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

BY

TUNA TÜRK

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

SEPTEMBER 2009

Approval of the thesis:

THE EFFECT OF SOFTWARE DESIGN PATTERNS ON OBJECT-ORIENTED SOFTWARE QUALITY AND MAINTAINABILITY

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

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

Prof. Dr. İsmet Erkmen _______________
Head of Department, **Electrical and Electronics Engineering**

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

Examining Committee Members:

Asst. Prof. Dr. Cüneyt Bazlamaçcı _____________________
Electrical and Electronics Engineering Dept., METU

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

Prof. Dr. Uğur Halıcı _____________________
Electrical and Electronics Engineering Dept., METU

Asst. Prof. Dr. Aysu Betin Can _____________________
Information Systems Dept., METU

(M. Sc.) Selma Dökmen _____________________
Lead Design Engineer, ASELSAN

Date: **10.09.2009**

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	: Tuna Türk
Signature	:

ABSTRACT

THE EFFECT OF SOFTWARE DESIGN PATTERNS ON OBJECT-ORIENTED SOFTWARE QUALITY AND MAINTAINABILITY

Türk, Tuna

M. Sc., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Semih Bilgen

September 2009, 68 pages

This study investigates the connection between design patterns, object oriented (OO) quality metrics and software maintainability. The literature on OO metrics, design patterns and software maintainability are reviewed, the relation between OO metrics and software maintainability is investigated, and then, in terms of obtained maintainability indicator metrics, the maintainability change of an application due to usage of design patterns is observed.

Keywords: Design Patterns, OO Metrics, Software Maintainability

ÖZ

TASARIM KALIPLARININ NESNE TABANLI YAZILIM KALİTESİNE VE YAZILIM BAKIM YAPILABİLİRLİĞİNE ETKİSİ

Türk, Tuna

Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Semih Bilgen

Eylül 2009, 68 sayfa

Bu tez çalışması tasarım kalıpları, nesne tabanlı metrikler ve yazılım bakım yapılabilirliği arasındaki ilişkiyi incelemektedir. Tasarım kalıpları, nesne tabanlı metrikler ve yazılım bakım yapılabilirliği ile ilgili literatür incelenmiş, nesne tabanlı metrikler ve yazılım bakım yapılabilirliği arasındaki ilişki araştırılmış ve daha sonra elde edilen bakım yapılabilirlik belirteci metrikler cinsinden bir uygulamanın tasarım kalıbı kullanımı ile bakım yapılabilirliğinin değişimi gözlemlenmiştir.

Anahtar Kelimeler: Tasarım kalıpları, nesne tabanlı metrikler ve yazılım bakım yapılabilirliği

to

My Grandfather,

My Family

and

Kubilay

ACKNOWLEDGEMENTS

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

I am grateful to ASELSAN Inc. for the resources and facilities that I utilize throughout the study. I also want to thank my colleagues at ASELSAN Inc. for their valuable support.

I appreciate the love, caring and support of my parents Ayşe, Orhan, and my brother, Cihan. Nothing compares finding them next to me whenever I need. I also thank to my extended family members Cavidan, Ebru and Selin for their understanding and companion. I am proud of being a member of such a family.

Lastly, I would not have been able to complete this work without sensible encouragements, precious support and endless love of Kubilay. Meeting him has been the best thing of my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AHF	: Attribute Hiding Factor
AIF	: Attribute Inheritance Factor
CBO	: Coupling Between Objects
COF	: Coupling Factor
DIT	: Depth of Inheritance Tree
FCM	: Factor Criteria Metric
GOF	: Gang of Four
IEEE	: Institute of Electrical and Electronics Engineering
ISO	: International Organization for Standardization
LCOM	: Lack of Cohesion in Methods
MHF	: Method Hiding Factor
MIF	: Method Inheritance Factor
MOOD	: Metrics for Object-Oriented Design
NOC	: Number of Children
OO	: Object-Oriented
POF	: Polymorphism Factor
RFC	: Response for a Class
UML	: Unified Modeling Language
WMC	: Weighted Methods per Class

CHAPTER 1

INTRODUCTION

The main goal of software engineering is to produce better software with high quality. However, control of quality is impossible unless it becomes quantifiable. Thus, fitting the software quality on a measurable basis is an important work on which many studies have been performed but is yet an open ended subject.

In the International Standard ISO/IEC 9126-1[1], a software quality model is established based on the Factor-Criteria-Metrics (FCM) Quality Model [2]. According to the standard, quality is affected by factors and factors are assessed via criteria. The main factors determining software quality are given as functionality, reliability, usability, efficiency, maintainability and portability. In the standard, mapping of these factors to criteria is described clearly but still it is ambiguous which criterion is mapped with which metrics. The factors defined are dependent on hardware used, technology involved, design of software, etc. The most design-dependent factor among all is maintainability. This makes this factor one of the favorites since most of the metrics are also design-dependent.

Establishing the bridge between software maintainability and the metrics derived from source code is among the most desirable accomplishments in software quality analysis domain. There are many software maintainability prediction models published in literature ([3],[4],[5]) that suggest a way to establish the relation between metrics and software maintainability.

When it comes to designing software, the most popular design technique is OO design. OO programming approach is more maintainable than the procedural ones [6] and it introduces the object concept as well as features like inheritance, encapsulation and polymorphism [7]. OO design increases modularity compared to functional design. Modularity increases understandability thus maintaining the code becomes easier [8]. As OO paradigm has introduced new concepts and features, new metrics like MOOD [9] for measuring inheritance, polymorphism, coupling and data hiding and Chidamber and Kemerer metrics [7] for class level code measurement has been developed.

There is accumulated experience on OO design. One concept that has helped this accumulation is the design pattern which provides reusable solutions for common OO problems [8]. Using design patterns will result in a better design with properly used OO concepts even for a novice developer because they carry the experience of previous developers. In addition to that, each design pattern addresses a problem-solution pair and even if the problem is very complex, what the pattern addresses is simply expressed with the name of the pattern. Thus design patterns constitute a language between designers to express problem solution pairs [11].

The objective of this study is, first, to investigate the relation between OO metrics and software maintainability, and then, in terms of the obtained maintainability indicator metrics, to observe the maintainability change of an application due to the usage of design patterns.

Within the scope of this thesis, an application that has common functionalities with real life software used in ASELSAN Inc. is developed. Five design patterns from the sources [12], [13], and [14] are applied to this software. Before and during the application of these design patterns OO metrics are measured. These metrics are examined and a discussion is held about how the software design patterns affected the maintainability of the software. Lastly, from these metrics results, correlation is observed by calculating correlation matrices.

The remainder of the thesis is organized as follows:

In Chapter 2, concepts of OO software analysis and design are reviewed. A brief literature survey on software quality, maintainability, OO metrics and their relation to maintainability is presented. Software design patterns that are used in the experimental study are listed and explained.

In Chapter 3, the experimental work is described. Used software functionality, the application procedure of design patterns and collected metrics are presented. Maintainability of the software is evaluated in terms of the collected metrics. In the last part of this thesis, considering all metric results, correlation between the used metrics is observed.

In Chapter 4, the work done in the thesis is summarized and obtained results are reviewed. The achievements are compared with previous works and probable future studies are suggested.

CHAPTER 2

LITERATURE REVIEW

This chapter presents an overview of the basic concepts that lie beneath the experimental work. OO design concepts, software quality, metrics for maintainability and software design patterns are described in their own section within this chapter.

2.1. OBJECT ORIENTED CONCEPTS

OO programming uses objects and their interactions to build software. An *object* can be simply defined as an entity that has mainly two parts, namely data (state) and functionality (behavior). Data is stored within objects and it is accessed by means of functionality that the object has. By this way, each object in the software has its own data. In [8], this definition is told to reflect just the implementation level description of object. At conceptual level an object is a set of responsibilities like functioning properly and knowing its type and state [8].

Class is a first level abstraction of set of objects with common responsibilities [7]. An object is an instance of a class and class defines the type of the object.

OO design has some key concepts that are addressed in this thesis. They are encapsulation, inheritance, polymorphism and coupling.

Encapsulation refers to hiding the implementation details of a class. By using encapsulation, a data item or the working principles within a class cannot be reached directly [7]. This introduces the differentiation of the user and the developer of the class. For the developer side, encapsulation gives the easiness of changing data or inner principles of the class without modifying the interface, as well as the classes that use this class's services. On the user side, encapsulation provides simplicity since the user deals only with the services provided by the class. So, as long as the services provided by the class are defined clearly, encapsulation also offers fault reduction in software with multiple developers. This benefit of encapsulation is given with the OO principle called "Encapsulate what varies" in [15].

Inheritance is the way to derive new classes by using the base properties of a pre-defined class and it is a way of defining higher levels of abstractions [7]. In the concept of inheritance, there are mainly two classes, *base class* (also called *ancestor class*) and the *derived class.* Inheritance provides diversity by which the derived class inherits the properties of the base class while implementing new features.

Polymorphism stands for the change in the behavior under the same interface. In other words, polymorphism allows different objects to respond in its own way to the same method [7].

Coupling is defined as the dependency of a program module on other modules. In [16] coupling is referred as "a count of the number of non-inheritance related couples with other classes". However, in [10], inheritance also treated as a coupling relation.

2.2. SOFTWARE QUALITY

One of the most important tasks of the software engineering is to assure the quality of the software. Throughout the software life cycle, as the product emerges, quality must be taken into account and empirical data about quality must be collected in each phase of the software development process to feed the next phase. However, "quality", from the nature of the word, is not countable thus measuring quality and collecting empirical data on quality of software is not very straightforward.

Throughout the software development process some quality characteristics of software become essential. In the literature there are many of these characteristics and they are associated with some measurable metrics.

Software quality models are established to determine characteristics that affect quality, to form a set of metrics that measure these characteristics, to collect data that will help to evaluate the quality of software. One of the established quality models is the ISO/IEC 9126-1 [1] quality model. This model is an international standard based on the works of McCall (1977), Boehm (1978) and others [17].

ISO/IEC 9126-1 quality model provides six main software characteristics (also referred as factors) and some sub-characteristics (also referred as criteria) of these characteristics (Table 1).

Table 1 ISO 9126-1 Quality Model Characteristics and Sub-Characteristics

Characteristics	**Sub-characteristics**
Functionality	Suitability, Accuracy, Interoperability, Security
Reliability	Maturity, Fault tolerance, Recoverability
Usability	Understandability, Learnability, Operability, Attractiveness
Efficiency	Time behavior, Resource utilisation
Maintainability	Analyzability, Changeability, Stability, Testability
Portability	Adaptability, Installability, Coexistence, Replaceability

According to ISO 9126-1, ***Functionality*** is the software's capability of performing important functions. It has four sub-characteristics.

1. Suitability is the software's capability of beyond being functional also performing appropriate functions.
2. Accuracy is the software's performance of functioning in a right way and with the right output.
3. Interoperability of the software indicates how well it communicates with its environment.
4. Security is the closeness of the software functions to the unwanted outside users.

Functionality and its sub-characters are omitted in this thesis since the focus is on the changes done in the source code without changing the functionality of the software.

Reliability, according to [18], is the ability of software to perform its required functions under stated conditions for a specified period of time. It has three sub-characteristics:

1. Maturity is the length of the time between failures.
2. Fault tolerance is the ability of software to continue to its normal operation under the existence of faults in the system.
3. Recoverability is the capability of the software to bring itself back to its normal operation after a failure.

Usability is the easiness for the users of software to learn, understand, and use its functions. It has four sub-characteristics:

1. Understandability is the degree of which the purpose of the software is clear to the evaluator.
2. Learnability is the degree of easiness of learning functions and usage of the software.
3. Operability is the degree of easiness of operating the system in right way.
4. Attractiveness is the ability of the software product to attract user's attention.

Maintainability, also referred as supportability, is the convenience about finding bugs in the software and fixing them or modifying the software for expanding its features or to adopt it to a new environment. It has four sub-characteristics.

1. Analyzability is the ability to find the reasons of the failures.
2. Changeability is the easiness of doing changes within the software.
3. Stability is the scarceness of risk of failures after a change in the software.
4. Testability is the verifiability of the new changes in the software.

Efficiency, also referred as performance, is the effective use of system resources, such as time and storage, to fulfill a task.

1. Time behavior covers the response and processing time and throughput rates of the software when a specific task is being completed.
2. Resource utilization is the usage of the CPU and memory appropriately when the software performs its functions.

Portability refers to how well the software can adapt to changes in its environment.

1. Adaptability is the degree of the system satisfying different system constraints and user needs.
2. Installability characterizes the effort needed to port the software to its usage environment.
3. Coexistence is the ability of the software to operate in an environment with other software.
4. Replaceability is the easiness of exchanging given software within specified environment.

It is very important to place these characteristics on to a measurable basis in order to improve the software in term of these characteristics. Software quality metrics are proposed to form this basis.

According to the ISO 9126-1 Quality Model, metrics to be used are not explicitly defined, rather left to the user to be defined. In [19] which is a continuation of the standard ISO/IEC 9126-1 and as of yet not an international standard but a prospective one, some quality metrics are presented for use in the quality assurance of software. Since these metrics do not measure directly the change in quality due to change in source code and since this thesis will focus on the changes in software due to applying design patterns, they are omitted and more suitable metrics from the literature are used.

The quality characteristics mentioned in ISO 9126-1 Quality Model may gain different levels of importance according to different domains. Indeed developing software whilst keeping the above characteristics at a high level is a hard job. A good design approach will be to determine which characteristics are more important for domain in use and to continue with a goal of keeping that attributes at a high level.

Maintainability, being the characteristic that is most dependent on the design of the software, will be considered within the scope of this thesis to see how it is affected by the application of design patterns. It is important to note here that ISO 9126-1 Quality Model is referenced here to see the important quality characteristics of software as a whole. However, it is not the only reference used in this thesis. In particular, maintainability of software is an attribute which is affected by the other sub-characteristics like fault-proneness and understandability. Even if they are categorized under different characteristics, for the sake of covering the meaning of maintainability in a more general way, we refer to these sub-characteristics as indicators for maintainability.

In the following sections, OO metrics which are chosen within the scope of this study are explained in detail. The literature that has investigated their relation to maintainability is reviewed.

2.3. METRICS FOR MAINTAINABILITY

In software engineering, reducing the cost and effort needed to complete a software product is as important as developing a software product meeting its functional requirements. That is why maintainability of the software is an important concept in software engineering. Highly maintainable software is;

- open to changes which means effort to adopt the software for new environments and to add new features is less;
- stable and close to faults which means as the code is changed the risk of failure is not increased;
- analyzable and testable which means it is easy to understand, debug the code and find bugs [17].

Software with such qualities can be upgraded and maintained, adapted (in)to new software projects easily and lastly new developers can be easily included to the software project and different developers can reuse code in their projects easily.

Therefore, maintainability is very important; however, it is not easy to develop highly maintainable software. Since it is hard to find directly coupled, well validated and code driven metrics measuring maintainability and to obtain a tool or methodology giving direct clues of increasing maintainability. There are many practices and investigations in the literature ([20], [3], [4], [5]) but still it is difficult to find a direct measure of maintainability.

In this part of the thesis, some widely used OO metrics are given with their definitions and their effect on maintainability is discussed. Throughout these discussions, clues relating the metrics to maintainability directly or indirectly, by first focusing on effects to testability, analyzability, changeability, understandability, stability and fault proneness, are investigated.

There are many OO metrics but the prominent ones among them were listed by Chidamber and Kemerer in 1994 [10]. They are:

- *Weighted Methods Per Class (WMC)*
- *Depth of Inheritance Tree (DIT)*
- *Number of Children (NOC)*
- *Coupling Between Objects (CBO)*
- *Response For a Class (RFC)*
- *Lack of Cohesion in Methods (LCOM)*

These metrics, except the last one, has been examined in detail with regard to fault proneness by Aydınöz in [21]. He has discussed the relation of these metrics to fault proneness by referencing to the literature. His conclusions are referenced in the individual metric sections.

The other remaining metric, LCOM, measures how much a class is far from being cohesive. Cohesiveness increases the *analyzability* and *testability* of the class and therefore, software. And since fault-proneness is closely related with the *stability* of the software these metrics are worth looking at closely and individually in terms of maintainability.

Other relevant and important OO metrics are six MOOD (Metrics for Object-Oriented Design) which are established by Abreu and Melo [9]. They are:

- *Method Hiding Factor (MHF)*
- *Attribute Hiding Factor (AHF)*
- *Method Inheritance Factor (MIF)*
- *Attribute Inheritance Factor (AIF)*
- *Coupling Factor (COF)*
- *Polymorphism Factor (POF)*

These metrics aim to measure goodness of the overall OO design in terms of encapsulation, inheritance, polymorphism and coupling.

Encapsulation effects analyzability by hiding complexity behind simple interfaces and stability by decreasing fault proneness by forbidding access to class from outside.

As the use of *Inheritance* increases in a design, it will result in wider and deeper inheritance trees, which are clues of complex design. So using inheritance excessively in design will decrease analyzability of the software. Also, since inherited classes may inherit methods or attributes or may override methods of its base classes, it is difficult to understand the exact behavior of the class with a glance to class only.

In [22] *Coupling,* also referred as the *message-passing,* is said to be indirectly related with the complexity, lack of encapsulation, lack of reuse potential, lack of understandability, lack of maintainability.

Polymorphism, being another OO concept addressed by the MOOD metrics, is expected to decrease the traceability of the software since polymorphic functions bring ambiguity. Excessive use of polymorphism will decrease analyzability and understandability of the software. In [22] first and second of the MOOD metrics are proposed to measure encapsulation, third and fourth are proposed to measure inheritance, fifth is proposed to measure coupling and sixth is proposed to measure polymorphism. Therefore, AHF, COF and POF can be concluded to be closely related with maintainability or its sub-characters due to effects of coupling, polymorphism and encapsulation to maintainability. When it comes to other MOOD metrics, in [23] MIF and AIF metrics are criticized to lack measuring inheritance and MHF metric lacks covering encapsulation paradigm. So these metrics' relation to maintainability needs more support rather than OO paradigm.

The discussion about these metrics is given in detail under the individual metric captions in the continuing part of this chapter. They are also grouped into two parts according to their scope, i.e. class and application level metrics.

2.3.1. CLASS LEVEL METRICS

This kind of metrics' scope is limited with the class only. They are useful when the parts of the software are to be compared within an application. So modules with low metric values or high metric values can be detected. However, when different applications are to be compared in terms of these metrics, an overall metric result is needed since the classes in applications are not the same and cannot be compared individually. In that case mean values of the metrics over classes in the application can be considered. This is the common way usually encountered in data mining works like [4].

2.3.1.1. WEIGHTED METHODS PER CLASS (WMC)

Definition: [10]

Given a class *C*,

$WMC(C) = c_1+c_2+...+c_n$,

where c_i is the complexity of M_i which is a method of *C* and *i* is from *0* to *n*, n being number of methods of *C*.

In [10], the method to calculate complexity of the methods is left to the concept in use. It is very common to choose it as the McCabe's Cyclomatic Complexity (CC). CC is defined as the maximum number of linearly independent execution paths in a program [24]. It is calculated as cyclomatic number of a program control graph; where cyclomatic number of a graph G is defined as $V(G)= e-n+2p$; where e is the number of edges, n is the number of nodes and p is the connected components in G. To obtain the control graph of a program e is calculated as the number of branches, n is taken as the number of code blocks and p as connected program segments.

It is a very widely used complexity measure [25]. In this thesis CC is also chosen to calculate WMC values of the classes.

Interpretation:

Complexity is an indicator of the effort used for developing and to understand a code segment. Complexity is a good indicator of fault prone software. Fault prone software is not stable. WMC is the OO version of the complexity. High values of this metric means a class with many complex methods. Therefore, this metric can be chosen to be an indicator of maintainability since it affects fault proneness, stability and understandability. In [26], WMC is found to be correlated with the effort needed to test that class. Since testability being a sub-character of maintainability, we can conclude that as WMC of a class increases maintainability of that class decreases. WMC has already used in some maintainability prediction models like MARS [5] and TreeNet [4].

2.3.1.2. Depth of Inheritance Tree (DIT)

Definition: [10]

Depth of inheritance tree for a class *C* is defined as DIT of *C*, which is the maximum length from *C* to the ancestor classes of *C*.

Interpretation:

Longer trees decrease understandability of the code since as the number of ancestor classes increases for a class, the number of inherited functions increases and considering dynamic binding it is very hard to understand the run time behavior of an object of that class because when a method of it is called method can show its own behavior or its ancestors behavior. In [10], it is mentioned that higher DIT increases the number of methods that a class will likely to inherit making it hard to predict the behavior of the class and also it increases design complexity. Former statement supports that higher DIT decreases understandability and latter indicates that higher DIT increases fault proneness therefore, maintainability. Also in [27], DIT is showed to be a very strong indicator of fault proneness through experiments. In [26], it is observed that DIT is not correlated with the unit testing effort of a class but it is pointed that if the testing approach requires testing of all inherited methods DIT will likely correlate with the testability of the class. With these clues we can conclude that DIT is indicator of maintainability but not a strong one like WMC.

2.3.1.3. NUMBER OF CHILDREN (NOC)

Definition: [10]

Number of children of a class *C* is defined as the number of first level subclasses of *C*.

Interpretation:

Both DIT and NOC are inheritance metrics. In [26], case studies showed that NOC is uncorrelated with the effort needed for testing the class however, a class with high NOC will be likely to inherit its errors to child classes, and must be thoroughly tested. On the other hand, since higher

NOC is an indicator of more inherited methods, testing cases used for the parent methods will be reused in child methods which points decrease in the testing effort. Experiments done in [27] shows that as NOC increases the fault proneness decreases, it is probably because as the children class number increases the attention cared for the base class increases since many classes depend on it, however, Briand at. al. conclude that NOC does not have a strong impact on fault-proneness. Looking from the testing and fault proneness side of maintainability NOC seems to have low indication of maintainability. On the other hand, Kemerer et. al. in [10] state that high values of this metric point to the likelihood of improper abstraction which leads to misunderstandings about the design. This can be considered as a negative impact on understandability and analyzability of the software. These discussions show that NOC is not a strong indicator of maintainability. It is included in our testing suite to evaluate its indication experimentally.

2.3.1.4. Coupling Between Objects (CBO)

Definition: [10]

CBO for class *C* stands for the number of other classes that *C* is coupled. A class is coupled to another if one uses other's methods, attributes or one is inherited from the other.

Interpretation:

There are two types of coupling, internal and external. Internal coupling of a class *C* (fan-out of *C*) stands for how much *C* uses other classes' services. On the other hand external coupling of *C* (fan-in of *C*) show how much *C*'s services are used by other classes. In [27], it is showed that internal coupling increases fault proneness of class. It is expected since a class highly coupled to other classes will be unaware of the changes done to other classes and this may cause the class to function wrongly or errors to be carried to the class. However, external coupling does not increase fault-proneness of the class itself since how much a class is used by others does not affect the class but external coupling means internal coupling for other classes and high external coupling increases overall coupling in software. So both coupling types affect fault-proneness of software. CBO, containing both of these couplings, is a predictor of fault proneness therefore, maintainability. High CBO signals poor and complex design, decreases modularity and reuse, complicate testing of the class and as a result decreases understandability and testability. So, CBO constitutes a measure for maintainability.

2.3.1.5. RESPONSE FOR A CLASS (RFC)

Definition: [10]

RFC of a class C is the cardinality of the set of methods that belong to C or is invoked by methods of C.

Interpretation:

RFC is another measure of internal coupling since it contains the number of methods that the class is coupled to. In [27], internal coupling is found to be the strong indicator of the class fault proneness. According to Chidamber and Kemerer [10], RFC indicates the complexity of a class and gives a measure of testing time. In [26], this statement is supported by experimental data since when a class is needed to be (unit) tested, tester has to cope with the initialization of the instances of classes whose methods are invoked by the class. To conclude, RFC affects the understandability, fault proneness and testability against maintainability. Therefore, as RFC increases, maintainability decreases.

2.3.1.6. Lack of Cohesion in Methods (LCOM)

Definition: [10]

Degree of similarity of two methods M_1 and M_2 of C_i is given by the number of common instance variables they use. Let *P* be the set of pairs of methods with degree of similarity zero and *Q* be the set of pairs of methods in *C* with degree of similarity being positive. Then LCOM of *C* is defined as:

$$LCOM = \begin{cases} |P|-|Q| & \text{if } |P|>|Q| \\ 0 & \text{otherwise} \end{cases}$$

Interpretation:

LCOM for a class being positive means methods of that class form disjoint sets of methods that are working on disjoint sets of attributes. Then this means as the LCOM increases number of uncorrelated services in the class increases. As an OO principle this type of classes should be divided according to the independent jobs they contain. Kemerer et. al. [10] proposed that high values of LCOM signal complexity and therefore, error proneness. However, in [27], Briend et. al. experimented that LCOM has no significant effect on error-proneness. In [26], two case studies showed different results; in one LCOM values increased with testability metrics whereas in the other decreased. It can be concluded that LCOM is uncorrelated with the maintainability of the software under these discussions. It is included in the metric suite to experimentally evaluate its level of meaningfulness.

2.3.2. APPLICATION LEVEL METRICS

Application level metrics give an overall result for an application. If partial results are to be measured the application code should be divided before measurement and separate metric results should be calculated. By this way modules inside an application can be compared. In this thesis, since comparison of applications is considered these metrics will be used as they are.

2.3.2.1. METHOD HIDING FACTOR (MHF) & ATTRIBUTE HIDING FACTOR (AHF)

Definition: [9]

Let C_1, C_2, …, C_{TC} be the ordered set of classes in an application with TC being the total number of classes and let $M_d(C)$ represent the number of methods in class C. Also let M_{mi} be the mth method in the ordered set of methods of C_i.

$$MHF = \frac{\sum_{i=1}^{TC}\sum_{m=1}^{M_d(C_i)}(1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)} \quad \text{where:}$$

$$V(M_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1} \quad \text{where:}$$

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 & \text{iff } j \neq i \wedge C_j \text{ may call } M_{mi} \\ 0 & \text{otherwise} \end{cases}$$

Interpretation:

MHF metric measures how much invisible the methods of an application are to the classes of the application and AHF metric measures the same thing for attributes in the application. Both of these metrics are not class level but application level metrics, giving an overall value for all methods or all attributes of all classes in the application. In [9], these metrics are given as the predictor of encapsulation, however, as criticized in [23], encapsulation is not a concept of *data-privacy* only but also concept of *unity* and this metric indeed does not measure the unity of a class. Since data-privacy is what these metrics measure only, it is good to focus on the clues that data-privacy reveal about maintainability.

Keeping data and behavior private decreases the access to them from everywhere in the design. This can be thought as a sanction to keep the coupling low. On the other hand less the visor of an attribute or method less the modifier of the data kept in the design and easier to keep track of the changes done to the data. This is good for understandability and traceability of code. From these it can be concluded that data privacy, therefore, MHF and AHF are in favor of maintainability. However, there are not enough experiments supporting this idea except the one Abreu et. al. constructed in [9]. In the case study to investigate the effect of MOOD metrics to maintainability and reliability, it is observed that increase in MHF results in decrease in defect density and effort to fix defects. However, in that study, AHF did not show a significant correlation with maintainability, contrary to what was expected.

As result, the correlation of these metrics with maintainability should be experimented more. In this study these metrics are not used as maintainability predictors but kept in measurement suite.

2.3.2.2. Method Inheritance Factor (MIF) & Attribute Inheritance Factor (AIF)

Definition: **[9]**

Let $C_1, C_2, \ldots, C_{TC}$ be the ordered set of classes in an application with TC being the total number of classes.

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$ where:

$M_a(C_i)=M_d(C_i)+M_i(C_i)$ and

$M_d(C_i)$ = the number of methods declared in C_i,

$M_a(C_i)$ = the number of methods that can be invoked in association with C_i

$M_i(C_i)$ = the number of methods inherited (and not overridden) in C_i.

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$ where:

$A_a(C_i)=A_d(C_i)+A_i(C_i)$ and the definition of $A_d(C_i)$, $A_a(C_i)$ and $A_i(C_i)$ are same as $M_d(C_i)$, $M_a(C_i)$ and $M_i(C_i)$ respectively, except they apply to attributes instead of methods.

Interpretation:

These metrics measure inherited methods/attributes ratio to the all methods/attributes in the application. They are aimed to measure inheritance. However, in [23], it is criticized that MIF counts overridden methods as non inherited but overridden methods may also use their super classes' methods which means indeed they are not overriding but reusing their super classes' methods which is another way of inheriting. MIF metric can be enhanced to capture this property but here it is used as it is, keeping this point in mind.

Maintainability critique of these metrics is based on the research by Abreu et. al. In [9], MIF is found to be correlated with defect density and rework needed where AIF is found to be moderately correlated with these properties. According to these results MIF and AIF seem to increase with increasing maintainability. However, it should be noted that authors point that in these works proper amount of inheritance is used. In case of excessive use of inheritance, maintainability will start to deviate since the code will be hard to understand and test due to complex structure inheritance introduces as discussed earlier in DIT metric. Since big inheritance trees will lead to excessive use of inherited methods and attributes in the code, high values of DIT and CBO may increase AIF and MIF.

As a result, these two metrics are not used in the experiments as maintainability predictors since results do not show strong correlation with maintainability.

2.3.2.3. COUPLING FACTOR (COF)

Definition: [9]

Let C_1, C_2, …, C_{TC} be the ordered set of classes in an application with TC being the total number of classes.

$$COF = \frac{\sum_{i=1}^{TC}\sum_{j=1}^{TC} is\_client(C_i, C_j)}{TC^2 - TC}$$ where:

$$is\_client(C_c, C_s) = \begin{cases} 1 & iff\ C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 & otherwise \end{cases}$$

where $C_c \Rightarrow C_s$ means C_c (a client class) has a non-inheritance relationship with C_s (a supplier class).

Interpretation:

This metric measures the amount of coupling between classes in an application. This metric and CBO are strongly related metrics except that CBO includes inheritance relations and measures coupling for a class only. The discussion about coupling and maintainability and CBO metric is valid for this metric as well. In [9], experiments showed that this metric is highly negatively correlated with defect density and failure density. So it can be easily concluded that COF has negative correlation with maintainability of the software.

2.3.2.4. POLYMORPHISM FACTOR (POF)

Definition: [9]

Let C_1, C_2, …, C_{TC} be the ordered set of classes in an application with TC being the total number of classes.

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} \left[M_n(C_i) \times DC(C_i) \right]}$$

where $M_d(C_i) = M_n(C_i) + M_o(C_i)$ and

$M_n(C_i)$ = the number of new methods,

$M_o(C_i)$ = the number of overriding methods,

$M_d(C_i)$ = the number of declared methods,

$DC(C_i)$ = the descendants count.

Interpretation:

This metric is defined as the ratio of the methods inherited from base classes but overridden to all methods taken from base classes not necessarily overridden or inherited. In [23], this metric is found to be correlated with maintainability predictor metrics like failure density and rework effort. But as in the case of MIF and AIF, for high values of the metric maintainability will start to decrease. So it can be concluded that for low value of POF as POF increases maintainability of the software increases.

2.3.3. METRIC vs MAINTAINABILITY OVERVIEW

Table 2 OO Metrics versus Maintainability

Metric	**Maintainability**
WMC	↓
DIT	↓
NOC	⇔
CBO	↓
RFC	↓
LCOM	⇔
MHF	⇔
AHF	⇔
MIF	↑ ↓
AIF	⇔
COF	↓
POF	↑↓

The meanings of symbols used in Table 2 is provided below:

↓ : maintainability decreases with the increasing metric value

↑ : maintainability increases with the increasing metric value

⇔ : maintainability and metric seems uncorrelated according to the survey

↑ ↓ : for low values of metric maintainability increases with the increasing metric value, but for high values maintainability decreases with the increasing metric value

In previous section, each metric is reviewed in terms of its effect on maintainability. In Table 2 the results of this investigation is summarized.

2.4. SOFTWARE DESIGN PATTERNS

In this section, interpretation of software design patterns will be given and some design patterns ([12], [13], [14]), which are considered to be applicable for the experimental phase of this thesis, will be examined and discussed in detail.

Software design patterns are one of the most important and appealing topics of the part of the software engineering interested in problem solving discipline in OO design. As in the other disciplines of engineering, it is important to construct a common vocabulary for expressing the concepts in software engineering too. OO design problems being the concept of software engineering under discussion; software design patterns can be thought of as the vocabulary for expressing the problem solution pairs. This vocabulary constructs a common language within software engineers, allowing them to carry their experience to a cumulative platform and to find solutions from this platform for problems they encounter.

Origins of the concept of software design pattern go back to 1967 but it has become popular with the publication of the book Design Patterns: Elements of Reusable Object-Oriented Software [15] whose the authors have been known as *Gang of Four (GoF).* And the patterns presented in the book are usually referred as *GoF Patterns*.

2.4.1. WHAT IS DESIGN PATTERN?

In [11], a definition for design pattern is given as:

Design Pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.

By this definition, the emphasis is drawn onto *name*, *recurring problem in a context* and *proven solutions* which are three main parts of a design pattern. A design pattern should have a name in order it to have its place in the software vocabulary. By this name, the problem solution pair it addresses will be clear. A design pattern should always point to a well proven solution, stating it clearly, for a common problem in a specific domain so that one having the same problem in the same context can easily adopt the solution the pattern addresses to its own context.

Patterns provide reuse of experience rather than reuse of code. They do not provide a solution for a whole application or subsystem, so they are not plug & play tools, where you can place in your code and obtain a working application; indeed it is very unlikely to be able to compile them alone. However, they pass the knowledge of developers that previously used them successfully to the new ones; they provide clear understanding of their design.

Design patterns are themselves presented in patterns which show a way of documenting a design pattern. It may slightly extend but a general pattern of design patterns will contain these essential elements [11]: Name, Problem, Context, Forces, Solution, Examples, Resulting Context, Rationale, Related Patterns, and Known Uses. Being a different example, GOF used these elements of patterns in their book: Pattern and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses and Related Patterns.

2.4.2. LIST OF SOME DESIGN PATTERNS

In this section, list of software design patterns which are used in Chapter 3 of this thesis will be given along with their description.

2.4.2.1. REACTOR PATTERN [12]

This pattern, also known as Dispatcher or Notifier, provides event-driven applications to demultiplex and dispatch service requests that are delivered from one or more clients in one place. This means each client's event handling are not done separately but done all in one place i.e. in Reactor Pattern. This pattern applies to applications that deals with multiple events simultaneously but consumes them synchronously and serially. A classical way would be having multiple tasks that will wait for the events to occur for each client. By reactor pattern a task will wait for any kind of event by the select facility of the operating system and send these events to the related concrete event handlers.

In Figure 1, the class diagram for Reactor Pattern is given and in Figure 2, the sequence diagram for the pattern is given. Whenever a file descriptor is created by the program it is registered to the Reactor and Reactor keeps a map of these file descriptors by the Handle object, coupled with a concrete Event Handler object. Reactor's *handle_events* method is called periodically in a loop by the main program. This methods checks if an event like *read, write, accept* exists in the handles and dispatches the event to the Event Handler coupled with the corresponding handle. Event Handler's *handle_event* method is a factory method (method that is overridden for the child classes and gains different behavior for each child, which is a direct use of polymorphism concept). By this way each concrete Event Handler handles the event in their own way.

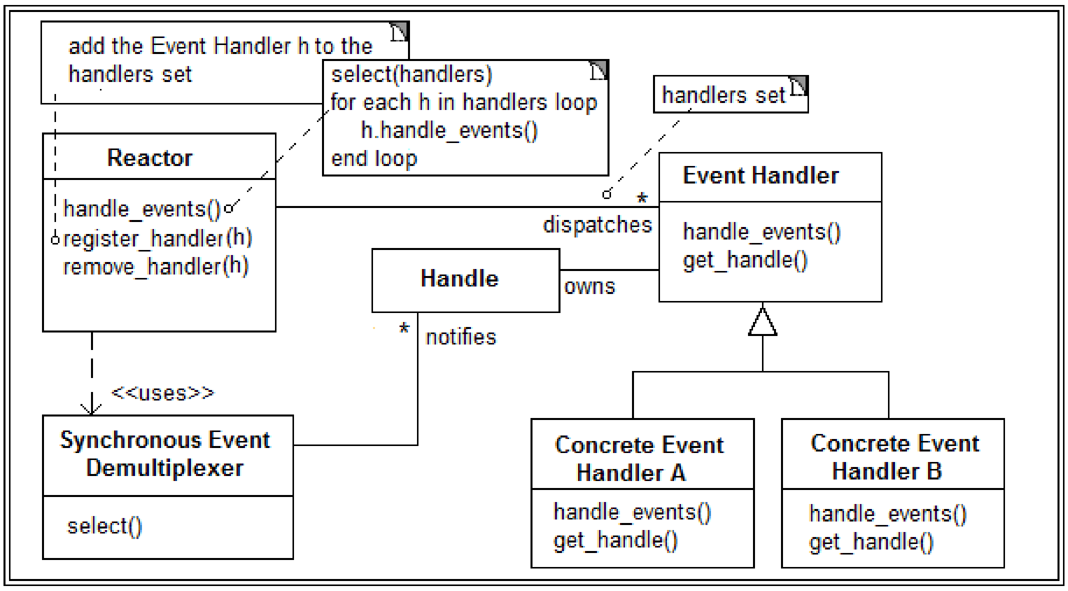

Figure 1 Reactor Pattern Class Diagram [12]

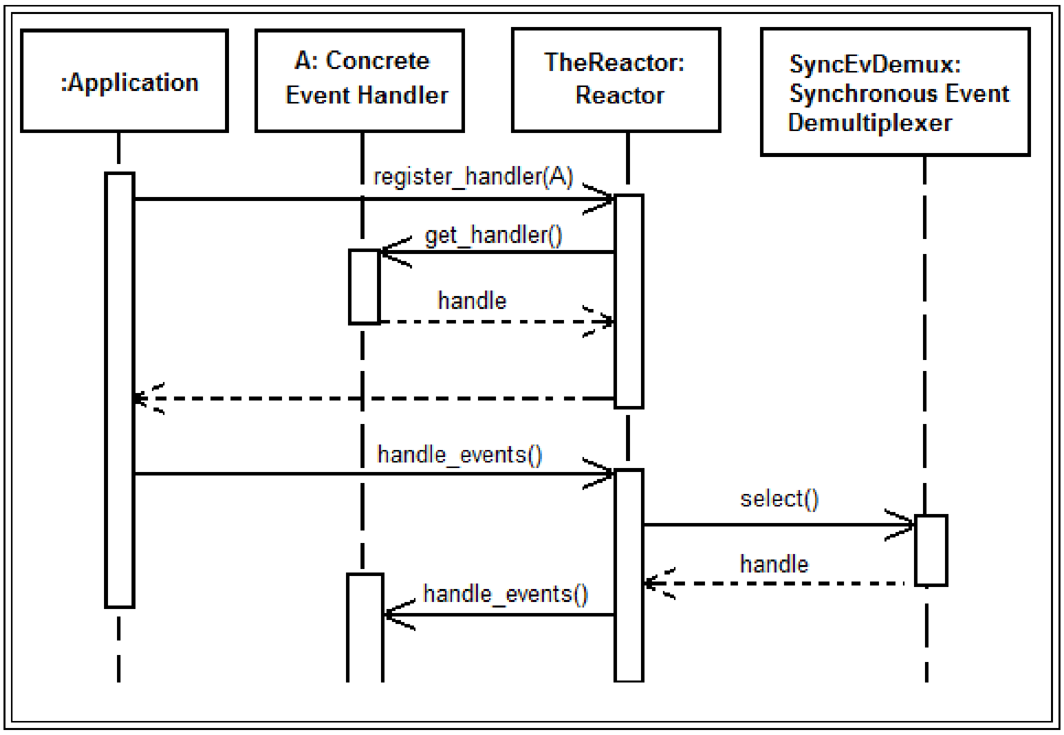

Figure 2 Reactor Pattern Sequence Diagram [12]

2.4.2.2. ACCEPTER CONNECTOR PATTERN [12]

Usually if two programs are communicating, one side is passive i.e. waits for connection, the other side is active i.e. initiates the connection. Whenever they establish connection they start to receive and forward data. This pattern separates the concept of establishing connection and handling other events than initializing connection.

In [14], this pattern is advised to be used with Reactor Pattern. So the class diagram given in Figure 3 is adapted to a case where these two patterns are implemented together. In Figure 4 the sequence diagram is given for a server side of the communication which listens to connections and accepts the arriving ones.

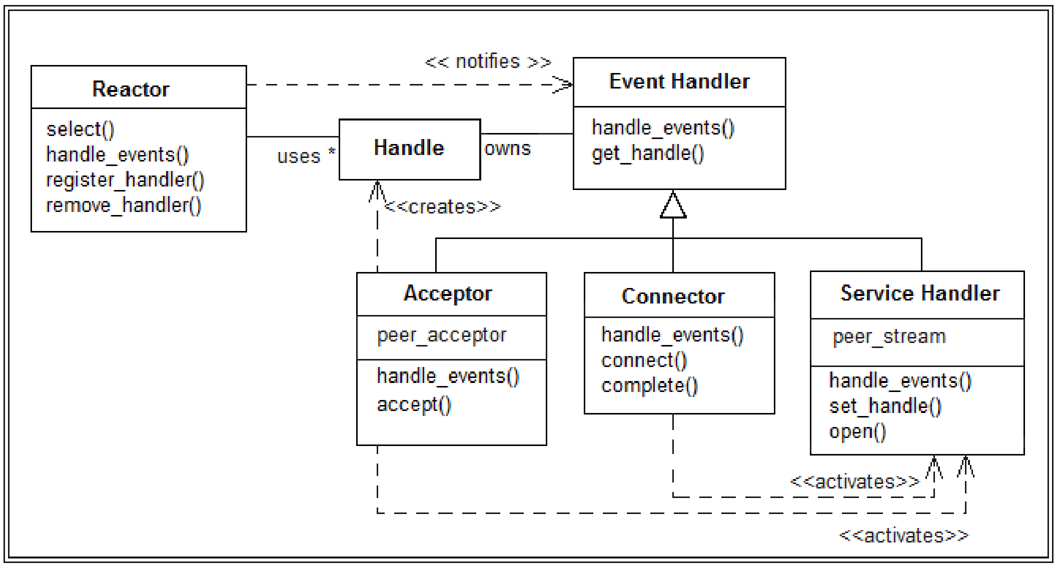

Figure 3 Acceptor Connector Pattern Class Relations [12]

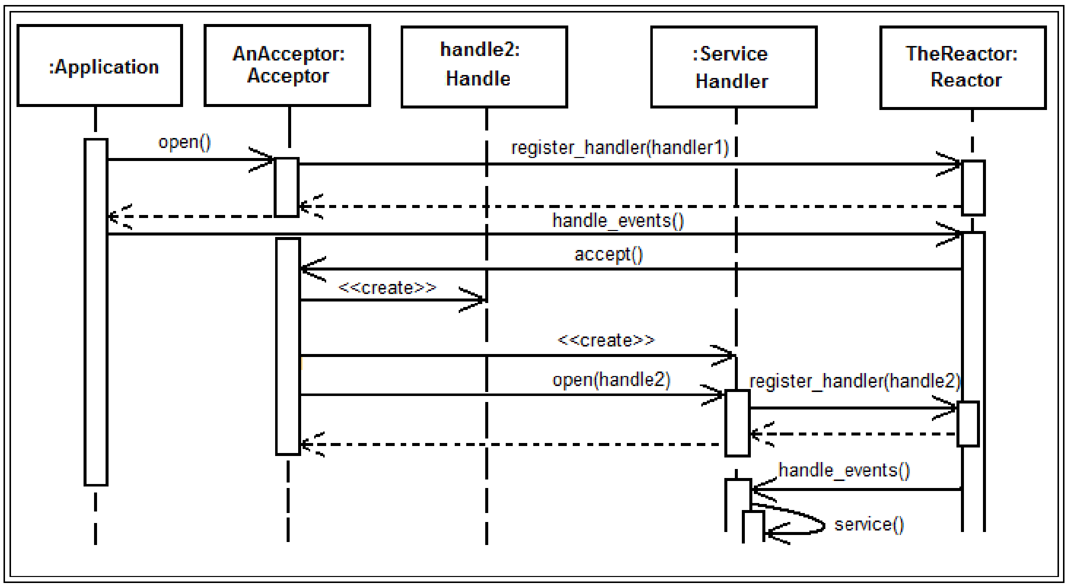

Figure 4 Sequence Diagram for Accepting Case [12]

2.4.2.3. Forwarder Receiver Pattern [14]

This pattern is applied to peer to peer systems and it provides decoupling of the communication details from peers. A Forwarder class for sending messages and a Receiver class for taking messages will hide and handle the communication details and protocols from their peer classes.

In Figure 5, the class diagram for the pattern is given by showing the both sides of the communication ends, and in Figure 6 the sequence diagram is given to show the run time characteristics.

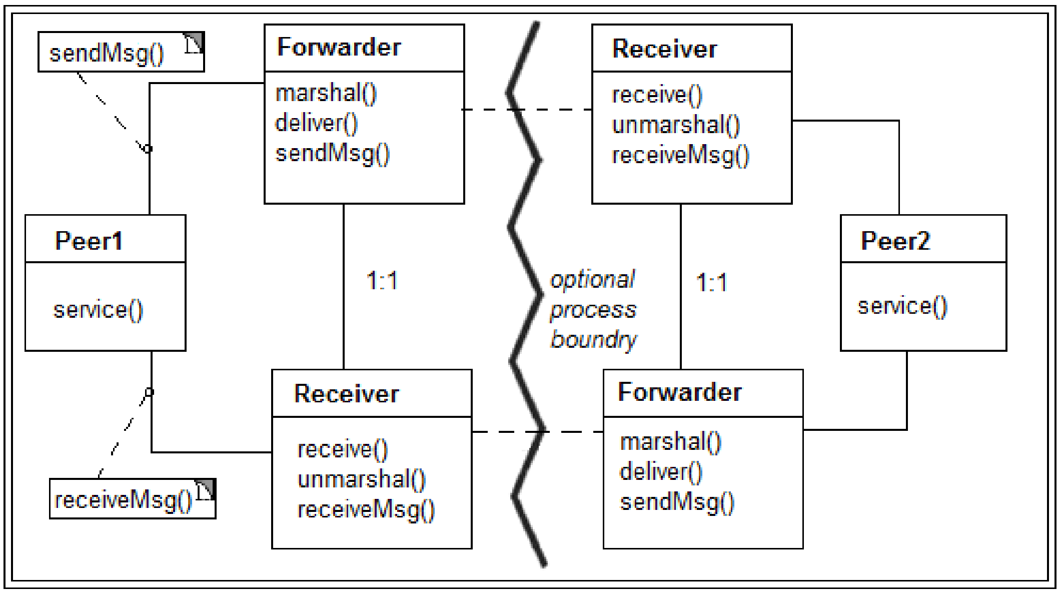

Figure 5 Forwarder Receiver Pattern Class Diagram [14]

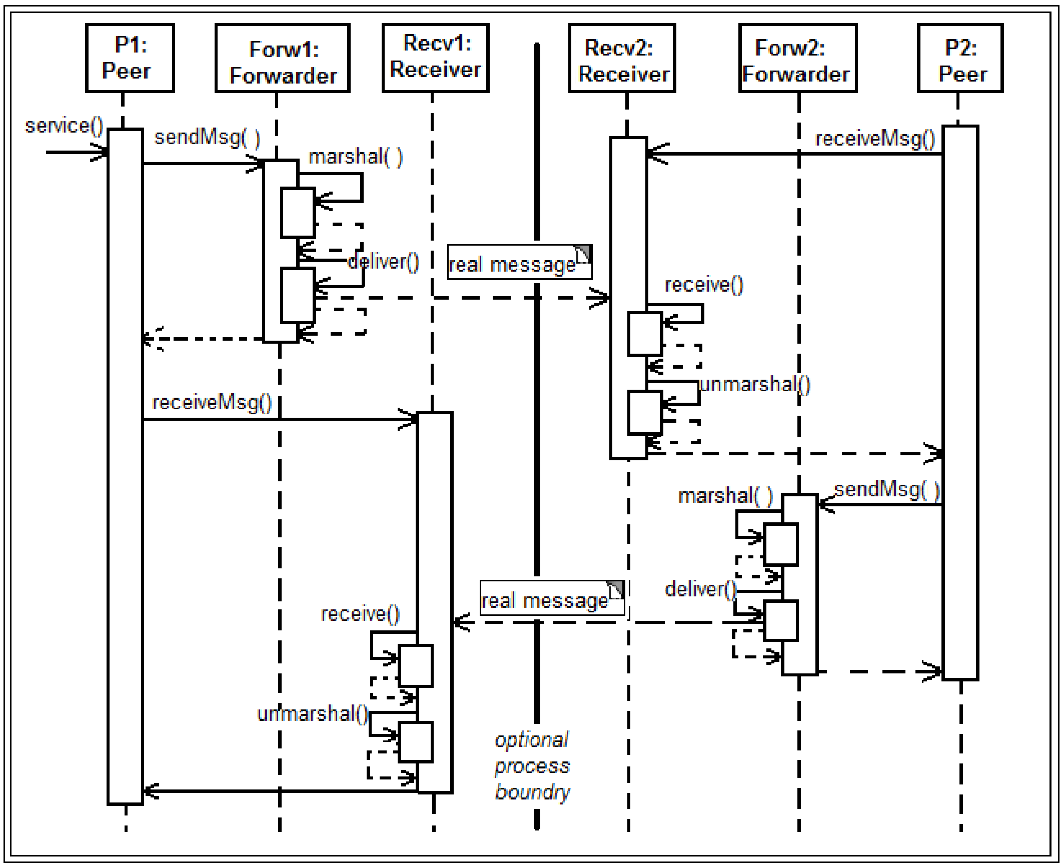

Figure 6 Sequence Diagram for Forwarder Receiver Pattern [14]

2.4.2.4. SMART POINTER PATTERN [13]

This pattern aims to overcome the deficiency of C++ language about pointers that is the unintentional pointer dereferencing. This mistake can cause the software to crash and sometimes it is very hard to debug these kinds of mistakes. Smart Pointer Pattern approaches the pointer as an object and adds to it the intelligence of holding the reference count to the object it points to. As the number of references vanishes the pointer itself destroys the object it points to so that there is no memory leak.

This pattern reminds Proxy pattern, a GoF Pattern since the Smart Pointer acts as a proxy for the real pointer. But the structure of the pattern is different which can be seen in Figure 7. This pattern wraps the pointer by a template class.

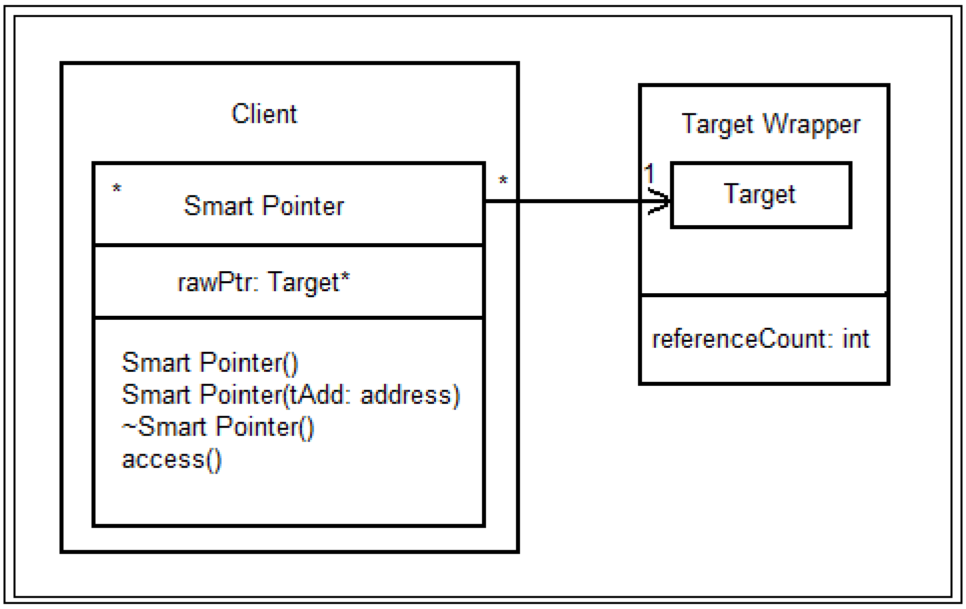

Figure 7 Smart Pointer Pattern Class Diagram [13]

2.4.2.5. COMMAND PROCESSOR PATTERN [14]

This pattern is for event driven applications where a request comes to the system and an action is performed according to the request. This pattern separates the request and the action. It has the ability of storing state and returning to the last state by undoing last action taken.

This pattern is slightly modified version of Command Pattern in GoF. The class diagram for the pattern is given in Figure 8 and in Figure 9 sequence diagram is given showing how a command object is created due to request in the system and how the request is handled.

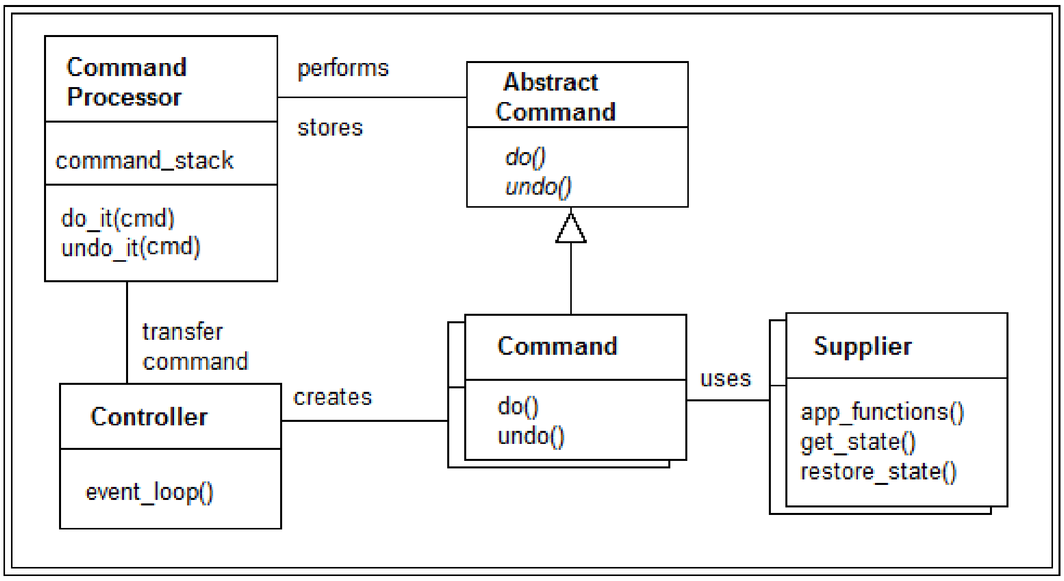

Figure 8 Command Processor Pattern Class Diagram [14]

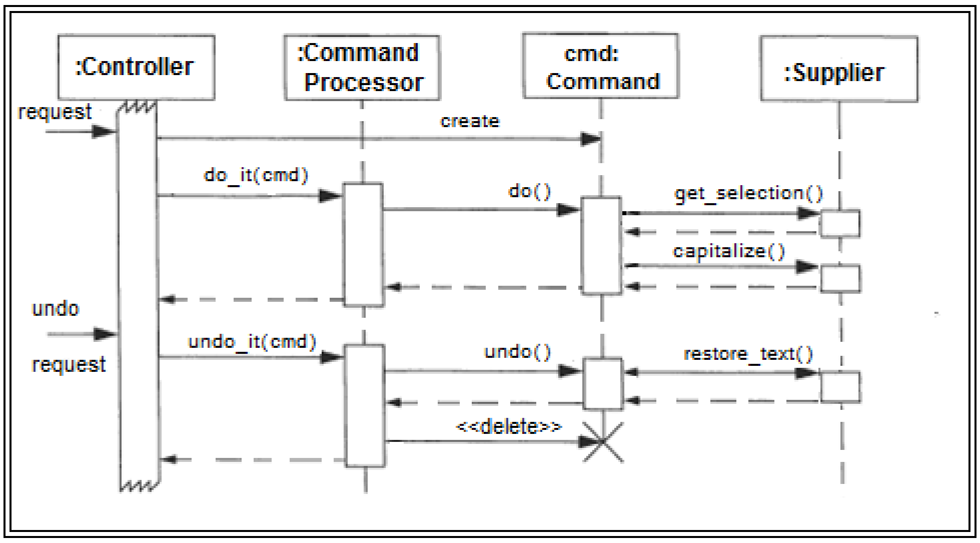

Figure 9 - Command Processor Pattern Sequence Diagram [14]

CHAPTER 3

EXPERIMENTAL WORK

By connecting the information pieces given in Chapter 2 the following question may arise:

"How will applying given design patterns affect maintainability of an application?"

It is this chapter along with the next where an element of the solution set lies. Finding even a good approximated answer to this question will need much more measurements on as many different cases like the one in this chapter. So changing the question a little bit, what this thesis will reveal the answer for will be clear.

"How will applying given design patterns affect the given maintainability metrics of the given application?"

In this chapter, the description and the use of software which lacks the design patterns given in Chapter 2 but is suitable for application of each of them will be given. The tools used for measuring and developing the software will be explained. The methodology of applying design patterns and taking measurements will be stated. Then the measures of metrics for the software and new versions of the software obtained by application of a design pattern will be given. The design patterns will be applied in a cumulative way to the software. So the whole process can be thought of refactoring of the software by the given design patterns and collecting the metric results at each refactoring step. Lastly, from obtained metric data correlation matrices will be calculated and the relation between metrics will be discussed.

3.1. DESCRIPTION of THE SOFTWARE

The application used in this thesis will be named as Routing Software. As the Routing Software is developed with application of new design patterns, a numeric suffix will be appended, like Routing Software 3, to indicate the version.

The very first version of the **Routing Software** is a simplified version of real life embedded software. Its main job is to communicate with other software over TCP/IP sockets. There are four other applications that this software communicates and the software routes predefined messages from one to another. The other applications that this software communicates are called peers throughout this thesis. Peers can connect to or accept connections from **Routing Software**. As soon as they establish connection they start sending messages to the other three peers.

The reason for choosing such functionality, i.e. routing, is that it is very common to have software that has to communicate with another module. For a system, as the number of applications that have to communicate with each other increases, the complexity of the design increases. So a solution will be the addition of new software that will communicate with each module and carry their messages to each other. This kind of software can be enriched by adding new functionalities. But the common functionality will be routing. So, **Routing Software** is developed with the aim of isolating this functionality.

3.2. EXPERIMENTAL METHODOLOGY

The practical work done for this thesis can be investigated under three main parts. The first part consists of obtaining the **Routing Software** and the peers that would be used in the experimental process. Second part is the application of the design patterns to **Routing Software** and taking measurements. Third part is examining the data obtained in the previous part.

In the first part, the applications that are developed in ASELSAN INC. and that have the functionality of communicating with other software units are examined. Then, the most common design to implement the communication of these applications is detected. With this design, **Routing Software** is developed isolating the functionality of communication. Since this software will need other software, i.e. peers, for communicating, an application simulating four different peers that will establish communication with **Routing Software** is developed**. Routing Software** is tested to be able to establish connections with four peers and route the defined messages among these peers. This version of the software is labeled **Routing Software 0** and the metrics values that are obtained by this version are called **Phase 0** results. The twelve metrics that are used in this process and whose detailed definitions can be found in Chapter 2, are:

- *Weighted Methods per Class (WMC)*
- *Depth of Inheritance Tree (DIT)*
- *Number of Children (NOC)*
- *Coupling Between Objects (CBO)*
- *Response For a Class (RFC)*
- *Lack of Cohesion in Methods (LCOM)*

- *Method Hiding Factor (MHF)*
- *Attribute Hiding Factor (AHF)*
- *Method Inheritance Factor (MIF)*
- *Attribute Inheritance Factor (AIF)*
- *Coupling Factor (COF)*
- *Polymorphism Factor (POF)*

The second part of the experimental process is divided into four phases. In each phase one or more design patterns are applied to **Routing Software** and metric values are taken for each version. The phases and applied design patterns are as follow:

- ***Phase 1:*** *Accepter Connector Pattern and Reactor Pattern*
- ***Phase 2:*** *Smart Pointer Pattern*
- ***Phase 3:*** *Forwarder Receiver Pattern*
- ***Phase 4:*** *Command Processor Pattern*

The metric values obtained for each phase is compared with the results of the previous phase and a discussion is held about how application of the pattern affected the maintainability of the software. In this discussion, increase in WMC, DIT, CBO, RFC and COF metrics will be considered to imply decrease in maintainability; increase in MHF will be considered to imply increase in maintainability. NOC, LCOM, AIF, MIF and AHF metrics will not be taken into account for maintainability predictions. And lastly increase in POF metric will be considered as increase in maintainability only if it has significantly low values. This method relies upon the discussions held in Chapter 2, and Table 2 serves as summary for this discussion.

In the third part, considering the results in all phases, a discussion is held about how the metrics are related with each other.

3.3. TOOLS THAT ARE USED DURING THE PROCESS

For developing **Routing Software,** Rational Rhapsody [28] is used. The peer applications are also developed with this tool. The reason for choosing this tool is that the applications that give the inspiration to **Routing Software** are also developed with this tool. So the results obtained will be more applicable to the existing software.

For taking metric measurements, IBM Rational Logiscope [29] metric evaluation tool is used. This tool has the ability to measure all the metrics explained in Chapter 2.

For calculating correlation matrices of metric data, Mathworks Matlab [30] is used.

3.4. EXPERIMENTAL PROCESS

3.4.1. PHASE 0

In this phase no design pattern is applied but the very first version of the **Routing Software** is formed and metric values are obtained for this version in order to compare it with next phases' results. **Routing Software** is formed by reusing the common parts of the software in ASELSAN Inc. that has the requirement of communicating with several other applications.

The **Routing Software** is designed to have a layered structure. Bottom layer is responsible for establishing the connection over TCP/IP, transmitting and receiving data to/from socket. So, this layer is called

Communication Layer. This layer provides communication with peers however, does not have any information about peers other than their physical address. This layer contains socket classes that try to establish connection with the given physical address and as the connection established they pend on the socket until data is received from the peer. Received data is sent to middle layer classes. Whenever the classes in the middle layer want to send data to other side, they send it to these socket classes and the raw data is sent over TCP sockets. For each peer in the system an instance of socket class which is reactive is created so there are threads as many as peer numbers.

Middle layer is responsible for handling the data specific to peers. This layer contains classes for each peer. They know about message protocol of their peer. When data arrives to this layer as a byte array from Communication Layer these classes know meaning of every byte. So they convert the received data into specific data classes' instances. This layer is called Interface Layer since the layer keeps the peer's interface details.

Upper layer is called Application Layer. This layer is responsible for what to do with the data objects i.e. what kind of data is received from peers and which data will be redirected to which peer. The architecture used in the application is a sample of *Embedded Control Software Architecture,* which is discussed in [31].

In Figure 10, these layers are shown with the units they contain. These units can be different instances of same class (as in Socket) or group of some class instances (as in Peer$_N$, it has objects of data classes as well as the related Peer$_N$ object).

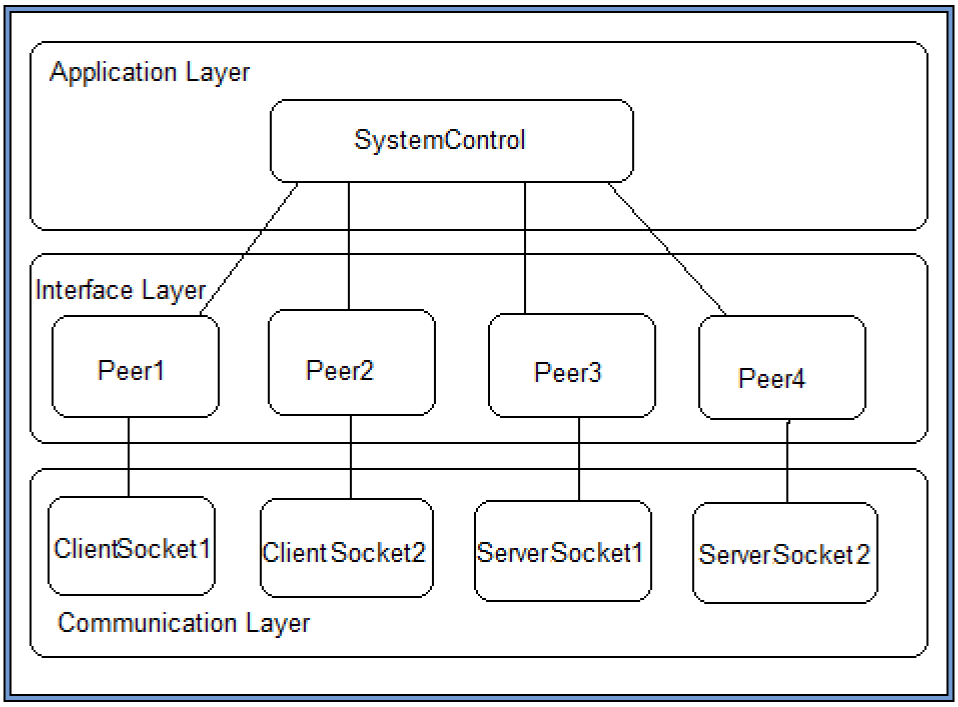

Figure 10 Layers and Units in Routing Software 0

Table 3 Application Level Metric Measurements for Phase 0

METRIC	Value
MHF	0.109898
AHF	0.961353
MIF	0.429612
AIF	0.457831
COF	0.076087
POF	0.216814

The statistical data for this version of Routing Software is provided in Table 3 and Table 4. The total number of classes in this phase is 18.

Table 4 Class Level Metric Measurements for Phase 0

METRIC	Min. Value	Max. Value	Avg. Value
WMC	2	44	17.28
DIT	0	3	1.06
NOC	0	4	0.72
CBO	0	29	10.56
RFC	2	59	22.22
LCOM	0.36	1	0.70

3.4.2. PHASE 1

In this phase two design patterns that go together are applied to the Routing Software. These patterns are Reactor and Acceptor Connector design patterns.

In Routing Software 0, number of threads increases with the number of sockets needed by the software. Increase in the number of threads introduces the overhead of context switching. The Reactor Design Pattern overcomes this problem by using select facility of the operating system. By this way, all socket handlers in the system are controlled from one place, i.e. in one thread, if there exists an event like arrival of data or connection request; event is distributed to the responsible objects.

On the other hand, in Routing Software 0, there are two kinds of socket classes; one for client connections and the other for server connections, these two classes both has the ability of receiving and transmitting data. Whenever their job of establishing connection is finished, they start to check on the socket if there is an event of data arrival. So these classes have two separate concepts establishing connection and data exchange. Acceptor Connector Design Pattern suggests to separate the data exchange responsibility to another class so as to increase cohesion.

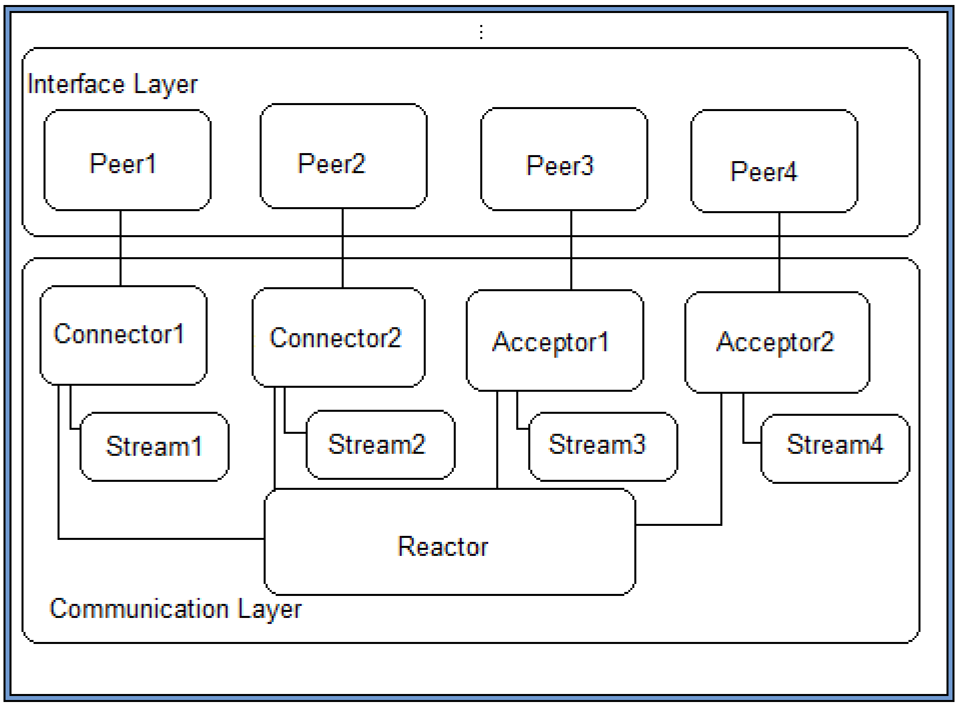

Figure 11 Interface and Communication Layer in Routing Software 1

These two patterns change the structure of Communication Layer which can be seen in Figure 11. The statistical data for this version of Routing Software is provided in Table 5 and Table 6. Results are compared with the previous phase results and the difference between these values given. Total number of classes in this phase is increased to 25.

Table 5 Application Level Metric Measurements for Phase 1

METRICS	Phase 1	Phase 0	Change
MHF	0.165976	0.109898	0.05607
AHF	0.909452	0.961353	-0.05190
MIF	0.339416	0.429612	-0.09020
AIF	0.300000	0.457831	-0.15783
COF	0.0412913	0.076087	-0.03480
POF	0.191304	0.216814	-0.02551

Table 6 Class Level Metric Measurements for Phase 1

METRICS	Phase 1			Phase 0	Change In Avg.
	Min.	**Max.**	**Avg.**	**Avg.**	
WMC	0	81	18.28	17.28	1
DIT	0	3	0.84	1.06	-0.22
NOC	0	4	0.64	0.72	-0.08
CBO	0	32	10.16	10.56	-0.4
RFC	2	62	20.96	22.22	-1.26
LCOM	0.36	1	0.68	0.70	-0.02

Results for DIT, CBO, RFC, MHF and COF are in favor of maintainability. However, WMC and POF showed contradicting behavior. Reactor class in

Reactor Design Pattern has a complex structure, since it checks for all the event handlers in the system periodically and distributes event to the related classes. Even though, the highest WMC belongs to this class and it increases the mean WMC, this class is still cohesive and compact in terms of the job it performs. If the Reactor was excluded, WMC would significantly decrease. Note that DIT and NOC values decrease in this phase. This means smaller inheritance trees are formed, because of this use of polymorphic methods decrease. That is why the POF is decreased.

Since metrics signal in favor of maintainability mostly, it can be concluded that Reactor Design Pattern leads more maintainable software.

3.4.3. PHASE 2

In this phase Smart Pointer Pattern is applied to Routing Software 1.

This pattern is a light pattern which consists of three template classes, to hold any kind of object pointers. It is easy to add this pattern to software; however, using this pattern needs attention. Whenever a pointer to an object is used it should be given to a Smart Pointer Class and all the pointers to that object should be made smart in order to avoid erroneous cases.

In Routing Software 1, data objects created in Interface Layer are passed to Application layer as pointers. These pointers are made Smart in this phase. During preceding phases new pointers of object are created as Smart pointers.

Table 7 Application Level Metric Measurements for Phase 2

METRICS	Phase 2	Phase 1	Change
MHF	0.187094	0.165976	0.02112
AHF	0.913782	0.909452	0.00433
MIF	0.322357	0.339416	-0.01706
AIF	0.292035	0.3	-0.00797
COF	0.0320513	0.0412913	-0.00924
POF	0.191304	0.191304	0.00000

Table 8 Class Level Metric Measurements for Phase 2

METRICS	Phase 2			Phase 1	Change
	Min.	Max.	Avg.	Avg.	In Avg.
WMC	0	81	16.32	18.28	-1.96
DIT	0	3	0.79	0.84	-0.05
NOC	0	4	0.57	0.64	-0.07
CBO	0	32	8.07	10.16	-2.09
RFC	2	62	19.75	20.96	-1.21
LCOM	0.36	1	0.72	0.68	0.04

The statistical data for this version of Routing Software is provided in Table 7 and Table 8. Number of classes in this phase has increased to 28 since this pattern contains three additional classes.

These classes are small and introduce low complexity, inheritance and coupling to the system. Thus, WMC, DIT, CBO, RFC, MHF and COF metric values are all in favor of maintainability. Only POF metric remained the same. It can be concluded that Smart Pointer Design Pattern provides a more maintainable application. And this improvement is better than the one The Reactor and the Acceptor Connector Design Pattern provided to the system.

3.4.4. PHASE 3

In this phase Forwarder Receiver Design Pattern is applied to Routing Software 2.

Different peers can support different message protocols. This variety should be handled in Peer classes in Routing Software 2. So if something about the message protocol of a peer changes, like the place of check sum bytes or value of header bytes etc. the peer class has to be changed. If the knowledge of message protocol details is separated from the peer class, peer will not be influenced from such changes. And if new classes specific for this job is added, then the resulting system will be more cohesive and open to changes. Then change in message protocol can be handled even in run time.

Forwarder Receiver Design Pattern suggests solution to this kind of problems. It separates communication details from peer classes. So peer classes deal with only the content of the messages not how it will be sent or received.

In Figure 12, it can be seen how Forwarder Receiver Design Pattern is applied to Routing Software 2. This pattern changes the structure of Interface Layer.

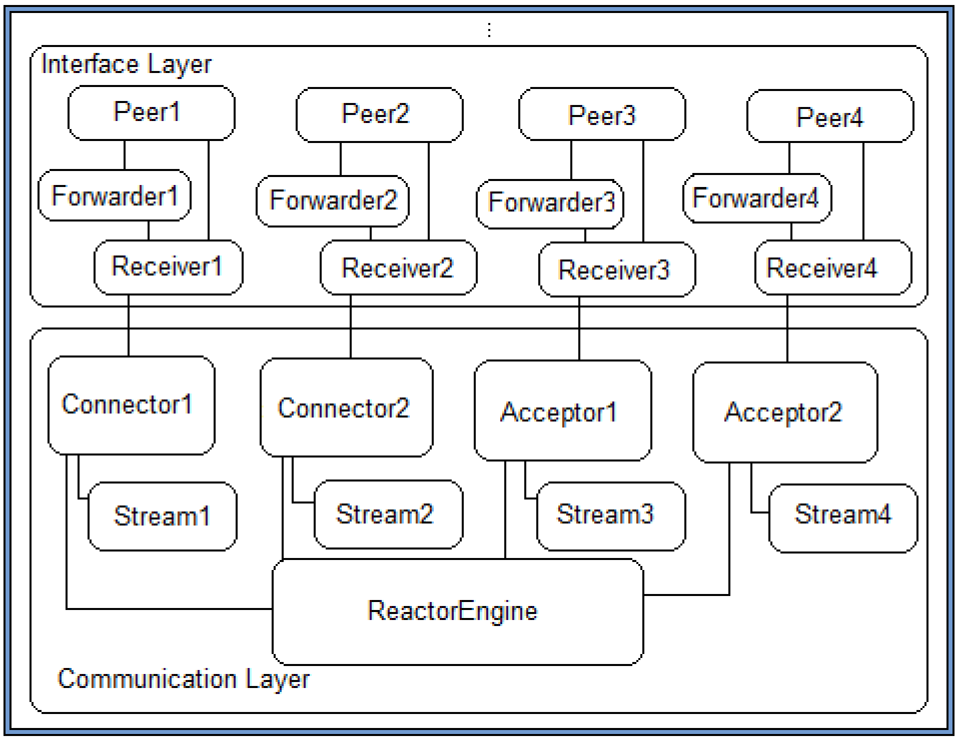

Figure 12 Interface and Communication Layer in Routing Software 3

The statistical data for this version of Routing Software is provided in Table 9 and Table 10. Number of classes in this phase has increased to 33.

In this phase DIT, RFC, MHF and COF showed an improvement in favor of maintainability. However, CBO and WMC increased. Here, the newly added classes to the code are coupled with each other and with the peer classes. This is the reason of the increase in CBO. It is worth to mention that Forwarder Receiver Classes has a complex structure which yields increase in the WMC.

Table 9 Application Level Metric Measurements for Phase 3

METRICS	Phase 3	Phase 2	Change
MHF	0.120718	0.187094	-0.06638
AHF	0.938312	0.913782	0.02453
MIF	0.271513	0.322357	-0.05084
AIF	0.253333	0.292035	-0.03870
COF	0.0318182	0.0320513	-0.00023
POF	0.211207	0.191304	0.01990

Table 10 Class Level Metric Measurements for Phase 3

METRICS	Phase 3			Phase 2	Change
	Min.	**Max.**	**Avg.**	**Avg.**	**In Avg.**
WMC	0	81	17.30	16.32	0.98
DIT	0	3	0.70	0.79	-0.09
NOC	0	4	0.52	0.57	-0.05
CBO	0	35	8.67	8.07	0.6
RFC	2	62	19.70	19.75	-0.05
LCOM	0.29	1	0.71	0.72	-0.01

It can be concluded that the Forwarder Receiver Design Pattern yields a more maintainable code. However, the improvement in this phase is less in comparison to the phase 2.

3.4.5. PHASE 4

In this phase Command Processor Design Pattern is applied to Routing Software 3.

This pattern is used in event driven applications. In Routing Software incoming messages are events that stimulate an action. The action that will be performed is decided by the SystemController in Application Layer. Command Processor Design Pattern suggests encapsulating the action into command objects and leaving the management of these objects to a separate class, i.e. CommandProcessor Class.

Applying the pattern, whenever a peer class informs the SystemController, it creates the associated command object and passes it to Command Processor Class. It executes command and stores it if an undo mechanism is needed.

The statistical data for this version of Routing Software is provided in Table 11 and Table 12. Number of classes in this phase has increased to 39.

In this phase all the maintainability predictor metrics showed an improvement except POF metric. Indeed, in this design pattern polymorphism is an important concept since all the command classes inherit from a common command class and override their specific methods so that Command Processor know each command from the interface of common command class but when it executes a command they all perform different tasks. However, the number of overriding methods is less compared to the inherited methods.

Table 11 Application Level Metric Measurements for Phase 4

METRICS	Phase 4	Phase 3	Change
MHF	0.12117	0.120718	0.00045
AHF	0.944762	0.938312	0.00645
MIF	0.28961	0.271513	0.01810
AIF	0.267442	0.253333	0.01411
COF	0.0286275	0.0318182	-0.00319
POF	0.192029	0.211207	-0.01918

Table 12 Class Level Metric Measurements for Phase 4

METRICS	Phase 4			Phase 3	Change
	Min.	**Max.**	**Avg.**	**Avg.**	**In Avg.**
WMC	0	81	16.33	17.30	-0.97
DIT	0	3	0.69	0.70	-0.01
NOC	0	4	0.54	0.52	0.02
CBO	0	35	8.15	8.67	-0.52
RFC	2	62	19.13	19.70	-0.57
LCOM	0.29	1	0.69	0.71	-0.02

Command Processor Design Pattern reserves a class for each task carried out as a result of a request. This increases the number of simpler and cohesive classes, improving mean values for the class metrics. In case of Routing Software it decreases the complexity of System Controller Class. This is the reason for the overall improvement in the code.

As a result it can be concluded that the Command Processor Design Pattern increases the maintainability of the Software.

3.4.6. DISCUSSION

In Figures 13 through 19, changes in metric values throughout the phases are shown graphically.

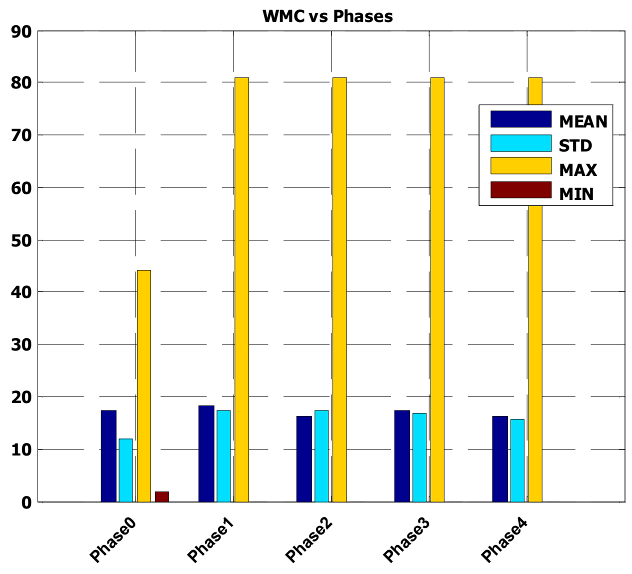

Figure 13 Change in WMC Metric throughout the Phases

In Figure 13, the mean values for WMC does not have monotone behavior. Two phases improved this metric, i.e. second and fourth phases.

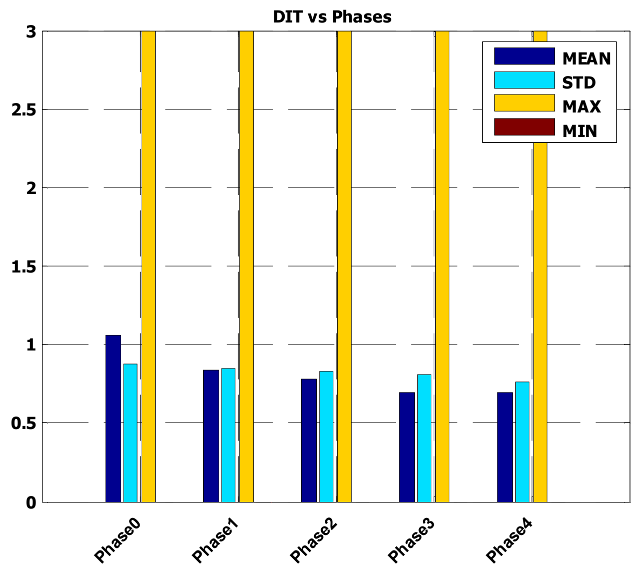

Figure 14 Change in DIT Metric throughout the Phases

In Figure 14, DIT mean values decreases monotonically. This decrease in metric is good sign of maintainability.

In Figure 15, NOC value decreases monotonically except the last phase. In Figure 16, CBO decreases throughout the phases except the third phase. Decrease in CBO is in favor of maintainability.

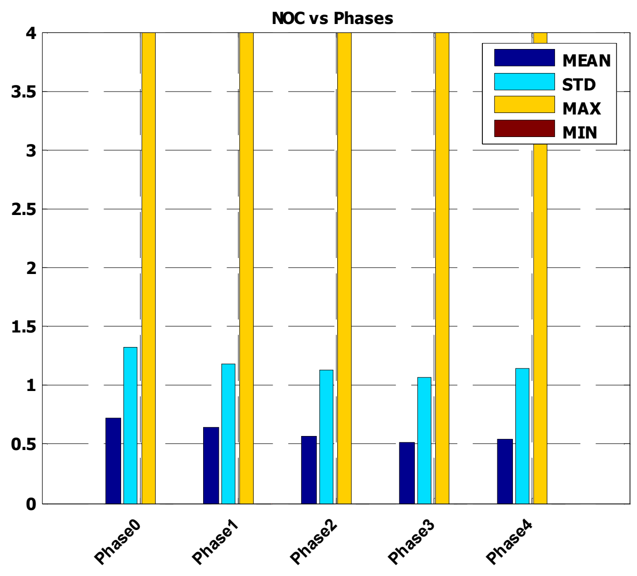

Figure 15 Change in NOC Metric throughout the Phases

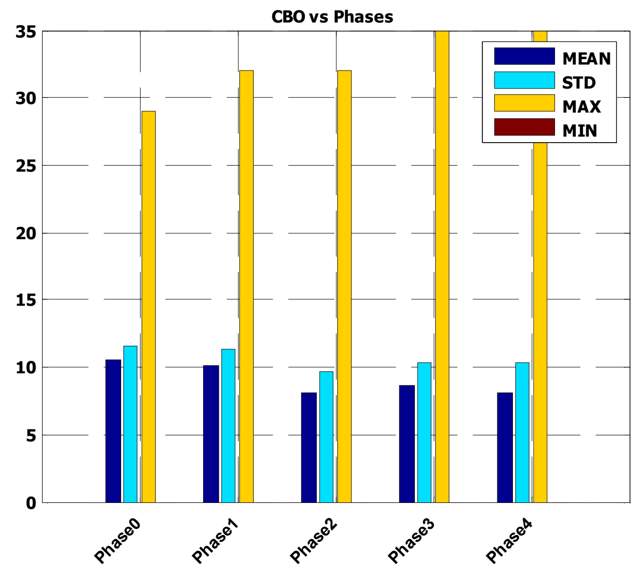

Figure 16 Change in CBO Metric throughout the Phases

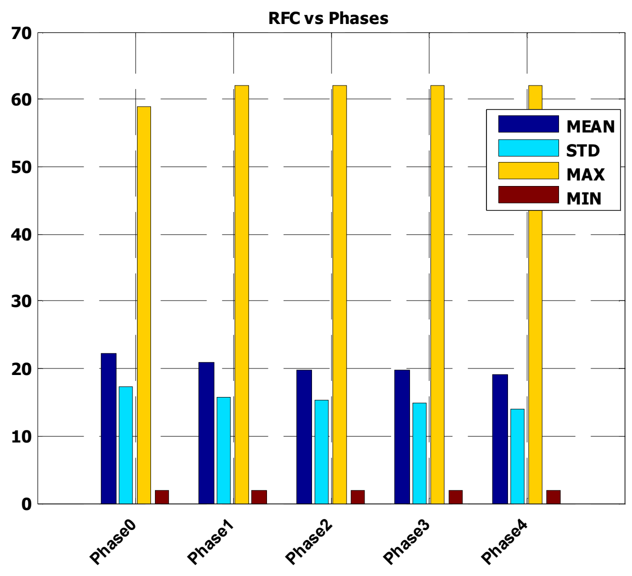

Figure 17 Change in RFC Metric throughout the Phases

In Figure 17, the monotonic decrease in RFC mean values can be seen. This is a sign of increase in maintainability throughout the phases.

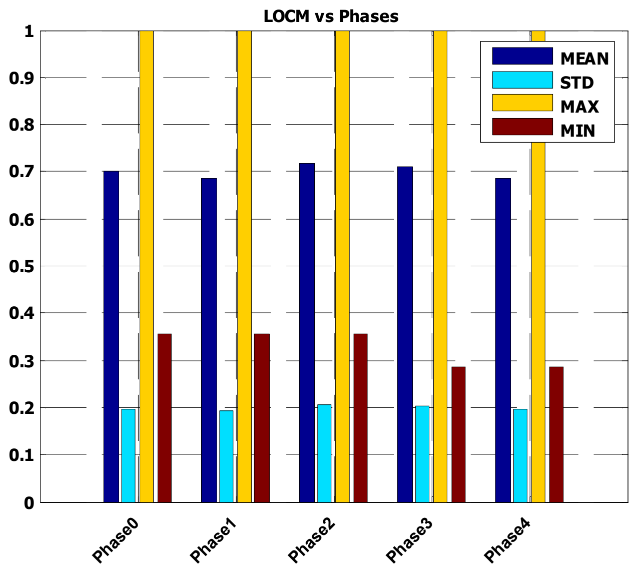

Figure 18 Change in LCOM Metric throughout the Phases

In Figure 18, LCOM mean values fluctuate. Since LCOM is not in the maintainability predicting metric set, this behavior does not give a sign about maintainability.

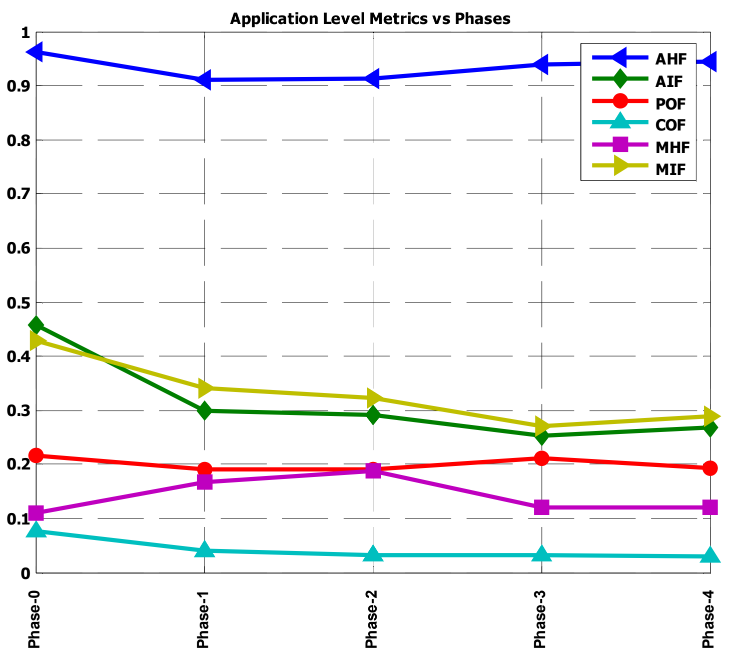

Figure 19 Application Level Metrics for All Phases

In Figure 19, AIF, MIF and COF values are monotonically increases throughout the phases while the other application level metrics fluctuates. Behavior of COF and MIF(since it has low values) is a good sign of maintainability increase. The other application level metrics are not in the set of maintainability indicator metrics except the POF. POF remains almost constant in low values.

According to the data obtained in all phases, correlation between these metrics can be observed. Since there are just five phases it would be misleading to attempt to derive a conclusion about the correlation of the metrics in general; however, some clues can be obtained about it with a discussion of the meaning of the metrics.

In Table 13, correlation matrix for application level metrics is shown. According to this table, MHF and AHF seem negatively strongly correlated. One reason for this result is the addition of public access methods to the code for each invisible attribute. So as the AHF increases by the addition of non-public attributes, MHF will decrease by the addition of public methods to reach them.

Also, MIF and AIF are strongly correlated metrics, this is due to the fact that when inheritance relation is established between classes usually both methods and attributes are used by the ancestor classes.

Table 13 Correlation Matrix for Application Level Metrics

	AHF	**AIF**	**POF**	**COF**	**MHF**	**MIF**
AHF	1	0.55	0.75	0.56	**-0.92**	0.37
AIF	0.55	1	0.59	**0.98**	-0.33	**0.98**
POF	0.75	0.59	1	0.67	-0.72	0.43
COF	0.56	**0.98**	0.67	1	-0.4	**0.95**
MHF	**-0.92**	-0.33	-0.72	-0.4	1	-0.15
MIF	0.37	**0.98**	0.43	**0.95**	-0.15	1

bold	Absolute value above 0.9

The correlation between MIF and COF and between COF and AIF is hard to explain. This could be specific to this case and it would not be appropriate to generalize this behavior.

Similar correlation analysis can be made for application level metrics. According to the class level metrics' mean values correlation matrix, shown in Table 14, DIT and NOC metrics are strongly correlated. Partial reason for this is due to the fact that increase in inheritance tree height also increases the number of inherited classes. For a parent class in this tree, the NOC value will increase by one increasing the mean value of NOC. However, increase in the width of the inheritance tree will increase the NOC value but will not affect the DIT value.

Table 14 Correlation Matrix for Class Level Metrics' Mean Values

MEAN	**WMC**	**DIT**	**NOC**	**CBO**	**RFC**	**LOCM**
WMC	1	0.34	0.5	0.81	0.68	-0.17
DIT	0.34	1	**0.93**	0.82	0.79	0.62
NOC	0.5	**0.93**	1	0.88	**0.92**	0.36
CBO	0.81	0.82	0.88	1	0.86	0.22
RFC	0.68	0.79	**0.92**	0.86	1	0.29
LOCM	-0.17	0.62	0.36	0.22	0.29	1

bold	Absolute value above 0.9

Table 15 Correlation Matrix for Class Level Metrics' STD Values

STD	**WMC**	**DIT**	**NOC**	**CBO**	**RFC**	**LOCM**
WMC	1	-0.83	**-0.94**	-0.7	-0.73	**0.98**
DIT	-0.83	1	0.78	0.69	**0.95**	-0.76
NOC	**-0.94**	0.78	1	0.78	0.79	**-0.91**
CBO	-0.7	0.69	0.78	1	0.75	-0.76
RFC	-0.73	**0.95**	0.79	0.75	1	-0.65
LOCM	**0.98**	-0.76	**-0.91**	-0.76	-0.65	1

bold	Absolute value above 0.9

Another strongly correlated metric pair is RFC and NOC. Their standard deviation seems also correlated in Table 15. It is hard to explain this situation with the nature of these two metrics, thus this result cannot be generalized.

Lastly, the correlation matrix for class level and application level metrics can be seen in Table 16.

Table 16 Correlation Matrix for Class and Application Level Metrics

	Mean Values					
	WMC	**DIT**	**NOC**	**CBO**	**RFC**	**LCOM**
MHF	-0.01	-0.17	-0.1	-0.25	-0.2	0.2
AHF	-0.22	0.37	0.26	0.22	0.3	0.01
MIF	0.27	**0.99**	**0.98**	0.79	**0.93**	-0.02
AIF	0.18	**0.97**	**0.92**	0.75	**0.9**	0.05
COF	0.33	**0.97**	**0.92**	0.84	**0.95**	0
POF	0.19	0.5	0.36	0.46	0.55	0.34

bold	Absolute value above 0.9

From the previous results MIF - AIF and DIT - NOC metric pairs' correlation is discussed. These four metrics are closely related with the inheritance concept. DIT and NOC measures the inheritance relation, MIF and AIF measure the amount of methods and attributes inherited through this relation. So it makes sense to conclude that these metrics are correlated.

It is hard to see the reason of high values of COF and DIT, NOC correlation since COF is related with the non-inheritance coupling. And the result should not be generalized. Similar discussion is valid for the RFC and MIF, AIF correlation results.

Both RFC and COF are coupling related metrics; RFC measures internal coupling where COF measures all kinds of coupling. This is the reason of these metrics correlation result. Another coupling metric is CBO and it is moderately correlated with RFC and COF metric.

In Figure 20, to summarize the foregoing discussion, the correlation among all considered metrics is presented visually.

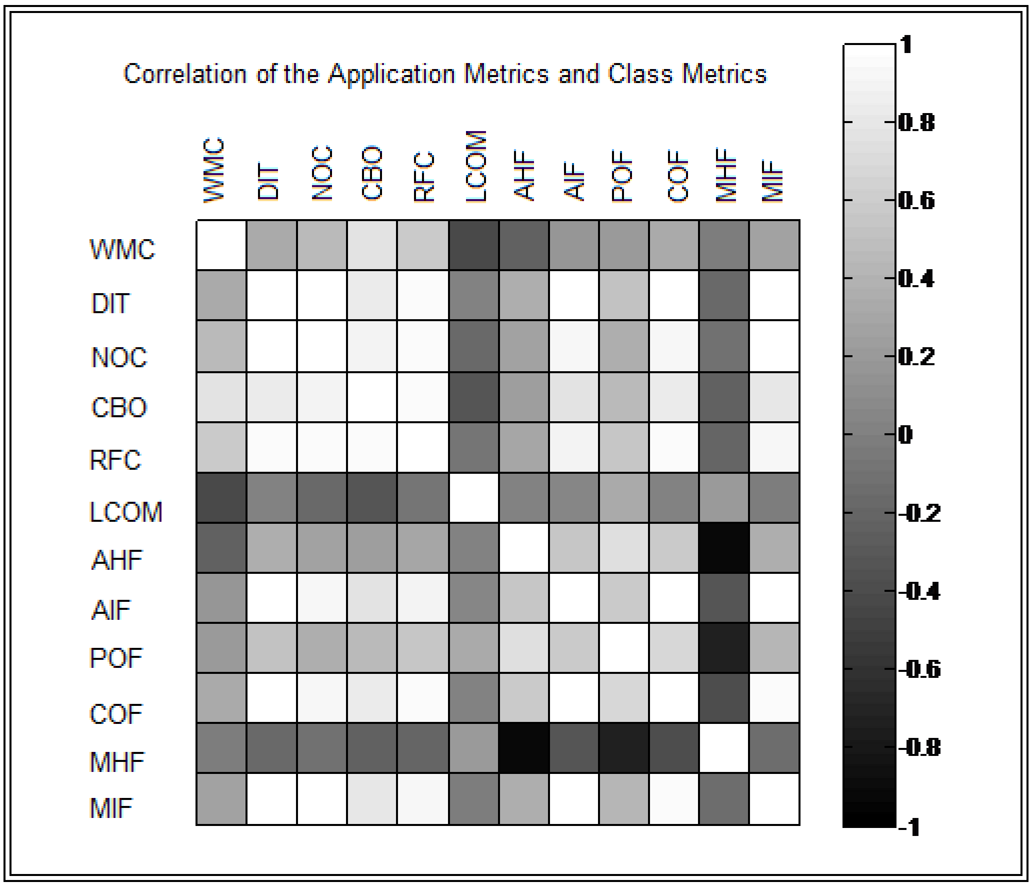

Figure 20 Visualization of Correlation of All Metrics

CHAPTER 4

DISCUSSION AND CONCLUSIONS

In this thesis, the relationship between the use of software design patterns and software maintainability, as measured by MOOD [9] and Chidamber and Kemerer OO metrics [10] has been investigated. Previous studies showing the effect of design patterns on maintainability have been reviewed and the metrics that can be used as maintainability predictors have been determined. Then an experimental case study has been constructed to apply software design patterns. For each applied pattern, OO metric values were calculated. The effect of the applied design patterns to maintainability has been discussed by investigating the numerical results of the metrics. Also the correlation between these metrics has been observed.

In the case study, Routing Software is constructed with the help of software basis of ASELSAN Inc. Five software design patterns were applied to the this software. Application of Reactor, Acceptor Connector, Forwarder Receiver Design Patterns improved most of the maintainability predicting metrics while Smart Pointer, Command Processor design patterns improved almost all the maintainability predicting metrics.

Improving maintainability of the code is very important in software quality assurance, so is its measurement. However, finding direct indicators for maintainability of the software is very difficult. For this purpose the maintainability history data of a lot of software samples should be examined. By maintainability history data; effort needed to test, effort needed to change/add functionality of/to the sample software, number of bugs found before release, failure density, time needed for a foreign developer to adopt the software can be considered. With these data, code metrics obtained from the sample software can be examined and correlation between these can be found. Then maintainability can be expressed more accurately by means of software metrics. There are studies like this in the literature (e.g. [20], [9] and [26]). In this thesis these kinds of studies have formed the basis of maintainability and metrics relationship. As a future work such a maintainability predicting suite can be constructed with a more comprehensive set of maintainability history data over a large variety of software samples and with a large set of software metrics.

This thesis has some similarities with the previous work ([21]) realized by B. Aydınöz. He observed the relation between error-proneness and OO metrics, and then he applied GOF design patterns to different software samples and examined the change in these metrics so concluded about the effect of these design patterns to software error-proneness. The present study differs from [21], in the use of a larger set of metrics, different design patterns and the different subject of prediction, i.e. maintainability. Aydınöz' research about the relation of metrics and error-proneness is referenced in this work, too, since error-proneness is related to stability of the software and stability is a factor on maintainability.

As a future work, this study can be improved with investigations of other quality characteristics like efficiency and reliability. Also the metric set can be increased with the inclusion of metrics that are not necessarily OO, like Halstead [32], McCabe [24] and size metrics.

Another aspect of software that may be affected by the use of design patterns is performance, especially important in the context of real-time embedded systems. Hence, investigation of the effects of design patterns on real-time performance is definitely another subject that deserves in-depth study.

REFERENCES

[1] International Standard ISO/IEC 9126-1:2001, “Software Engineering - Product Quality - Part 1: Quality Model”.

[2] McCall, J., Richards, P., Walters, G., “Factors in Software Quality, Volume I”, NTIS Springfield, 1977.

[3] Tian, Y., Chen, C., Zhang, C., “AODE for Source Code Metrics for Improved Software Maintainability”, Fourth International Conference on Semantics, Knowledge and Grid, 2008.

[4] Elish, M. O., Elish, K. O., “Application of TreeNet in Predicting Object-Oriented Software Maintainability: A Comparative Study”, European Conference on Software Maintenance and Reengineering, 2009.

[5] Zhou, Y., Leung, H., “Predicting Object-Oriented Software Maintainability Using Multivariate Adaptive Regression Splines”, Journal of Systems and Software, 2007.

[6] Henry, S., Humphrey, M., Lewis, J., “Evaluation of the Maintainability of Object-Oriented Software”, IEEE Region 10 Conference on Computer and Communication Systems, 1990.

[7] Unhelkar, B., “Practical Object Oriented Design”, Cengage Learning Australia, 2005.

[8] Shalloway, A., Trott, J., “Design Patterns Explained: A New Perspective on Object-Oriented Design”, Addison-Wesley, 2002.

[9] Abreu, F., Melo, W., “Evaluating the Impact of Object-Oriented Design on Software Quality”, Proceedings of the 3rd International Software Metrics Symposium, 1996.

[10] Chidamber, S., Kemerer, C., “A Metrics Suite for Object-Oriented Design”, IEEE Transactions on Software Engineering, Volume 20, No 6, June 1994.

[11] “Patterns and Software: Essential Concepts and Terminology”, http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html, Access date: 05/08/2009.

[12] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., “Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects”, Wiley, 2000.

[13] Douglass, B. P., “Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems”, Addison-Wesley, 2002.

[14] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., “Pattern-Oriented Software Architecture Volume 1: A System of Patterns”, Wiley, 1996.

[15] Gamma, E., Helm, R., Johnson, R., Vlissides, J., “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1998.

[16] Mitchell, A., Power, J. F., “Toward a Definition of Run-Time Object-Oriented Metrics”, 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, 2003.

[17] “ISO 9126 Software Quality Characteristics, An Overview of the ISO 9126–1 Software Quality Model Definition“, www.sqa.net/iso9126.html, Access date: 05/08/2009.

[18]ANSI/IEEE, “IEEE standard glossary of software engineering terminology”, IEEE Std. 729 - 1983, 1983.

[19] International Standard ISO/IEC 9126-3:2003, “Software engineering — Product quality — Part 3: Internal Metrics”.

[20] Dagpinar, M., Jahnke, J. H., “Predicting Maintainability with Object-Oriented Metrics”, 10th Working Conference on Reverse Engineering, 2003.

[21] Aydınöz, B., “The Effect of Design Patterns On Object-Oriented Metrics and Software Error-Proneness”, MS Thesis, EEE Department, METU, 2006.

[22] Harrison, R., Counsell, S. J., Nithi, R.V., “An Evaluation of the MOOD Set of Object-Oriented Software Metrics”, IEEE Transactions on Software Engineering, Volume 24, No 6, June 1998.

[23] Mayer, T., Hall, T., “Measuring OO Systems: A Critical Analysis of the MOOD Metrics”, Technology of Object-Oriented Languages and Systems, 1999.

[24] McCabe, T. J., “Complexity Measure”, IEEE Transactions on Software Engineering, Volume 2, No 4, December 1976.

[25] Weyuker, E. J., “Evaluating Software Complexity Measures”, IEEE Transactions on Software Engineering, Volume 14, No 9, September 1988.

[26] Bruntink, M., Deursen, A., “Predicting Class Testability Using Object-Oriented Metrics”, Source Code Analysis and Manipulation, Fourth IEEE International Workshop, 2004.

[27] Briand, L., Wuest, J., Daly, J., Porter, V., "Exploring the Relationships Between Design Measures and Software Quality in Object-Oriented Systems", J. Systems and Software, Vol. 51, 2000.

[28] "IBM Software - Rational Rhapsody", http://www-01.ibm.com/software/awdtools/rhapsody/, Access date: 05/08/2009.

[29] "Telelogic is now IBM - IBM Rational Logiscope: Software Quality Assurance, Bug Tracking", http://www.telelogic.com/products/logiscope/index.cfm, Access date: 05/08/2009.

[30] "MATLAB - The Language of Technical Computing", http://www.mathworks.com/products/matlab/index.html?ref=pfo, Access date: 05/08/2009.

[31] Gören, H. Ö., Gürler, E., "Elektronik Harp Sistemleri Gömülü Kontrol Yazilimi Mimarisi", Ulusal Yazılım Mimarisi Konferansı, P. 20-25, 2006.

[32] Halstead, M. H., "Elements of Software Science", Elsevier, 1977.