABTM for Compaction at GCMP Using RCA

	ABTM
CPP WDP	690780.69

Table 29 shows the ABTM required to compact the heap memory of MMS. It is very large relative to a memory request operation seen in Table 22.

Being a memory pattern, it is seen that, GCMP solves the memory de-allocation and fragmentation problems with the expense of increasing ABTM. Likewise GCLP, GCMP also removes the responsibility of the programmer for de-allocating the previously allocated memory in a designed MMS. The most important difference between GCLP and GCMP is that, GCMP also solves the fragmentation problem. However, there is important overhead. Especially, compaction requires a very large execution time. Moreover there is some initialization overhead in terms of memory and execution time. The amount of dedicated memory for MMS should be considered and decided carefully by the designer of the system. Another important responsibility of the designer is to decide whether to use GCMP or not because it has some advantages and disadvantages. The applicability of GCMP is definitely dependent on the system requirements and properties. Note that since there is no task running for GCMP, TMM is not applicable for GCMP in this project, so it is not recorded just like GCLP in step 6.

Note that, there exist different algorithms for GCMP implementation as in GCLP case. Of course, the performance metrics for GCMP are also affected by the algorithm used for collection and compaction. Therefore, these metrics should be seen as the performance of GCMP with reference algorithm implementation.

Moreover, the client code that used this system also affects the performance metrics. Different client codes will result in different performance metrics.

3.4.7 Step 7: Garbage Compactor Pattern Using Reference Counting Algorithm + State Pattern

3.4.7.1 Introduction

In this section, the state pattern will be used together with the GCMP. During the implementation of GCMP, it is seen that the MMS has different states. For instance, at the start-up of the system, memory requirements are served easily because there is already a huge free memory block in the heap. This state continues up to where all the memory in the heap is served. On the next state, when memory is required, a previously allocated but no more used memory block with enough size is selected and returned. When this method fails, a collection algorithm is applied to the heap. This can also be thought as another state of MMS. Moreover, likewise collection, compaction can also be thought as a state of MMS. Note that, the memory requirements are served differently at these states. Therefore, the state pattern is an applicable GOF design pattern for RT design pattern GCMP. State pattern would also be applied to the GCLP. However, GCMP already includes collection state. Therefore, it is enough and more meaningful to apply it to GCMP.

3.4.7.2 Before Applying State Pattern to GCMP

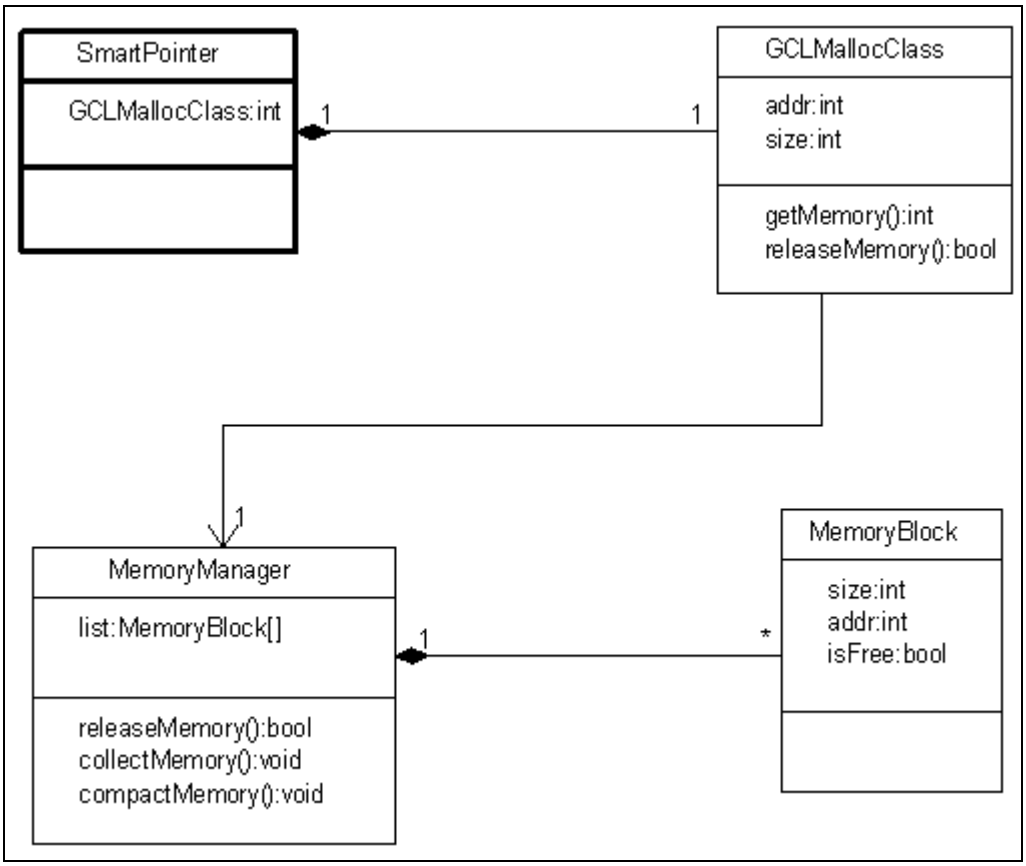

Figure 13: UML Diagram Before Applying State Pattern to GCMP

As seen in Figure 13, MemoryManager class has the responsibility to manage all the heap alone. It responds to memory requests differently according to its states. However this is done with conditional statements. This is the actual case in section 3.4.6.

3.4.7.3 After Applying State Pattern to GCMP Using RCA

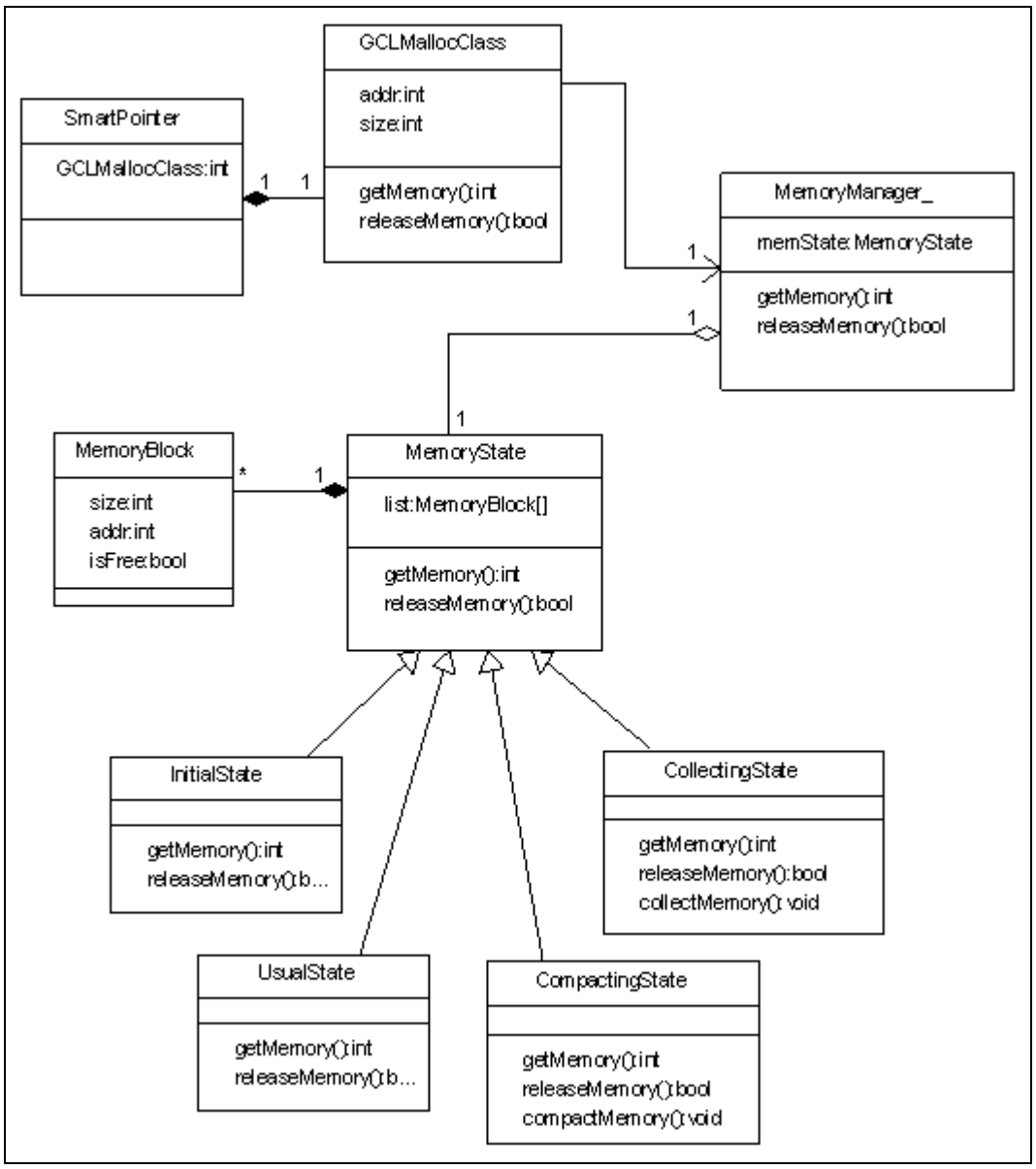

Figure 14: UML Diagram After Applying State Pattern to GCMP Using RCA

Figure 14 shows the UML diagram after applying state pattern to GCMP. In this case, MemoryManager class becomes a contact between GCLMallocClass and

MemoryState class. MemoryState class has 4 sub state classes. Each state responds differently to memory related work.

3.4.7.4 Results, Comparison and Discussion

Table 30: ABTM for GCMP Using RCA + State Pattern

	# of Tests	Maximum	Minimum	Average	Change (%)
C	200	708.33	645.83	652.02	-
CPP + GCMP	200	23958.3	2812.4955	12820.98	1866.35
CPP + GCMP + State	200	27395.79	3020.8285	12510.61	-2.42

Table 30 shows that state pattern has slightly decreased ABTM. However, it is still far from the C case. The ABTM for GCMP + state case is much higher than the regular memory allocation.

Table 31: OFSM for GCMP Using RCA + State Pattern

Cases	Size	Change (%)
C	9204	-
CPP + GCMP	18051	96.12
CPP + GCMP + State	1337351	14430.11

Table 31 shows that the state pattern has brought a huge amount of OFSM overhead. This is a result of the fact that the heap memory became global instead of a constant class variable. Therefore, the heap memory required for MMS is included in the object file size during compilation. Since heap is implemented as a global variable; it will not effect the IMM overhead any more.

Table 32: EMM for GCMP Using RCA + State Pattern

Cases	Before	After	Consumption
C	5768544	3304	5765240
CPP + GCMP	8047800	7954144	93656
CPP + GCMP + State	8124664	8021352	103312

Table 32 shows one of the most important benefits of GCMP. C case decreases the memory in time whereas CPP WDP has just a very smaller effect on the free memory size. There is an incomparable difference between C and CPP GCMP cases. However, state pattern has increased memory consumption. This is because of the objects and their own variables created for states of GCMP.

Table 33: ABTM for Initialization of GCMP Using RCA + State Pattern

	Number of Tests	Maximum	Minimum	Average	Change (%)
CPP + GCMP	50	16.25	14.17	14.78	-
CPP + GCMP + State	50	90.62	65.42	84.59	472.41

As in the earlier cases, there is some initialization overhead for GCMP + state pattern also. The required time for creating objects is stated above. State pattern has increased the ABTM for initialization overhead by almost 5 times. This is because of the objects created for states. Note that, this initialization is done once and will not be repeated until a reset of the system. The 84.59 microsecond overhead seen in Table 15 is quite small in comparison with the ABTM in Table 33. However, according to system requirements; the usage of the state pattern would be a matter of decision for the system designer likewise other memory patterns.

Table 34: IMM for GCMP Using RCA + State Pattern

Cases	Before	After	Consumption
CPP + GCMP	9358560	8047800	1310760
CPP + GCMP + State	8124856	8124776	80

Table 34 shows the amount of memory consumed in initialization of MMS. The consumption for GCMP includes the heap memory of size 1 MB. Remaining part is consumed for variables and objects required for initialization of MMS. However, state pattern has a very small amount of memory consumption for initialization. Since the heap memory for state pattern implementation is already declared as global and therefore included in OFSM, this metric is quite small compared to GCMP alone. On the other hand, C case has no any memory consumption for initialization but it consumes memory more and more in time with no limit. When the memory is consumed totally, the system becomes open to crash. As stated before, the heap memory size is the choice of the designer of the system.

Table 35: SBBM for GCMP Using RCA + State Pattern

Cases	SBBM	Change (%)
C	1024	-
CPP + GCMP	1045276	101977.73
CPP + GCMP + State	1045276	0.00

Table 35 shows that the state pattern has no effect on SBBM as expected in Table 1.

Table 36: FBM-AS for GCMP Using RCA + State Pattern

Cases	FBM-AS	Change (%)
C	542.4	-
CPP + GCMP	87397	16013.02
CPP + GCMP + State	87397	0.00

It is seen form Table 36 that the state pattern also has no effect on FBM-AS.

Table 37: ABTM for Compaction at GCMP Using RCA + State Pattern

	ABTM
CPP + GCMP	690780.69
CPP + GCMP + State	670623.93

Table 37 shows the ABTM required to compact the heap memory of MMS. It is very large relative to a memory request operation seen in Table 30. However, it seems that the state pattern has slightly decreased the compaction time.

It is seen form the results that the state pattern has no effect on fragmentation. It is very interesting that the state pattern has also improved ABTM for GCMP as seen in Table 2 in section 3.4.1.4. [8] states that the object that has the highest amount of information about something should be responsible for that thing. This principle is called information expert. This is also consistent with the high cohesion principle [8]. The state pattern decreases coupling and increases cohesion. Note that since there is no task running for GCMP, TMM is not applicable for GCMP in this project, so it is not recorded.

Note that, there exist different algorithms for GCMP implementation as in GCLP case. Of course, the performance metrics for GCMP + State are also affected by the algorithm used for collection and compaction. Therefore, these metrics should be seen as the performance of GCMP + State with reference algorithm implementation. Moreover, the client code that used this system also affects the performance metrics. Different client codes will result in different performance metrics.

CHAPTER 4

DISCUSSION AND CONCLUSIONS

In this thesis, effects of various design patterns on the performance of RTS have been investigated. Selected patterns from both GOF and RT design pattern sets are studied. Various performance related metrics for RTS are selected and used. The results of measurements have given the opportunity to discuss and compare the effects of various design patterns on RTS.

In the literature review section, a number of design patterns are discussed. Not all of GOF and RT design patterns are expected to effect performance of RTS. Moreover, design patterns should be applied to a specific problem addressed by that design pattern. There were a limited number of available projects available within the scope of this study, so a limited number of design patterns could be used. Therefore, many of GOF and RT design patterns were eliminated. The remaining design patterns that could be used in this study were examined in detail. The state and strategy patterns are examined as GOF design patterns. SPP, GCLP and GCMP patterns are examined as RT design patterns. The observer pattern is examined as a pattern that belongs to both of the GOF and RT design pattern categories. The metrics for measuring the performance of RTS were examined. Execution time ([6], [4], [9] and [11]), memory consumption ([9]) and memory fragmentation ([15], [6]) related performance metrics are analyzed,

measured, compared and discussed. Some metrics are specific to the projects and are subsets of other metrics. These are selected according to the system properties. [4].

In step 1 of the experimental work, the state pattern is applied to the RTS. It is seen that the CPP language has increased the execution time relative to C case. Nevertheless when the state pattern is applied to CPP, the degradation in execution time has been compensated for and it is slightly better than the C case. The memory consumption metric is much better than the C case, but this is the effect of CPP language, not the state pattern. There is also some initialization overhead for the state pattern. It was expected (Table 38) that the state pattern would slightly decrease the execution time and have almost no effect on memory metrics. This expectation was fully justified in the experiments as seen in Table 38.

In step 2, the strategy pattern is applied to the RTS. It is seen that the CPP language has increased the execution time by half relative to the C case. However, when the strategy pattern is applied to CPP, the degradation in execution time has been decreased. The memory consumption metric is much better than the C case and this is an effect of both the CPP language and the strategy pattern. There is also some initialization overhead for strategy pattern. It was expected (Table 38) that strategy pattern would decrease the execution time. According to the results, as seen in Table 38, strategy pattern has a worse execution time than C case, but it has improved the execution time in CPP case. The degradation in execution time can or can not be ignored according to system requirements and this is the choice of the designer of the system. Moreover, it was expected that (Table 38) the strategy pattern would have no effect on memory metrics. However, as seen from Table 38, it has also decreased memory consumption.

In step 3, the observer pattern is applied to the RTS. It is seen that the CPP language has decreased the execution time relative to the C case and observer pattern has increased the execution time. However, it is still better than the C case.

Memory consumption is best with the observer pattern. There is also some initialization overhead for observer pattern. It was expected (Table 38) that observer pattern slightly increase the execution time. This expectation is justified in experiments as seen in Table 38. However, as seen in Table 38, it has decreased the memory consumption unlike the expectation.

In step 4, SPP is implemented in RTS. It is seen that SPP has increased the execution time a lot. However, it has solved the memory leak problems after creating reference objects. It was expected (Table 38) that, SPP will increase the ABTM, decrease the memory consumption and increase memory fragmentation. As seen in Table 38, it is worse in execution time and better in memory consumption. Besides, memory fragmentation metrics could not be measured. (See Sec. 3.4.4.4)

In step 5, an MMS for RTS has been implemented using GCLP. It is seen that GCLP has increased the execution time metrics a lot. There is also much initialization overhead in GCLP. However, GCLP has solved the memory allocation/de-allocation problems. It has improved the memory performance about consumption a lot. Nevertheless, the memory fragmentation problem has not been solved by GCLP. It was expected (Table 38) that GCLP will increase the execution time metric (ABTM), decrease the memory consumption and increase memory fragmentation. This expectation was fully justified in the experiments as seen in Table 38.

In step 6, an MMS for RTS has been implemented using GCMP. It is seen that, like GCLP, GCMP has increased the execution time metrics a lot and there is much initialization overhead. However, GCMP has done more than GCLP. It has solved the memory allocation/de-allocation problems. Moreover, the memory fragmentation problem is also solved by GCMP with the cost of large execution time for compaction. It was expected (Table 38) that GCMP will increase the execution time metric (ABTM), decrease the memory consumption and memory

fragmentation. This expectation was fully justified in the experiments as seen in Table 38.

In step 7, the state pattern is applied to GCMP. It is seen that it has no effect on memory performance again. However, it has slightly improved the execution time performance of GCMP. There is also some initialization overhead while deploying state pattern. GCMP + state pattern combination has the same memory performance in consumption and fragmentation with GCMP case alone. However, it is better in execution time relative to GCMP alone. It was expected (Table 38) that state pattern will decrease the execution time metric (ABTM). This expectation is also justified in this step (Table 38). Note that, GCMP has solved the memory consumption and fragmentation problems and state pattern has improved the ABTM (Table 38).

Table 38: Actual Effect of GOF and RT Design Patterns on Performance

Pattern Name	Performance Metrics								
	ABTM			Memory Cons.			Memory Frag.		
	E	M		E	M		E	M	
		C	CPP WODP		C	CPP WODP		C	CPP WODP
1. State	D	D	D	NC	D	NC	NC	NC	NC
2. Strategy	D	I	D	NC	D	D	NC	NC	NC
3. Observer	I	D	I	NC	D	D	NC	NC	NC
4. SPP Using RCA	I	-	I	D	-	D	I	-	NP
5. GCLP Using RCA	I	I	-	D	D	-	I	I**	-
6. GCMP Using RCA	I	I	-	D	D	-	D	D	-
7. State + GCMP	I	I*	-	D	D	-	D	D	-

"I": This term indicates that the related metric increased in terms of quantity.
"D": This term indicates that the related metric decreased in terms of quantity.
"E": This term means "Expected"
"M": This term means "Measured"
"NC "(No Change): This term indicates that the related metric did not change.
"NP" (Not Possible): This term indicates that the related metric could not be measured.
"-": This term indicates that the corresponding case is not implemented.
*: Although GCMP increased ABTM a lot, state pattern has slightly decreased this metric.
**: GCLP and C cases have both increased memory fragmentation in the same quantity.

Table 38 shows both the expected and measured metrics. Note that, the "Expected" columns were stated earlier in Table 1 and are copied here. The "Measured" columns in Table 38 are filled according the results of experimental steps from 1 through 7. In each column, CPP WDP case is compared with C and CPP WODP cases separately in terms of RT performance metrics.

The measurements are not handled as a simulation and/or a prediction. The design patterns are applied to various real time communication systems and the results are collected in a real time manner. The performances of the systems are investigated with and without design patterns. They are compared and discussed at the end of each experimental section.

In earlier studies the effects of software design patterns on maintainability and software error-proneness were investigated ([1], [17]). There are similarities between [1], [17] and this study; since all of them have investigated the effects of design patterns. However, there are three main differences. First of all, this study focuses on the effects on performance. Another important difference is the experimental platform. In this study, all the experiments have been conducted on real time systems. However, [1] and [17] investigate the effects of design patterns on more general software systems. Finally, the measurements about the metrics are not predicted or calculated. Unlike [1] and [17], they are measured at run-time. Besides, the effects on maintainability obtained from [1] and [17] must be considered as complementary to the results of the experiments reported in this study. Consequently, this thesis can be seen as a continuation of [1] and [17]. In fact [17] states that, the effects of design patterns on RTS performance should be handled as another subject which requires in depth study.

It should be kept in mind that, the performance metrics measured in this study are dependent on the design of the system, the client code and the compiler etc. as stated before in section 3.2. These metrics will most probably be different in another system, even in the same system with different client code. For instance,

the arrangement of conditional statements in the system may affect performance. Moreover, the algorithms used to apply design patterns are also factors on performance. For example, there exist different algorithms for garbage collection. The performance measurement of applying GCLP is done using RCA. Another algorithm for GCLP will probably result in different performance metric results. This algorithm dependent performance is also under consideration for GCMP.

In this thesis, the results are measured for a specific system with a specific client code. There is no generalization. The generalization of the effects of design patterns on RTS performance can be investigated as another in-depth study. To assess the level of generalizability, statistical evaluation of the effects of design patterns on large samples of software systems must be considered. While this has been outside the scope of the present study, such an exercise would help establish confidence levels and provide guidelines for general usage.

Some RT and GOF design patterns that can affect the performance of RTS have not been studied in this thesis because of the availability of projects. As a future work, these design patterns may also be investigated.

As another item of future work, the effect of design patterns on real time software safety and reliability can be investigated. Especially, safety and reliability patterns [2] are natural candidates for such a study. The effects of resource and concurrency patterns on RTS are also subjects that merit investigation.

REFERENCES

[1] Aydınöz B., "The Effect of Design Patterns On Object-Oriented Metrics and Software Error-Proneness", A Thesis Submitted to the Graduate School of Natural and Applied Sciences of Middle East Technical University, Sep. 2006

[2] Douglass, B.P., "Real-Time Design Patterns : Robust Scalable Architecture for Real-Time Systems", Pearson Education, 2003

[3] Gamma E., Helm R., Jonhson R., Vlissides J., "Design Patterns, Elements of Reusable Object Oriented Software", Addison-Wesley, 1994

[4] Gao, J. Z., Tsao, H. S., Wu, Y., "Testing and Quality Assurance For Component-Based Software", 229-260, Artech House Inc., 2003

[5] Grenning J.W., "Why are You Still Using C?", March 2002-2003
http://www.renaissancesoftware.net/files/articles/WhyAreYouStillUsingC.pdf
Access Date: Feb. 19th, 2010

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

[7] Laplante, P. A., "Real-Time Systems Design and Analysis", Wiley & Sons Publication, 3rd Edition, (2004), Page 393.

[8] Larman C., "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", Third Edition, Addison Wesley Professional, October 20, 2004

[9] Meier, J.D., Farre, C., Bansode, P., Barber, S., Rea, D., "Performance Testing Guidance for Web Applications, patterns practices", (2007), 28-34, 217-218.

[10] Meyer, B. 1988., "Object-Oriented Software Construction", 1.st edition. Englewood Cliffs, NJ.: Prentice-Hall.

[11] Mills, E. E., "Software Metrics", SEI Curriculum Module SEI-CM-12-1.1, Seattle University, December 1988.

[12] Min, Z., Smith, J., "A Survey of Garbage Collection Techniques", (1998), http://pages.cs.wisc.edu/~zhong/termproj_surveyGC.doc,
Access Date: Feb. 19th, 2010

[13] Raghavan, G., Salomaki, A., Lencevicius, R., "Model Based Estimation and Verification of Mobile Device Performance", Conference on Computing Frontiers, (2008)

[14] Rational Rhapsody,
http://www.ibm.com/developerworks/rational/products/rhapsody/, (2009),
Access Date: Feb. 19th, 2010

[15] Rosso, D. C., "The Method, the Tools and Rationales for Assessing Dynamic Memory Efficiency in Embedded Real-Time Systems in Practice", (2006)

[16] Tornado Version 2.2,
http://www.windriver.com/products/product-notes/tornado2/tornado22_relnote.pdf, (2002), Access Date: Feb. 19th, 2010

[17] Türk, T.,"The Effect Of Software Design Patterns On Object-Oriented Software Quality And Maintainability", A Thesis Submitted To The Graduate School Of Natural And Applied Sciences Of Middle East Technical University, Sep. 2009

[18] Schildt H., "Teach yourself C++", page 3, 3. Edition, Osborne, 1997

[19] Wind River Workbench, http://www.windriver.com/products/workbench/, (2006), Access Date: Feb. 19th, 2010

[20] Venners, B., "Inside the Java Virtual Machine", Chapter 9-Garbage Collection, (2000), http://www.artima.com/insidejvm/ed2/gc.html,
Access Date: Feb. 19th, 2010