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

BY

BARIŞ AYDINÖZ

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

SEPTEMBER 2006

Approval of the Graduate School of Natural and Applied Sciences

_____

Prof. Dr. Canan Özgen
Director

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

_____

Prof. Dr. İsmet Erkmen
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

Prof. Dr. Hasan C. GURAN (METU, EE) _____

Prof. Dr. Semih BİLGEN (METU, EE) _____

Dr. Altan KOÇYİĞİT (METU, EE) _____

Dr. Şenan Ece (GÜRAN) SCHMIDT (METU, EE) _____

Asst. Prof. Dr. Cüneyt F. BAZLAMAÇCI (METU, EE) _____

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: Barış AYDINÖZ

Signature :

ABSTRACT

THE EFFECT OF DESIGN PATTERNS ON OBJECT-ORIENTED METRICS AND SOFTWARE ERROR-PRONENESS

AYDINÖZ, Barış

M.S., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Semih BİLGEN

September 2006, 134 pages

This thesis study investigates the connection between design patterns, OO metrics and software error-proneness. The literature on OO metrics, design patterns and software error-proneness is reviewed. Different software projects and synthetic source codes have been analyzed to verify this connection.

Keywords: Design Patterns, OO Metrics, Software Error-Proneness

ÖZ

TASARIM ÖRÜNTÜLERİNİN NESNE TABANLI METRİKLER VE YAZILIM HATA EĞİLİMİ ÜZERİNDEKİ ETKİLERİ

AYDINÖZ, Barış

Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Semih BİLGEN

Eylül 2006, 134 pages

Bu tez çalışması, tasarım örüntüleri, nesne tabanlı metrikler ve yazılım hata eğilimi arasındaki ilişkileri incelemektedir. Tasarım örüntüleri, nesne tabanlı metrikler ve yazılım hata eğilimi ile ilgili literatür incelenmiştir. Farklı yazılım projeleri ve yapay kaynak kodlar ilgili bağlantının doğrulanması için analiz edilmiştir.

Anahtar Kelimeler: Tasarım Örüntüleri, Nesne Tabanlı Metrikler, Yazılım Hata Eğilimi

ACKNOWLEDGMENTS

I would like to express my gratitude to following people:

- Prof. Dr. Semih Bilgen, for his guidance, criticism, and advices during the development of this thesis.
- My father, Fevzi Aydınöz; my mother, Hikmet Öğüt; my sister, Dr. Özgür Canbay; my brother-in-law, Dr. Alper Canbay, for their great motivations, encouragements and morale support.
- Executives of Software Engineering Department of MST Division in ASELSAN Inc., for their contributions on this study.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

LIST OF TABLES

TABLES

LIST OF CODE LISTINGS

LISTINGS

LIST OF ABBREVIATIONS AND ACRONYMS

CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
GOF	Gang of Four
GPL	General Public License
IEEE	Institute of Electrical and Electronics Engineering
ISO	International Organization for Standardization
LCOM	Lack of Cohesion in Methods
MFC	Microsoft Foundation Classes
MTBF	Mean Time between Failures
NMA	Number of Methods Added
NMO	Number of Methods overridden
NOC	Number of Children
OO	Object-Oriented
OOAD	Object-Oriented Analysis and Design
RFC	Response for a Class
SIX	Specialization Index
STL	Standard Template Library
UML	Unified Modeling Language
WMC	Weighted Methods per Class

CHAPTER 1

INTRODUCTION

1.1. Software Error-Proneness and Design Patterns

Computer systems sometimes crash and fail to perform requested actions or operations. There are many reasons for a failure to happen, misunderstood requirements, architectural or detailed design faults, user or operator mistakes, environmental factors and much more. The immediate effects of software faults can range from minor inconveniences (e.g., having to restart a hung personal computer) to catastrophic events (e.g., software in an aircraft that prevents the pilot from recovering from an input error).

Moreover, there is no definite solution to remove design faults. Since there are inherent properties of modern software systems: complexity, conformity, changeability, and invisibility, these properties are hard to remove and also as stated before there is no formal solution for these inherent and irreducible properties [FPB87]. This is the point where software error-proneness arises. This term is related to the probability of failure for a given class or module in software.

Software metrics have many application areas such as **quality management, product size, and maintenance** in the software development life-cycle. Moreover, in architectural and detailed phases of software development cycle, there are some object oriented (OO) metrics, which have been proposed to measure software error-proneness in the literature, can help engineers to determine and eliminate majority of the design errors. In this thesis, metrics that are related with software error-proneness in the context of OO design have been researched in detail in Chapter 2.

Design patterns provide reusable solutions for common OO design problems. Moreover, they identify participating class associations, abstractions, instances, class roles and key aspects for design structure that simply defines the solution context in OO designs. Also, design patterns are the elegant solutions of design problems and they have evolved iteratively. Moreover, [CA77] states, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

1.2. The Purpose and Scope of the Study

The objective of the study is to investigate the effect of design patterns on software error-proneness. There are many motivations for this objective. First of all, in the literature, design patterns are mainly known as reusable solutions for different problem contexts. But, application of design patterns has different effects on software. One of them is expected to be the effect on software error-proneness. This study mainly investigates this effect. Secondly, this study aims to show that the effects of design patterns on software are measurable and therefore interpretable. Thirdly, design pattern implementations and refactorings in software are highly recommended. Also, design patterns have positive effects on design, and increase reusability, analyzability, and so on. But, design patterns may introduce side effects like increase in error-proneness. Therefore, this study aims to show that design patterns have different effects on software, when researched on the basis of different approaches and technical views.

Within the scope of this study, different source codes of sample software projects and synthetic projects have been refactored with design patterns. Fluctuations on metrics have been evaluated mainly from the perspective of the software error-proneness. Also, in this thesis, [GOF98] patterns have been researched and the effects of these patterns on OO metrics have been measured in different cases with refactorings.

1.3. Outline

This thesis is organized as follows:

In chapter 2, in order to determine software error-proneness related metrics and to connect metrics and design patterns, **literature review** has been performed. Conceptually different design patterns and OO metrics have been researched, in order to investigate the relations between them and software error-proneness.

In chapter 3, **experimental work** that has been performed on different cases is explained. Samples for these cases consist of open source software projects and synthetic source codes. Also, each case or experiment has six steps in its implementation to empirically show the effect of design patterns on software error-proneness related OO metrics. Also, effects of design patterns on software error-proneness have been interpreted. Moreover, metric collection tool [TTL00], which is used to measure OO metrics in performed experiments, and phases of experimental work cases have been discussed and described.

In chapter 4, discussions and comparisons of the effects of different design patterns have been made, **conclusions** of the study have been presented, and some future work alternatives have been suggested.

CHAPTER 2

LITERATURE REVIEW

2.1. Introduction

In this chapter, the concept of software error-proneness, object-oriented metrics, design patterns, and possible effects of design patterns on OO metrics that are related with error and fault proneness are defined and researched.

Moreover, after determining error-proneness related metrics, the connections between these metrics and specific design patterns are established, this phase has crucial importance, since all the work performed in chapter 3 mainly uses these connections as a basis. In addition, OO metric and design pattern relation stimulates that refactorings using design patterns can decrease error-proneness in software.

To sum up, this chapter is the basis of the succeeding chapters. Conclusions of this chapter have helped us to understand the connections between error-proneness of software, OO metrics and design patterns.

2.2. Definitions

Institute of Electrical and Electronics Engineering (IEEE) standard glossary of software engineering terminology [IEEESG90] defines error, fault, and program failure as given below:

- **Error:** As an inappropriate action performed by a programmer or designer.
- **Fault:** The manifestation and the result of an error during the coding of a program.

- **Program failure:** Some incorrect program behavior caused by the existence of faults in the program

These definitions show us that failure or crash are, in fact, a chain reaction, propagate from executable code to the system boundaries. These definitions also demonstrate that some software faults are all design faults. Therefore, key point is that design of software carries great importance and fault tolerance techniques should be added to the system during requirements or architectural design phases of the used software process model. Moreover, there is no definite solution to remove design faults. Since, there are inherent properties of modern software systems: complexity, conformity, changeability, and invisibility. These properties are hard to remove and also as stated before there is no formal solution for these inherent and irreducible properties [FPB87].

In addition, from a system perspective, software fault tolerance can be discussed under the two topics of software engineering; first of these is reliability. Reliability, informally, is the probability, over a given period of time that the system will correctly deliver services as expected by the user [IS04]. The motivation for fault tolerance in the content of reliability is that under failure or crash, system or software should continue to perform and deliver outputs for requested operations. The second is availability, which is defined as the probability that a system, at a point of time, will be operational and able to deliver the requested services [IS04]. The main difference between availability and reliability is that reliability is defined for a period of time, but availability has importance for a point of time.

Consequently, software error-proneness has strong a composition relationship between availability and reliability. They can be used to measure fault tolerance of a system or software indirectly and from a higher perspective. For example, Mean Time between Failure (MTBF) is a reliability metric, which shows the time elapsed between two failures that cause system to fail.

Software fault tolerance refers to the use of techniques to increase the likelihood that the final design embodiment will produce correct and/or safe outputs. Moreover, main property for a fault tolerant system is that system can continue operation after some faults have occurred. Embedded fault tolerance mechanisms ensure that these system faults do not cause system failure. In practice, fault tolerance is applied in situations where system failure could cause a catastrophic accident or where a loss of system operation would cause large economic losses. For example, air traffic control system should continue operation continuously while planes in the air.

But, these metrics do not reflect the actual error-proneness of the software. That is, different environments will result in different reliability values [ARJ90]. Moreover, the key assumption that enables the design and reliability estimation of highly reliable hardware systems is that the components fail independently [BF91]. This does not apply to software, where the results of one component are directly or indirectly dependent on the results of other components, and thus errors in some modules can result in problems in other modules. When it comes to software reliability estimation, the only reasonable approach is to treat the software as a black box [PSK90].

To sum up, the aim of this introductory part about software faults is to emphasize the crucial importance of errors for software and to state that it is almost impossible to remove all the errors or misinterpretations in software. However, in the source code level, in architectural and detailed design phases of software development cycle, there are some OO metrics can help engineers to determine and eliminate majority of the design errors. In the succeeding part, these metrics are evaluated, and related papers reviewed from the view of software error-proneness.

2.3. Object Oriented Metrics

The focus on software process improvement has increased the demand for software measures, and consequently new metrics are developed. Software metrics have many

application areas such as **quality management, product size, and maintenance** in the software development life-cycles. But in this section, some OO metrics will be presented briefly and their relations with software error-proneness will be investigated.

Investigated metrics are all validated in the literature with the application of different validation methods, which are beyond the scope of this thesis. Therefore, these metrics are all commonly accepted in the community. Moreover, OO metric and fault-proneness association is confirmed by applying these validation methods [EBGR01].

OO metrics could be good indictors for determining fault or error prone modules and eliminate them, consequently help to increase software fault tolerance. [EBGR01] explains that some software metrics are important for software fault tolerance, also these metrics show error-prone modules, and consequently help designer to identify weak chains in the detailed design.

Chidamber and Kemerer, in their paper [CK94], define six new metrics for OO software, these metrics are:

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

Main purpose of [CK94] is to devise a solution for overcoming ineffective non-OO metrics. These traditional software complexity metrics do not have appropriate mathematical properties, and they fail to display predictable behavior of software. In addition, [CK94] strongly emphasizes that software metrics should depend on stronger

degree of theoretical and mathematical precision. Another key point for them is that OO software design is much more different than traditional function based software design, so as obvious, OO design approach needs more effective and realistic metrics.

[CK94] and [LK94] metrics are important for software fault tolerance, because these metrics show complexities for software modules, namely classes, and complexity is a good indicator for error-prone modules. Modules with high complexities are prone to error and hard to maintain.

Also, class design is the highest priority in the Object-Oriented Analysis and Design (OOAD), and since it deals functional requirements of the system, it must occur before system design and program design. Consequently, if these metrics evaluated correctly, completely, and also if complexities are removed by using these measures with application of design patterns and/or some intellectual designer approaches, fault tolerance is maximized, and number of error-prone modules is minimized. As a result, length of maintenance, and test phases, also labor effort will be limited to a lower degree.

In [TKC99], authors validated [CK94] metrics using software faults. [TKC99] states that traditional testing techniques are not viable for detecting OO faults. To overcome this problem they propose OO metrics to determine viable testing technique. That is to say identifying correct metrics can help designer to dig out fault prone parts, which would be very hard to find if traditional techniques were performed.

The results of [TKC99] suggest that Weighted Methods per Class (WMC) can be a good indicator for faulty classes and Response for a Class (RFC) is a good indicator for OO faults. Moreover, in [BBM96], authors conducted an empirical experiment based on eight medium sized school projects. Their results suggest that five of the six [CK94] metrics are useful quality indicators for predicting fault and error prone classes.

Another significant relationship between error-proneness and software metrics has been experimentally shown by [BDPW98]. Again in [BDPW98], authors analyze many OO metric relations with error-proneness, but in this thesis, main concern is the metrics of [CK94] and [LK94]. The results of [BDPW98] show that highly import coupled classes are more error prone. In other words, how much a class uses other classes has significant effect on the probability for the class to contain a fault. Moreover, [BPDW98] concludes that method invocation is the main coupling mechanism having impact on error-proneness.

Moreover, [BPDW98] shows that there is no evidence that classes, which have high export coupling values, are more likely to be error prone. In other words, how much a class is being used by other classes has little effect on the probability for the class to contain a fault. But, Coupling between Objects (CBO) [CK94] count both import and export coupling. Thus, from the view of fault proneness, when measuring CBO metric and detecting error-prone modules, weak influence of export coupling should be considered.

In [BPDW98], class cohesion metrics and fault-proneness analysis shows that there is a weak relation with cohesion and fault-proneness, which provides only a weak support for the hypothesis high cohesion of a class has a casual relationship to error-proneness. This result indicates that Lack of Cohesion in Methods (LCOM) [CK94] is irrelevant for determining error-prone modules in software; therefore this metric is excluded and not evaluated from the view of error-proneness.

In [BPDW98], result of association between inheritance metrics and error-proneness suggests that the deeper the class is located in the hierarchy, the more error-prone it is. Moreover, the more superclasses a class inherits from, and the more methods a class inherits, the more error-prone the class is, and classes with a high Number of Children (NOC) or subclasses are less error-prone. Perhaps, a greater attention (e.g., through inspections) is given to them as they are developed since many other classes depend on

them. As a result, fewer defects are found during testing. Also, the more use of method overriding is being made; related metrics are Number of Methods Overridden (NMO) and Specialization Index (SIX), the more fault-prone it will be. Lastly, the more methods are added to a class; related metric is Number of Methods Added (NMA), the more fault-prone it is.

In conclusion, research of [BDPW98] has shown that many coupling and inheritance metrics are strongly related probability of fault detection in a class. However, cohesion metrics does not seem to have strong impact on error-proneness. These metrics, mentioned in [CK94] and [LK94], are deeply researched and highly approved, also validated by software community, lots of papers that validate aforementioned metrics issued like [TKC99].

But, there are many of metrics in literature and some of them have many common attributes, which makes some metrics redundant [BPDW98]. But, validation, reexamination and determining the redundant metrics are out of scope in this thesis. Therefore, most commonly approved metrics will be examined from the view of software fault tolerance and error-proneness. This last statement implicitly emphasizes the assumption of perfectly validation of [CK94] and [LK94] for this thesis.

In the succeeding part error-proneness related metrics of [CK94] and [LK94] are explained and discussed. These metrics are:

- **Weighted Methods Per Class (WMC) [CK94]**
- **Depth of Inheritance Tree (DIT) [CK94]**
- **Number of Children (NOC) [CK94]**
- **Response For a Class (RFC) [CK94]**
- **Coupling Between Objects (CBO) [CK94]**
- **Number of Methods Added (NMA) [LK94]**

- **Number of Methods Overridden (NMO) [LK94]**
- **Specialization Index (SIX) [LK94]**

2.3.1. Weighted Methods per Class (WMC)

- **Definition of [CK94]:** Let C be a class, with methods M_1, M_2... M_k let c_1, c_2... c_k be the complexity of the methods. Chidamber and Kemerer do not define "Complexity" deliberately here in order to allow for the most general application of this metric. They advise that designer should or may determine the complexity of each method by using traditional approaches.

$$\mathbf{WMC = c_1 + c_2 + ... + c_k}$$

- **Applicability:**

1. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.

2. The larger the number of methods in an object, the greater the potential impact on the subclasses.

3. Objects with large number of methods are likely to be more application specific, limiting the possible reuse.

Applicability 2 and 3 are concerned with software design and reuse mainly. But, viewpoint 1 is important for software fault tolerance, as stated above, complexity shows error-prone modules. And, as WMC for a class rises, complexity of the related class increases, therefore it becomes hard to maintain and hard to repair also. Consequently, from the view of software fault tolerance, designer should keep this

metric as low as possible, and try to lessen WMC, by subclassing or separating the related class into two or more class.

2.3.2. Depth of Inheritance Tree (DIT)

- **Definition of [CK94]:** The depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritances, the DIT will be the maximum length from the node to the root of tree.
- **Applicability**:

1. Deeper trees constitute greater design complexity, since more methods and classes are involved.
2. The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

Again, for software error-proneness, complexity is an important indicator and should be limited; but inheritance is an inevitable fact of OOAD. Designers generally use inheritance and abstract classes in order to increase reuse and portability. But, trade-off between reusability and complexity should be well investigated. A deliberate strategy to reduce complexity for class inheritance trees to place as much functionality as close as possible to the root of the inheritance tree.

2.3.3. Number of Children (NOC)

- **Definition of [CK94]:** Metric simply defines the number of immediate subclasses subordinated to a class in the class hierarchy.
- **Applicability:**

1. Greater the number of subclasses, greater the reuse, since inheritance is a form of reuse.

2. Greater the number of subclasses, the greater the likelihood of improper abstraction of the superclass class. If a class has a large number of subclasses, it may be a case of misuse of subclassing.
3. The number of subclasses gives an idea of the potential influence a class on the design. If a class has large number of subclasses, it may require more testing of the methods in that class.

From the perspective of software error-proneness, [EBGR01], [BWIL98], and [BWDP00] prove that NOC is correlated with software fault tolerance and error-proneness. Moreover, in [BH01] author shows that application of Bridge pattern has a reducing effect on average NOC metric value.

2.3.4. Response for a Class (RFC)

- **Definition of [CK94]:** RFC = NLM + NRM

Where:

NLM = number of local methods in the class

NRM = number of remote methods called by methods in the class

A given remote method is counted only once

- **Applicability:**

1. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.
2. The larger the number of methods that can be invoked from a class, the greater the complexity of the class.

3. A worst case value for possible responses will assist in appropriate allocation of testing time.

[CK94] states that RFC metric is directly related with the complexity of the class. This viewpoint makes this metric valuable for fault-tolerance, and its association with fault tolerance was proved in the current literature [BWIL98], [BWDP00]. RFC carries great importance and should be limited in order to increase software fault tolerance, but, this metric should be well understood and should be limited in the early design phases of the software. In addition, this metric is related with count of associations, compositions and aggregations in a class, since in the methods of a class usually these relations are used and remote calls to these objects are performed. If designer can limit these relations, and remote method calls, he/she automatically achieve to decrease the value of RFC for a class.

2.3.5. Coupling between Objects (CBO)

- **Definition of [CK94]:** CBO for a class is a count of the number of other classes to which it is coupled.

- **Applicability** :

1. Excessive coupling between classes is detrimental to modular design and prevents reuse.
2. In order to improve modularity and promote encapsulation, inter-class coupling should be kept to a minimum.
3. A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be.

[CK94] does not state the CBO is related with complexity of a class, and with fault-proneness of a class. But, in [EBGR01], authors prove that CBO is strongly associated

with fault-proneness. This is consistent with previous literature that found CBO to be associated with fault-proneness [BWDP00], [BWIL98]. Also, CBO metric shows the fan-out of a class, and CBO values above custom/standard defined threshold in a class can be an indication of a rotten design. Developer should refactor the design and devise some solutions to limit the value of CBO metric.

2.3.6. Number of Methods Added (NMA)

- **Definition of [LK94]:** This is the normal expectation for a subclass, if it will further specialize the superclass' type of object.
- **Applicability** :

1. The number of new methods should usually decreases as you move down through the layers of the hierarchy.
2. Anomalies detected with this metric are usually detected by the number of methods overridden (NMO) metric.

As stated in [BPDW], more new methods added by subclass, the more fault-prone the class will be. [LK94] determined some threshold values for this metric, which are 20 for inheritance level 1, four for inheritance level six.

2.3.7. Number of Methods overridden (NMO)

- **Definition of [LK94]:** This metric defines the number of methods overridden by a subclass.
- **Applicability** :

1. A large number of overridden methods indicate a design problem.
2. Abstract classes and used framework classes should be excluded when counting this metric.

[BPDW98] states that the more use of method overriding is being made, the more fault-prone it will be.

2.3.8. Specialization Index (SIX)

- **Definition of [LK94]: SIX** = (NMO * class hierarchy inheritance level) / total number of methods
- **Applicability** :

1. Abstract classes and used framework classes should be excluded when counting this metric.
2. This metric helps designer to verify the location of a class in the hierarchy.

2.4. Crucial Effect of Determining Error Prone Modules on Software Life Cycle

In the OO metrics part, association between error-proneness and metrics was investigated. The main result of preceding part was inheritance and coupling metrics are strongly related with error-proneness, also help to indicate or detect error-prone classes.

In [BDPW98-2], authors experimentally show that when combined, a subset of the measures enables the construction of a very accurate model to predict in which classes most of the faults will lie in. Outcome of this paper is very important, for instance before test phase of software, predicting error-prone classes and devising solutions to lower their related metric scores may reduce testing significantly.

[BDPW98-2] classifies a class error-prone if its predicted probability after aforementioned analysis performed to contain a fault higher than a certain threshold. Interested readers can look at [BDPW98-2] for details.

The results of [BDPW98-2] also show that many coupling and inheritance metrics are strongly related to the probability of fault detection in a class.

To sum up, in [BDPW98-2] authors show that by using some of the coupling and inheritance measures, very accurate models can be derived to predict in which classes most of the faults actually lie.

2.5. Design Patterns

In this section, concept of a design pattern and most of the design patterns, which are deeply investigated in literature and related with the metrics that are also related with fault or error proneness, will be explained; also some interpretations about the applicability of these design patterns on software fault tolerance will be given.

Moreover, design patterns and software metrics serve for the common aims, which are promotion to adaptable, portable designs and reducing maintenance efforts. But, design patterns and metrics are completely different actually. For example, a metric high score for coupling between objects indicates detrimental design, low reusability, whereas design patterns also imply how to solve the context depended problem.

However, there is one consideration that designer or reader should not underestimate that is if established design patterns are accepted as building blocks for design, then their usage should be compatible with metric suite being applied [VP95]. Moreover, refactorings that have been applied to reform metric scores should not destroy pattern structures, and applied design patterns should not worsen metric values. In this case metric, the pattern, or both may prove to be suspect.

2.5.1. What is a design pattern?

Design pattern is a general solution to a problem in a context. Aim is to generalize the solution, in order to prevent invent the same solutions again and again.

According to [BPD02] there are different types of design patterns from the view of Design phase, which are:

- Architectural Design Patterns
- Mechanistic Design Patterns

In [BPD02], there are detailed descriptions of Architectural Design Patterns; these patterns are applied to software development process during architectural and detailed design phases.

The main difference between [GOF98] and [BPD02] is that [BPD02] explains design patterns in a more abstract way than [GOF98]; also [BPD02] broadens the scope of a design pattern from detailed view to architectural view. But, the common reality is that all the patterns provide a solution to a general problem. In other words, they provide solution reuse, not the details of the solution. Developers are responsible to apply the solution to the problem context. But, they should first find the problem and determine whether a design pattern is applicable or not. If a design pattern is not applicable, developer should devise a pattern, or at least a reusable solution.

As stated before, design patterns does not provide code, or class reuse, they provide solution reuse. That is design patterns do not emphasize code reusability, they define a general solution to a common occurring problem in a context. Underlying objects of a problem and their relations are determined after deciding which pattern is best to apply.

In general, a pattern has four essential elements [GOF98]:

- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- The **problem** describes when to apply the pattern.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.
- The **consequences** are the results and trade-offs of applying the pattern.

The details of the investigated design patterns are out of the scope of this research. Thus, patterns involved will be explained briefly and mainly the applicability on software error-proneness will be presented, if possible.

2.6. The Relation between Software Error-Proneness and Design Patterns

2.6.1. Gang of Four (GOF) Patterns

[GOF98] separates design patterns into three main groups:

- **Creational**
- **Structural**
- **Behavioral**

Creational patterns mainly solve the problem of the instantiation process. Usage or application of these creational patterns makes a system/software free of how its objects are created, composed, and represented. Structural patterns help designers to solve the problem of composition of classes or objects to make larger objects or collaborations. Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

In the succeeding sections, [GOF98] patterns will be argued under aforementioned groups.

2.6.2. Creational Design Patterns

These patterns are:

- **AbstractFactory**
- **Builder**
- **Factory Method**
- **Prototype**
- **Singleton**

Creational patterns do not affect structural relation of the functional or semantic classes in software, since these patterns have little or no direct effect on OO metrics. Also, Creational patterns are simple in logic and structure when compared to other type of patterns. They abstract the creation of entity or functional classes for their clients, which makes clients a bit simpler.

Also, application of creational patterns decreases import coupling, if client is responsible for the creation of many other classes to perform a service, which decreases CBO metric for client. However, in this thesis, the effect of application of creational patterns on OO metrics is not empirically evaluated. Creational patterns have actually no direct relation with software error-prone modules, but they make system more flexible and portable. Therefore, they may corporate with other design patterns.

2.6.3. Structural Design Patterns

2.6.3.1. Bridge Pattern

The intent of this pattern is to decouple an abstraction from its implementation so that the two can vary independently. Figure 1 shows UML structure of Bridge pattern.

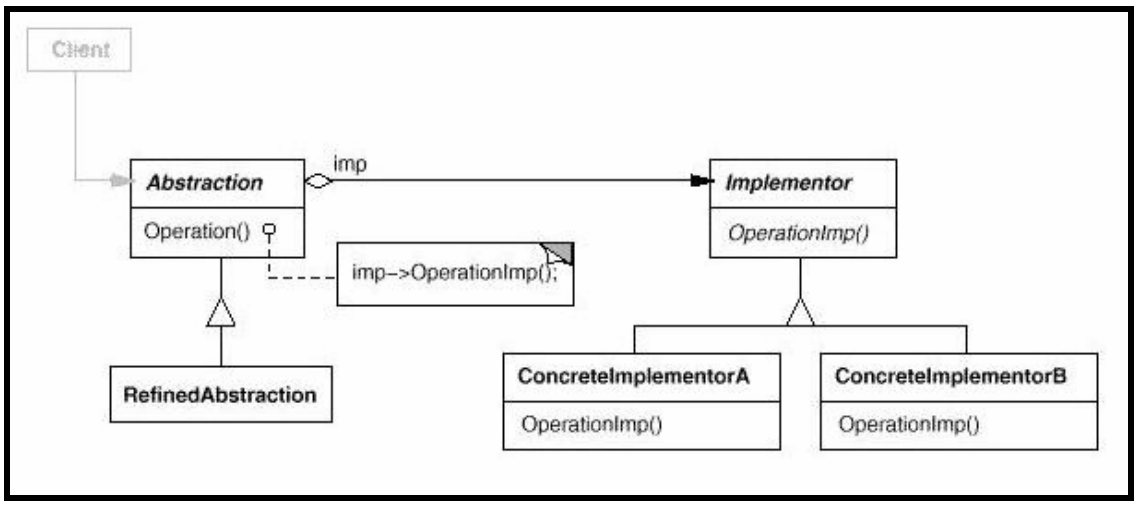


Figure 1 - Bridge Pattern [GOF98]

This pattern is important from the view of error-proneness and fault-tolerance, since this pattern provides decoupling interface and implementation, improved extensibility, and hiding implementation details from clients. By using this pattern, developer can devise implementation independent actuator, sensor, algorithm, and complex-structured classes. In other words developer gains the key to flexibility, portability, and fault tolerance; failed implementations can be changed at run-time easily if this pattern is used during the detailed design of related module.

The effect of Bridge pattern on DIT was investigated in [BH01], where the author shows that application of this pattern reduces average DIT in the circumstances covered by the constraint conditions. But applicability of this patterns are problem context depended. All the above patterns are discussed in the design patterns section.

But, as stated by [CK94] the DIT is a metric that related with class or module complexity, and also with fault tolerance intuitively. But, subclassing a complex fault prone module decreases WMC metric, moreover contributes to the fault tolerance of the related module. Furthermore, this metric doesn't help developers to write well designed systems [AV04]. It only says how deep the inheritance hierarchy is in the

written project. It doesn't matter if you collect a number of 3 for instance; it just tells you that the length from this node to the root is 3.

In addition, [BH00] has empirically showed that application of this pattern reduces the average DIT metric, as explained in preceding parts, association between DIT metric and fault-proneness is validated in literature. But, as stated before, the effect of DIT metric on error-proneness is less than other OO metrics associated with error-proneness.

Moreover, [BH00] has showed that this pattern has a distinct reducing effect on average NOC. The relation of NOC metric with error-proneness has been stated in the preceding sections.

But, scalability is a concern, application of Bridge pattern may show a very small part of the software, and consequently increase in overall tolerance will be negligible. Also, as stated in the viewpoints large number of subclasses indicates the misuse or overuse of inheritance, also NOC is an indicator the level of subclassing. Designer can interpret problem domain into application domain wrongly and can misuse or overuse inheritance property in object-oriented languages. Design techniques for the interpretation of the real world into software are beyond the scope of this research, but if large number of subclassing is inevitable, some design patterns can solve this problem.

Consequently, Bridge pattern reduces DIT and NOC metric values; this result leads to an interpretation that is Bridge pattern reduces average number of error-prone classes in software. But, before the application of this pattern, if the aim is to reduce number of faulty classes, detection of these classes carries great importance. Another factor that affects applicability is the fault class or classes should define a specific problem context, which defines Bridge pattern as a solution.

2.6.3.2. Composite Pattern

The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Figure 2 shows UML structure of Composite pattern.

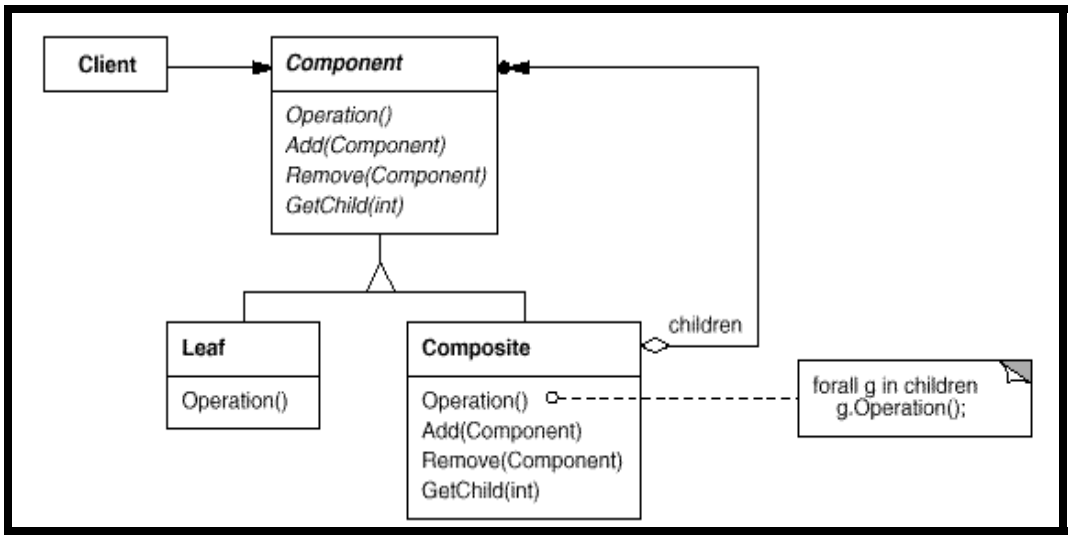


Figure 2 - Composite Pattern [GOF98]

Moreover, this pattern has several consequences. First of all, primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object. Secondly, Clients can treat composite structures and individual objects uniformly. Moreover, this pattern makes it easier to add new kinds of components. But, the disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Run-time checks should be added to the code.

Composite pattern simplifies the client class coupling, which reduces CBO values of client classes, if client has to manage, store, or perform operations on different “Leaf” classes. Since, these responsibilities can be transferred to “Composite” class without exposing “Leaf” classes to the clients.

2.6.3.3.Decorator Pattern

The intent of this pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. Figure 3 shows UML structure of Decorator pattern.

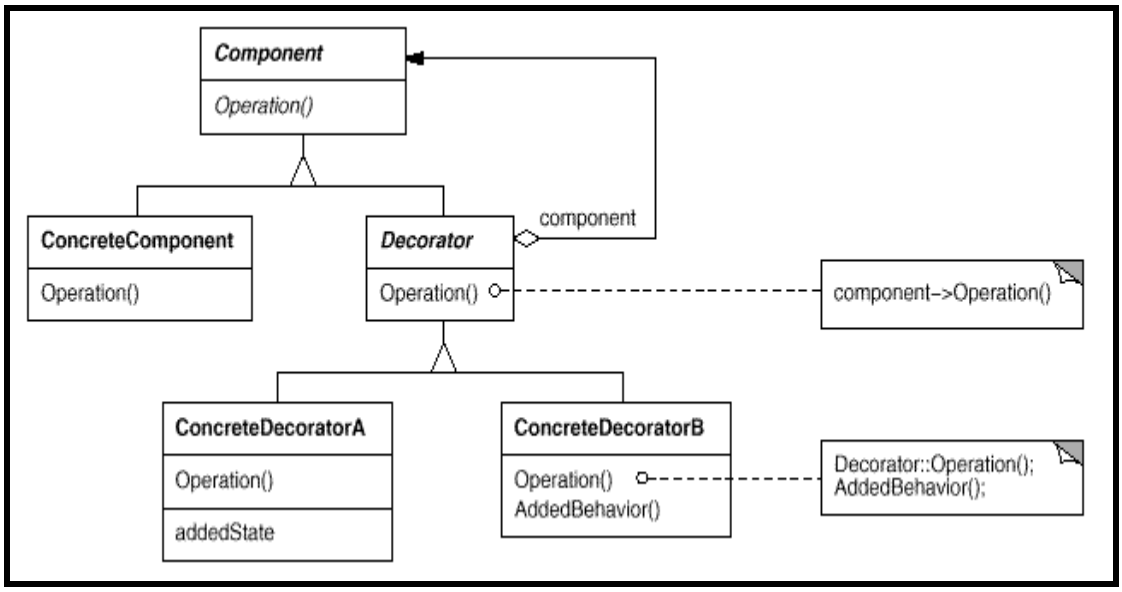


Figure 3 - Decorator Pattern [GOF98]

In addition, this pattern provides more flexibility than static inheritance, but it can introduce lots of little objects to the detailed design, also a decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. But, Decorator pattern helps to designer add additional responsibilities or features to a hierarchic class design in flexible way.

Decorator pattern helps designer to reduce extensive subclassing [MC96]. Therefore, this pattern mainly reduces NOC metric in software.

2.6.3.4.Façade Pattern

The intent of this pattern is to provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. Figure 4 shows UML structure of Façade pattern.

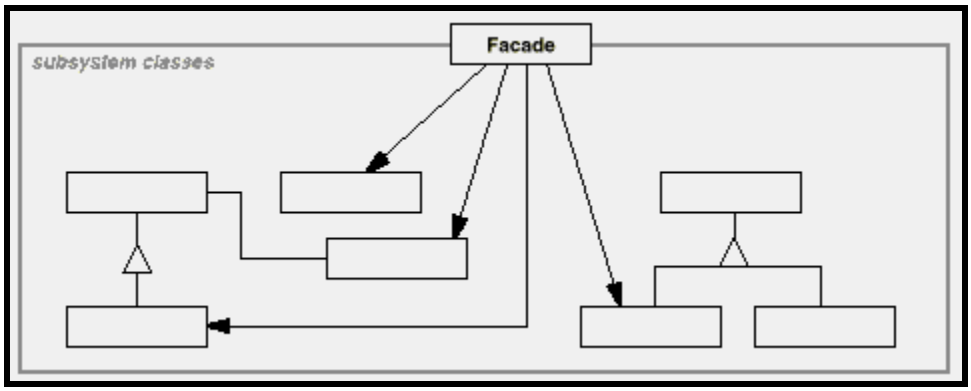


Figure 4 - Façade Pattern [GOF98]

As it can be seen, Façade class or Façade pattern provides only one entry point for a package or a subsystem [RJB99]. Therefore, all the clients that use one or more classes in related package or subsystem do not have relations with those classes; they only know "Façade" class and all the services are requested over that class. In short, it shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use. Façade classes can eliminate complex or circular dependencies.

Façade pattern increases the stability of the package. Stability is related to the amount of work required to make a change. In software engineering, stability of a package is measure by instability metric (*I*) [RCM00]. This metric simply shows stability of a package. If there are no outgoing dependencies, then *I* will be zero and the package is stable. If there are no incoming dependencies then *I* will be one and the package is instable. This pattern makes instability metric (*I*) of a package approach zero.

Moreover, Façade pattern can also decrease CBO metric, since it sets a wall behind the semantic classes and all operation flow over package façade. That is to say that this

pattern reduces instability metric as stated before. And instability metric is highly correlated with CBO metric. Therefore, this design approach significantly decrease export/import coupling from/to the related package or subsystem.

2.6.4. Behavioral Patterns

These patterns characterize complex control flow that's difficult to follow at run-time. They shift the developers focus away from flow of control to let them concentrate just on the way objects are interconnected.

2.6.4.1. Chain of Responsibility Pattern

The intent of this pattern is to avoid coupling the client of a request to its server by giving more than one object a chance to handle the request. Figure 5 shows UML structure of Chain of Responsibility pattern.

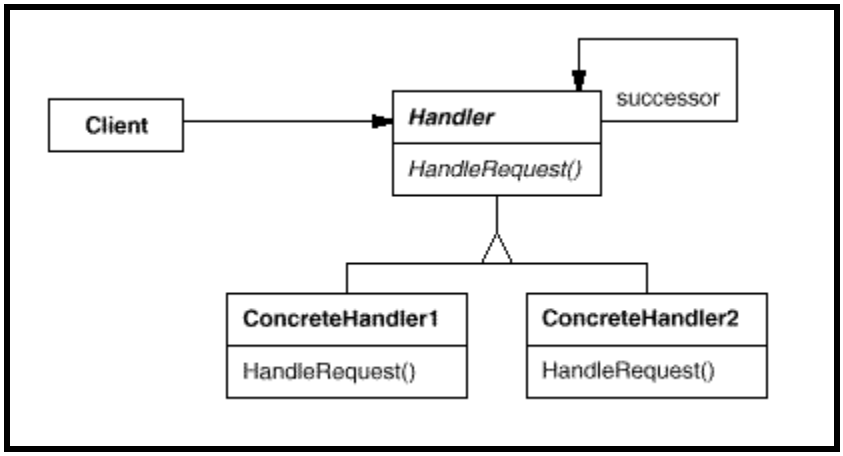


Figure 5 - Chain of Responsibility Pattern [GOF98]

This pattern as stated above provides reduced coupling between the sender and receiver. That is both the receiver and the sender have no explicit knowledge of each other, and an object in the chain doesn't have to know about the chain's structure. Moreover, this pattern provides added flexibility in assigning responsibilities to objects.

Structure of the chain can be changed at run-time, or combine with subclassing to specialize handlers. But, there is one important disadvantage clients or senders are not informed about whether the request is handled or not. Since, low coupling between the sender and the receivers, and the chain like structure prevents handlers to inform sender.

As it can easily be seen after some inspection, there is a similarity between the "Composite" and "Chain of Responsibility". But, in this pattern, concrete classes reach, or invoke themselves through common interface. Therefore, each concrete class may have association with a successor or a handler. But, the problem comes from inherent weakness of this pattern, which is that sender or client of an operation will never be guaranteed that the operation is handled completely and correctly.

This pattern, if applied correctly and all the constraints are met, decreases CBO metric, since client does not have to know all handlers in the design which decreases client's CBO metric value.

2.6.4.2. Command Pattern

The intent of this pattern is to encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. Figure 6 shows UML structure of Command pattern.

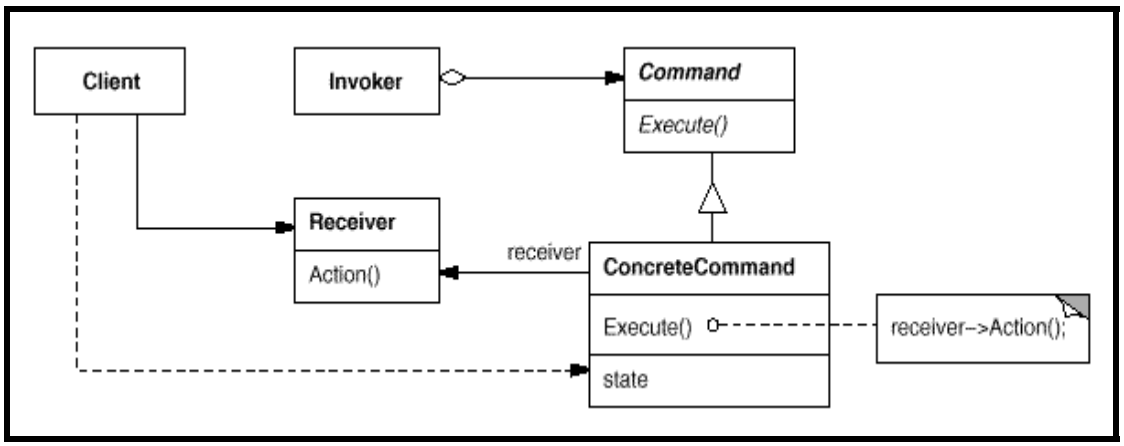


Figure 6 - Command Pattern [GOF98]

Command pattern decouples the object that invokes the operation from the one having the knowledge to perform it. This property makes this pattern crucial for user interface designs. An application can provide both a menu and a push button interface to a feature just by making the menu and the push button share an instance of the same concrete Command subclass. Command pattern is highly flexible from the view of clients, because, the object that issues request only needs to know how to issue it; it doesn't need to know how the request will be carried out. This last statement implicitly declares that application of this pattern may reduce CBO and WMC values of "Invoker" classes.

2.6.4.3. Mediator Pattern

Intent of this pattern is to define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. Figure 7 shows UML structure of Mediator pattern.

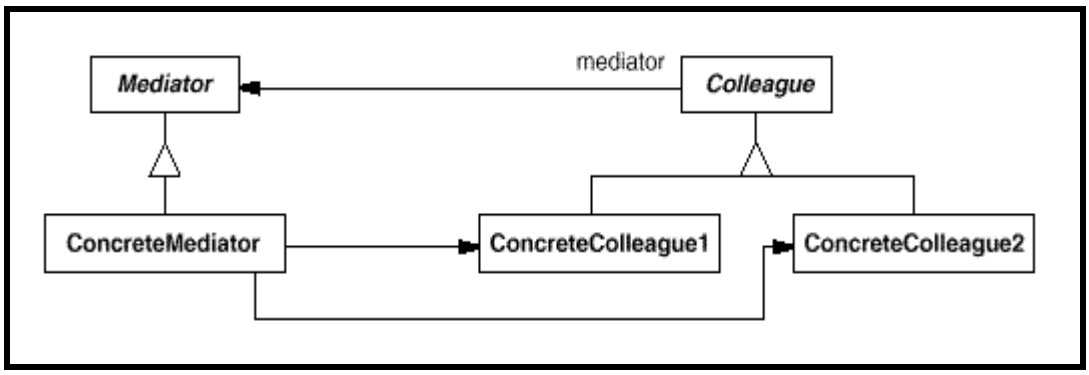


Figure 7 - Mediator Pattern [GOF98]

Object-oriented design encourages the distribution of behavior among objects. Such distribution can result in an object structure with many connections between objects; in the worst case, every object ends up knowing about every other. This property makes this pattern crucial for OO designs and also for fault tolerance. Since, Mediator reduces overall complexity in the design, can help designer to vary their relations implicitly at run-time. Moreover, objects only know the mediator, thereby reducing the number of interconnections. This property also reduces the subclassing for implementing different behaviors. In other words, this pattern reduces the overall size of the classes that have many relations with other classes, the number of classes in the packages, the overall complexity, and increases reusability. Also, a mediator centralizes the control; mediator pattern trades complexity of interaction for complexity in the mediator, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain. But, knowing where a nightmare may occur is much better than trying to find or predict where it may occur.

The effect of Mediator pattern on coupling between object was scrutinized in [BH01]. In this paper, author shows that application of Mediator pattern decreases coupling between objects. This conclusion helps to link the connection between software error-proneness and mediator pattern. But, one drawback pointed out in [BH01] is that author devises the scenario and uses the small number of the classes for graphical comparison,

without being concerned with cases that have very large number of classes, or a framework.

2.6.4.4. Memento Pattern

Intent of this pattern is, without violating encapsulation to capture and to externalize an object's internal state so that the object can be restored to this state later. Figure 8 shows UML structure of Memento pattern.

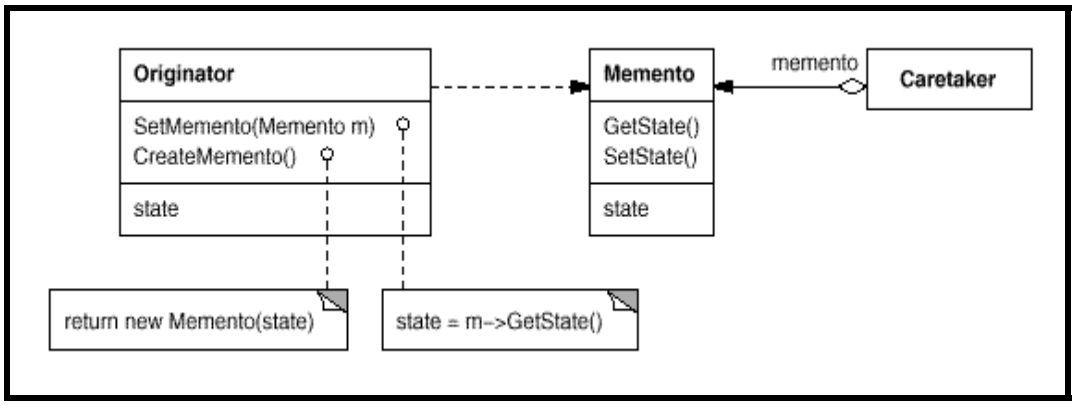


Figure 8 - Memento Pattern [GOF98]

Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors. This property makes this pattern crucial for fault tolerance when recovering from a crash. Some error prone classes determined during design can be applied to the system with Memento pattern. Consequently, recovering these modules and putting them into their normal operating states will be easier. These approach increases system fault tolerance to higher level, but never makes it completely error or fault resistant. One drawback of this pattern is that it violates encapsulation, because objects usually hide their member variables or internal states, in order to apply this pattern, an originator, an object that wants to be remembered, should save its encapsulated variables into another class -memento-, and should have the ability to reach hidden variables of that memento class. Also, this pattern introduces

new memento classes to the system that increases the size and complexity. Designer should trade between these advantages and drawbacks, and should not overuse this pattern.

2.6.4.5. Strategy Pattern

Intent of this pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. Figure 9 shows UML structure of Strategy pattern.

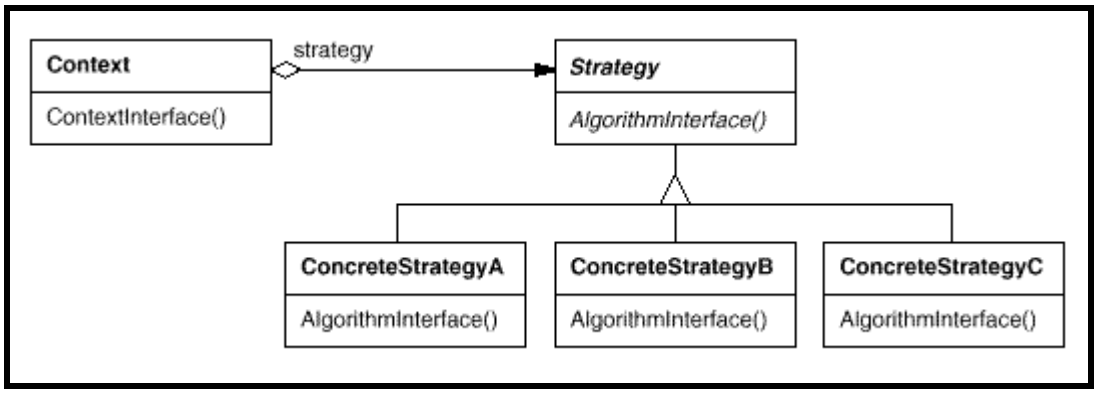


Figure 9 - Strategy Pattern [GOF98]

Moreover, this pattern helps designer to eliminate conditional statements in source code of "Context" type classes [GOF98]. Therefore, it can reduce complexity related metric values, like WMC value.

2.6.4.6. Template Method Pattern

Intent of this pattern is to define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. Figure 10 shows UML structure of Template Method pattern.

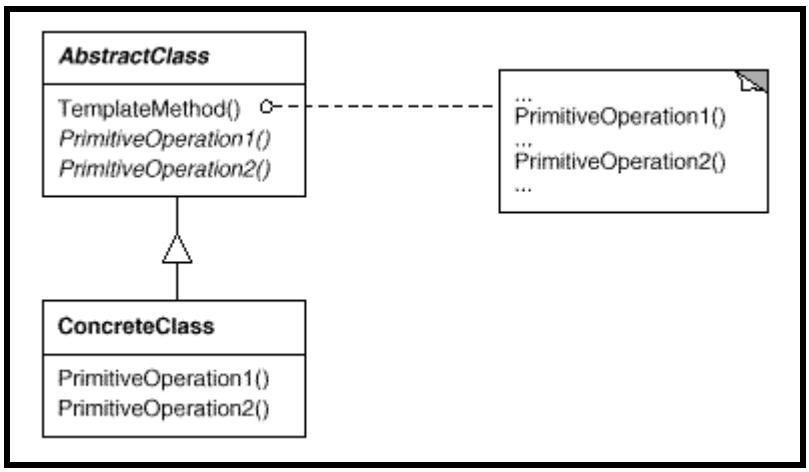


Figure 10 - Template Method Pattern [GOF98]

Template methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes. Therefore, from the view of software error-proneness, usage of this pattern is ambiguous. But, according to [GOF98], this pattern can limit the number of primitive methods that a subclass must override to flesh out the algorithm. This property helps designer to reduce the values of RFC, WMC [CK94] metrics for the related classes. As explained in section 2.1.2, RFC and WMC are associated with software fault tolerance or error-proneness [EBGR01].

2.6.4.7. Visitor Pattern

Intent of this pattern is to represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. Figure 11 shows UML structure of Visitor pattern.

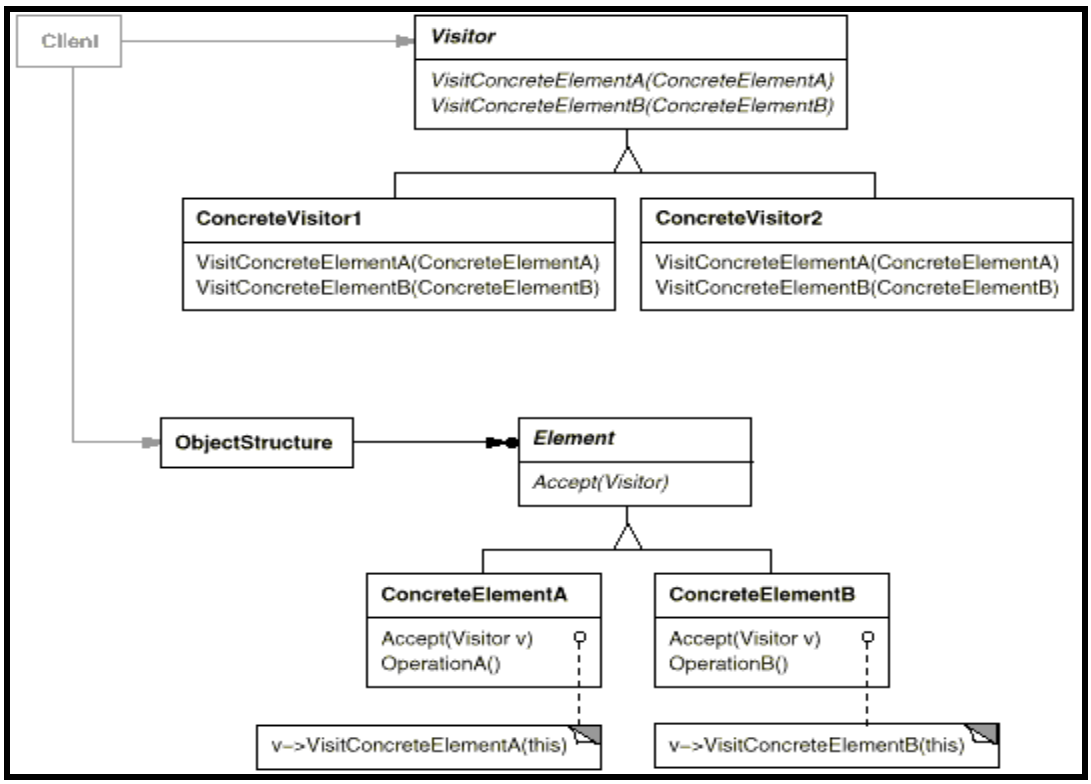


Figure 11 - Visitor Pattern [GOF98]

Intent of this pattern states that this pattern adds new operations to complex objects structures without changing their internal structures. The basic effect of this pattern is the removal of some operations from the class hierarchy and replaces them with new "Visitor" hierarchy. This last statement tells us implicitly that application of this pattern changes metrics associated with method count and class complexity, e.g. WMC, and NMO.

Application of Visitor pattern on OO metrics has been researched in [BH01]. The results of this paper about application of aforementioned pattern have a significant reducing effect on WMC. And also, as stated and proven by [TKC99] WMC is related with software fault proneness, therefore Visitor pattern should be used when ever possible to decrease software error-proneness relatively by reducing the number of faulty classes or error prone class associations.

And also, [BH01] has empirically showed that Visitor pattern has a significant reducing effect on NMO, but for obviously not limited or low number of overridden Visitor methods. This result helps us to observe the relation of Visitor pattern and error-proneness in software. If the context is compatible with the solution that this pattern offers, “Visitors” reduce the number of error-prone modules.

2.6.4.8. Iterator Pattern

Intent of this pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Figure 12 shows UML structure of Iterator pattern.

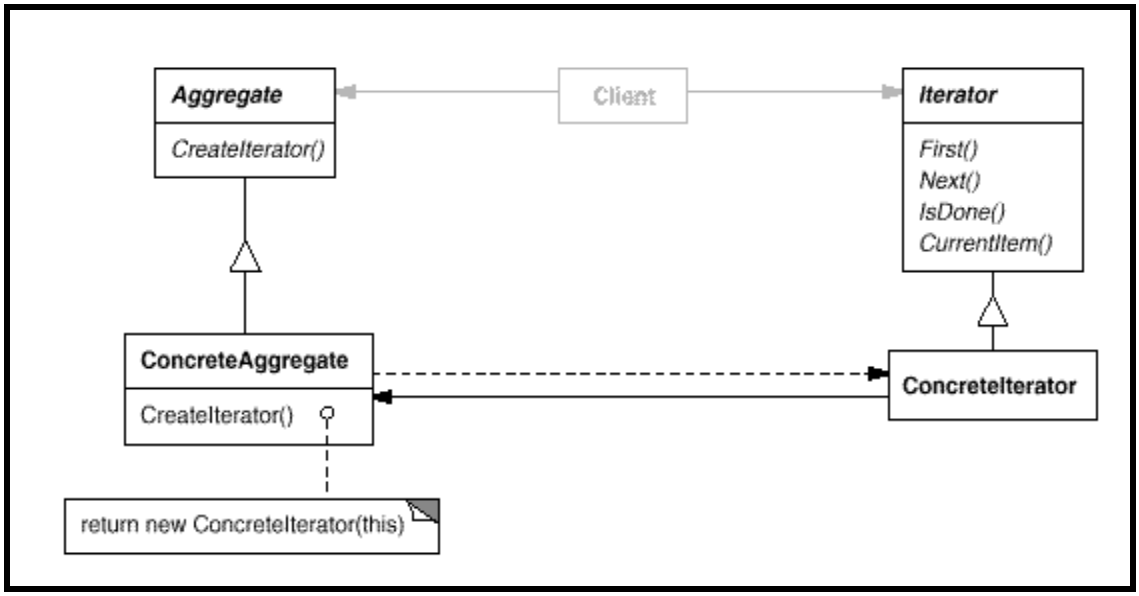


Figure 12 - Iterator Pattern [GOF98]

This pattern provides many benefits for multiple aggregate relations. First, provides access to aggregate’s objects without exposing their internal structure. Secondly, Iterator pattern supports multiple traversals of aggregate objects and lastly, provides a uniform interface for traversing different aggregate structures. Moreover, this last statement implicitly states that this pattern helps designer to support different traversing

algorithms without contaminating the client code, which may decrease or limit WMC value for these client classes.

2.6.4.9. Observer Pattern

Intent of this pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Figure 13 shows UML structure of Observer pattern.

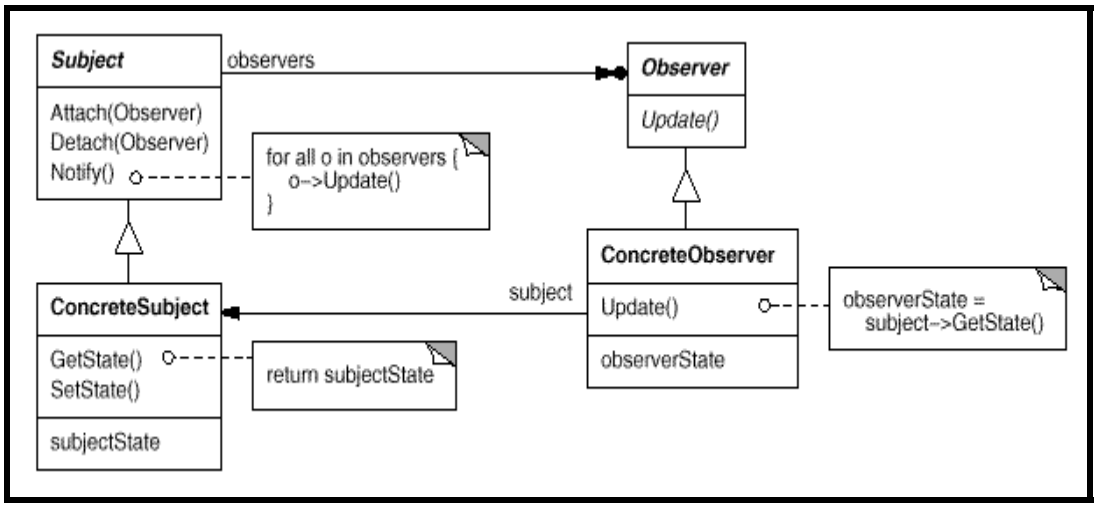


Figure 13 - Observer Pattern [GOF98]

This pattern helps designers to vary subjects and observers independently, and to increase reusability. Therefore, from the view of software error-proneness this pattern may have no direct usage. But, this pattern has investigated deeply in chapter 3 to see its effects on software error-proneness related metrics.

2.6.4.10. State Pattern

Intent of this pattern is to allow an object to alter its behavior when it's internal state changes. Figure 14 shows UML structure of State pattern.

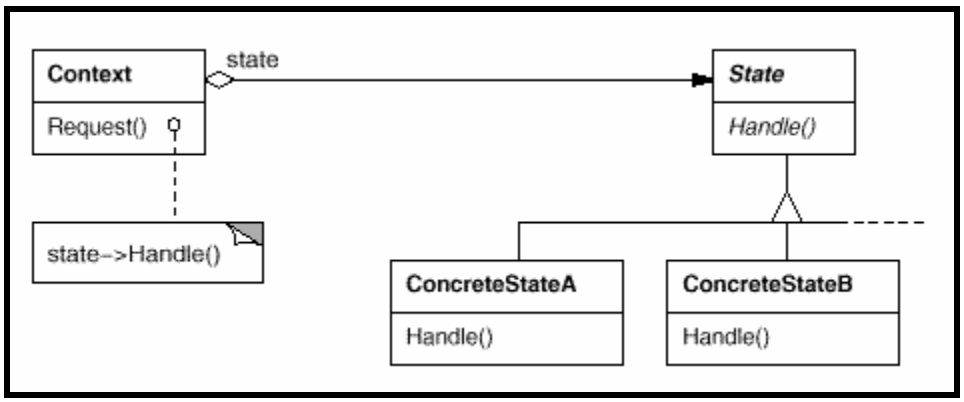


Figure 14 - State Pattern [GOF98]

State pattern separates object's state form its implementation code. Therefore, this pattern simplifies object's source code. That is to say State pattern reduces the WMC value for classes that have intrinsic behavior in different circumstances. But, on the other hand, this pattern introduces new state associations and consequently increases the CBO value. Moreover, application of this pattern breaks encapsulations with "friend" type inclusions [GOF98].

2.6.5. Omitted GOF Patterns

Some of the structural and behavioral patterns of [GOF98] has been omitted in this chapter and not further investigated in chapter 3. These patterns are:

- Adapter Pattern
- Flyweight Pattern
- Proxy Pattern
- Interpreter Pattern

Adapter and Proxy patterns have been omitted, since these patterns have relatively simple structure and are easy to implement. Therefore, they have almost no effect on OO metrics. Moreover, Flyweight pattern is almost a "creational" pattern, and its applicability provides only space optimizations [GOF98]. Lastly, the omission reason

for Interpreter pattern is that it has a narrow application area restricted to when the used language is simple and performance is not a critical concern [GOF98]. Moreover, Interpreter pattern is very similar to Visitor pattern in its structure [GOF98]. Therefore, Visitor pattern experiments are considered to be enough to show effects of Interpreter pattern.

2.6.6. Overview of Real-Time Design Patterns

These patterns are widely known in the literature, but for simplicity and consistency, [BPD02] has been referenced for these patterns.

Furthermore, showing the effects and comparison results of [BPD02] patterns are problematical, since these patterns approach a system or software from an architectural level, and also offer pattern structures for embedded systems, where real-time performance, code size, effective memory usage, safety, system size are main concerns. Besides, [BPD02] patterns are conceptually different from [GOF98] patterns that cope with problems in class level and therefore have direct effect on object-oriented metrics. While, on the other hand, [BPD02] patterns cope with embedded software development problems and devise patterns for the generalized solutions for them.

Consequently, the real time design patterns of [BPD02] approach software in a more abstract way than [GOF98] patterns. This makes patterns, mentioned in [BPD02], hard to measure and analyze from perspective of the OO metrics that are related with software error-proneness. Therefore, real-time design patterns are excluded from the scope of this study.

CHAPTER 3

EXPERIMENTAL WORK

3.1. Introduction

As stated in the previous chapter, there are direct relations with design patterns and OO oriented metrics. Some of the OO metrics are strongly associated with the probability of error-proneness of a module in software. In addition, the literature review has shown that design patterns like "Mediator" (see. Sec. 2.3.4.3.) are related with OO metrics. This important result is the most influential stimulus factor for this chapter. Since, it implicitly declares that design patterns can affect error-proneness.

This chapter will attempt to give empirical answers for the following questions:

- What is the effect of design patterns on the class error-proneness?
- Does correct design pattern application solve metric defined problem?
- Does application of design patterns add asset to software?

Based on the results and consequences of previous chapter, in this part, effects, benefits, and drawbacks of application of design patterns on error-prone modules or classes will be shown empirically. Also, important interpretations will be made, which will be a basis for related future researches.

First of all, in this chapter, six phases for measuring the effects of application of chosen design patterns are implemented. These phases are:

1. Measuring software to obtain OO metrics
2. Determining error prone classes based on OO metrics
3. Hatching correct design pattern for the problem described by metrics

4. Applying related pattern(s)
5. Re-measuring software
6. Comparing and interpreting the results of step 1 and step 5

There are many OO metrics in the literature, but in step 1 and step 5, below metrics will be collected from software. These metrics were also reviewed and their relations with error-proneness were also stated in chapter 2.

- **Weighted Methods Per Class (WMC) [CK94]**
- **Depth of Inheritance Tree (DIT) [CK94]**
- **Number of Children(NOC) [CK94]**
- **Response For a Class (RFC) [CK94]**
- **Coupling Between Objects (CBO) [CK94]**
- **Number of Methods Added (NMA) [LK94]**
- **Number of Methods overridden (NMO) [LK94]**
- **Specialization Index (SIX) [LK94]**

In step 1, metric values for software classes or modules are obtained. After step 1, in step 2, error-prone classes will be determined.

Step 3 actually is the most difficult part, since there is no standard procedure for pattern hatching using metric values. In next step, useful pattern(s) that is/are compatible with metric defined problem will be applied to software.

In step 4, results of step 3 that are compatible design patterns are implemented, also required structural and code level alterations are made. But, there is an implicit declaration that refactoring will also be applied to code in this step. Since source code fragments need refactoring before and during the application of design patterns. Usage

of refactoring for this part does change the structure and associations of components, modules, and moreover, eliminates common programming errors and misusages.

Refactoring is another outcome, not a necessity. In view of the fact that, the application of a design pattern needs refactoring. Therefore, after this step, structure of software components or modules will be changed. The result of this step will be the input of next step.

In step 5, altered software has to be re-measured, in order to interpret and compare metric values that are acquired in this step and in step 1. Collection of these metrics values for these steps will be made by using a software engineering tool that is available in market. The details of this tool will be given in succeeding parts.

Step 6 is the one of the most crucial part for this thesis. Because, comparisons and interpretations of application of design pattern(s) on OO metrics will be stated in this step. Moreover, according to the results, validation or repudiation of the effect of implementing design patterns on decreasing class error-proneness will be performed.

The weakness of this empirical approach is that there is not a standard approach or procedure for the application of a design pattern in software as stated before. In step 4, solution to the metric defined problem will be made by developers. However, knowledge levels of developers are variable and different. That means some developers may apply wrong design patterns, or apply correct pattern but writes code and refactor poorly, which can make software even more error-prone.

On the other hand, results may show that talented and experienced developers can decrease the number of error-prone classes by the help of design patterns and OO metrics. This means much for software development cycle, since less error-prone classes shortens the test and debug phases, also increase analyzability and reusability for software. These positive side effects decrease the cost of development also.

3.2. The General Characteristics of Researched Software

The researched software should be written in an OO language, like C++, Ada, or Java. Since, OO metrics will be collected and also design patterns can only be applied into an OO project.

Moreover, software should or could be design pattern free. Since, one of the aims of this thesis study is to show empirically the effects of design patterns on error-prone class probability. The most effective and persuasive way for this objective is to choose software that originally lacks design patterns in its architecture.

Also, software or components should be researched after their detailed design phase. Since, completing immature software is out of the scope for this thesis.

In conclusion, software that will be researched should have to be object-oriented, design pattern free, and in test phase level.

3.3. Metric Evaluation Tool

In the market, there are many metric evaluation tools, but unfortunately many of these tools do not provide interface for developing user-defined metrics or define existing important metrics and also they are incapable to help developer/user to evaluate metrics that they collect.

Moreover, many of these tools do not recommend preferred or acceptable threshold levels, and do not warn user about error-prone classes or modules. In addition, they do not focus on metric evaluation, but they exactly work as reverse engineering tools.

In this study, measurement tool used for projects, written in C++ language, is Logiscope [TTL00]. This analysis tool has been developed in 2000 by Telelogic. Logiscope is commercial and has many OO, functional and application metrics in its metric database. Moreover, Logiscope provides its users to add custom metrics, and also generates project metric analysis reports. Consequently, Logiscope has been used for this thesis to evaluate OO software projects.

But, main problem for metric collection and evaluation tools is that they do not have common metric computation methods. Most of them are not compatible with standards, except for Logiscope. In Logiscope, software metrics templates used to evaluate the code are ISO 9126 compliant.

In addition, there is not any measurement tool that computes NMA and SIX metric values directly. Therefore, in this thesis, these metric values have been computed externally by using other metric values.

3.4. Researched Software Projects

In the experiments that have been performed in this section, all the average metric scores will be reported as per class, whereas maximum and minimum values will be presented for overall project classes.

3.4.1. Case 1: NetClass [NC01]

NetClass is a very simple to use socket and thread wrapper that works under UNIX, LINUX based systems as well as WIN32 based systems. It is written in C++ and is focused on simplicity and performance.

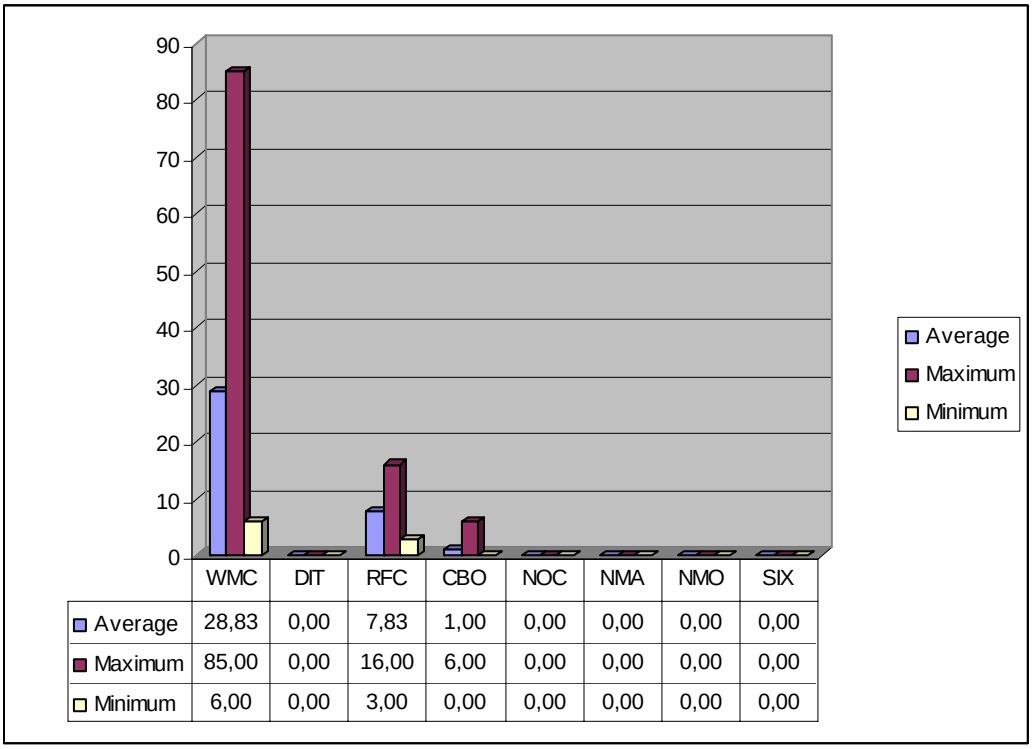
NetClass project is an open source project, hosted by sourceforge.net and distributed under General Public License (GPL).

3.4.1.1. Phase One: OO Metrics Measurement of Overall Project

Below project summary is given:

Project Summary:

- Classes: 6
- Files: 15
- Functions: 39
- Lines: 2161
- Version: 0.3.3



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	28,83	0,00	7,83	1,00	0,00	0,00	0,00	0,00
Maximum	85,00	0,00	16,00	6,00	0,00	0,00	0,00	0,00
Minimum	6,00	0,00	3,00	0,00	0,00	0,00	0,00	0,00

Figure 15 - Average, Maximum and Minimum Values of Thesis Metrics for NetClass Project before Refactorings

Figure 15 shows to metric interpreter or software metric analyzer that WMC value of this project has high average and maximum values. Also, Figure 15 reveals that WMC metric values should be decreased in order to make software less error-prone.

3.4.1.2. Phase Two: Determining Error Prone Modules

In the NetClass project, three classes have approximately 83% of sum of WMC value among six classes. These classes are the rotten parts or error-prone modules of this project and they should be refactored.

WMC values of this classes and total WMC are given graphically in Figure 16.

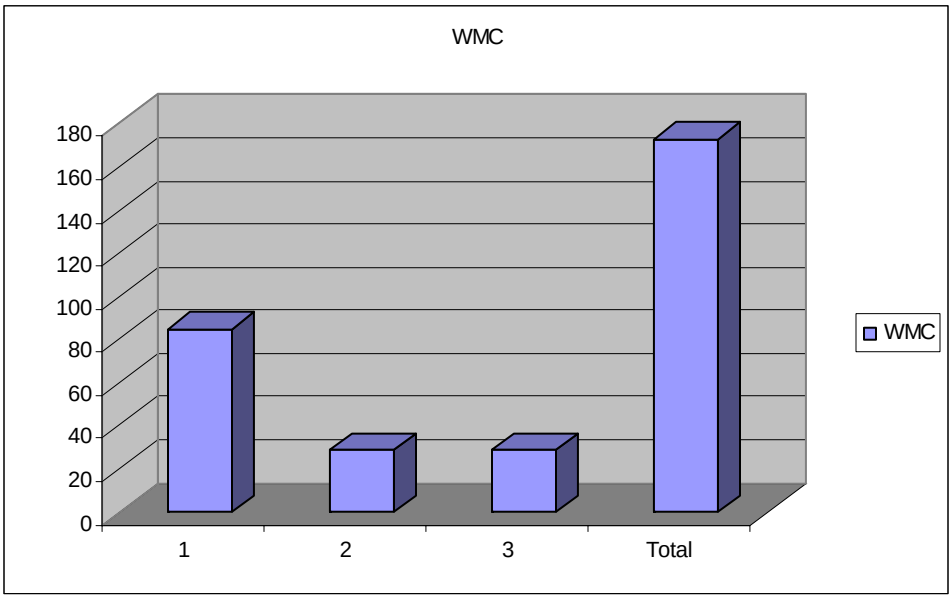


Figure 16 - Top Three classes for WMC metric in NetClass Project

3.4.1.3. Phase Three: Hatching Design Pattern(s)

Evaluations in chapter 2 have showed that “Template Method” and “Visitor” design patterns may decrease WMC values for classes.

But, NetClass project includes different implementations to support different operating systems. Also, in Figure 16, the class, labeled as 1, includes two different implementations in it. This property stimulates the usage of "Bridge" for this class.

Applied design patterns to NetClass project are given below:

- **Bridge Pattern**: used to separate abstraction and implementation
- **Template Method Pattern**: To simplify the complex methods. But, this pattern will be evaluated in detail in succeeding sections.

3.4.1.4. Phase Four: Applying related pattern(s)

After determining which pattern should be best for project, source code has been modified accordingly. For NetClass project, Bridge pattern introduced three new classes for the project. One has been added for base abstraction and the others have been added for two different implementations.

Moreover, Template Method has also been used. The purpose of the Template Method pattern is to divide complex methods into smaller sub methods and also to control subclasses extensions. In NetClass project, this pattern has been implemented in the base class of different implementations, and also to simplify complex methods in the class, labeled as 1, in Figure 16. But, this pattern has not reduced WMC value for this class; in contrast, it has increased WMC value. Since, even the simplest method, add 1 to the overall WMC value for a class. In addition, poor implementation of this rotten class did not allow the separation into simpler logical/procedural division of complex methods.

To sum up, "class 1" has been refactored, that is some complex methods divided by Template method. In addition, its two different implementations have been separated

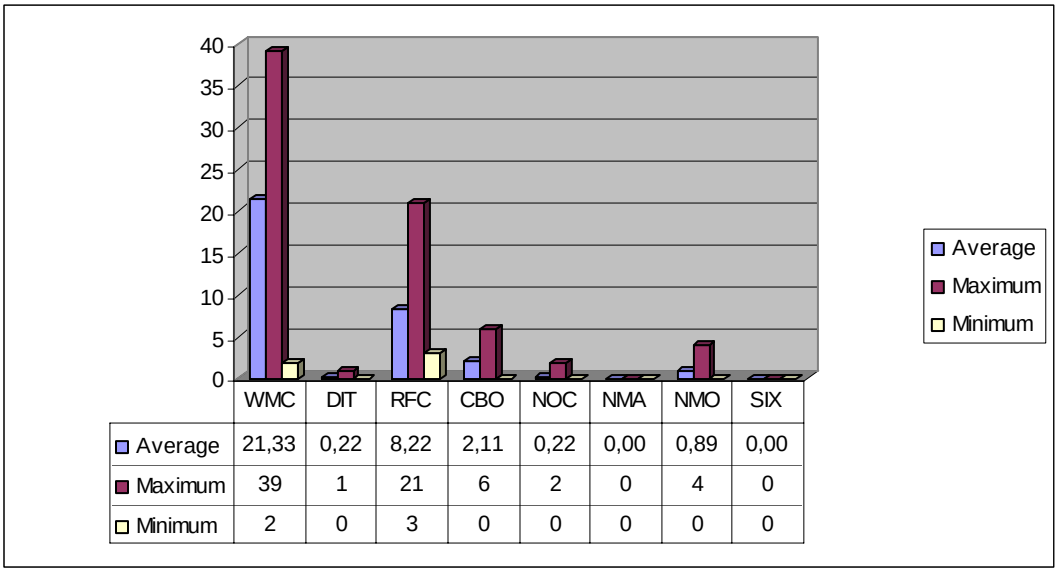
by applying “Bridge” pattern. Also, in the base class of these implementations, “Template Method” pattern has been used.

3.4.1.5. Phase Five: Re-measuring software

Below refactored project summary is given:

Project Summary:

- Classes: 9
- Files: 20
- Functions: 60
- Lines: 2321

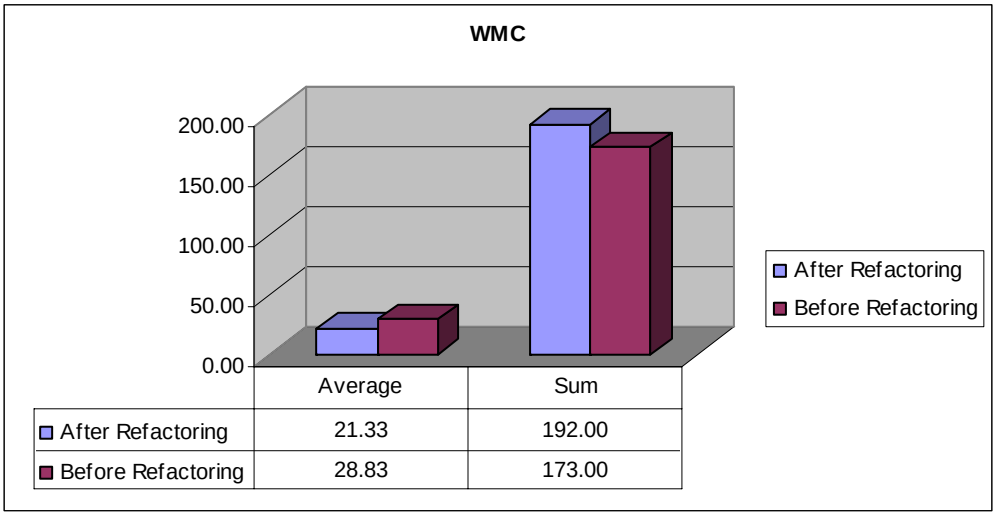


	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	21,33	0,22	8,22	2,11	0,22	0,00	0,89	0,00
Maximum	39	1	21	6	2	0	4	0
Minimum	2	0	3	0	0	0	0	0

Figure 17 - Average, Maximum and Minimum Values of Thesis Metrics for NetClass Project after Refactorings

Figure 17 graphically shows average, maximum, and minimum values of metrics for NetClass project after refactorings.

3.4.1.6. Phase Six: Comparisons and Interpretations



	Average	Sum
After Refactoring	21.33	192.00
Before Refactoring	28.83	173.00

Figure 18 - WMC Comparisons for NetClass Project

In Figure 18 graphically reveals that WMC and values for overall NetClass project has reduced approximately %24 for average and %50 for standard deviation after the implementations of Bridge and Template Method patterns.

Moreover, the main reason for the increase in overall project WMC value is 21 added methods. These methods have been imperatively added to implement Template Method and Bridge patterns into the project.

On the other hand, application of these patterns has slightly increased RFC, DIT, NOC and NMO metric values. Increase in RFC, DIT, NOC values is the side effect of Bridge pattern, whereas Template Method pattern has caused an increase in NMO metric value.

According to Figures 17 and 18, applied patterns has improved NetClass project design and reduced error-proneness with minor side effects, like slight increases in metrics except for WMC.

3.4.2. Case 2: Power-Grab [PG02]

Power-Grab project is a 32-bit Windows program written in Visual C++ 6.0 for scanning and downloading binary files from usenet news servers [PG02].

This project is an open source project, hosted by sourceforge.net and distributed under General Public License (GPL).

3.4.2.1. Phase One: OO Metrics Measurement of Overall Project

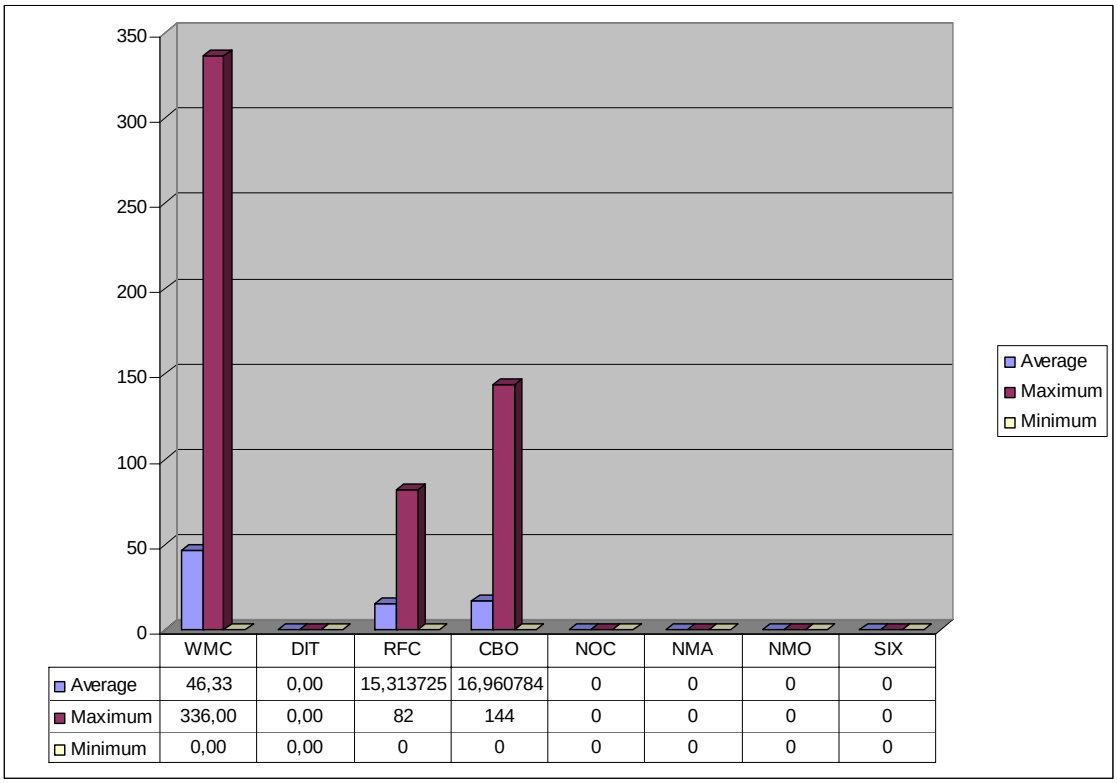
Below project summary is given:

Project Summary:

- Classes: 49
- Files: 97
- Functions: 884
- Lines: 26608
- Version: 2.0

Power-Grab project [PG02] uses Microsoft Foundation Classes (MFC), and the developer creates new classes that inherit from MFC. But, inheritance related metric values of these classes have been omitted. Since, the framework of MFC is beyond the scope of this thesis, and also Logiscope does not collect correct inheritance metric values of these classes. Omitted metrics for these classes are DIT, NMA, NMO and SIX. But, other metrics that are WMC, RFC, CBO, and NOC have been collected for all classes correctly. In addition, other classes, which do not inherit from one or more classes of MFC, have been measured and all the required metrics are collected for these classes.

Figure 19 reveals that developer of Power-Grab project has not used inheritance except for MFC classes. This makes DIT, NOC, NMA, NMO, and SIX metric values meaningless for this case.



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	46,33	0,00	15,313725	16,960784	0	0	0	0
Maximum	336,00	0,00	82	144	0	0	0	0
Minimum	0,00	0,00	0	0	0	0	0	0

Figure 19 - Average, Maximum and Minimum Values of Thesis Metrics for Power-Grab Project before Refactorings

3.4.2.2. Phase Two: Determining Error Prone Modules

In Figures 20 and 21, error-prone modules that have relatively high WMC, CBO, and RFC metric values are given. According to these graphics, there are 14 error-prone classes for WMC metric, 24 error-prone classes for CBO metric, and 2 error-prone classes for RFC metric. There are not standard threshold values for OO metrics. Therefore, these classes have been determined by using default threshold values that are assigned by Logiscope.

Moreover, a class, labeled as **CInt96** in Power-Grab project, has been excluded from the CBO measurement, since, this class actually has not been designed to be a class, it is indeed a type like integer, but it has been extended to manage 12 bytes.

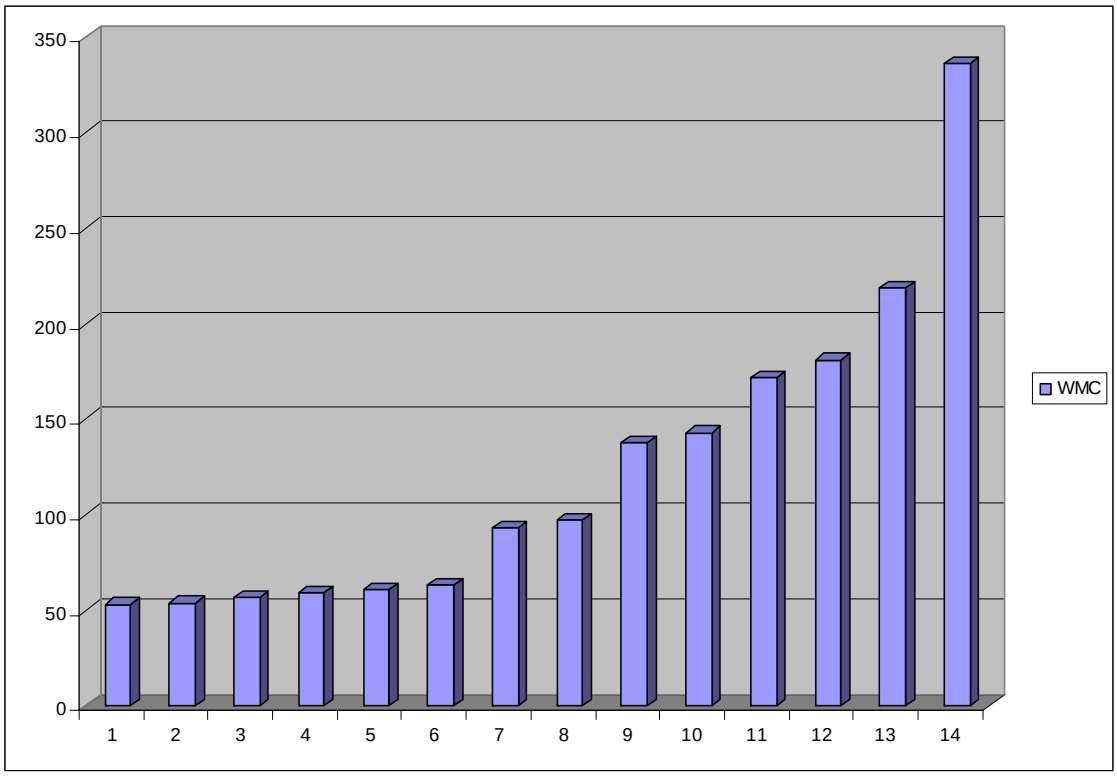


Figure 20 - Classes that have high WMC values for Power-Grab Project

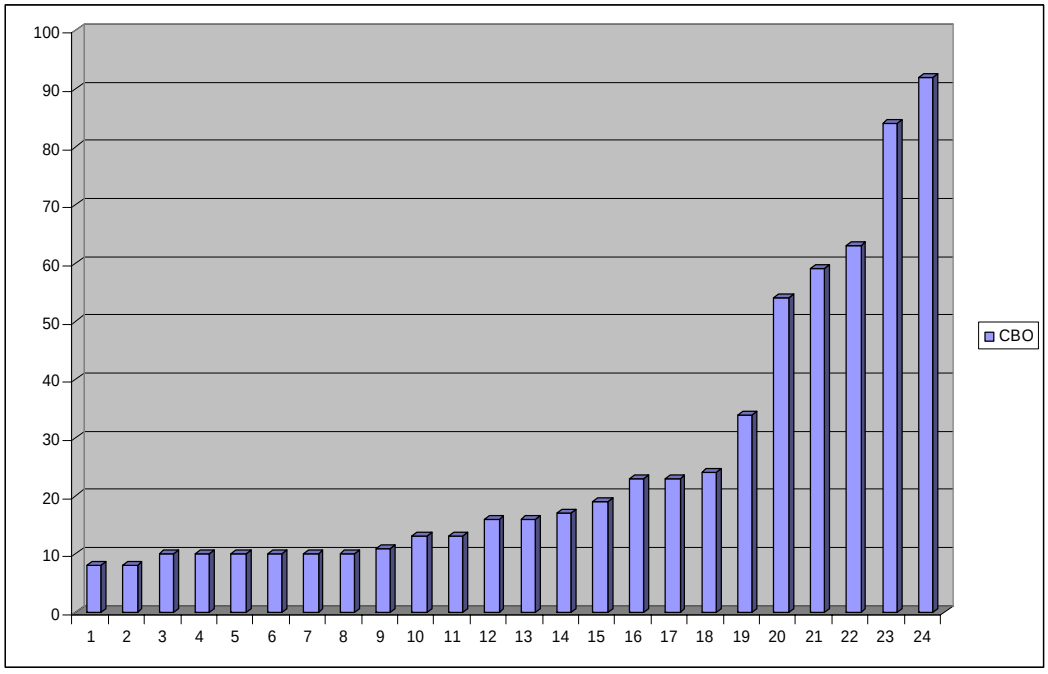


Figure 21 - Classes that have high CBO values for Power-Grab Project

3.4.2.3. Phase Three: Hatching Design Pattern(s)

First of all, as a refactoring decision, classes that have WMC and CBO metric values above non-acceptable default threshold values, mentioned in section 3.4.2.2., will be refactored first. In other words, classes that are labeled as 9 to 14 in Figure 20, and labeled as 9 to 24 in Figure 21 have superior priority among other classes.

Applied design patterns to Power-Grab project are given below:

- **Façade Pattern**:
  - The class, labeled as 24 in Power-Grab, is a mediator like class. That is it handles user events, and performs some application specific operations. This pattern isolates this class from application specific classes, which will reduce the CBO value for this error-prone class.

- **State Pattern**:

- This pattern reduces the complexity of a class that has different state and behaviors. Therefore, State pattern reduces WMC value, but slightly increases CBO value. Since, this pattern introduces new associations to the class, where it is being applied.

- **Singleton Pattern:**
  - Extern or global classes have been transformed into "Singletons", and they have been associated with Façade pattern. Thus, these global classes are abstracted from other semantic classes. But, the effect of this pattern is minor, and will not be evaluated as stated in chapter 2.

3.4.2.4. Phase Four: Applying related pattern(s)

State pattern has applied to **CTaskAgent** class in the project. Also, this pattern has introduced five new classes. Figure 22 displays the associations between these classes.

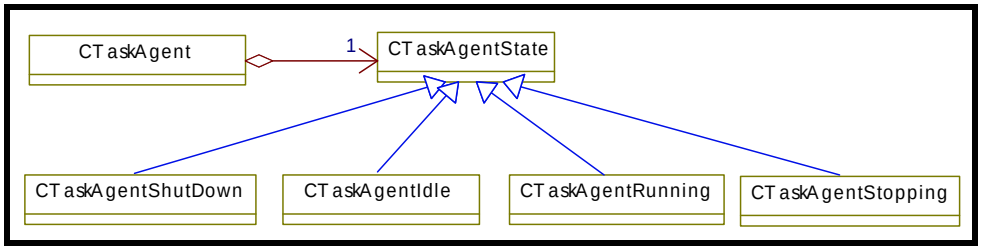


Figure 22 - CTaskAgent Class Relations after State Pattern Applied

Moreover, this pattern breaks encapsulation in **CTaskAgent** class. Since, **CTaskAgent** class has "friend" type inclusions with subclasses of **CTaskAgentState**. Therefore, these classes can access private member attributes and methods of **CTaskAgent** class.

Moreover, State pattern only affects the metric values of **CTaskAgent** class. Since, other classes that are inserted to the project to realize this pattern are isolated from other project classes. That is to say only **CTaskAgent** class has relations with them.

Other pattern that has been implemented is Façade pattern. This pattern has introduced only one class, which is named as **ApplicationEngine.** This façade class isolates user interface classes from semantic classes and vice versa. But, refactoring with Façade pattern took so much time to figure out relations and implement them. Lots of code and class relation modifications have been made.

3.4.2.5. Phase Five: Re-measuring software

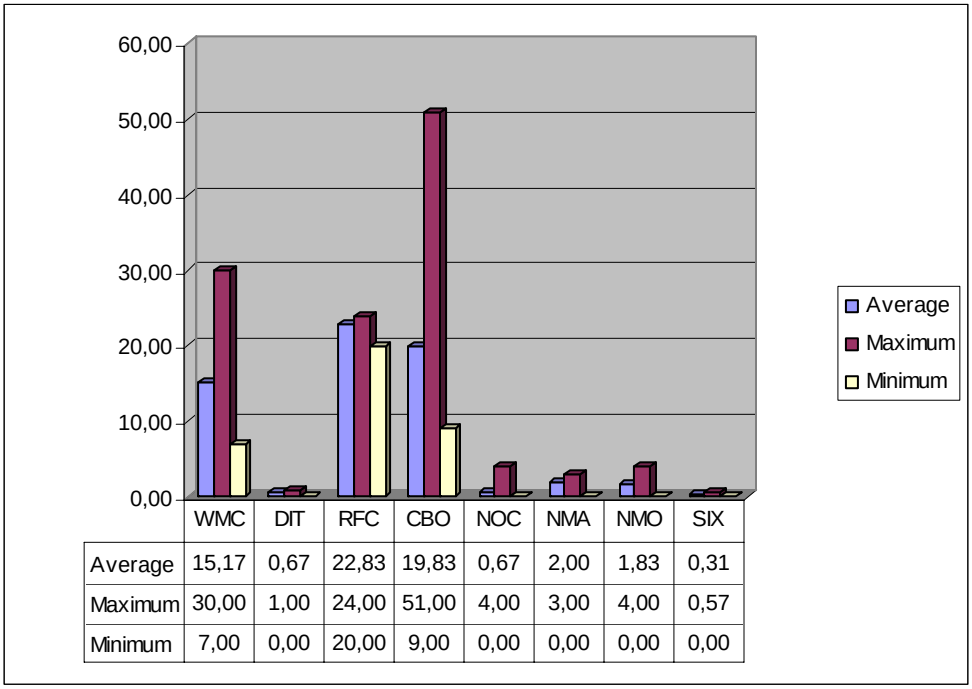
3.4.2.5.1. State Pattern Implementation

Below refactored project summary is given:

Project Summary:

- Classes: 54
- Files: 107
- Functions: 929
- Lines: 27179

Since, State pattern does only affect the class where it has been applied. Figure 23 displays metric values of the classes that are inserted to the project for realizing this pattern.



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	15,17	0,67	22,83	19,83	0,67	2,00	1,83	0,31
Maximum	30,00	1,00	24,00	51,00	4,00	3,00	4,00	0,57
Minimum	7,00	0,00	20,00	9,00	0,00	0,00	0,00	0,00

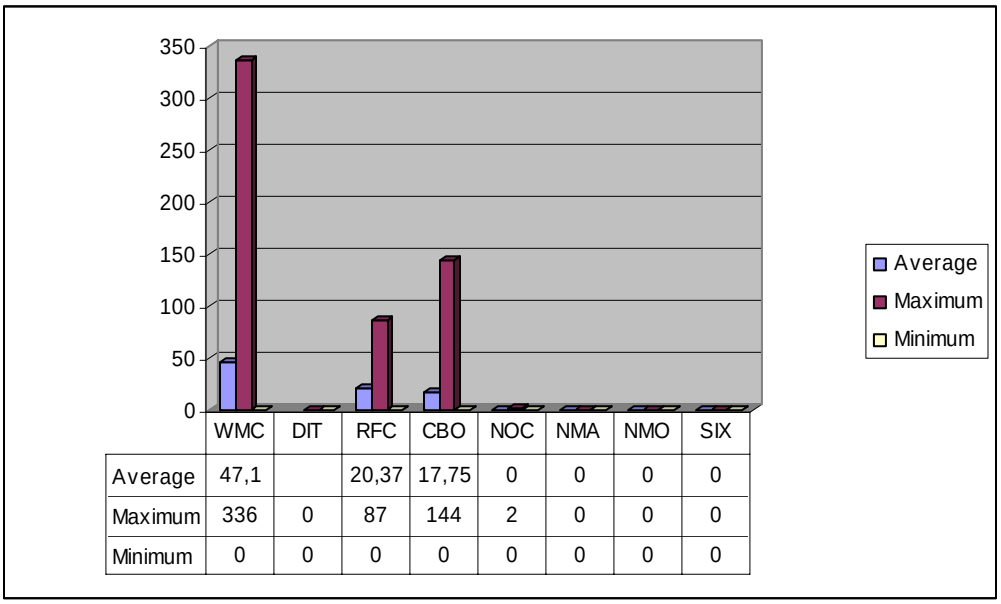
Figure 23 - Average, Maximum and Minimum Values of Thesis Metrics for State Pattern Case after Refactorings

3.4.2.5.2. Façade Pattern Implementation

Below refactored project summary is given:

Project Summary:

- Classes: 50
- Files: 99
- Functions: 965
- Lines: 27782



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	47,1		20,37	17,75	0	0	0	0
Maximum	336	0	87	144	2	0	0	0
Minimum	0	0	0	0	0	0	0	0

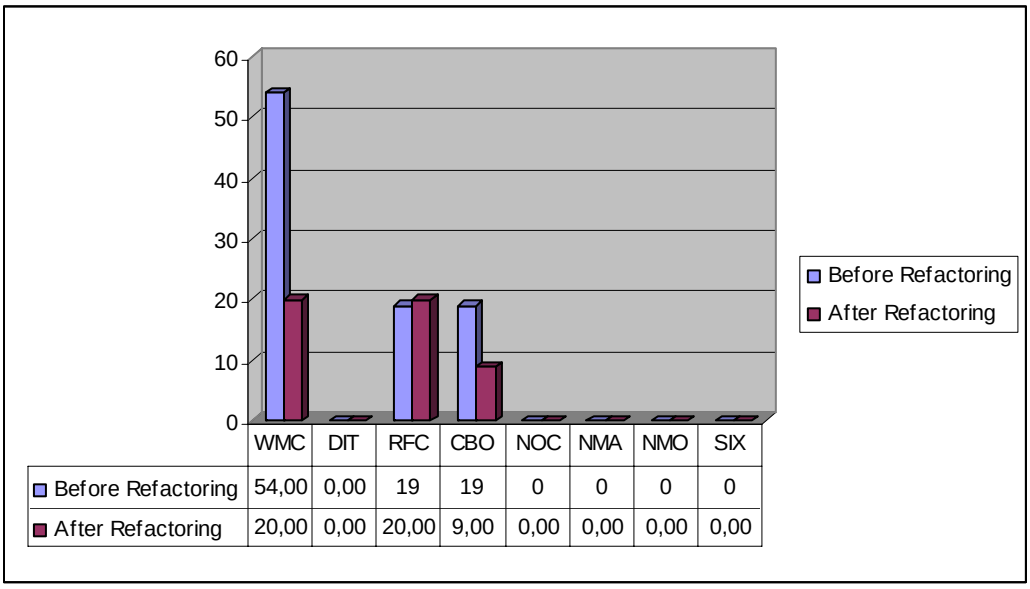
Figure 24 - Average, Maximum and Minimum Values of Thesis Metrics for Façade Pattern Case after Refactorings

Figure 24 displays the new metric values after Façade pattern implementation for Power-Grab project.

3.4.2.6. Phase Six: Comparisons and Interpretations

3.4.2.6.1. State Pattern Implementation

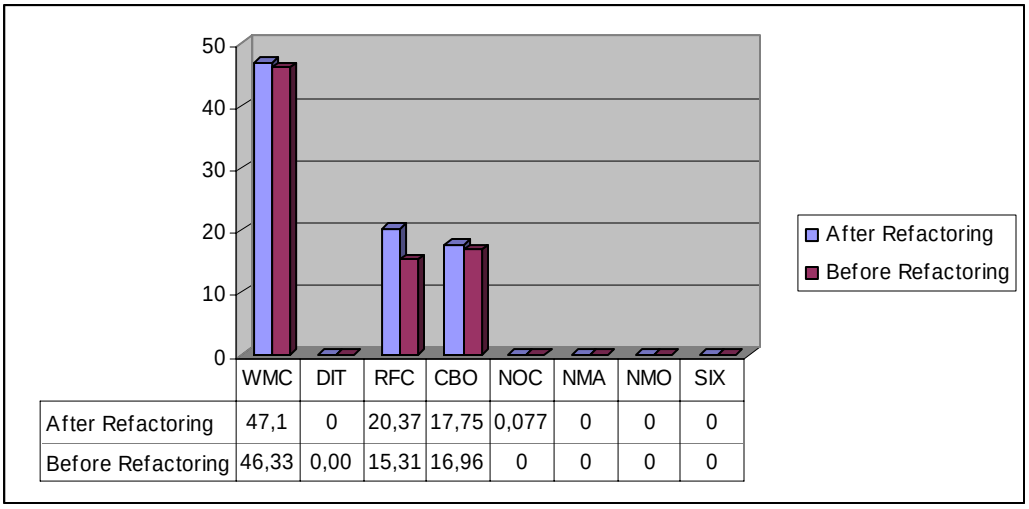
Figure 25 clearly indicates that State pattern refactoring for **CTaskAgent** class has significantly reduced WMC and CBO values. Since, this pattern has separated intrinsic state behavior from the class. Moreover, this refactoring has simplified **CTaskAgent** class and reduced testing, maintainability efforts. Consequently, State pattern reduces the error-proneness of a class, which has intrinsic state behavior.



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Before Refactoring	54,00	0,00	19	19	0	0	0	0
After Refactoring	20,00	0,00	20,00	9,00	0,00	0,00	0,00	0,00

Figure 25 - Metric Values Comparisons for CTaskAgent Class

3.4.2.6.2. Façade Pattern Implementation



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
After Refactoring	47,1	0	20,37	17,75	0,077	0	0	0
Before Refactoring	46,33	0,00	15,31	16,96	0	0	0	0

Figure 26 - Metric Values Comparisons for Façade Pattern Implementation

Façade pattern implementation unfortunately did not affected average metric values positively, as can be seen in Figure 26. There are many reasons for this result. Most important of them is that project source code was deteriorated and classes were highly-coupled with each other. Therefore, Façade pattern refactoring did not produce enough

improvements. Moreover, code complexity, time limitations and bad design of project source code made **ApplicationEngine** façade class highly coupled and complex.

3.4.3. Case 3: Mediator Pattern

Source codes investigated under cases 3 through 13 are synthetic. In other words, all the object relations and project code are synthetically arranged within the scope of this study to see the effect of relevant patterns on software error-proneness related metrics.

3.4.3.1. Phase One: OO Metrics Measurement of Overall Project

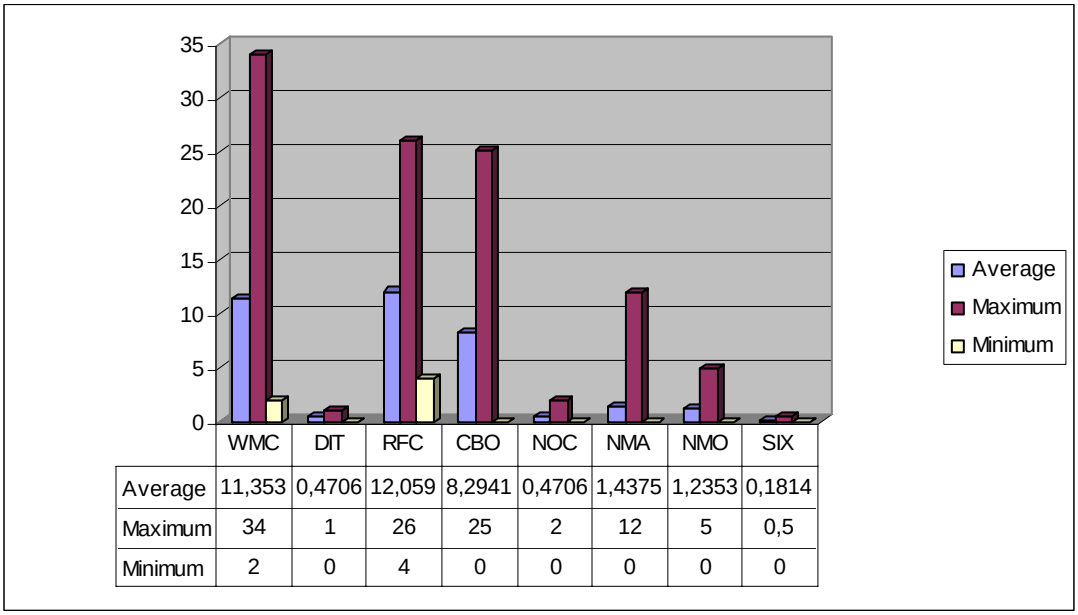
Unrefactored project summary is given below:

Project Summary:

- Classes: 17
- Files: 38
- Functions: 150
- Lines: 3310

3.4.3.2. Phase Two: Determining Error Prone Modules

As previously stated, in this phase, error-prone classes are detected. According to Figure 27 and 28, average and maximum values of CBO metric have relatively high values as expected. Moreover, in this case, the primary objective is to observe the fluctuation on the CBO metric after refactorings has been made.



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	11,353	0,4706	12,059	8,2941	0,4706	1,4375	1,2353	0,1814
Maximum	34	1	26	25	2	12	5	0,5
Minimum	2	0	4	0	0	0	0	0

Figure 27 - Average, Maximum and Minimum Values of Thesis Metrics for Mediator Pattern Case

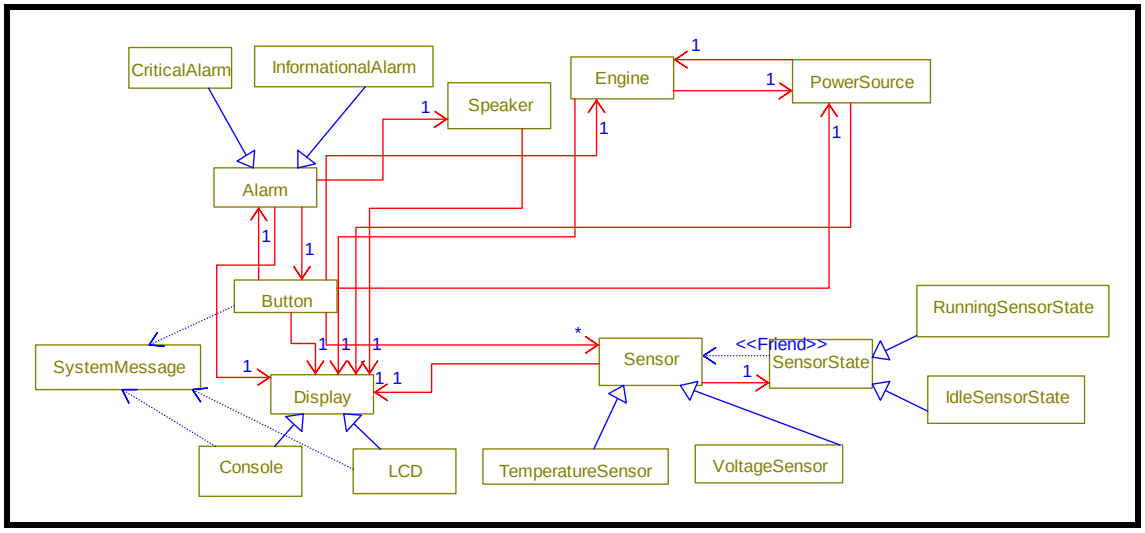


Figure 28 - Design of Mediator Case before Refactorings

3.4.3.3. Phase Three: Hatching Design Pattern(s)

As stated in 3.4.3.2, in this case, project classes has intense relations among themselves, which is detected after evaluating CBO metric values and remedy for this problem, defined by CBO, is to apply mediator pattern.

- **Mediator Pattern:**

This pattern helps developer to centralize functionality that doesn't belong in any one of the semantic classes. Also, implementation of this pattern provides semantic classes communicate over this mediator class. Therefore, this pattern assists developer to reduce coupling between many classes, where functionality is embedded. But, on the other hand, mediator class will have a high CBO, since it will know and regulates all other semantic classes.

Figure 29 shows the CBO values of all project classes before refactoring.

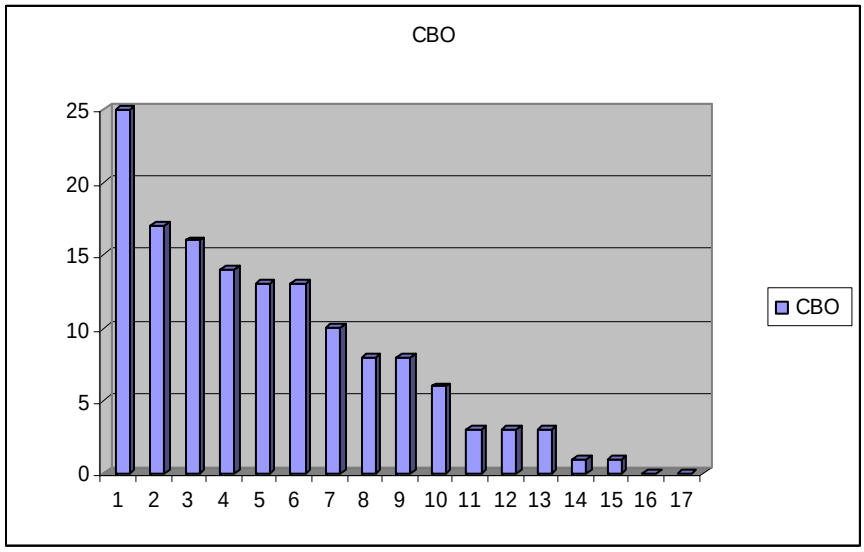


Figure 29 - CBO Values for Classes

3.4.3.4. Phase Four: Applying related pattern(s)

Implementation of Mediator pattern has introduced 2 interface classes and a concrete class that is the “Mediator” to the project, as Figure 30 shows. The classes that send events or messages to others have been abstracted to reduce the mediator coupling. Moreover, all one-to-one and one-to-many relations have been transferred to the

mediator, class. Finally, some of the semantic code has also been transferred to the mediator class.

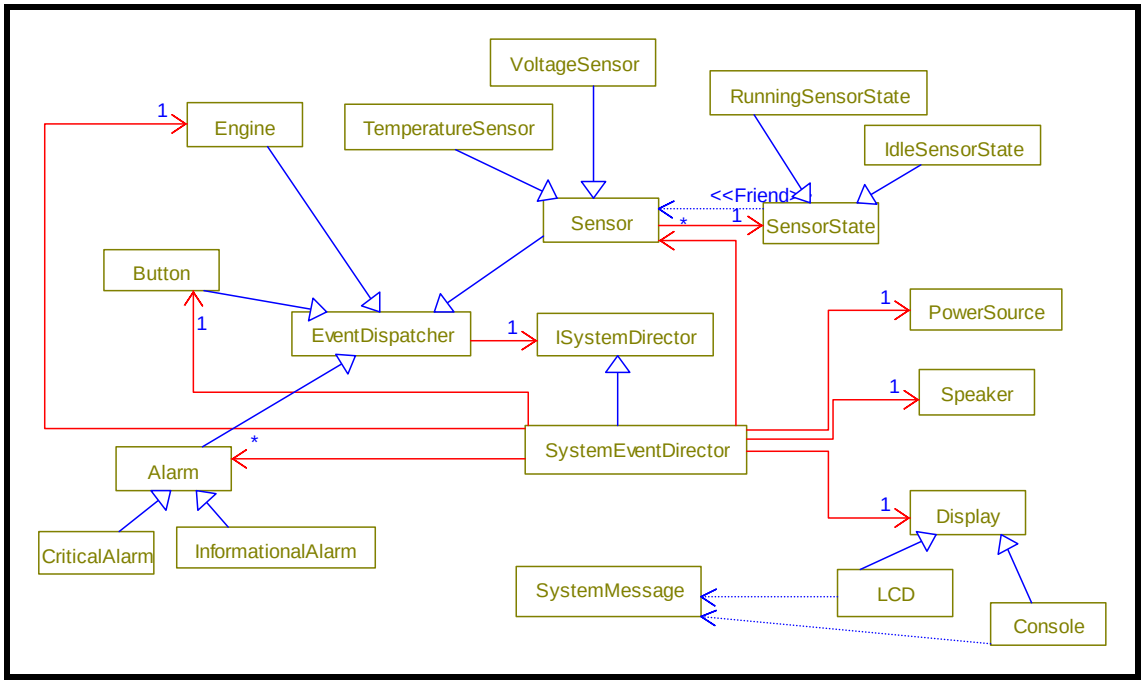


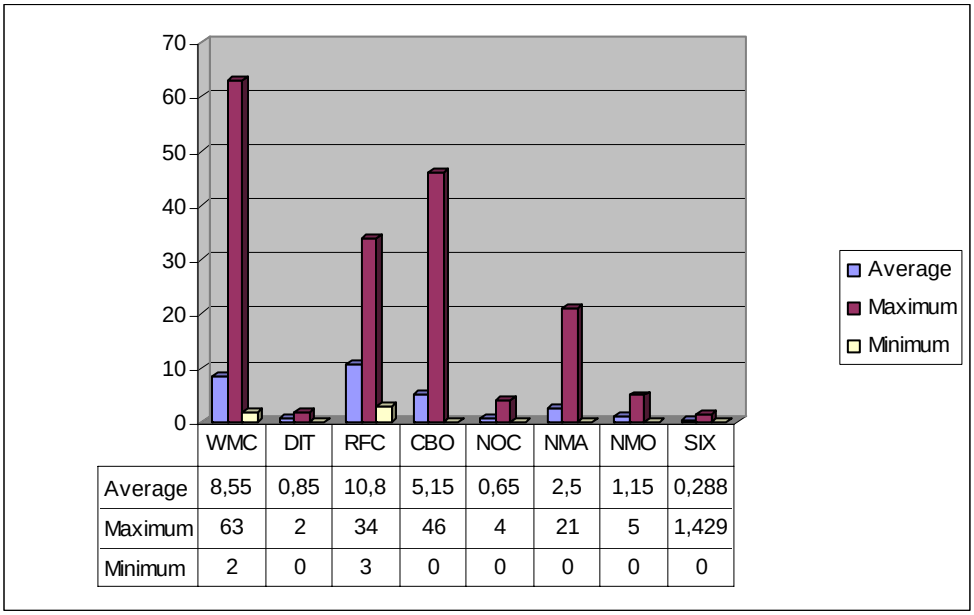
Figure 30 - Design of Mediator Case after Refactorings

3.4.3.5. Phase Five: Re-measuring software

Below refactored project summary is given:

Project Summary:

- Classes: 20
- Files: 44
- Functions: 130
- Lines: 3310



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	8,55	0,85	10,8	5,15	0,65	2,5	1,15	0,288
Maximum	63	2	34	46	4	21	5	1,429
Minimum	2	0	3	0	0	0	0	0

Figure 31 – Average, Maximum and Minimum Values of Thesis Metrics for Mediator Pattern Case after Refactoring

Figure 31 displays metric values after Mediator pattern refactoring.

3.4.3.6. Phase Six: Comparisons and Interpretations

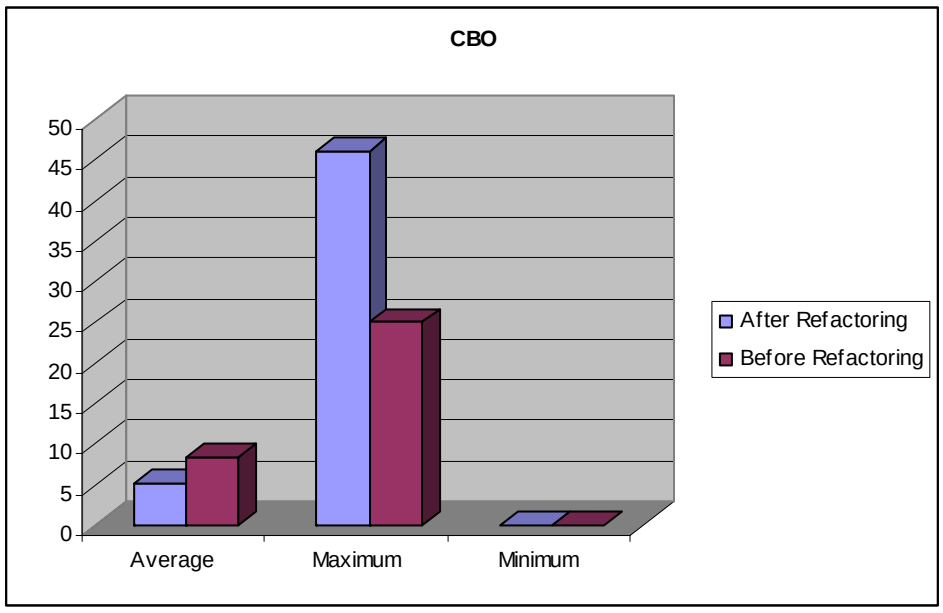


Figure 32 - Variations in Average, Maximum and Minimum Values of CBO

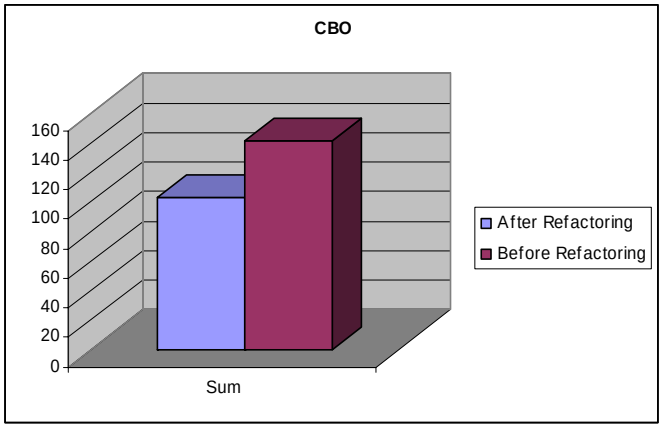


Figure 33 - Variation in Sum Value of CBO

Figure 32 and 33 clearly indicate that "Mediator" has reduced average CBO value approximately 38% and reduced sum value of CBO approximately 27%.

To sum up, "Mediator" moderates CBO to lower values as expected. But, as a side effect, this pattern introduces new mediator class that has generally high CBO value. This side effect, of course, is undesirable, but this pattern reduces total CBO value and concentrates it in one or more classes. This concentration helps developer to locate and solve the problem in some cases.

3.4.4. Case 4: Chain of Responsibility Pattern

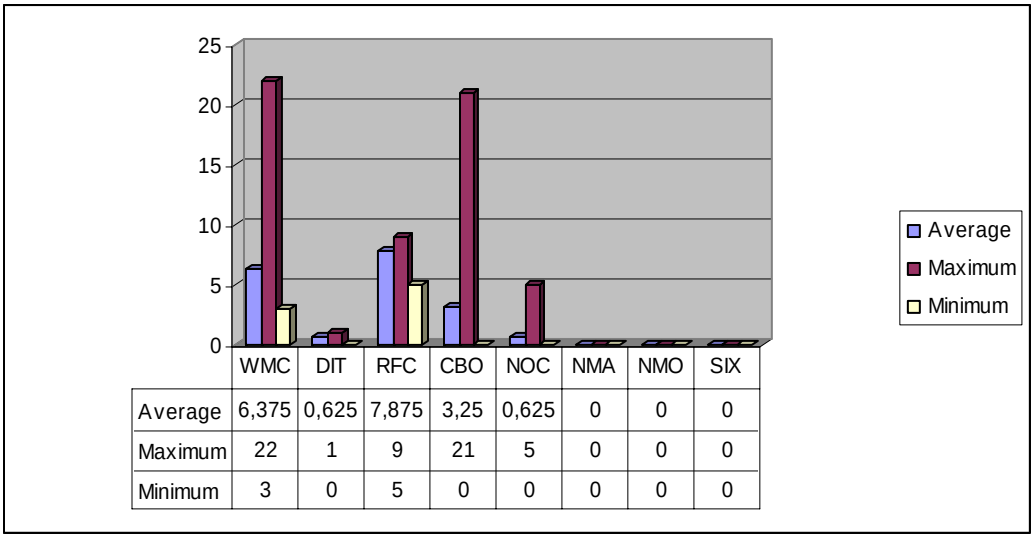
3.4.4.1. Phase One: OO Metrics Measurement of Overall Project

Unrefactored project summary is given below:

Project Summary:

- Classes: 8
- Files: 20
- Functions: 39

- Lines: 1190



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	6,375	0,625	7,875	3,25	0,625	0	0	0
Maximum	22	1	9	21	5	0	0	0
Minimum	3	0	5	0	0	0	0	0

Figure 34 - Average, Maximum and Minimum Values of Thesis Metrics for Chain of Responsibility Pattern Case before Refactorings

3.4.4.2. Phase Two: Determining Error Prone Modules

Figure 34 does not clearly indicate the possibility that project code embraces any error-prone classes; the main reason is that the code and relations are synthetically constructed. But, in this case main interest is on a specific class. Many of the Chain of Responsibility pattern refactorings will be made for observing and interpreting metric fluctuations in this class. In addition, Figure 35 shows metric values for this class.

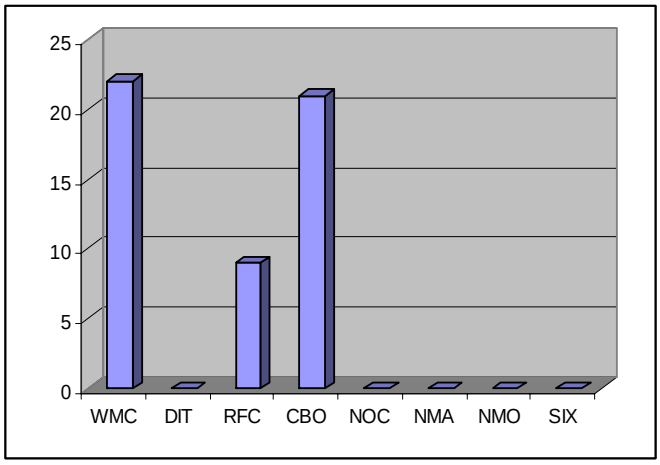


Figure 35 - Metric Values for Sample Class

3.4.4.3. Phase Three: Hatching Design Pattern(s)

Aforementioned class, in its implementation, includes a code segment given in Listing 1.

This code segments includes nested if clauses and pointer to class instances, which rises CBO and WMC values for the sample class, under investigation. Moreover, in the method given in Listing 1, semantically, there is an implicit chain; hoops of chain are bind by “for” loop, which is a mistake. Also, in “if” clauses, instances are checked for their eligibility to change the given argument, which is another mistake. All the instances should check their eligibility in their implementations.

Chain of Responsibility pattern solves the coding problems, just discussed. That is to say this pattern adds flexibility to the code and reduces coupling between classes. But, above code guarantees that given argument will be handled, on the other hand, Chain of Responsibility pattern does not promises a proper handling.

```
std::vector<Approver*>::const_iterator iter;
iter = itsApprover.begin();

for(;iter != itsApprover.end(); ++iter){


    Assistant* assistant = dynamic_cast<Assistant*>(*iter);
    if(assistant != NULL){
        if(request->getLevel() <= UNCLASSIFIED)
            request->setIsAllowed(true);
        continue;
    }

    Manager* manager = dynamic_cast<Manager*>(*iter);
    if(manager != NULL){
        if(request->getLevel() <= COMPANY_SPECIFIC)
            request->setIsAllowed(true);
        continue;
    }

    Director* director = dynamic_cast<Director*>(*iter);
    if(director != NULL){
        if(request->getLevel() <= LIMITED_ACCESS)
            request->setIsAllowed(true);
        continue;
    }


    VicePresident* vice_president = dynamic_cast<VicePresident*>(*iter);
    if(vice_president  != NULL){
        if(request->getLevel() <= SECRET)
            request->setIsAllowed(true);
        continue;
    }

    CEO* president = dynamic_cast<CEO*>(*iter);
    if(president  != NULL){
        if(request->getLevel() <= TOP_SECRET)
            request->setIsAllowed(true);
        continue;
    }
}
```

Listing 1 - Code Segment before Refactorings for Chain of Responsibility Case

3.4.4.4. Phase Four: Applying related pattern(s)

Application of Chain of Responsibility pattern does not introduce new classes to the project, if the all argument handlers have a common interface or an abstract class, but requires code and relation (association, aggregation, and est.) changes for classes where discussed problems occur. For example, after the implementation of Chain of Responsibility pattern, code segment, given in Listing 1, transformed into the code given in Listing 2.

```
assert(itsApprover != NULL);
itsApprover->approve(request);
```

Listing 2 - Code Segment after Refactorings for Chain of Responsibility Case

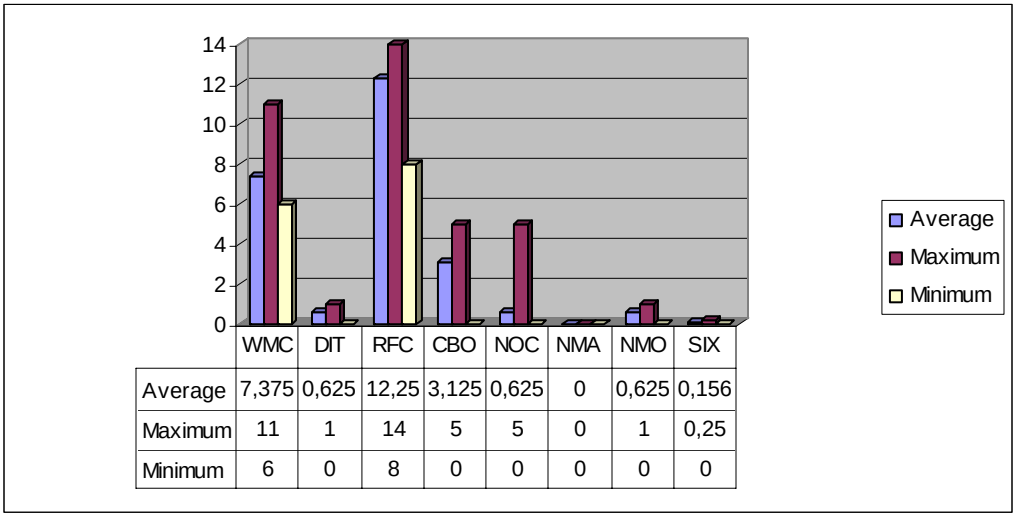
Another problem in the application of this pattern is the construction, modification, destruction tasks of the chain. The basic solution is to use client class to perform these tasks. But, if many clients use the same chain, another class should be added to the project to perform these tasks, and stabilize the internal structure of chain.

3.4.4.5. Phase Five: Re-measuring software

Below refactored project summary is given:

Project Summary:

- Classes: 8
- Files: 20
- Functions: 47
- Lines: 1294



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	7,375	0,625	12,25	3,125	0,625	0	0,625	0,156
Maximum	11	1	14	5	5	0	1	0,25
Minimum	6	0	8	0	0	0	0	0

Figure 36 - Average, Maximum and Minimum Values of Thesis Metrics for Chain of Responsibility Case after Refactorings

Figure 36 displays new metric values after refactorings for Chain of Responsibility case.

3.4.4.6. Phase Six: Comparisons and Interpretations

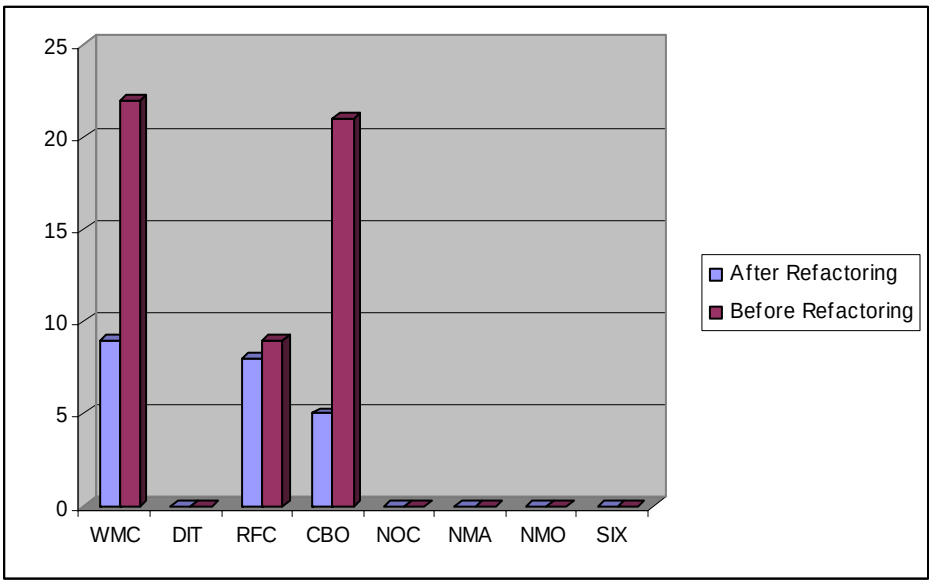


Figure 37 - Metric Values Comparisons for Sample Class

Figure 37 clearly shows that after the application of Chain of Responsibility pattern, WMC and CBO values for sample class has 59% and 76% decreased respectively. Also, there is a slight decrease in RFC value.

This graphic shows that this pattern helps designer to decrease high coupling and class method complexity, if problem context is similar or same with the problem context defined by Chain of Responsibility pattern.

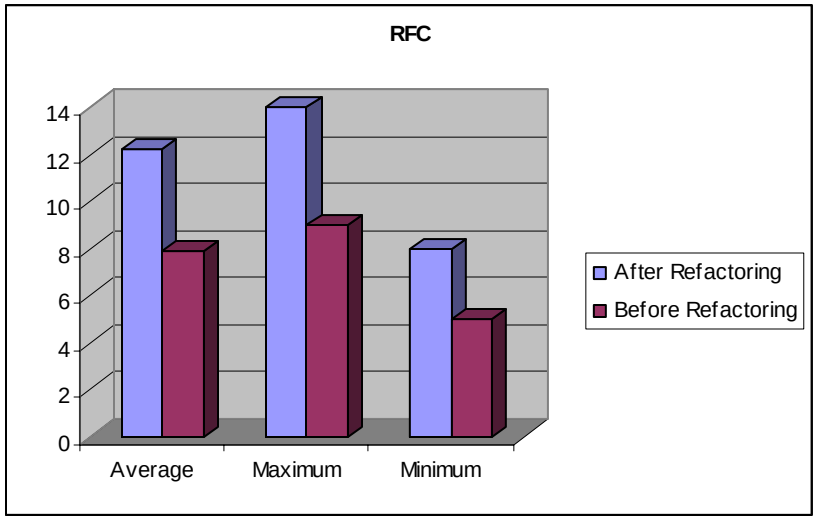


Figure 38 – Average, Maximum and Minimum Values for RFC

On the other hand, Figure 38 shows that this pattern has significantly increased average RFC value. As stated, in chapter 2, this metric mainly indicates test and debug complexity for a class. This outcome mainly results from new relations and methods that are created to realize Chain of Responsibility pattern.

To sum up, this pattern is a good remedy for high WMC and CBO values. But, as a primary side effect, it increases RFC values in the set off classes, which are effected by this pattern.

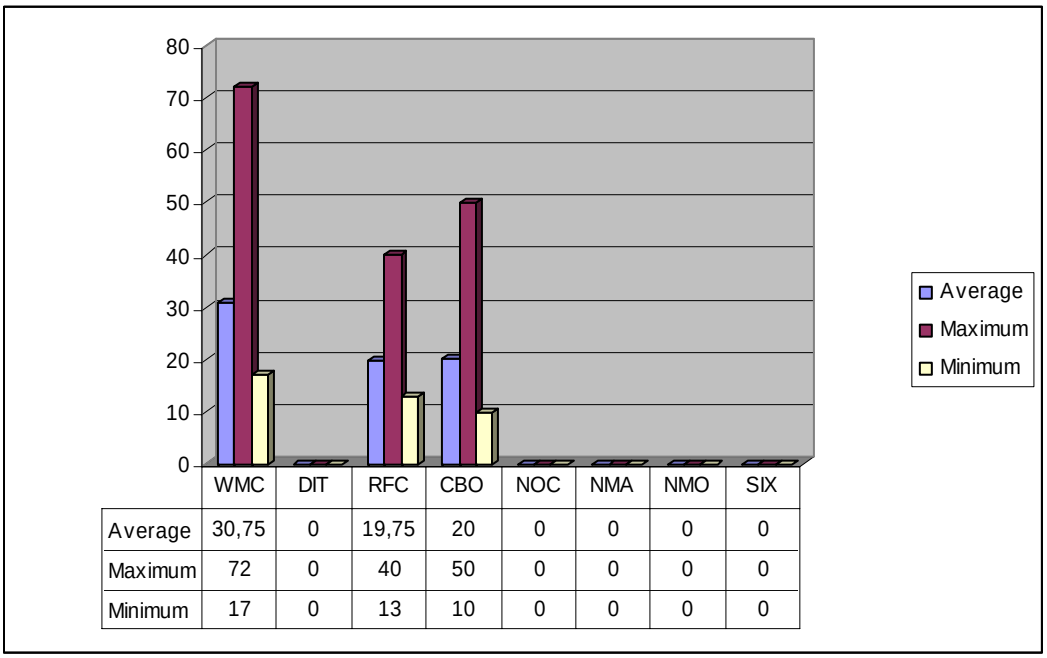
3.4.5. Case 5: Composite Pattern

3.4.5.1. Phase One: OO Metrics Measurement of Overall Project

Unrefactored project summary is given below:

Project Summary:

- Classes: 4
- Files: 12
- Functions: 53
- Lines: 1018



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	30,75	0	19,75	20	0	0	0	0
Maximum	72	0	40	50	0	0	0	0
Minimum	17	0	13	10	0	0	0	0

Figure 39 - Average, Maximum and Minimum Values of Thesis Metrics for Composite Pattern Case before Refactorings

3.4.5.2. Phase Two: Determining Error Prone Modules

As Figure 39 clearly shows that, the project classes are complex and highly coupled. As expected, since design of this project was implemented to understand and to show the

effects of Composite pattern, after refactorings. Figure 40 figure shows classes and the relations among them.

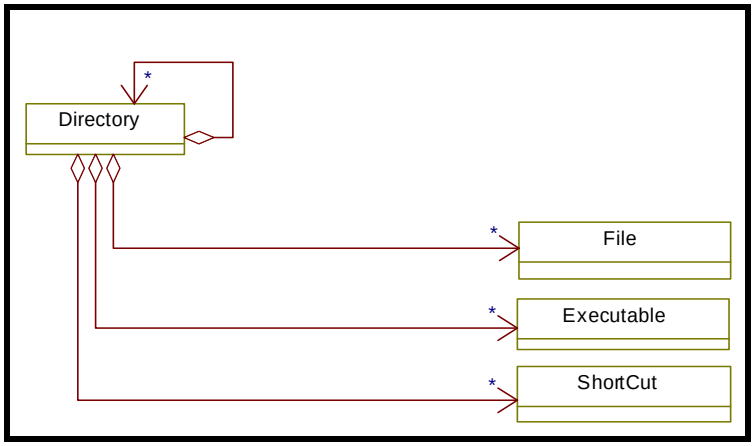


Figure 40 - Design of Composite Pattern Case before Refactorings

In this case, “Directory” class has one to many relations with the other class and uses these links in its methods. This structure makes this class complex and highly coupled. As a result, “Directory” class has a rotten design. Another point was omitted by the designer that all the classes could have inherited from a common interface. Since, they represent nodes in the file structure of an operating system.

3.4.5.3. Phase Three: Hatching Design Pattern(s)

Discussed design problems overlap with applicability context of the Composite pattern. Therefore, this pattern is remedy for high WMC and CBO values in this case.

Composite pattern has below applicability characteristics [GOF98]:

- Clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
- Represent part-whole hierarchies of objects

3.4.5.4. Phase Four: Applying related pattern(s)

Composite pattern application has introduced a new interface class and has eliminated three one-to-many relations. That design structure alteration reduced CBO values for **Directory** class. However, after pattern implementation, these four classes have inheritance relations. In detail, these relations have increased inheritance related metrics, like DIT. Figure 41 shows refactored design for case 5.

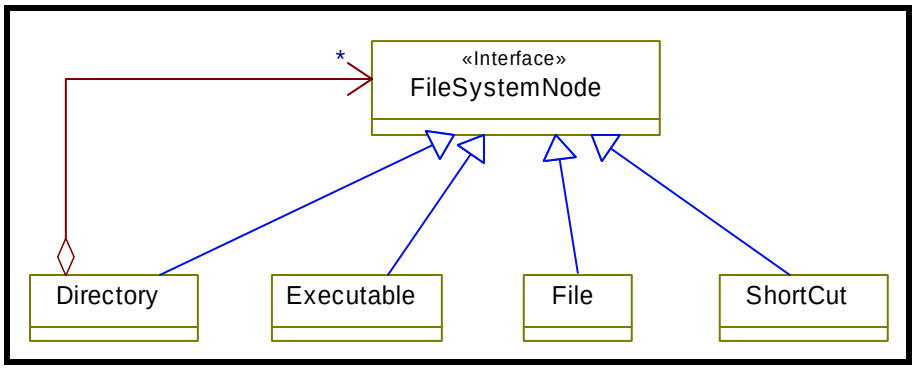


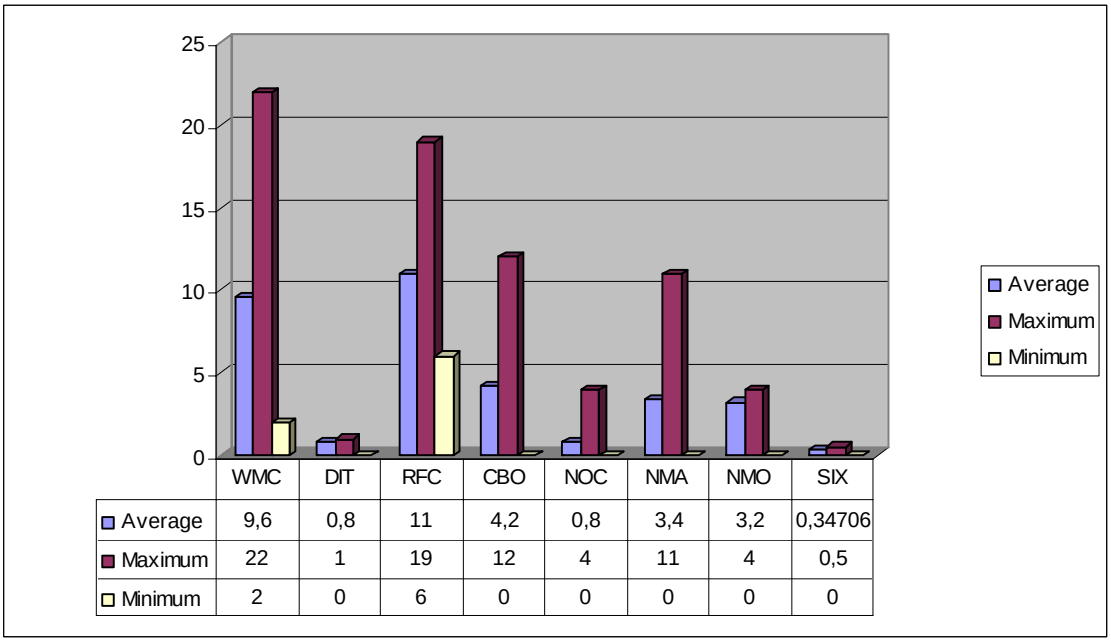
Figure 41 - Design of Composite Pattern Case after Refactorings

3.4.5.5. Phase Five: Re-measuring software

Below refactored project summary is given:

Project Summary:

- Classes: 5
- Files: 14
- Functions: 44
- Lines: 970

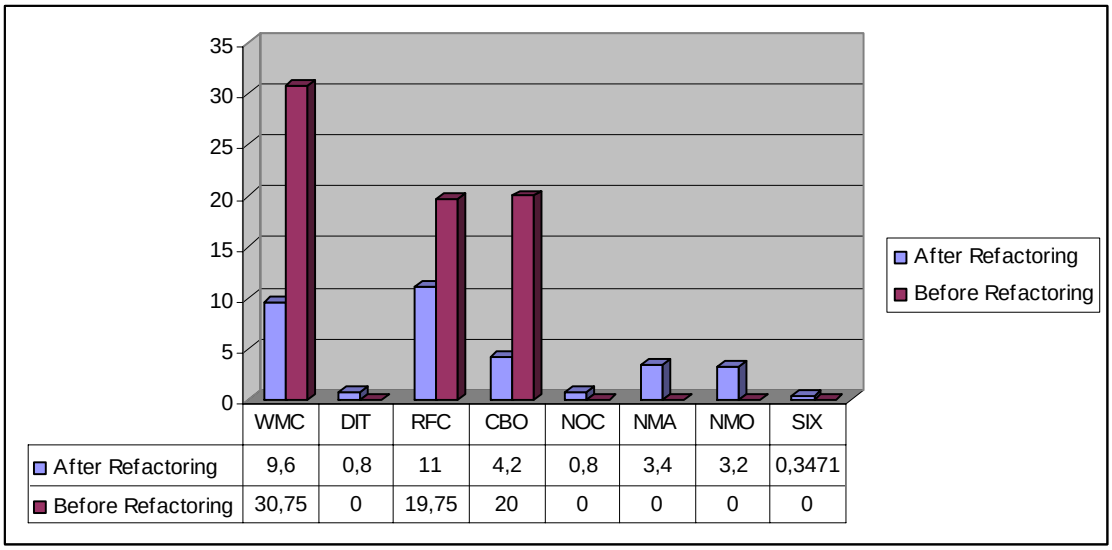


	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	9,6	0,8	11	4,2	0,8	3,4	3,2	0,34706
Maximum	22	1	19	12	4	11	4	0,5
Minimum	2	0	6	0	0	0	0	0

Figure 42 - Average, Maximum and Minimum Values of Thesis Metrics for Composite Pattern Case after Refactorings

Figure 42 shows new metric values after refactorings.

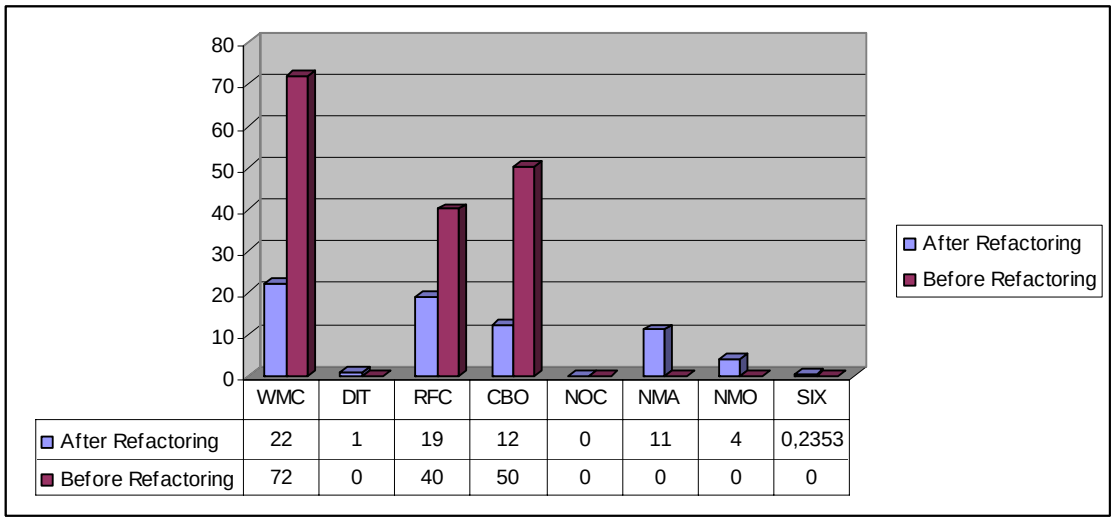
3.4.5.6. Phase Six: Comparisons and Interpretations



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
After Refactoring	9,6	0,8	11	4,2	0,8	3,4	3,2	0,3471
Before Refactoring	30,75	0	19,75	20	0	0	0	0

Figure 43 - Average Value Comparisons of Thesis Metrics for Composite Pattern Case

Primary objective of this synthetic code is to empirically show the effect of CBO fluctuation after refactorings. Figure 43 clearly shows that application of **Composite pattern** has reduced average CBO value approximately 79%. Moreover, **Directory** class had higher CBO values than others, and below figure shows CBO and other metric differences after and before refactorings in **Directory** class.



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
After Refactoring	22	1	19	12	0	11	4	0,2353
Before Refactoring	72	0	40	50	0	0	0	0

Figure 44 - Metric Comparisons of Directory class after and before Refactorings

As can be seen from Figure 44, main effect of Composite pattern is on **Directory** class. Since, it was misinterpreted and designed poorly. WMC, RFC, and CBO values of this class have decreased approximately 69%, 53%, and 76% respectively. Composite pattern has a significant impact on complexity related metrics. However, as a side effect, inheritance related metrics has increased slightly. But, when compared with the reduction in WMC, RFC, and CBO, increase in other metrics is inconsiderable.

3.4.6. Case 6: Strategy Pattern

Strategy pattern has two aspects for this thesis. Firstly, this pattern eliminates conditional statements. Secondly, Strategy pattern can be used as an alternative to subclassing. Therefore, in this case, mentioned aspects have been evaluated separately.

3.4.6.1. Phase One: OO Metrics Measurements of Overall Projects

3.4.6.1.1. Conditional Statements Problem Aspect

Unrefactored project summary is given below:

Project Summary:

- Classes: 1
- Files: 2
- Functions: 14
- Lines: 402

As it can be seen from project summary, this aspect has one class and Figure 45 shows its thesis metric values.

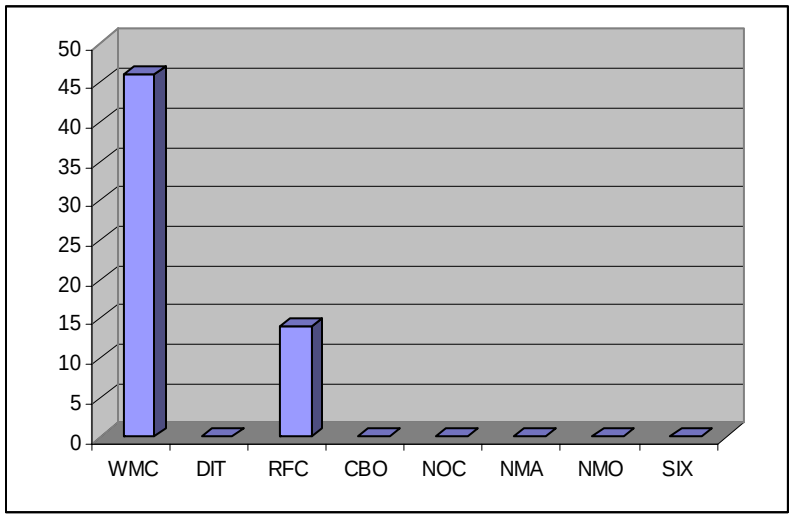


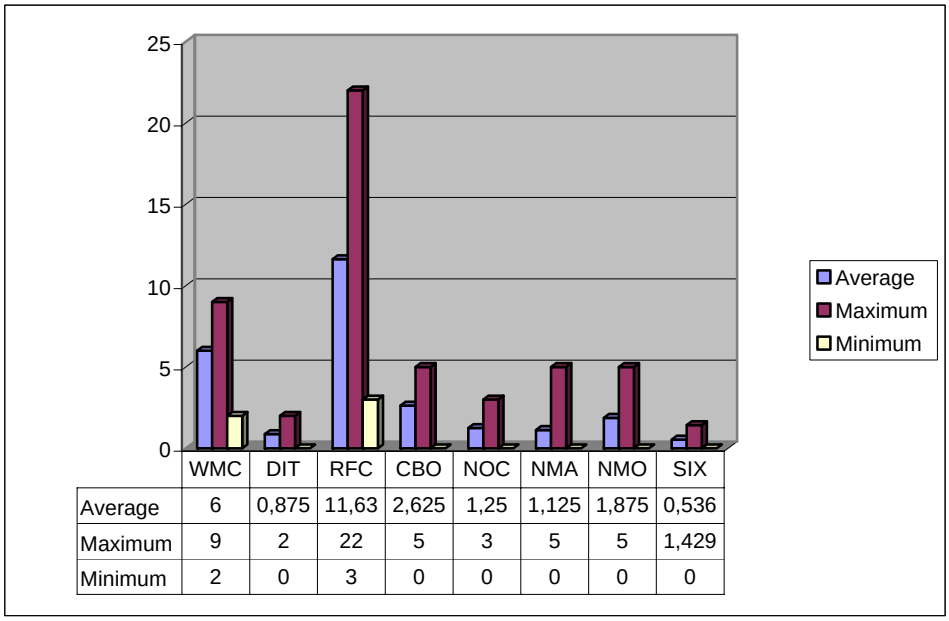
Figure 45 - Thesis Metrics for Strategy Pattern Case (Conditional Statements Aspect)

3.4.6.1.2. Subclassing Alternative Aspect

Unrefactored project summary is given below:

Project Summary:

- Classes: 8
- Files: 20
- Functions: 42
- Lines: 1179



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	6	0,875	11,63	2,625	1,25	1,125	1,875	0,536
Maximum	9	2	22	5	3	5	5	1,429
Minimum	2	0	3	0	0	0	0	0

Figure 46 - Average, Maximum and Minimum Values of Thesis Metrics for Strategy Pattern (Subclassing Alternative Aspect)

Figure 46 displays average, maximum, and minimum values of thesis metrics for Strategy pattern in subclassing alternative aspects before refactorings.

3.4.6.2. Phase Two: Determining Error Prone Modules

3.4.6.2.1. Conditional Statements Problem Aspect

Synthetic code of this project aspect has only one class, and this class was designed to understand and observe the effect on Strategy pattern on this class.

3.4.6.2.2. Subclassing Alternative Aspect

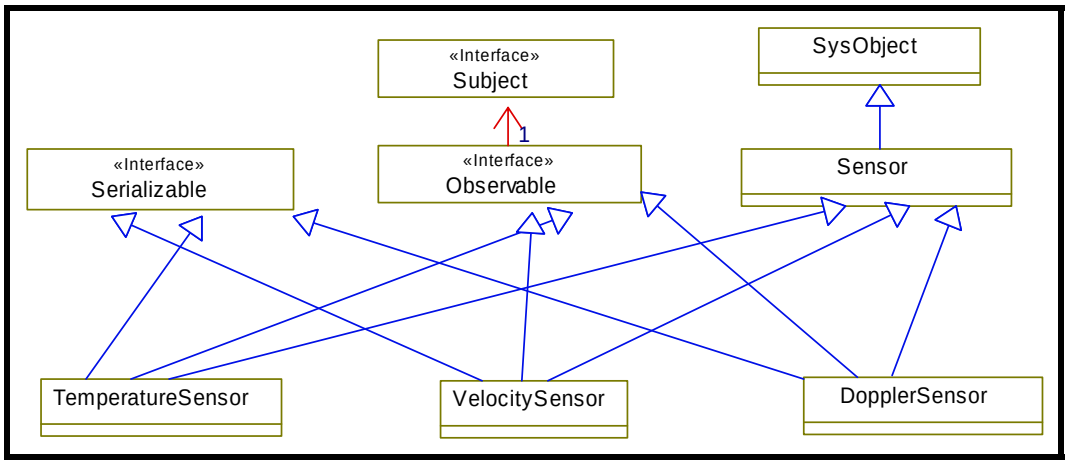


Figure 47 - Design of Subclassing Alternative Aspect before Refactorings

As Figure 47 displays, **TemperatureSensor**, **VelocitySensor**, and **DopplerSensor** classes suffer from multiple inheritance. The outcome of this design is that these classes have relatively high RFC, CBO, NMO values. To sum up, these three classes pop up as error-prone modules.

Below table displays metric values of previously mentioned classes.

Table 1 - Metric Values of Sensor Classes

	DopplerSensor	TemperatureSensor	VelocitySensor
WMC	9	9	9
DIT	2	2	2
RFC	22	22	22
CBO	5	5	5
NOC	0	0	0

NMA	0	0	0
NMO	5	5	5
SIX	0	0	0

3.4.6.3. Phase Three: Hatching Design Pattern(s)

3.4.6.3.1. Conditional Statements Problem Aspect

In this aspect, project code contains one class and this class includes different methods and algorithms for sorting. To visualize class behavior, code segment in Listing 3 is given.

```
std::vector<int> value;
switch(sortType){
    case SHELL:
        value = shellSort(array);
        break;
    case QUICK:
        value = quickSort(array);
        break;
    case MERGE:
        value = mergeSort(array);
        break;
    case BUBBLE:
        value = bubbleSort(array);
        break;
    case HEAP:
        value = heapSort(array);
        break;
    default:
        break;
}

return value;
```

Listing 3 - Code Segment before Refactorings for Strategy Pattern Case

As we know that the Strategy pattern offers an alternative to conditional statements for selecting desired behavior [GOF98]. Therefore, Strategy pattern can abolish

conditional statements and embedded algorithms from the implementation of error-prone class.

3.4.6.3.2. Subclassing Alternative Aspect

Inheritance offers another way to support a variety of algorithms or behaviors. You can subclass a Context class directly to give it different behaviors. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend [GOF98]. The importance of previous sentence is that, in this aspect, Strategy pattern can transform serializing and observing interfaces into strategies. This design modification or refactoring can decrease inheritance related metrics, on the other hand, it will surely increase coupling related metrics for some classes. In the succeeding parts, trade-off among them will be discussed.

3.4.6.4. Phase Four: Applying related pattern(s)

3.4.6.4.1. Conditional Statements Problem Aspect

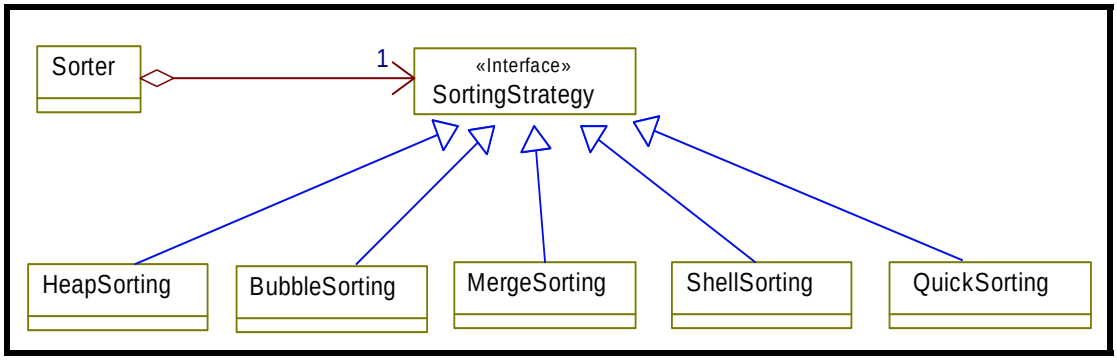


Figure 48 - Design of Strategy Pattern for Conditional Statements Aspect after Refactorings

Application of Strategy pattern, in this aspect, introduced six new classes to the project, as Figure 48 indicates. All the algorithms have been abstracted by **SortingStrategy** class. In addition, sorting algorithms have been transformed into concrete classes. To

realize that pattern, Sorter class is now constructed with its selected strategy, which introduced flexibility and extensibility to the software. However, a one-to-one shared aggregation has been added between **Sorter** and its strategy. That association increases coupling and complexity.

3.4.6.4.2. Subclassing Alternative Aspect

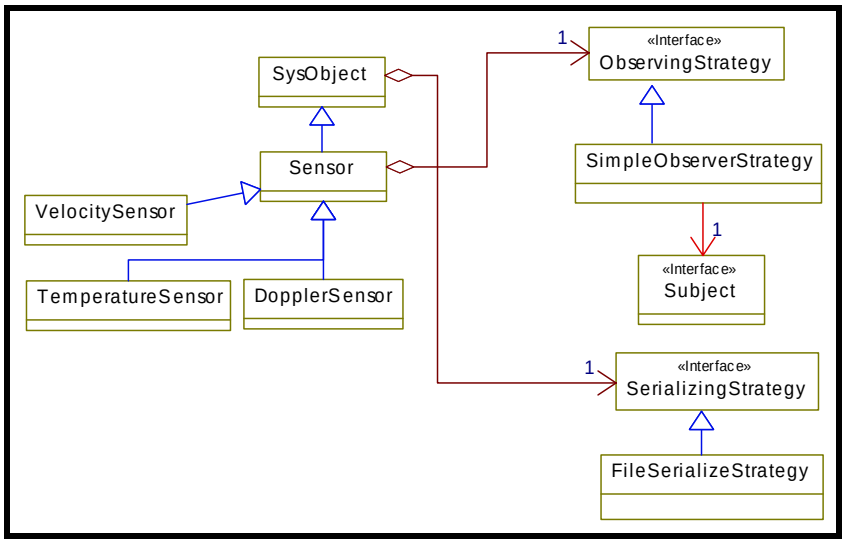


Figure 49 - Design of Strategy Pattern for Subclassing Alternative Aspect after Refactorings

Observable and **Serializable** interfaces of sensor concrete classes have been transformed into abstract strategy classes. As a consequence, two new concrete strategy classes, **SimpleObserverStrategy** and **FileSerializeStrategy**, have been added to software.

In the design, shown in Figure 49, inheritance relations changed into one-to-one aggregations. This structure modification has solved multiple inheritance problems, and inheritance behaviors have been transferred to strategies. But, this time client of these sensor classes should have to know proper strategies. This side effect increases client side coupling. Another drawback is the communication overhead, which means

separation of context and behavior causes process delay due messaging between objects.

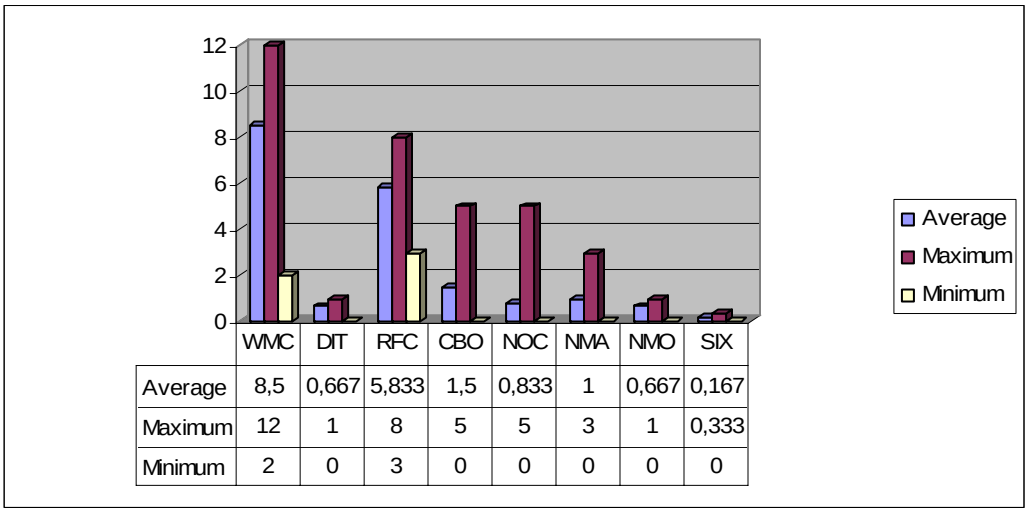
3.4.6.5. Phase Five: Re-measuring software

3.4.6.5.1. Conditional Statements Problem Aspect

Refactored project summary is given below:

Project Summary:

- Classes: 7
- Files: 16
- Functions: 29
- Lines: 1064



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	8,5	0,667	5,833	1,5	0,833	1	0,667	0,167
Maximum	12	1	8	5	5	3	1	0,333
Minimum	2	0	3	0	0	0	0	0

Figure 50 - Average, Maximum and Minimum Values of Thesis Metrics for Conditional Statements Problem Aspect

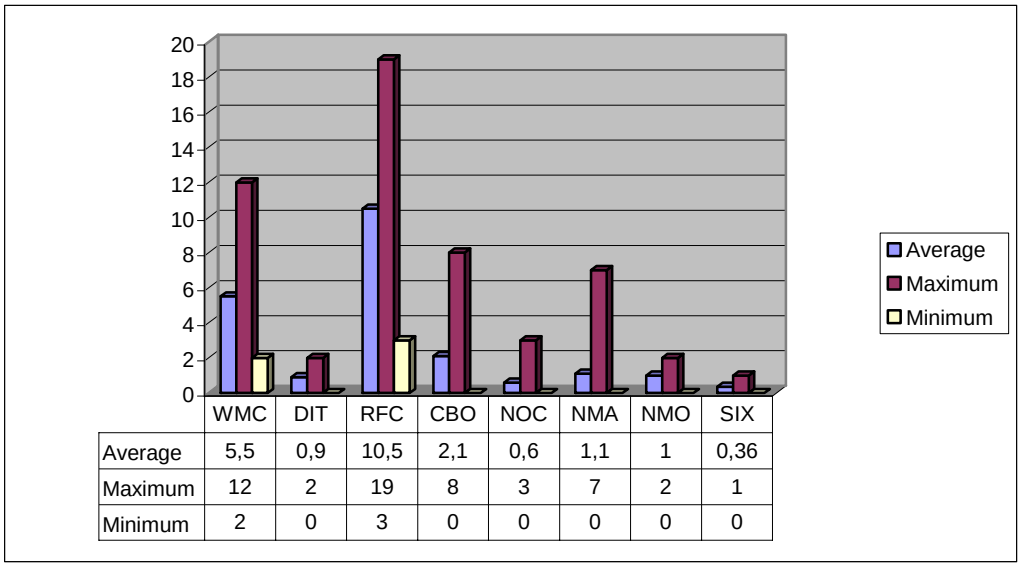
Figure 50 shows new metric values after refactorings for conditional statements problem aspect.

3.4.6.5.2. Subclassing Alternative Aspect

Refactored project summary is given below:

Project Summary:

- Classes: 10
- Files: 22
- Functions: 47
- Lines: 1379



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	5,5	0,9	10,5	2,1	0,6	1,1	1	0,36
Maximum	12	2	19	8	3	7	2	1
Minimum	2	0	3	0	0	0	0	0

Figure 51 - Average, Maximum and Minimum Values of Thesis Metrics for Subclassing Alternative Aspect

Average, maximum and minimum values of thesis metrics after refactorings for subclassing alternative aspect have been given in Figure 51.

3.4.6.6. Phase Six: Comparisons and Interpretations

3.4.6.6.1. Conditional Statements Problem Aspect

As stated before, in this case, there was only one class, and its metric values have been given in Figure 45. After applying Strategy pattern, design structure significantly changed. As a result, all the complexity, concentrated on "Sorter" class, has been distributed over strategy classes. But sum values of WMC, CBO, RFC and other inheritance related metrics has increased.

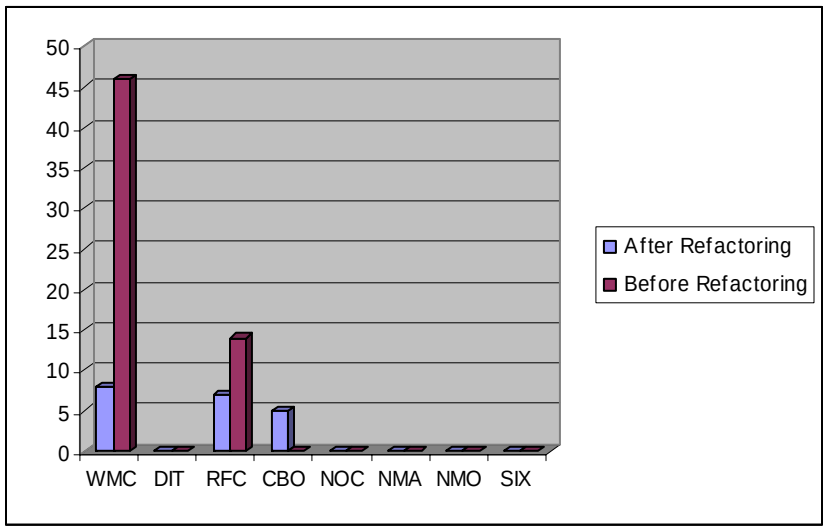


Figure 52 - Metric Comparisons of Sorter class after and before refactorings

Moreover, Figure 52 graphically reveals that application of Strategy pattern significantly diminishes WMC and RFC values. But, as a side effect, this design alteration increases CBO metric value. Since, implementation of this pattern imperatively introduces aggregation relation between strategy and context.

3.4.6.6.2. Subclassing Alternative Aspect

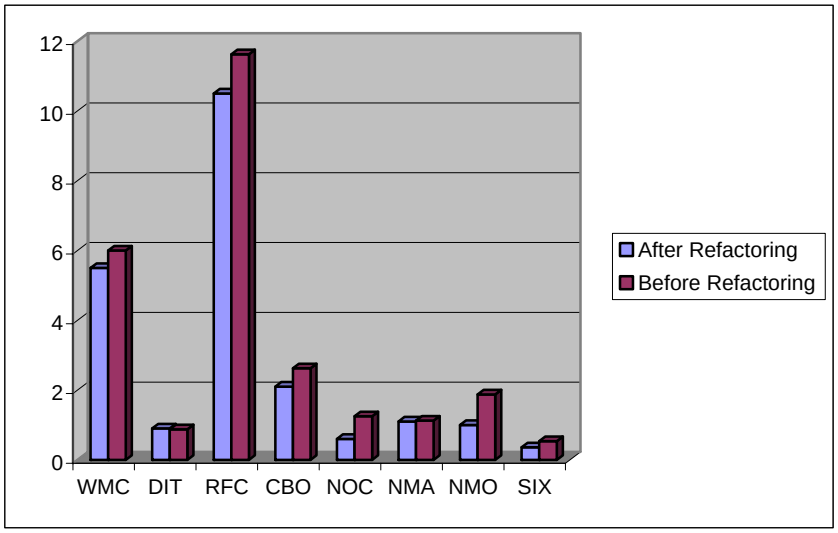


Figure 53 - Comparison of Average Values of Thesis Metrics for Subclassing Alternative Aspect in Strategy Pattern Case

Figure 53 points out that all the metrics have been affected by the application of Strategy pattern. Percentage of average metric value alterations for each metric are given in the below figure.

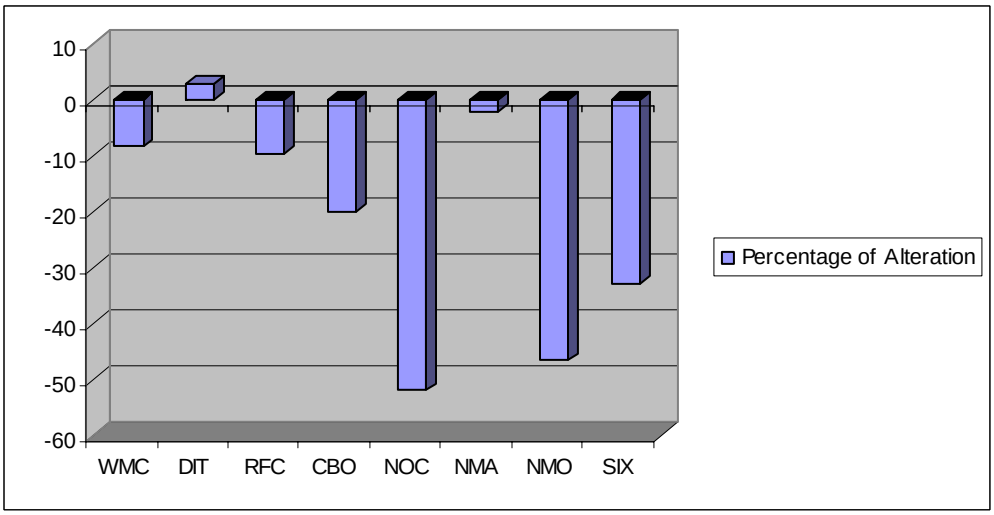


Figure 54 - Average Metric Value Alterations for Subclassing Alternative Aspect in Strategy Pattern Case

Figure 54 indicates that this pattern has a considerable affect on NOC metric value as expected. Since, it was embedded to the design to eliminate excessive inheritance relations. This result empirically shows that this pattern is successful for that purpose. In addition, NMO that is another inheritance related metric, has reduced significantly. As stated in chapter 2, a large number of overridden methods indicate a design problem [LK94]. This pattern is capable of partially eliminating this bad design smell and reducing the probability of error-prone classes among software. Another important result is 20% drop in average CBO metric value. This fluctuation shows that complexity of software has reduced on average.

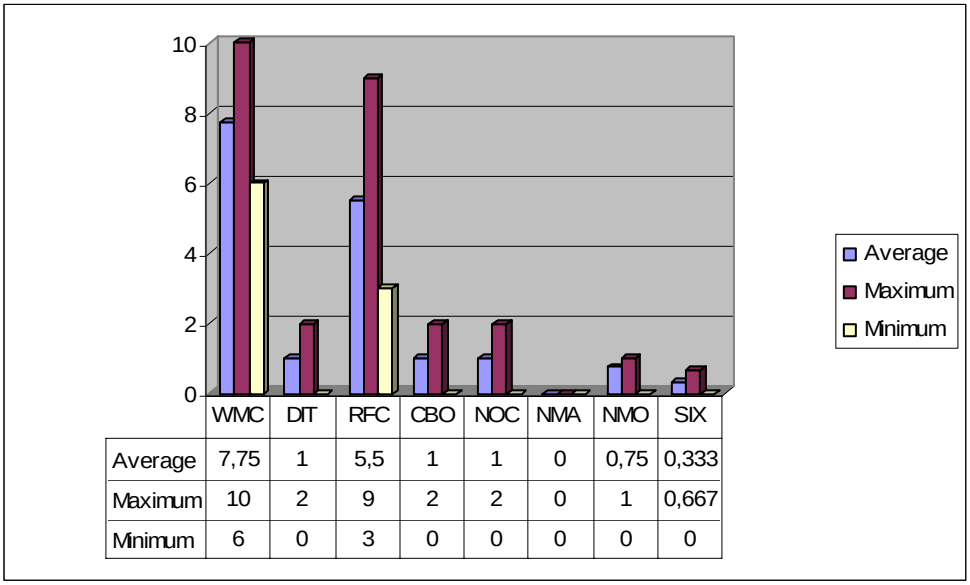
3.4.7. Case 7: Template Method Pattern

3.4.7.1. Phase One: OO Metrics Measurement of Overall Project

Unrefactored project summary is given below:

Project Summary:

- Classes: 4
- Files: 12
- Functions: 13
- Lines: 651



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	7,75	1	5,5	1	1	0	0,75	0,333
Maximum	10	2	9	2	2	0	1	0,667
Minimum	6	0	3	0	0	0	0	0

Figure 55 - Average, Maximum and Minimum Values of Thesis Metrics for Template Method Pattern

3.4.7.2. Phase Two: Determining Error Prone Modules

Synthetic code of this project has four classes, and these classes were designed to understand and observe the effect on Template Method pattern on their metric values.

Figure 55 may not display any indicators for error-prone classes, but source code of the project contains design faults. There are four classes, shown in Figure 56, and they implement four different variants of bubble sorting algorithm. The problem is that these classes implement invariant and variant parts of the algorithm in their own implementation code. This problem is the motivation for the application of Template Method pattern.

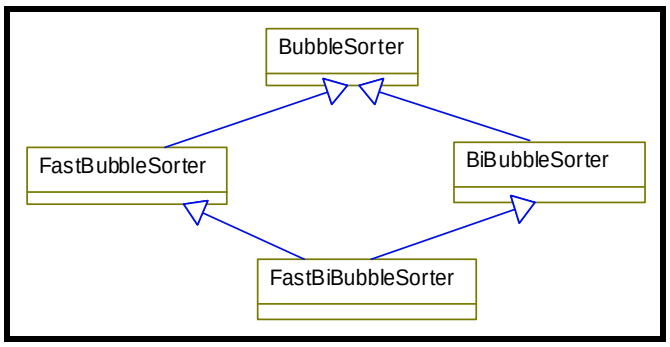


Figure 56 - Design of Template Method Pattern Case before Refactoring

3.4.7.3. Phase Three: Hatching Design Pattern(s)

Template Method pattern is the generalized solution for the problem, described in 3.4.7.2. Moreover, [GOF98] states that one of the applicability of this pattern is that designer should implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary. Therefore, Template Method perfectly suits for refactoring of this project.

3.4.7.4. Phase Four: Applying related pattern(s)

Template Method implementation does not introduce new classes, but brings in new methods. These methods are the variant and invariant parts of the bubble sorting algorithm for each class. Details of the variant and invariant parts of the aforementioned algorithm are out of the scope.

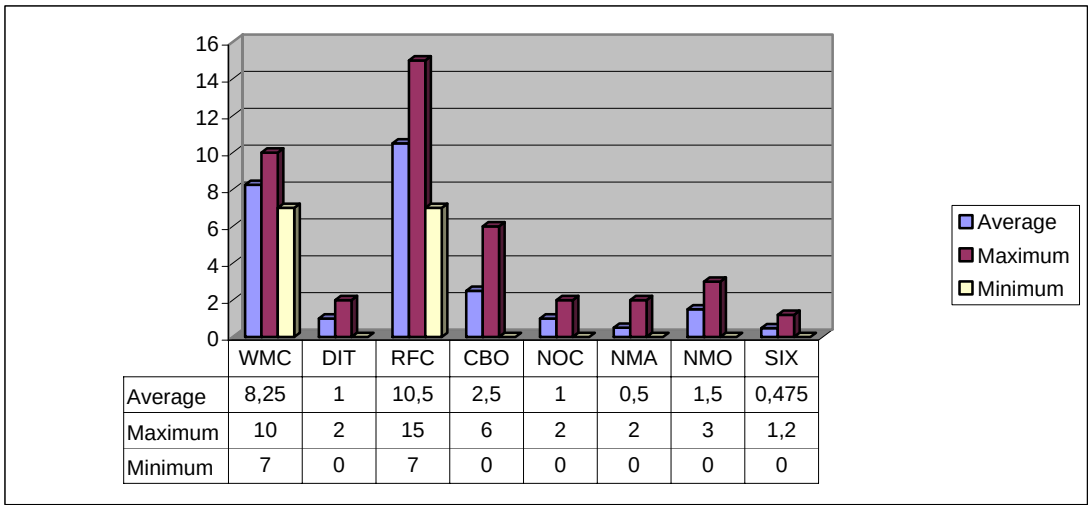
As stated, this pattern does not introduce new classes and also new class relations. This feature implies that coupling and inheritance related metrics should not alter after refactorings.

3.4.7.5. Phase Five: Re-measuring software

Refactored project summary is given below:

Project Summary:

- Classes: 4
- Files: 12
- Functions: 22
- Lines: 715



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	8,25	1	10,5	2,5	1	0,5	1,5	0,475
Maximum	10	2	15	6	2	2	3	1,2
Minimum	7	0	7	0	0	0	0	0

Figure 57 - Average, Maximum and Minimum Values of Thesis Metrics for Template Method Pattern after Refactorings

Figure 57 shows new metric values after refactorings for Template Method pattern case.

3.4.7.6. Phase Six: Comparisons and Interpretations

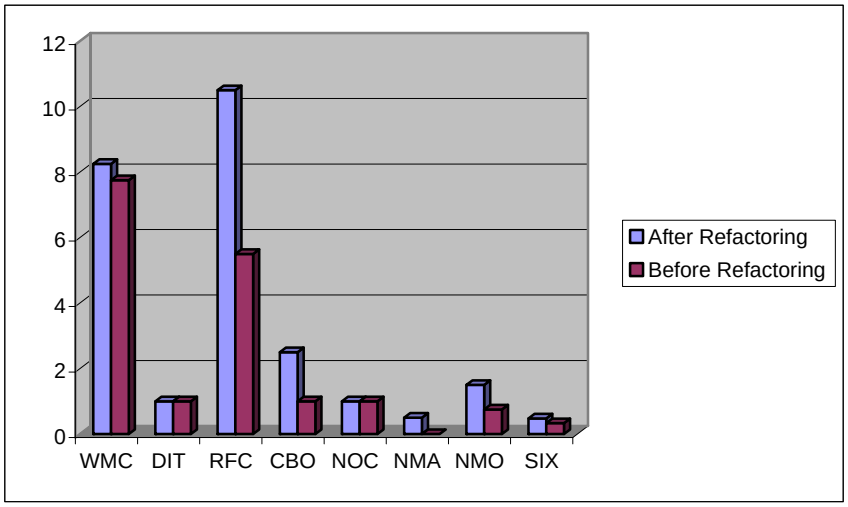


Figure 58 - Comparison of Average Values of Thesis Metrics for Template Method Pattern

Figure 58 shows that after the Template Method implementation, average values of all metrics have increased, except DIT and NOC. Since, this pattern does not change inheritance tree structure. In addition, average RFC and CBO metric values have increased significantly. The main reason for CBO increase is multiple inheritances in **FastBiBubbleSorter**. Since, this class, after refactoring in its methods, used different super classes for the implementation of some invariant parts. But, boost in RFC is the direct result of Template Method refactoring. Because, this pattern has divided algorithm into parts and positioned them to different methods in these classes.

To sum up, this increase in metric values supports the idea that Template Method pattern does not bring any improvements in project. But, the primary objective for this case is to separate variant and invariants parts of the algorithm. Consequently, that objective has been achieved, this design became more flexible. But, designer does not get any measurable improvement from the perspective of software error-proneness.

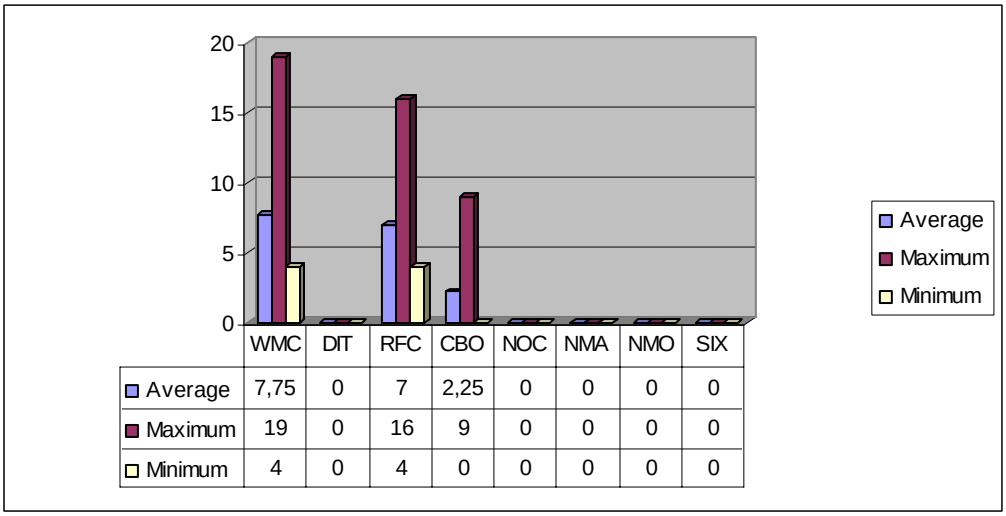
3.4.8. Case 8: Command Pattern

3.4.8.1. Phase One: OO Metrics Measurement of Overall Project

Unrefactored project summary is given below:

Project Summary:

- Classes: 4
- Files: 10
- Functions: 29
- Lines: 636



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	7,75	0	7	2,25	0	0	0	0
Maximum	19	0	16	9	0	0	0	0
Minimum	4	0	4	0	0	0	0	0

Figure 59 - Average, Maximum and Minimum Values of Thesis Metrics for Command Pattern Case before Refactorings

Figure 59 shows metric values before refactorings for Command pattern case.

3.4.8.2. Phase Two: Determining Error Prone Modules

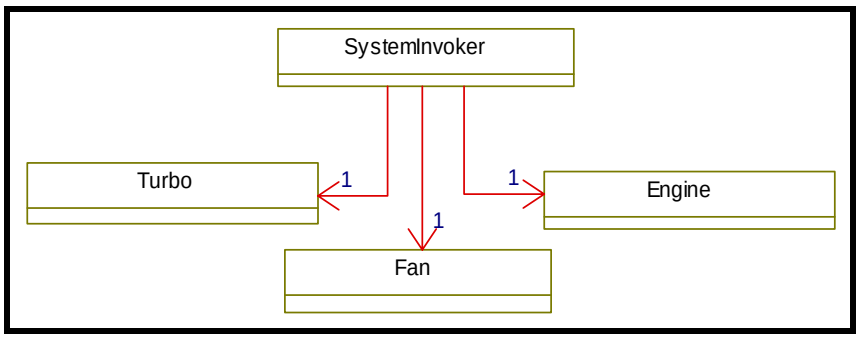


Figure 60 - Design of Command Pattern Case before Refactorings

In this case, **SystemInvoker** class, shown in Figure 60, is tightly coupled with other functional classes and consequently has higher CBO value than any other class in the project. In addition, this class invokes operations to other classes that know how to execute them. These features make this class error-prone and relatively complex.

3.4.8.3. Phase Three: Hatching Design Pattern(s)

Command pattern decouples the object that invokes the operation from the one that knows how to perform it [GOF98]. Therefore, this pattern is the convenient solution for the problem that is described in section 3.4.8.2.

3.4.8.4. Phase Four: Applying related pattern(s)

Figure 61 indicates that application of this pattern to the project introduced six concrete classes and one interface class. All the operations that have been invoked by **SystemInvoker** before refactorings have been shifted into command classes. Therefore, **SystemInvoker** class has been isolated from the functionality.

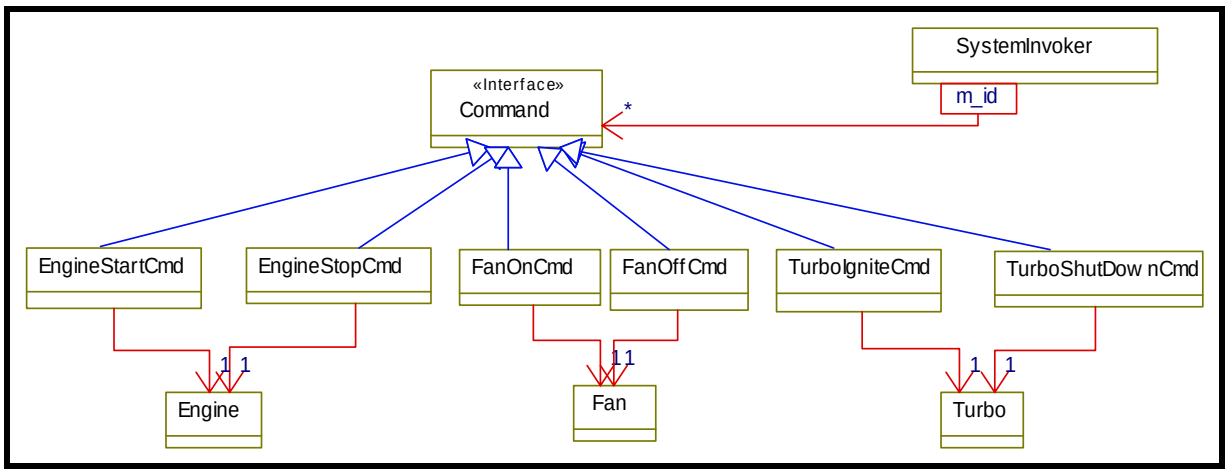


Figure 61 - Design of Command Pattern Case after Refactorings

On the other hand, this pattern introduced new inheritance hierarchies and new associations to software. This side effect increases the total CBO and inheritance related metric values.

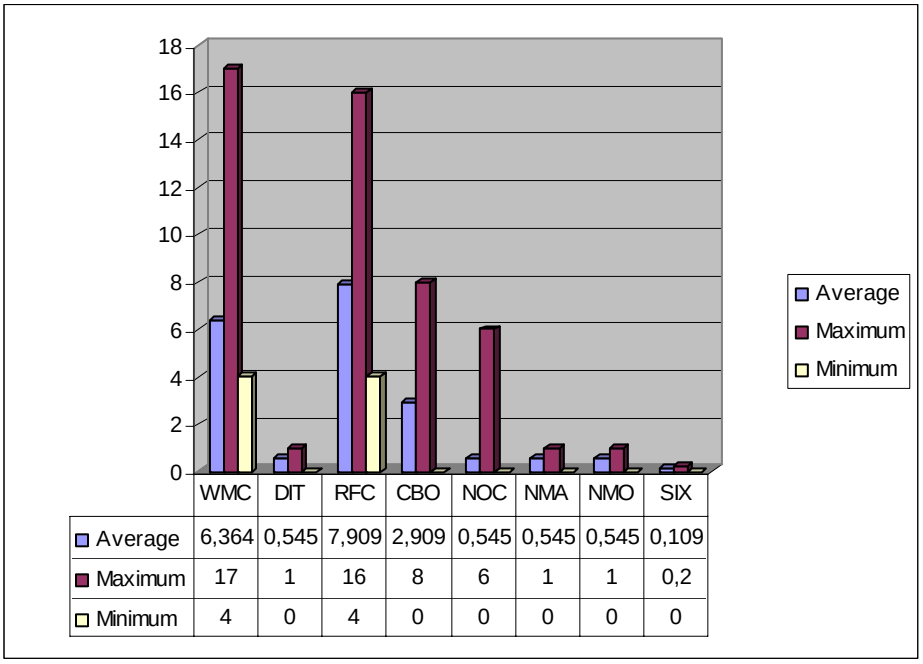
Moreover, instead of direct relations to command classes, an "id" has been given to these classes and **SystemInvoker** instances should have to be initialized with these relations in its internal map data structure before the application runs.

3.4.8.5. Phase Five: Re-measuring software

Refactored project summary is given below:

Project Summary:

- Classes: 11
- Files: 26
- Functions: 64
- Lines: 1685



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	6,364	0,545	7,909	2,909	0,545	0,545	0,545	0,109
Maximum	17	1	16	8	6	1	1	0,2
Minimum	4	0	4	0	0	0	0	0

Figure 62 - Average, Maximum and Minimum Values of Thesis Metrics for Command Pattern Case after Refactorings

Figure 62 displays new metric values after refactorings for Command pattern case.

3.4.8.6. Phase Six: Comparisons and Interpretations

As can be seen in Figure 63, Command pattern did not bring expected improvement in software. In contrast, Command pattern refactoring has increased all average metric values except for WMC. But, primary objective was to reduce CBO and WMC values for **SystemInvoker** class. Figure 64 shows the fluctuations in metrics for this class.

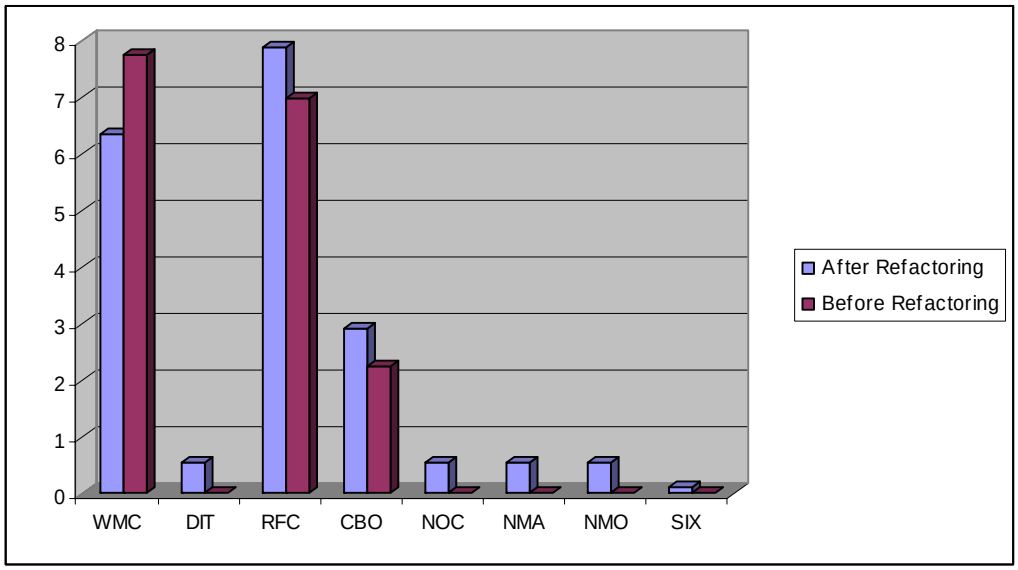


Figure 63 - Comparison of Average Values of Thesis Metrics for Command Pattern

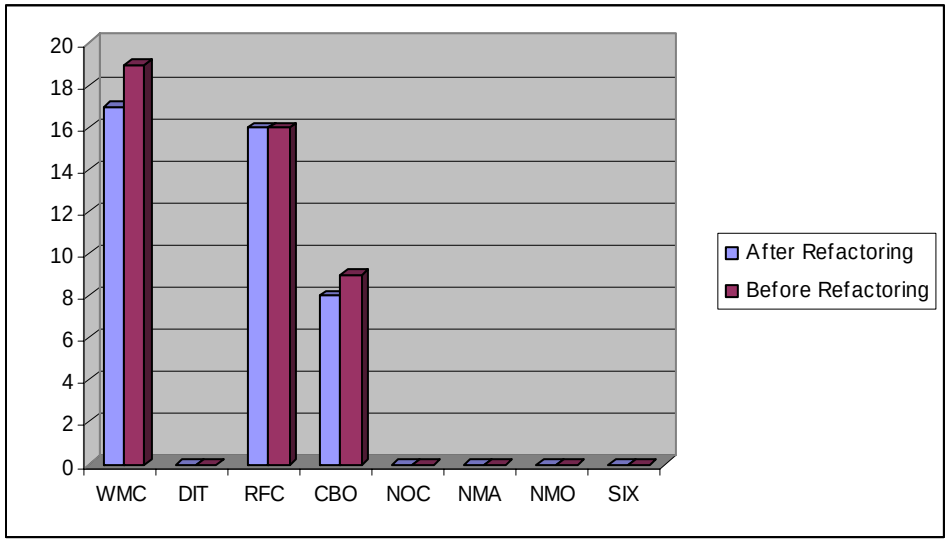


Figure 64 - Comparison of Metric Values of SystemInvoker Class after and before Refactorings

Moreover, Figure 64 indicates that this pattern did not bring any significant improvement in **SystemInvoker** class, either. The main reason is that after refactorings this class had an embedded map data structure and its management methods. These mentioned methods and member variable increased CBO, RFC, and WMC values.

To sum up, Command pattern increased overall complexity and coupling between classes. But, on the other hand, from the view of software maintainability and reusability perspective this pattern isolated functionality and invocation, which is a desired design improvement.

3.4.9. Case 9: Visitor Pattern

Visitor pattern has two aspects for this thesis. One of them is adding tools aspect and the other is structure traversing aspect. In this case, mentioned aspects have been evaluated separately.

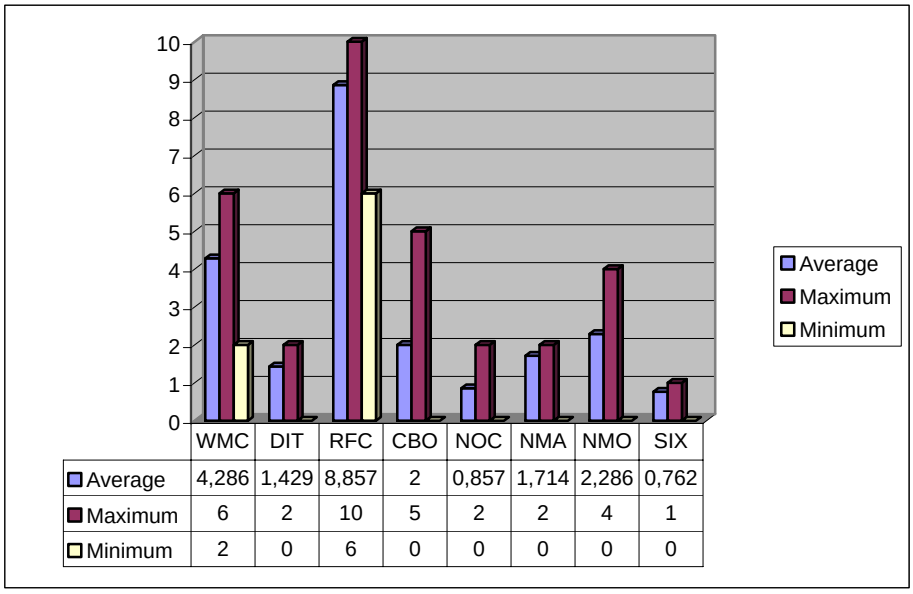
3.4.9.1. Phase One: OO Metrics Measurement of Overall Project

3.4.9.1.1. Adding Tools Aspect

Unrefactored project summary is given below:

Project Summary:

- Classes: 7
- Files: 14
- Functions: 30
- Lines: 802



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	4,286	1,429	8,857	2	0,857	1,714	2,286	0,762
Maximum	6	2	10	5	2	2	4	1
Minimum	2	0	6	0	0	0	0	0

Figure 65 - Average, Maximum and Minimum Values of Thesis Metrics for Adding Tools Aspect Case before Refactorings

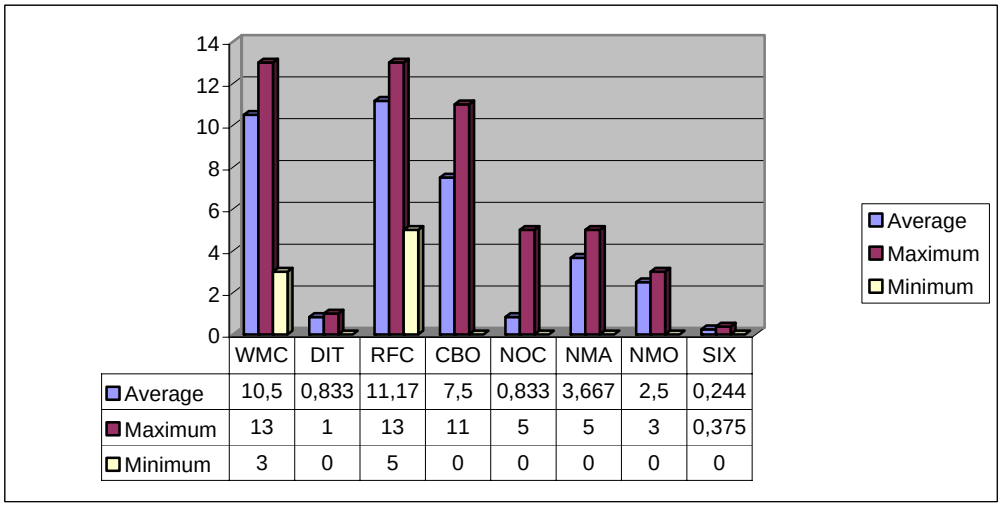
Figure 65 displays metric values before refactorings for adding tools aspect in Visitor pattern case.

3.4.9.1.2. Structure Traversing Aspect

Unrefactored project summary is given below:

Project Summary:

- Classes: 6
- Files: 14
- Functions: 56
- Lines: 1133



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	10,5	0,833	11,17	7,5	0,833	3,667	2,5	0,244
Maximum	13	1	13	11	5	5	3	0,375
Minimum	3	0	5	0	0	0	0	0

Figure 66 - Average, Maximum and Minimum Values of Thesis Metrics for Structure Traversing Aspect Case before Refactorings

Figure 66 displays metric values before refactorings for structure traversing aspect in Visitor pattern case.

3.4.9.2. Phase Two: Determining Error Prone Modules

3.4.9.2.1. Adding Tools Aspect

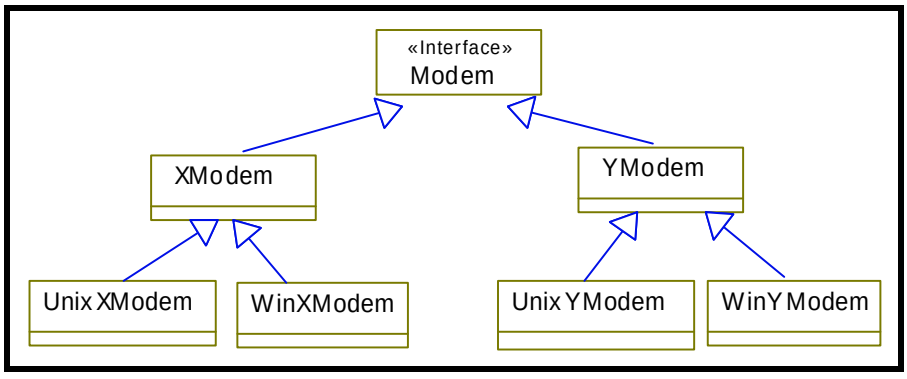


Figure 67 - Design of Adding Tools Aspect before Refactorings

Figure 67 shows that in the above hierarchy, adding new operations is not an easy task. Defining a new operation in **Modem** interface penetrates over other classes, and also

pollutes interface. For example, in this case, there are related and unrelated operations in the modem interface. These unrelated operations should be separated from the interface. But, in this case, these operations are implemented in leaf classes. The result of this solution is the increase in DIT and complexity of the inheritance structure. Moreover, another outcome is the excessive subclassing.

Therefore, modifications on **Modem** interface are hard to perform. Moreover, if new requirements arrive, modifications will cause waste of the time and increased complexity in the inheritance tree structure. To be brief, this design needs refactoring to remove unrelated methods pollution and to make it more flexible.

3.4.9.2.2. Structure Traversing Aspect

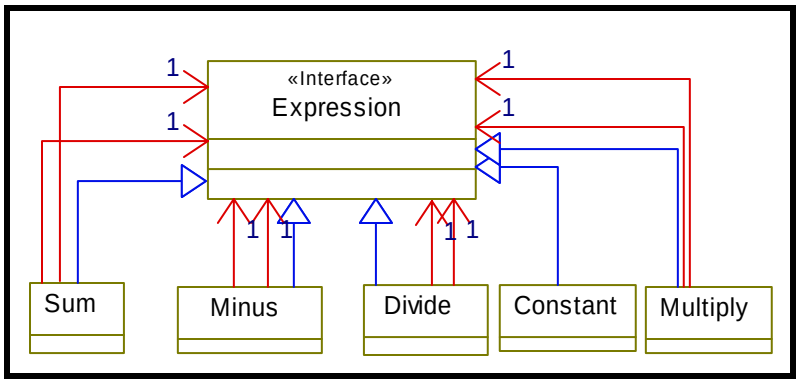


Figure 68 - Design of Structure Traversing Aspect before Refactorings

For the above case, same problems are valid, which are adding new tools and interface pollution. Moreover, in Figure 68, concrete classes actually are nodes in the expression, but they are also responsible for the computational algorithms. This is the problem for the above design. Therefore, leaf classes should be refactored, and algorithms should be separated from them. An Iterator approach can not solve described problem, since Iterators do not have the ability of mounting up the evaluation results after each expression performed in this case.

3.4.9.3. Phase Three: Hatching Design Pattern(s)

According to [GOF98], Visitor pattern has below consequences:

- Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would be passed as extra arguments to the operations that perform the traversal, or they might appear as global variables.
- A visitor gathers related operations and separates unrelated ones.

These consequences are the solution context for the problems described in 3.4.9.2.1 and 3.4.9.2.2. Therefore, Visitor pattern has been applied to these aspects for refactoring purposes.

3.4.9.4. Phase Four: Applying related pattern(s)

3.4.9.4.1. Adding Tools Aspect

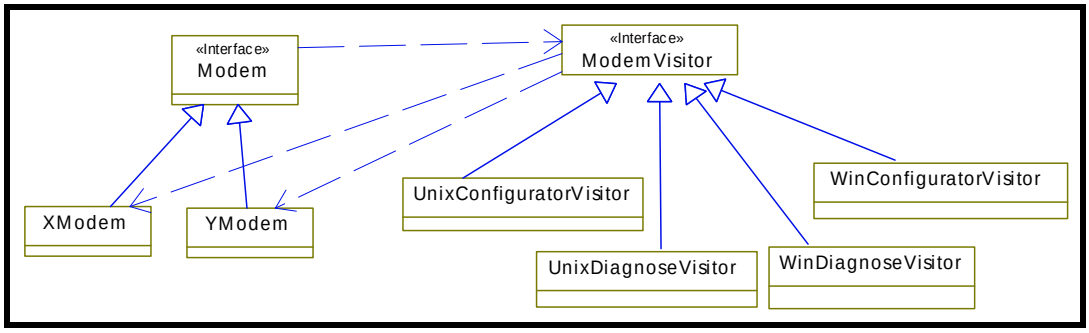


Figure 69 - Design of Visitor Pattern for Adding Tools Aspect after Refactorings

As can be seen from Figure 69, application of Visitor pattern for this aspect eliminated four leaf classes whose superclass classes were **XModem** and **YModem**. But, this modification introduced four concrete classes and one interface class to the project. These four classes are implementation of interface polluting methods of Modem interface for different operating systems.

3.4.9.4.2. Structure Traversing Aspect

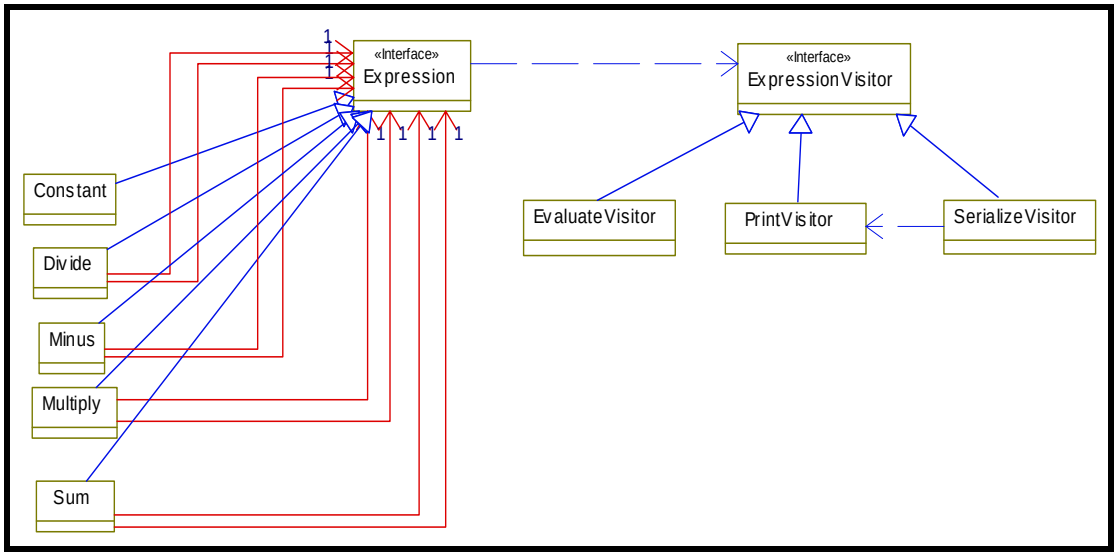


Figure 70 - Design of Visitor Pattern for Structure Traversing Aspect after Refactorings

Figure 70 shows that application of this pattern for this aspect did not alter the inheritance structure but introduced three concrete classes and one interface class to software, and also eliminated interface pollution for **Expression** class. But, also this refactoring managed to separate algorithms and traversing concepts.

3.4.9.5. Phase Five: Re-measuring software

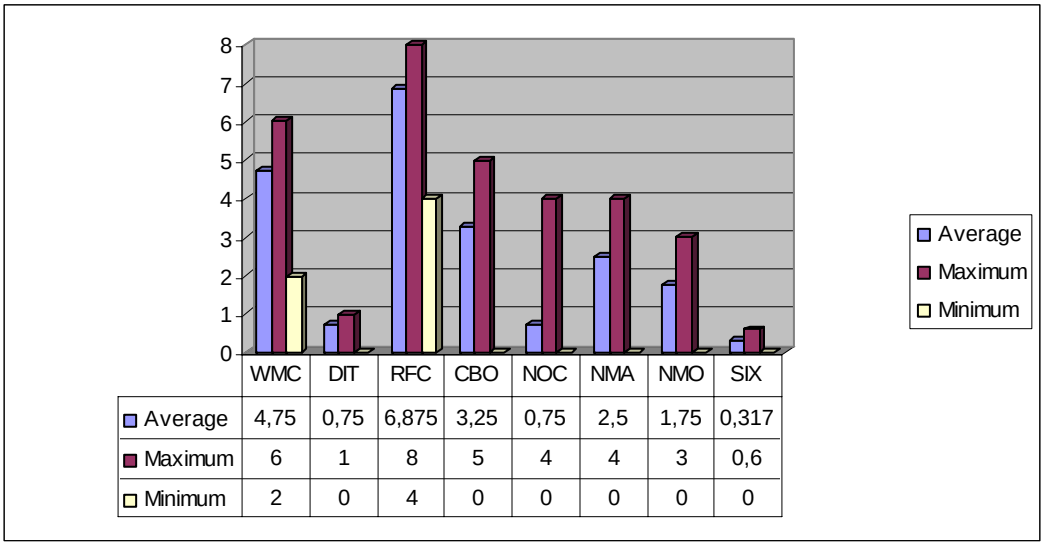
3.4.9.5.1. Adding Tools Aspect

Refactored project summary is given below:

Project Summary:

- Classes: 8
- Files: 16
- Functions: 38

- Lines: 1023



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	4,75	0,75	6,875	3,25	0,75	2,5	1,75	0,317
Maximum	6	1	8	5	4	4	3	0,6
Minimum	2	0	4	0	0	0	0	0

Figure 71 - Average, Maximum and Minimum Values of Thesis Metrics for Adding Tools Aspect Case after Refactorings

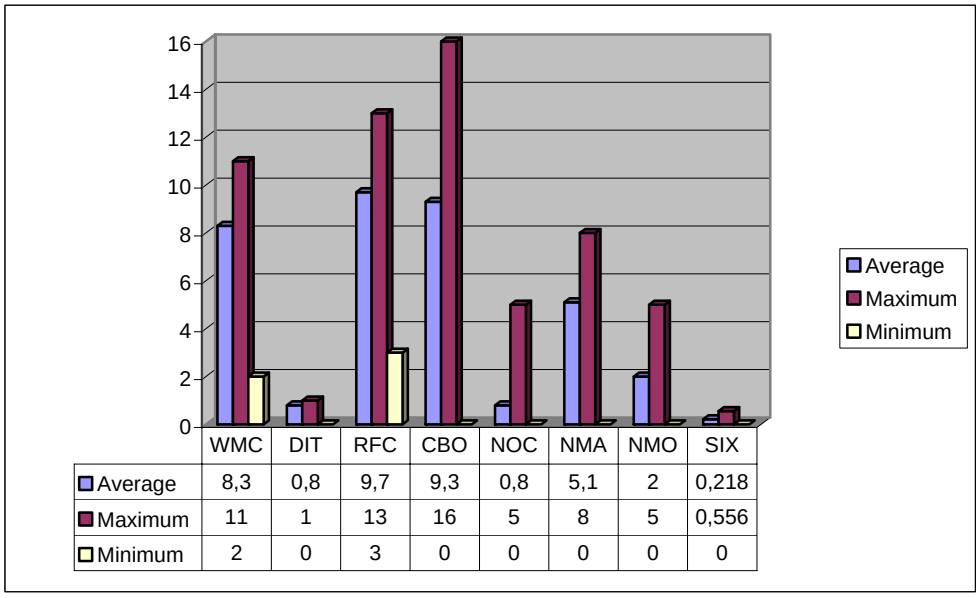
Figure 71 displays new metric values of metrics after refactorings for adding tools aspect in Visitor pattern.

3.4.9.5.2. Structure Traversing Aspect

Refactored project summary is given below:

Project Summary:

- Classes: 10
- Files: 22
- Functions: 76
- Lines: 1771



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	8,3	0,8	9,7	9,3	0,8	5,1	2	0,218
Maximum	11	1	13	16	5	8	5	0,556
Minimum	2	0	3	0	0	0	0	0

Figure 72 - Average, Maximum and Minimum Values of Thesis Metrics for Structure Traversing Aspect Case after Refactorings

Figure 72 displays new metric values of metrics after refactorings for structure traversing aspect in Visitor pattern.

3.4.9.6. Phase Six: Comparisons and Interpretations

3.4.9.6.1. Adding Tools Aspect

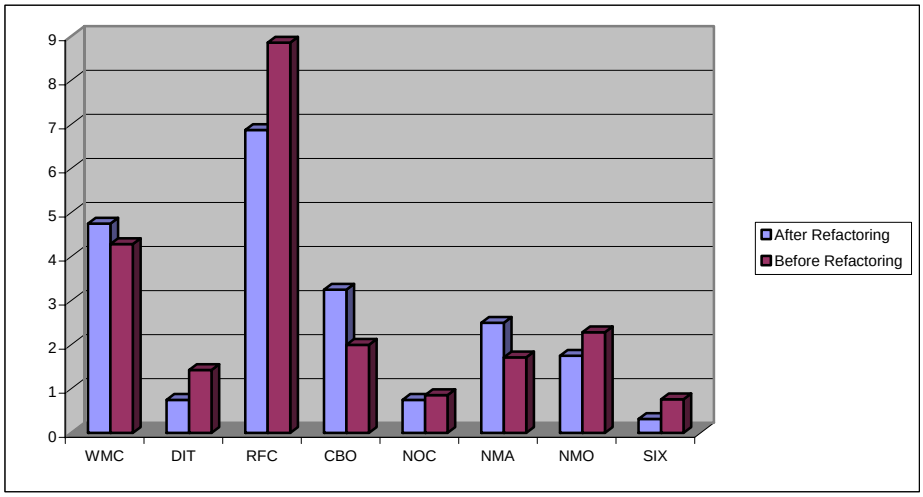


Figure 73 - Comparison of Average Values of Thesis Metrics for Adding Tools Aspect

Implementation of Visitor pattern has significantly reduced average DIT and SIX metric values, as can be seen in Figure 73. This result indicates the fact that Visitor pattern is a remedy for excessive subclassing and adding tools to a solid interface problem. Moreover, the main objective was to achieve to add new behaviors to an existing structure without polluting interface and hierarchy tree. Also, decrease in SIX metric shows that more classes in the design are in better positions in inheritance tree structure.

On the other hand, Visitor pattern causes design to be more complex, as CBO and WMC metric scores indicates. This is the trade-off that designer should decide. But, this pattern for this case eliminated poor design and prevented future design corruption.

3.4.9.6.2. Structure Traversing Aspect

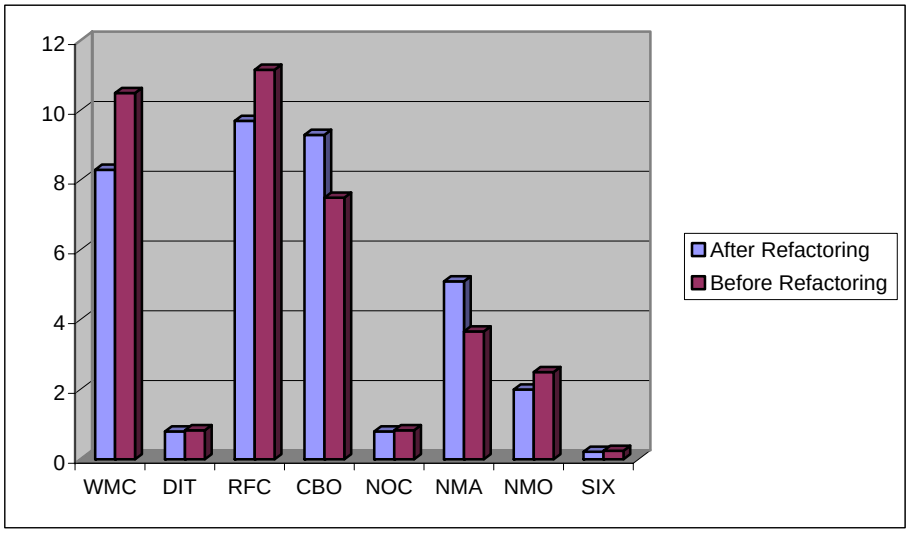


Figure 74 - Comparison of Average Values of Thesis Metrics for Structure Traversing Aspect

Figure 74 graphically reveals that effect of this pattern has distributed over complexity and inheritance related metrics. It has reduced class complexity but increased class coupling. In addition, Visitor pattern has minor effects for average DIT and NOC. But, effect of this pattern on other inheritance related metric is more influential. Because, traversing and computational algorithms are separated, and new classes had to override less methods. But, as a side effect more methods were added to provide same functionality.

To sum up, for this case, this pattern completed desired objectives and provided flexibility to the design. Moreover, Visitor pattern positively affected software error-proneness.

3.4.10. Case 10: Observer Pattern

3.4.10.1. Phase One: OO Metrics Measurement of Overall Project

Unrefactored project summary is given below:

Project Summary:

- Classes: 7
- Files: 18
- Functions: 60
- Lines: 1360

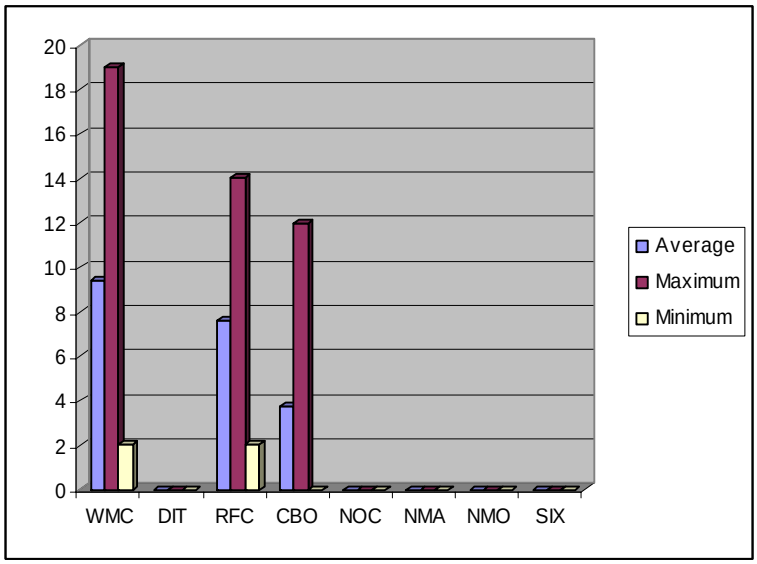


Figure 75 - Average, Maximum and Minimum Values of Thesis Metrics for Observer Pattern Case before Refactorings

Figure 75 displays metric values before refactorings for Observer pattern case.

3.4.10.2. Phase Two: Determining Error Prone Modules

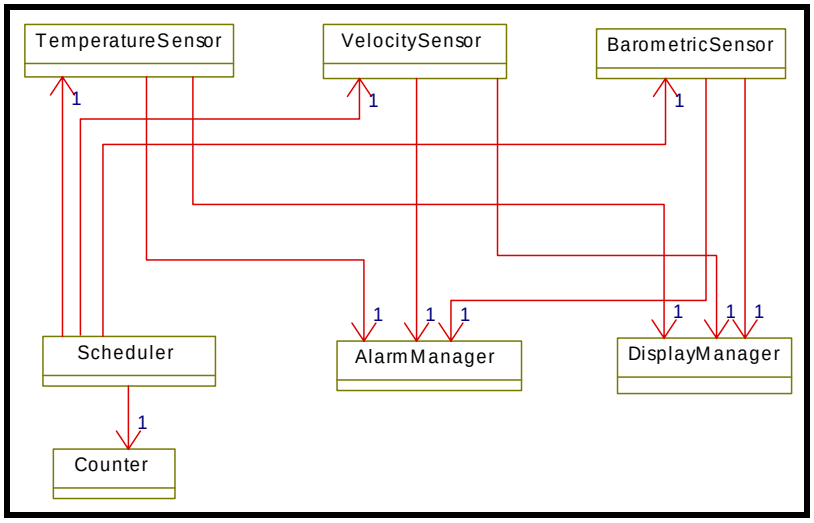


Figure 76 - Design of Observer Pattern Case before Refactorings

In this case, classes are highly coupled with each other, as can be seen in Figure 76. Moreover, objects notify each other directly. In other words, when an event or behavior change occurs in an object, this object calls related methods of its associated classes. This design makes classes highly coupled, resisted to modifications and error-prone.

Moreover, direction of the coupling is more critical. Since, less modification-prone classes are associated with more modification-prone ones. This direction problem limits reuse and changeability.

3.4.10.3. Phase Three: Hatching Design Pattern(s)

Design problem described in 3.4.10.2 mainly results from object notification requirements. Every event dispatcher object specifically knows the other associated class to notify them. But, they know too much. Designer should implement an abstraction to solve high coupling problem. The solution seems to be Observer pattern. Because, this pattern has the applicability that is given below:

- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled [GOF98].

3.4.10.4. Phase Four: Applying related pattern(s)

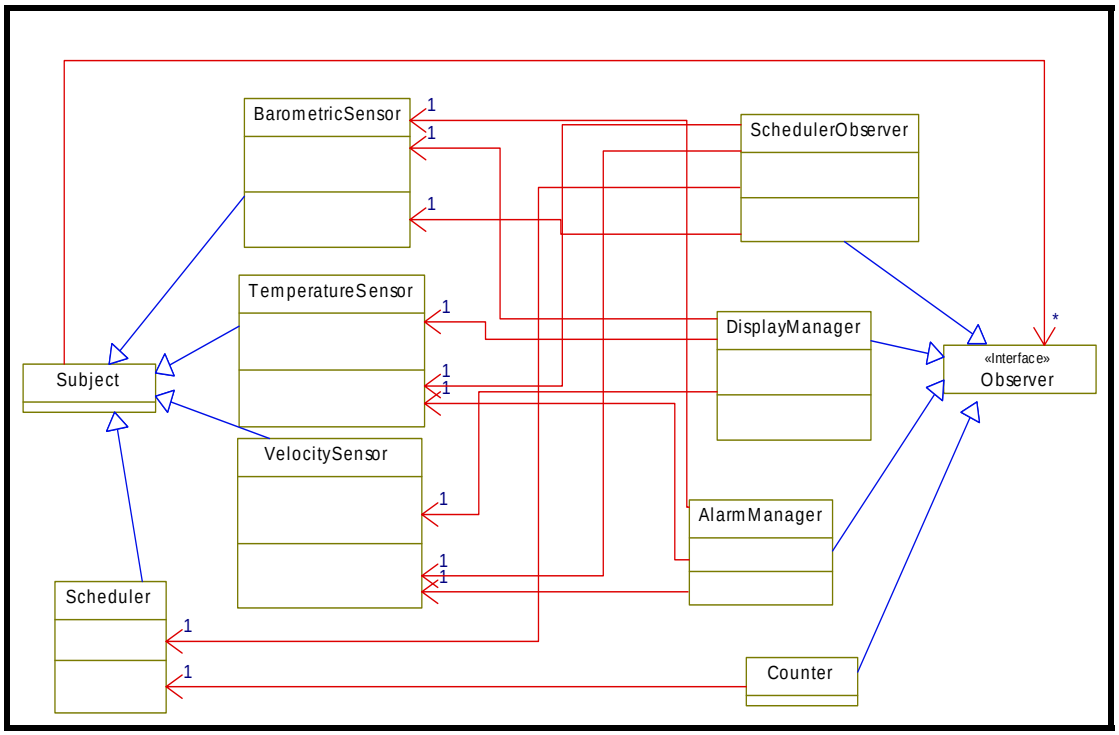


Figure 77 - Design of Observer Pattern Case after Refactorings

Figure 77 shows that refactoring with Observer pattern has introduced two interfaces, which are Observer and Subject classes. These interfaces hide the details of notification, and change the direction of association. Moreover, between Subject and Observer interfaces, there is one-to-many relation to assure that one event dispatcher object or subject can notify many others that have to inherit from Observer class.

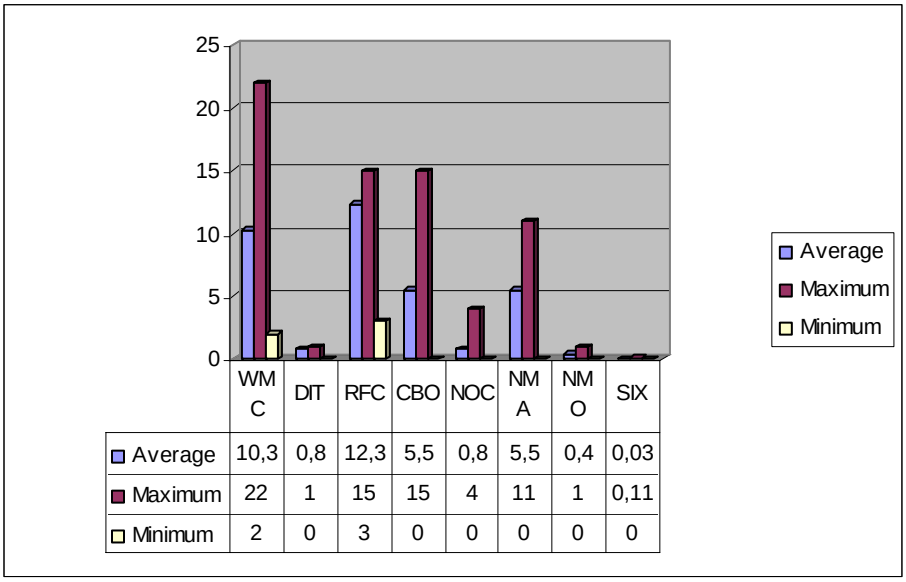
In addition, to prevent multiple inheritances for sensor classes, **SchedulerObserver** class has been added to design. This class notifies others that have been notified by Scheduler before.

3.4.10.5. Phase Five: Re-measuring software

Refactored project summary is given below:

Project Summary:

- Classes: 10
- Files: 24
- Functions: 79
- Lines: 1846



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	10,3	0,8	12,3	5,5	0,8	5,5	0,4	0,03
Maximum	22	1	15	15	4	11	1	0,11
Minimum	2	0	3	0	0	0	0	0

Figure 78 - Average, Maximum and Minimum Values of Thesis Metrics for Observer Pattern Case after Refactorings

Figure 78 displays new metric values after refactorings for Observer pattern case.

3.4.10.6. Phase Six: Comparisons and Interpretations

Figure 79 clearly shows that Observer pattern has increased all the error-proneness metrics of the project. This result empirically indicates that this pattern eliminates association direction problem, but increases complexity and error-proneness probability of the classes. However, if Mediator pattern had been implemented together with Observer pattern for this case, increase in complexity and error-proneness would have been lower as showed in Case 3.

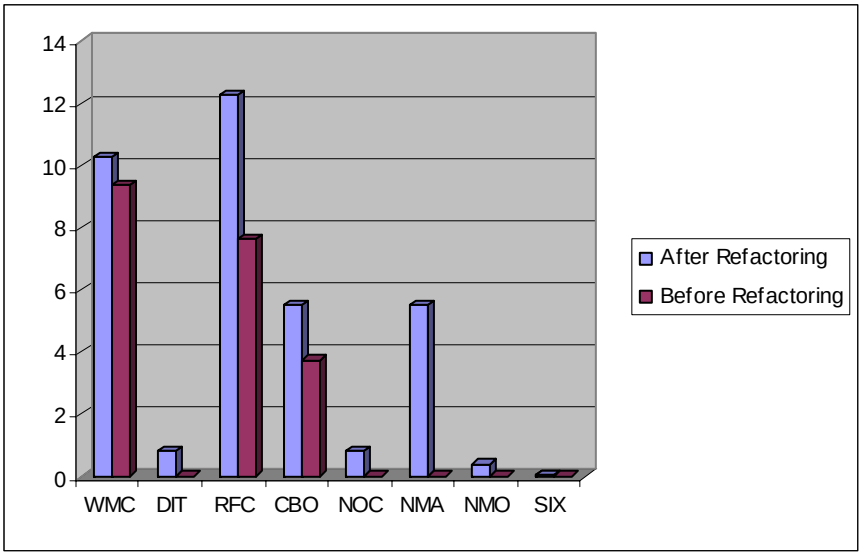


Figure 79 - Comparison of Average Values of Thesis Metrics for Observer Pattern

To sum up, Observer pattern is not a remedy for high coupling, alone. If designer want to decrease high coupling and change the direction of association between classes where notifications are present, he/she should use Mediator with Observer pattern.

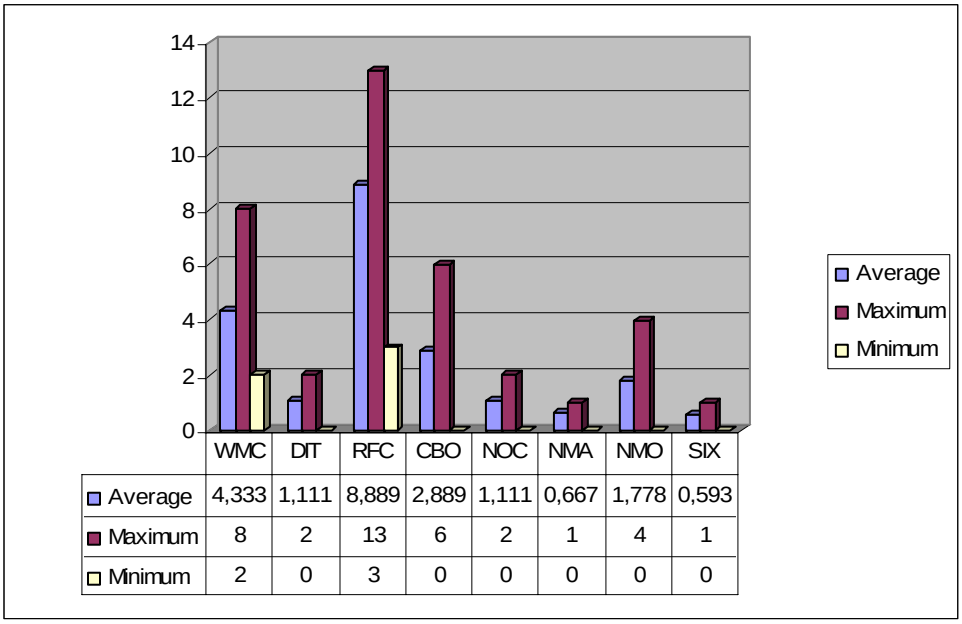
3.4.11. Case 11: Decorator Pattern

3.4.11.1. Phase One: OO Metrics Measurement of Overall Project

Unrefactored project summary is given below:

Project Summary:

- Classes: 9
- Files: 20
- Functions: 38
- Lines: 1090



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	4,333	1,111	8,889	2,889	1,111	0,667	1,778	0,593
Maximum	8	2	13	6	2	1	4	1
Minimum	2	0	3	0	0	0	0	0

Figure 80 - Average, Maximum and Minimum Values of Thesis Metrics for Decorator Pattern Case before Refactorings

Figure 80 displays metric values before refactorings for Decorator pattern case.

3.4.11.2. Phase Two: Determining Error Prone Modules

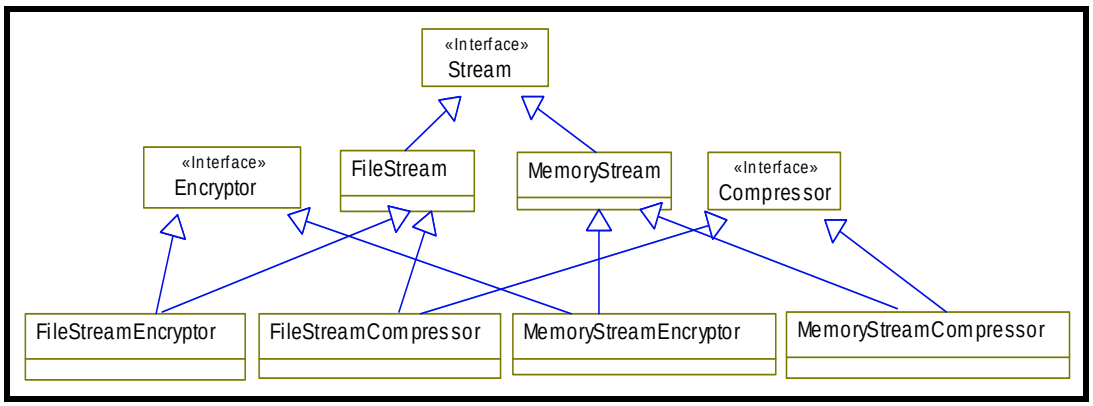


Figure 81 - Design of Decorator Pattern Case before Refactorings

For this case, error-prone modules are the leaf classes of the design, shown in Figure 81. These four classes have multiple interfaces that cause relatively high CBO, RFC and DIT metric values for these leaf classes.

3.4.11.3. Phase Three: Hatching Design Pattern(s)

The problem for this design is excessive subclassing and multiple inheritances. Moreover, adding every combination for compression and encryption is unfeasible and causes bursts in error-proneness related metrics. However, there is a solution for this excessive subclassing problem. Solution is Decorator pattern. Since, in [GOF98], Decorator pattern has below applicability:

- When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.

3.4.11.4. Phase Four: Applying related pattern(s)

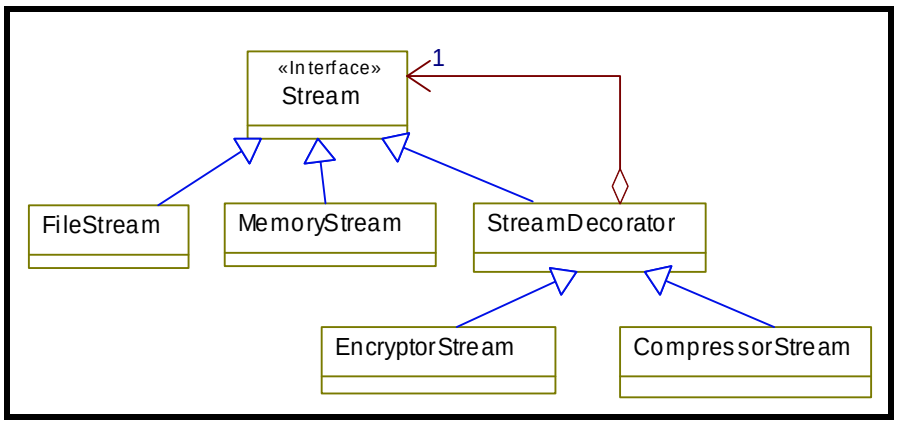


Figure 82 - Design of Decorator Pattern Case after Refactorings

Figure 82 reveals that refactoring with Decorator pattern has eliminated four leaf classes plus **Compressor** and **Encryptor** interfaces. However, this pattern has introduced **StreamDecorator**, **EncryptorStream**, and **CompressorStream** classes. These three classes confirmed Decorator pattern. Also, they provided designer to implement all possible compression and encryption combinations over stream classes, elegantly.

Moreover, refactoring with Decorator pattern has eradicated excessive subclassing and multiple inheritances problems. Moreover, this pattern has brought software flexibility and reusability. However, **StreamDecorator** inherits from **Stream** interface, but it is much different from the other stream classes; therefore designer shouldn't rely on object identity when he/she uses decorators [GOF98].

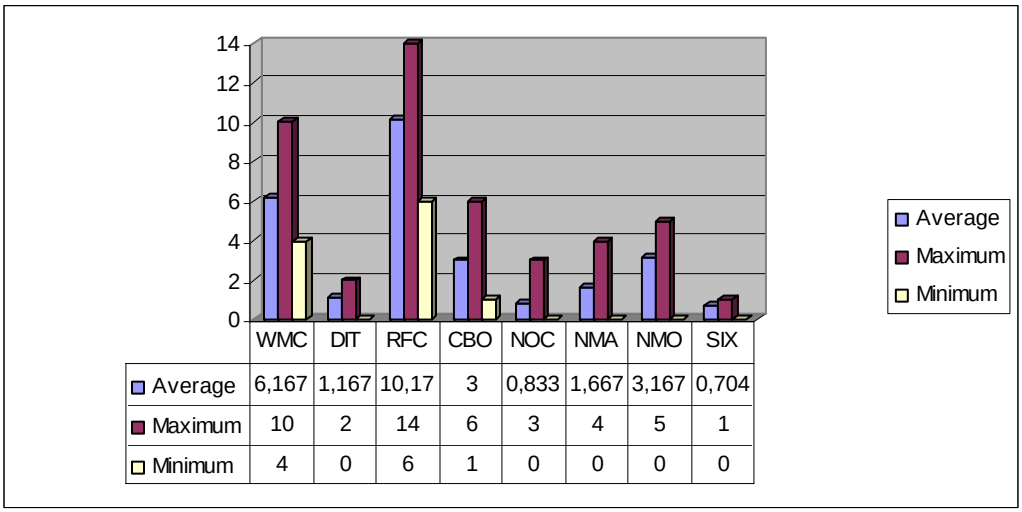
3.4.11.5. Phase Five: Re-measuring software

Refactored project summary is given below:

Project Summary:

- Classes: 6
- Files: 14

- Functions: 35
- Lines: 858



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	6,167	1,167	10,17	3	0,833	1,667	3,167	0,704
Maximum	10	2	14	6	3	4	5	1
Minimum	4	0	6	1	0	0	0	0

Figure 83 - Average, Maximum and Minimum Values of Thesis Metrics for Decorator Pattern Case after Refactorings

Figure 83 shows metric values after refactorings for Decorator pattern case.

3.4.11.6. Phase Six: Comparisons and Interpretations

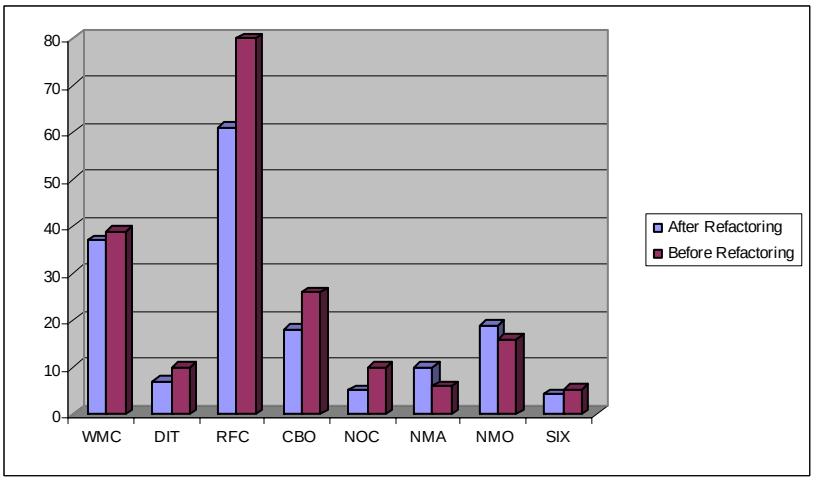


Figure 84 - Comparison of Sum Values of Thesis Metrics for Decorator Pattern

Figure 84 graphically exposes that refactoring with Decorator pattern has significantly reduced sum values of DIT, RFC, CBO, and NOC. This reduction was the primary objective. However, increases in sum values of NMA, and NMO are undesired and caused as a side effect by reduction in DIT and RFC. Also, this pattern has reduced total number of classes in the project. Since, Decorator pattern eliminated unnecessary leaf classes. But, this reduction caused boost in average class metrics except for NOC as Figure 85 indicates.

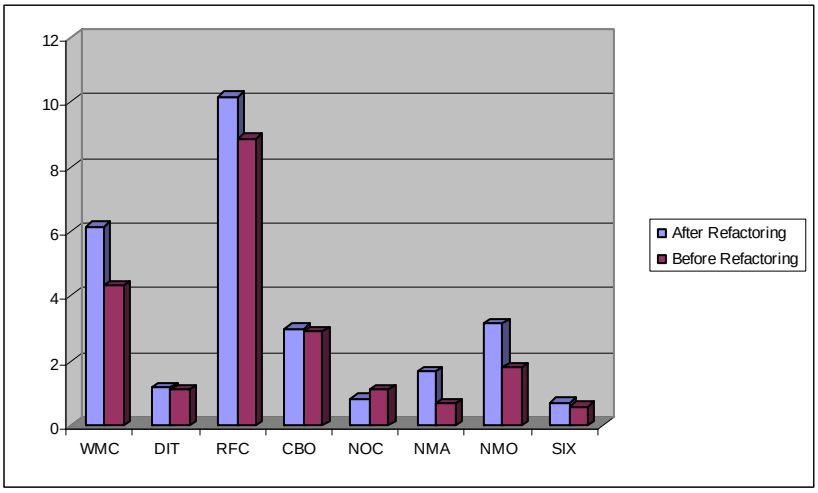


Figure 85 - Comparison of Average Values of Thesis Metrics for Decorator Pattern

In conclusion, Decorator pattern has procured necessary improvement for the project. It prevented excessive subclassing in the design. Moreover, increases in average values of NMO, NMA and SIX were inevitable. Since, after refactorings number of classes has decreased and new design had to provide same functionality. In brief, the main result of this case is that Decorator pattern has brought improvement in design and also positively effected software error-proneness.

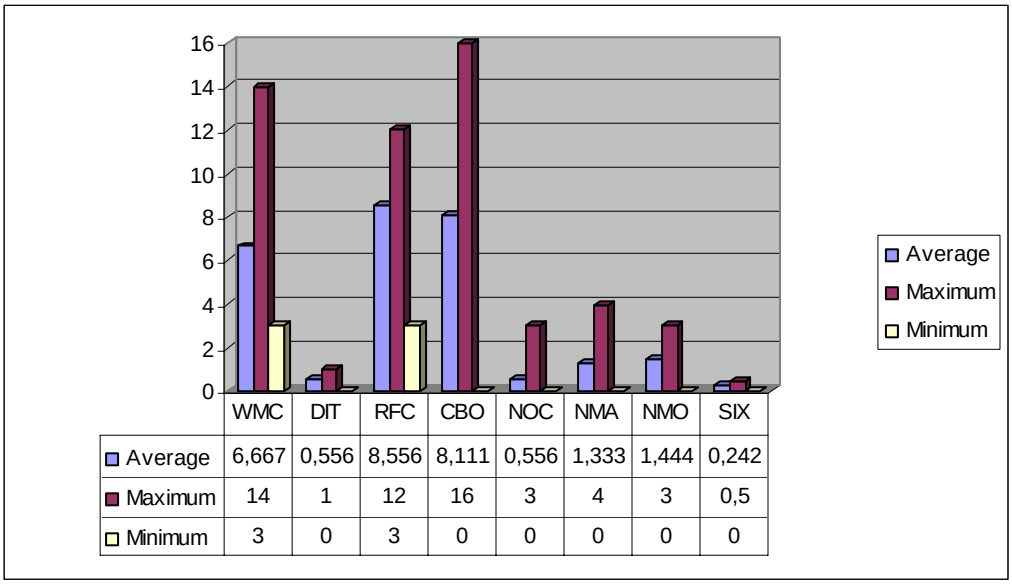
3.4.12. Case 12: Memento Pattern

3.4.12.1. Phase One: OO Metrics Measurement of Overall Project

Unrefactored project summary is given below:

Project Summary:

- Classes: 9
- Files: 20
- Functions: 56
- Lines: 1383



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	6,667	0,556	8,556	8,111	0,556	1,333	1,444	0,242
Maximum	14	1	12	16	3	4	3	0,5
Minimum	3	0	3	0	0	0	0	0

Figure 86 - Average, Maximum and Minimum Values of Thesis Metrics for Memento Pattern Case before Refactorings

Figure 86 shows metric values before refactorings for Memento pattern case.

3.4.12.2. Phase Two: Determining Error Prone Modules

The problem for this design originates from **RobotMovementController** class. This class saves its state when a command executes. State management requirement makes

RobotMovementController class complex and error-prone. In other words, this class has relatively high WMC, CBO and RFC values when compared with others.

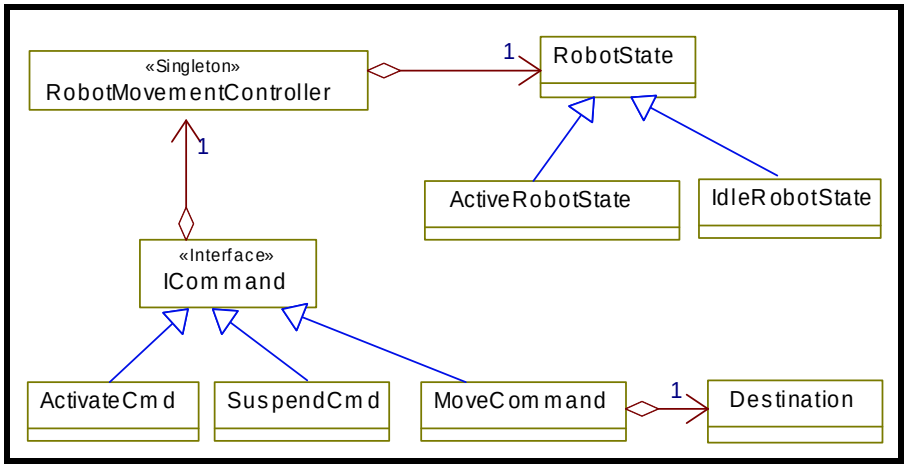


Figure 87 - Design of Memento Pattern Case before Refactorings

As Figure 87 shows, State pattern was also used for reducing complexity of **RobotMovementController** class. But, requirement for saving internal state caused management problems for this class. Therefore, weak chain of this design is **RobotMovemmentController** class.

3.4.12.3. Phase Three: Hatching Design Pattern(s)

The described problem, in part 3.4.12.2, motivates the refactoring with Memento pattern. Since, this pattern has below applicability [GOF98]:

- Snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later.

Memento pattern will simplify **RobotMovementController** class. Since, this class will not be responsible of saving and managing its current state information. All these responsibilities will be transferred and distributed to command classes.

3.4.12.4. Phase Four: Applying related pattern(s)

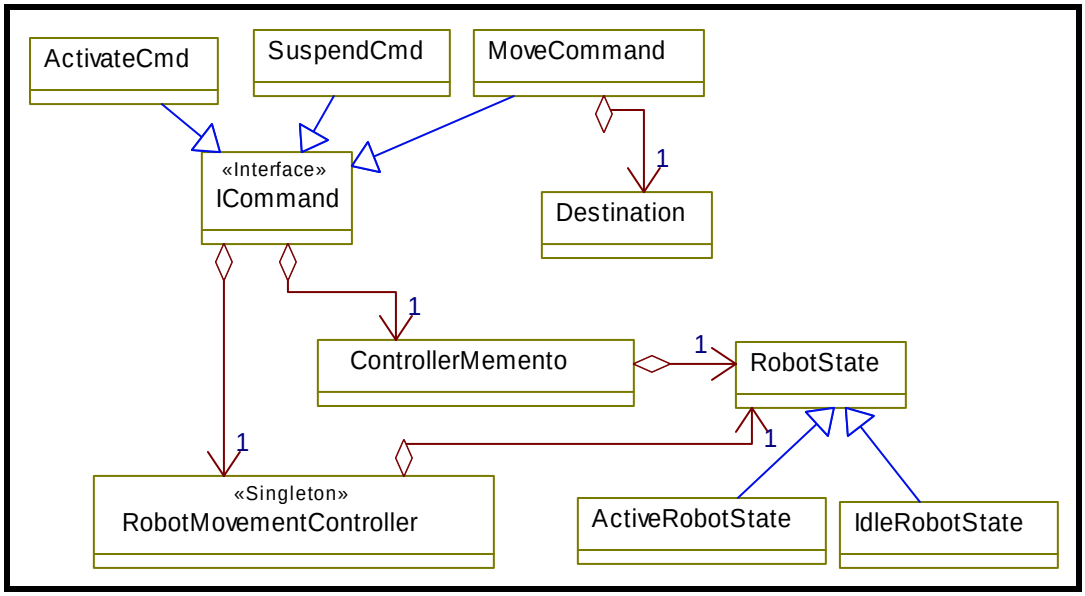


Figure 88 - Design of Memento Pattern Case after Refactorings

Refactoring with Memento pattern has introduced a new class, called **ControllerMemento,** also two associations. One association is between **ICommand** and ControllerMemento. The other one is between **ControllerMemento** and **RobotState** classes. These modifications were inevitable for the refactoring with this pattern.

Moreover, command classes do not know memento interface, which is only exposed to **RobotMovementController** class. In other words, there is a friend type inclusion between **ControllerMemento** and **RobotMovementController** classes. The reason for this inclusion is to preserve encapsulation for the state information.

As can be seen from Figure 88, command classes keep a reference of the memento of **RobotMovementController** in this refactored design. But, command classes do not know how to create and restore this memento information.

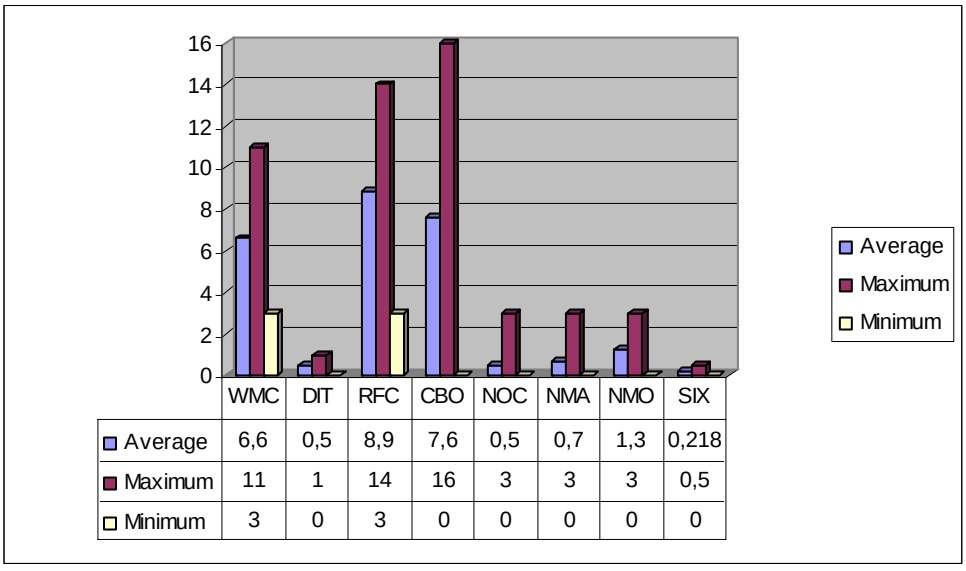
Moreover, **RobotMovementController** knows how to perform the transformation from memento into state information, which keeps state information hidden from command classes.

3.4.12.5. Phase Five: Re-measuring software

Refactored project summary is given below:

Project Summary:

- Classes: 10
- Files: 22
- Functions: 62
- Lines: 1546

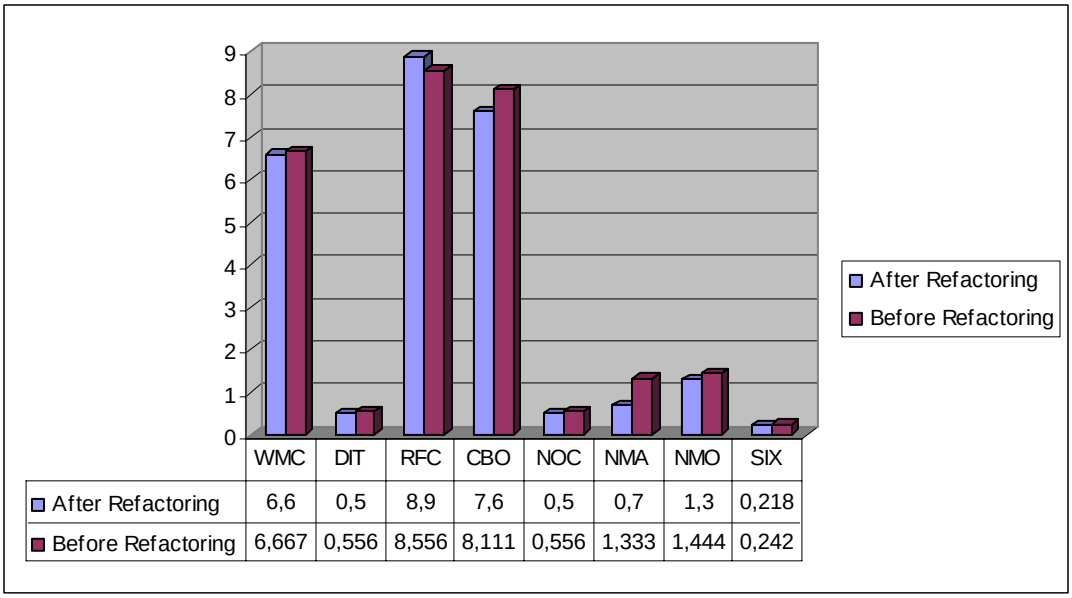


	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	6,6	0,5	8,9	7,6	0,5	0,7	1,3	0,218
Maximum	11	1	14	16	3	3	3	0,5
Minimum	3	0	3	0	0	0	0	0

Figure 89 - Average, Maximum and Minimum Values of Thesis Metrics for Memento Pattern Case after Refactorings

Figure 89 shows metric values after refactorings for Memento pattern case.

3.4.12.6. Phase Six: Comparisons and Interpretations

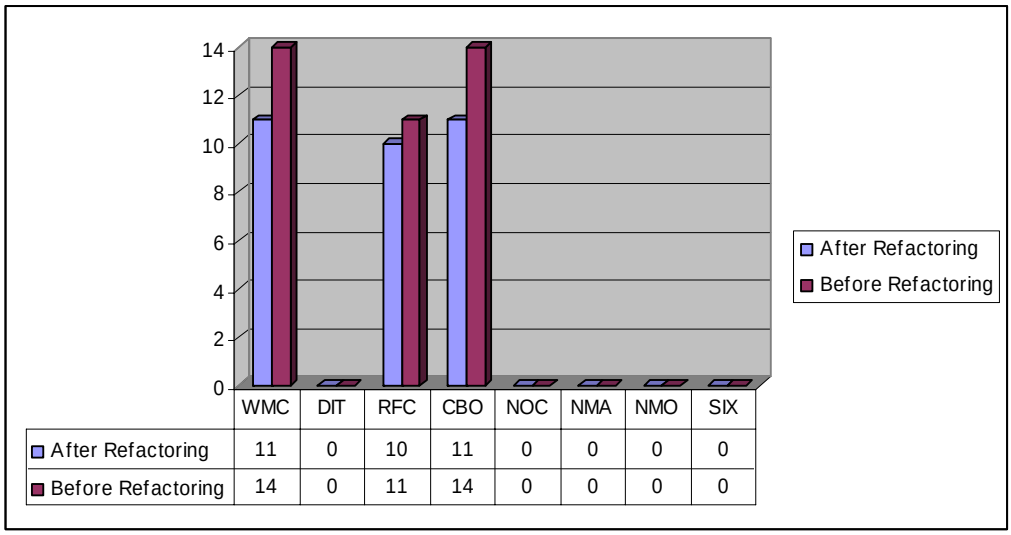


	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
After Refactoring	6,6	0,5	8,9	7,6	0,5	0,7	1,3	0,218
Before Refactoring	6,667	0,556	8,556	8,111	0,556	1,333	1,444	0,242

Figure 90 - Comparison of Average Values of Thesis Metrics for Memento Pattern

Figure 90 graphically reveals that Memento pattern did not affect average values of metrics significantly except for NMA and SIX. But, decreases in NMA and SIX mainly results from **ControllerMemento** class. Because, this class has zero SIX and NMA value, which decreases averages for these metrics.

However, main concern was to observe the effects on **RobotMovementController** class. Figure 91 clearly shows that Memento pattern, without significantly effecting overall average of complexity related metrics, has improved and simplified **RobotMovementController** class. In conclusion, this design pattern helps designer to reduce complexity and error-proneness of a class when its state has to be saved and restored later.



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
After Refactoring	11	0	10	11	0	0	0	0
Before Refactoring	14	0	11	14	0	0	0	0

Figure 91 - Comparison of Metric Values for RobotMovementController Class

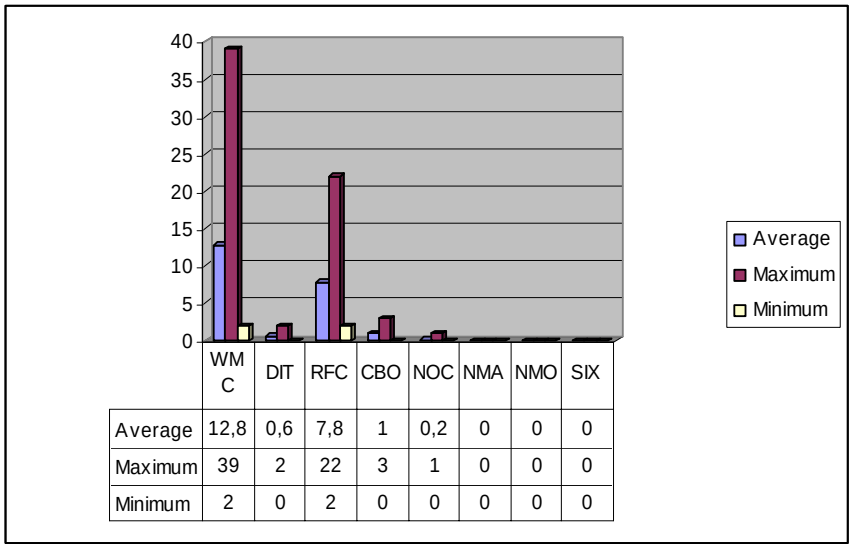
3.4.13. Case 13: Iterator Pattern

3.4.13.1. Phase One: OO Metrics Measurement of Overall Project

Unrefactored project summary is given below:

Project Summary:

- Classes: 5
- Files: 14
- Functions: 38
- Lines: 965



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	12,8	0,6	7,8	1	0,2	0	0	0
Maximum	39	2	22	3	1	0	0	0
Minimum	2	0	2	0	0	0	0	0

Figure 92 - Average, Maximum and Minimum Values of Thesis Metrics for Iterator Pattern Case before Refactorings

Figure 92 shows metric values before refactorings for Iterator pattern case.

3.4.13.2. Phase Two: Determining Error Prone Modules

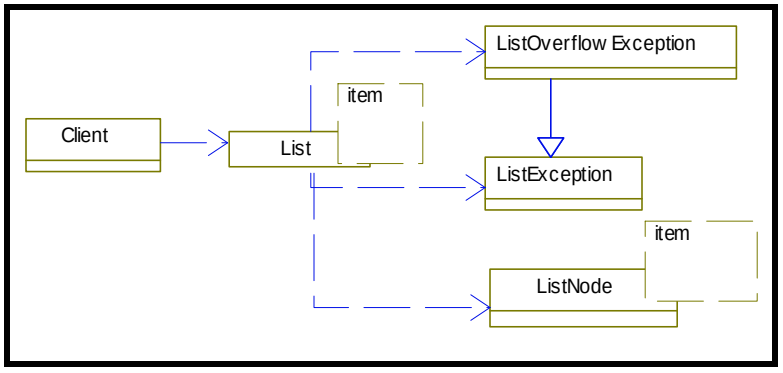


Figure 93 - Design of Iterator Pattern Case before Refactorings

In this design, as can be seen from Figure 93, template **List** class is the weak chain. Since, it has relatively higher WMC and RFC values. The reason of these high values is that this class is responsible for serving access and modification methods to the list nodes, also traversing over them. Moreover, these methods expose the internal structure

and break encapsulation. In brief, this class has more responsibilities than it should have. Metric values of **List** class are given in Table 2.

Table 2 - Metric Values of List Class

	List
WMC	39
DIT	0
RFC	22
CBO	3
NOC	0
NMA	0
NMO	0
SIX	0

In addition, supporting different traverse algorithms will complicate either **Client** or **List** classes. In this case, **Client** class supports two different traversing algorithms, which makes this class relatively more error-prone when compared with other classes in the project.

3.4.13.3. Phase Three: Hatching Design Pattern(s)

[GOF98] suggests Iterator pattern solution for the problem defined in 3.4.13.2. Because, Iterator pattern has below applicabilities:

- To access an aggregate object's contents without exposing its internal representation [GOF98]. That solves the extensive responsibility and breaking encapsulation problems for **List** class.
- To support multiple traversals of aggregate objects [GOF98]. This applicability is the solution for high WMC value problem in **Client** class.

The solution context that is offered by Iterator pattern is suitable to handle the problems described in preceding sections. Therefore, this pattern is used for refactoring purposes in this case.

Moreover, Standard Template Library (STL) uses a different implementation or a variance of Iterator pattern to solve these problems described above. Therefore, this pattern carries vital importance in design, if designer implements custom container classes and flexibility and reusability have superior priorities.

3.4.13.4. Phase Four: Applying related pattern(s)

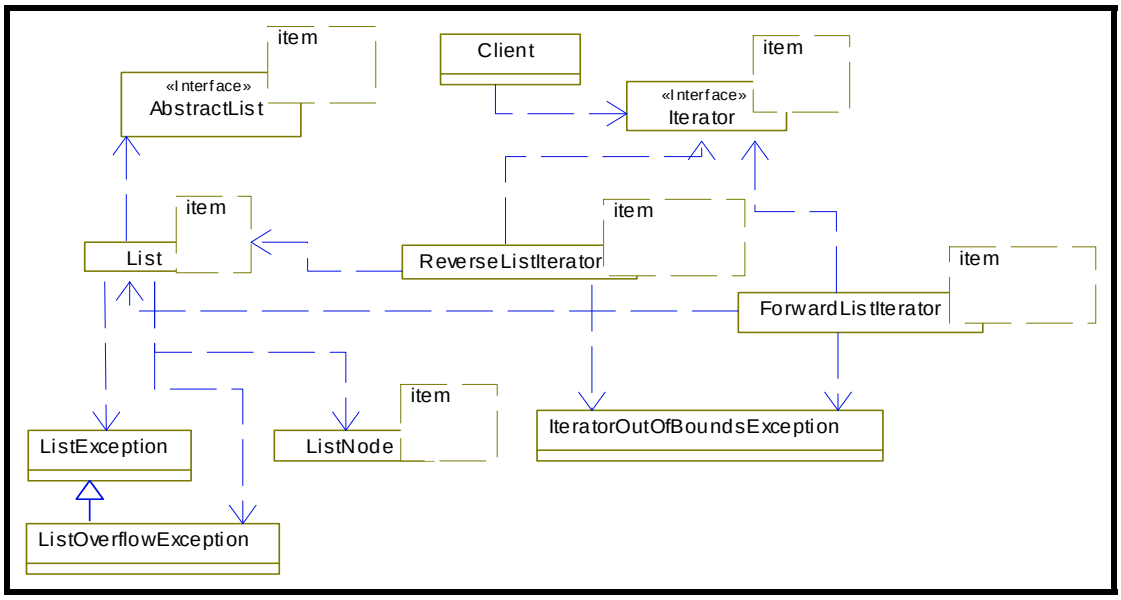


Figure 94 - Design of Iterator Pattern Case after Refactorings

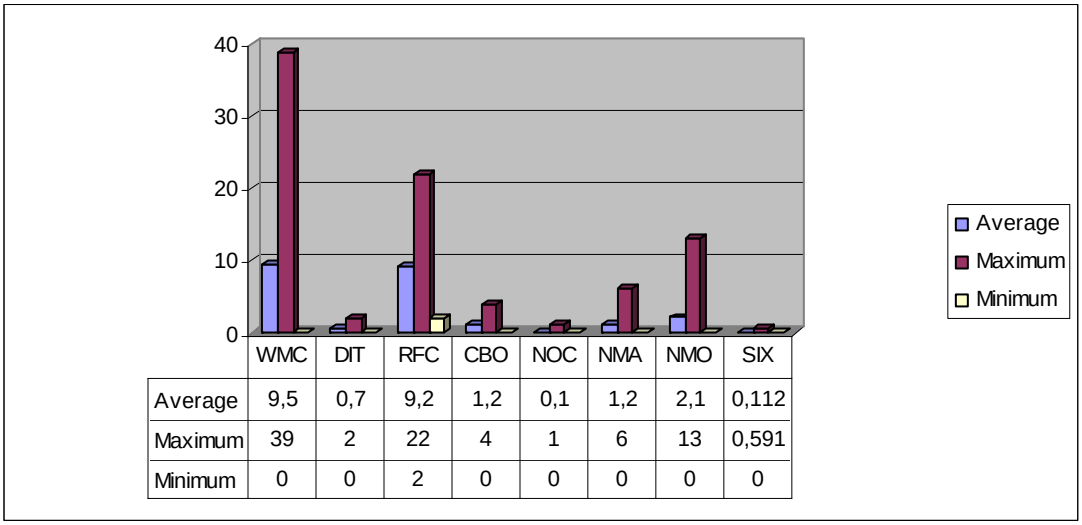
Figure 94 shows that refactoring with Iterator pattern has introduced five new classes. Two of them, called **AbstractList** and **Iterator**, are the abstract classes to isolate concrete list and iterator classes. Other two are concrete Iterator classes, which are **ReverseListIterator** and **ForwardListIterator**. These iterator classes support different traversing algorithms and simplifies list interface. The last one is an exception class, named as **IteratorOutOfBoundsException**. This class is used to throw exception when an illegal iteration performed.

3.4.13.5. Phase Five: Re-measuring software

Refactored project summary is given below:

Project Summary:

- Classes: 10
- Files: 24
- Functions: 60
- Lines: 1632



	WMC	DIT	RFC	CBO	NOC	NMA	NMO	SIX
Average	9,5	0,7	9,2	1,2	0,1	1,2	2,1	0,112
Maximum	39	2	22	4	1	6	13	0,591
Minimum	0	0	2	0	0	0	0	0

Figure 95 - Average, Maximum and Minimum Values of Thesis Metrics for Iterator Pattern Case after Refactorings

Figure 95 shows metric values after refactorings for Iterator pattern case.

3.4.13.6. Phase Six: Comparisons and Interpretations

Table 3 shows that Iterator pattern significantly reduces the complexity of client classes, if these classes support different traversing algorithms. In addition, this pattern simplifies the interface of **List** class, without creating substantial effects on WMC,

RFC, and CBO values. But, inheritance related metrics of this class has increased, because of **AbstractList** class.

Table 3 - Comparisons of Metric Values for Client and List Classes

	After Refactoring		**Before Refactoring**	
	Client	**List**	**Client**	**List**
WMC	7	39	14	39
DIT	0	1	0	0
RFC	4	22	4	22
CBO	0	4	0	3
NOC	0	0	0	0
NMA	0	6	0	0
NMO	0	13	0	0
SIX	0	0,6	0	0

In conclusion, Iterator pattern refactorings has elegantly solved the breaking encapsulation and high WMC value problems. Moreover, different traversing algorithms and container classes can be easily adapted to this refactored design and work together. This improvement increases reusability, flexibility, also limit complexity for classes that sustain different traversing strategies.

CHAPTER 4

DISCUSSION AND CONCLUSIONS

In this thesis study, software error-proneness related OO metrics have been determined and reviewed. Moreover, design patterns that affect these metrics have been found out, and their effect on metrics have been shown empirically on individual experiments.

13 different cases have been refactored with design patterns, and fluctuations on metrics have been evaluated mainly from the perspective of the software error-proneness. Also, in this thesis, [GOF98] patterns have been researched and the effects of these patterns on OO metrics have been measured with refactorings.

As stated in chapter 2, software error-proneness related OO metrics have a significant impact on software development cycle. Since, by using coupling and inheritance metrics, very accurate models can be derived to guess error-prone classes in software [BDPW98-2]. These models will obviously help developers to reduce software testing phase of the development cycle.

Design patterns, as stated before, are elegant and reusable solutions to common occurring design problems. But, this study showed that design patterns may have many effects on software quality.

In chapter 3, Power-Grab [PG02] project has shown that refactorings, which intend to reduce high coupling between classes, with design patterns are meaningful, if design pattern free source code is not deteriorated and stable in its design. This result seems to be a limitation for design pattern refactorings. Therefore, designer(s) should investigate and interpret source code before applying any design pattern, which proposes a remedy for high class coupling. Moreover, they may perform additional pre-refactorings with different methods, like eliminating repeated code segments. Consequently, Façade

pattern case has showed that refactoring with design patterns to solve high coupling problem does not work flawless under time limitations and deteriorated design.

Power-Grab Project [PG02] has been refactored with two design patterns. One of them is Façade pattern and the other is State pattern. These patterns have been implemented separately. Also, the effects of these patterns have been evaluated under different subsections. Façade pattern did not bring any improvement in Power-Grab Project. It affected error-proneness related OO metrics negatively. The reason for this result mainly originated from deteriorated OO design and class structures. However, State pattern has positively affected the design and reduced complexity in contrast of Façade pattern. Because, State pattern is not a remedy for high class coupling problem. Moreover, State pattern only influences the class that has under refactoring with this pattern.

In NetClass project [NC01], Bridge pattern has been applied. In this case, this refactoring has reduced average WMC value, which means simplification for the overall project. But, on the other hand, this pattern has slightly increased RFC, DIT, NOC metric values as a side effect.

In case 3, the effect of Mediator pattern on OO metrics has been investigated. Mediator pattern mainly affected CBO metric. This pattern has reduced average and sum values of CBO metric for the overall project. In other words, that result simply indicates that refactoring with Mediator pattern reduces the probability of error-proneness among project classes. But, as a side effect this pattern introduces new mediator class that has generally high CBO value.

In case 4, the application of Chain of Responsibility pattern has reduced WMC and CBO values. In other words, this pattern decreases high coupling and class method complexity. But, as a primary side effect, it increases RFC values of the classes, which are effected by this pattern.

In case 5, the application of Composite pattern has significantly reduced average CBO value. Also, refactoring with this pattern has reduced WMC and RFC values for respected classes. However, as a side effect, inheritance related metrics has increased slightly after the implementation of this pattern.

In case 6, the effects of two different aspects of Strategy pattern has been evaluated on synthetic code. These aspects are related with complexity and excessive subclassing. In both aspects, refactoring with Strategy pattern has brought improvements in software, also reduced complexity and inheritance related OO metrics, which are also correlated with software error-proneness.

In case 7, Template Method pattern implementation has increased all software error-proneness related metrics except for DIT and NOC. But, design became more flexible after refactorings with this pattern. However, project has been negatively influenced from the perspective of software error-proneness.

In case 8, Command pattern refactoring has increased all average metric values except for WMC. In addition, this pattern did not bring any significant improvement in classes, where this pattern has focused on. Command pattern increased overall complexity and coupling between classes. But, on the other hand, from the view of software maintainability and reusability perspective this pattern isolated functionality and invocation.

In case 9, the effects of two different aspects of Visitor pattern has been evaluated on synthetic code. These aspects are related with complexity and excessive subclassing. In the excessive subclassing related aspect, this pattern has reduced DIT and SIX metric values, but design became more complex in its structure. In other words, Visitor has increased WMC and CBO values for this aspect.

In complexity related aspect, Visitor has reduced class complexity, but increased class coupling. However, this pattern provided flexibility to the design and positively affected majority of the error-proneness related metrics.

In case 10, Observer has increased all the error-proneness metrics of the project. This result indicates that this pattern increases complexity and error-proneness probability of the classes. However, if Mediator had been implemented together with Observer for this case, increase in complexity and error-proneness would have been lower as showed in Case 3.

In case 11, Decorator pattern has been applied during refactorings and this design improvement has significantly reduced sum values of DIT, RFC, CBO, and NOC metric values.

Decorator pattern has reduced the total number of classes in the project. Therefore, there has been a boost in average class metrics. But, Decorator pattern has prevented excessive subclassing and positively effected software error-proneness.

In case 12, Memento has not altered average values of OO metrics significantly for the overall project classes. But, this pattern has improved OO design and simplified the class whose state had to be saved and restored later.

In case 13, synthetic code has been refactored with Iterator pattern. Refactorings has elegantly solved the breaking encapsulation and high WMC value problems in the design. Moreover, Iterator pattern has brought reusability, flexibility in software and limited complexity for classes that perform different traversing strategies.

In general, chapter 3 has empirically showed that some design patterns like, Mediator, State, and Chain of Responsibility have improved software quality and reduced error-proneness much more than others in the experiments performed. Also, some design

patterns like, Template Method, and Observer pattern have affected software error-proneness negatively. On the other hand, these patterns have increased reusability and analyzability of software. In brief, every design pattern affected software error-proneness related OO metrics in different levels. But, in common, they have enhanced software quality in these experiments.

A point that has to be made after having completed this study is that metric collection tools, which are available in the market lack standards. Most of them were designed as reverse engineering tools, and they do not focus on metrics. Moreover, calculation of metric values may differ for each of them.

Also, in the market, there is not any tool that interprets the source code and offers design patterns for software improvements. But, software developers use design patterns intensively in their projects, however, there is not any criteria, whether they apply the correct pattern(s) or not. It has been noticed that the market has to supply tools to overcome these design problems. Moreover, these tools will bring great assets in software development process.

In this study, every metric measurement, comparisons and interpretations have been presented over one design pattern only. As a recommendation for future work, the effect of a pattern language implementation on software may be measured and interpreted. Also, as stated before, there is no way to determine, whether software is ready for design pattern refactorings or not. A tool can be designed and implement to solve predefined problem.

Moreover, as stated in chapter 3, there is not a standard way to determine and hatch design patterns in software. Actually, designers will need similar tools in near future. Therefore, a software tool can be designed and implemented to dig and find out design patterns in software.

In addition, similar studies should be performed to observe and show the effects of design patterns on different aspects of software, like maintainability, analyzability, and so on.

Furthermore, design patterns that are mentioned in [BPD02] have been excluded from the scope of this study. But, the effects of real time design patterns on software error-proneness in particular, and software quality in general, is also an area that deserves in-depth study.

In chapter 2, possible effects of design patterns on software testing phase has been mentioned. As another future work recommendation, these effects can be measured. For example, a simple project can be refactored or developed with design patterns by a group of developers and same project can be developed without design patterns by another group of developers. After this phase, testing efforts will be measured for these projects. Consequently, comparisons will empirically reveal the effect of design patterns on software test phase.

As another point that has to be emphasized before having completed this section is that this study does not claim any generalization of the observations on the experiments. Obviously, other experiment structures and alternative refactorings are always possible, with possibly different effects. The aim has simply been to illustrate possible effects of design patterns on software error-proneness.

In conclusion, in this thesis study, the relations between software error-proneness associated metrics and design patterns have been investigated. Also, this study emphasizes the importance of design patterns, the lack of standard metric analysis tools, and the lack of standard ways for hatching design patterns.

REFERENCES

[ARJ90]: Russell J. Abbott, "Resourceful Systems for Fault Tolerance, Reliability, and Safety", ACM Computing Surveys, Vol. 22, No. 1, March 1990, pp. 35 – 68.

[AV04]: Magnus Andersson, Patrik Vestergren, "Object-Oriented Design Quality Metrics", Uppsala Master's Theses in Computer Science, 2004, ISSN 1100 – 1836.

[BBM96]: V. R. Basili, L. C. Briand, W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", IEEE Transactions on Software Engineering, Vol. 20, No. 10, October 1996, pp. 751 – 761.

[BDPW98]: Lionel C. Briand, John Daly, Victor Porter, Jürgen Wüst, "A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems", Fifth Int'l Symposium on Software Metrics, 1998, pp. 246.

[BDPW98-2]: Lionel C. Briand, John Daly, Victor Porter, Jürgen Wüst, "Predicting Fault-Prone Classes with Design Measures in Object-Oriented Systems", Proc. Ninth Int'l Symposium on Software Reliability Engineering, 1998, pp. 334.

[BF91]: Ricky W. Butler, George B. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software", IEEE Transactions on Software Engineering, Vol. 19, No. 1, January 1991, pp. 3 – 12.

[BH01]: Brian Huston, "The Effects of Design Pattern Application on Metric Scores", the Journal of Systems and Software 58, 2001, pp. 261 – 269.

[BPD02]: Bruce Powell Douglas, "Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems", Addison Wesley, 2002.

[BWIL98]: L. Briand, J. Wuest, S. Ikonomovski, H. Lounis, “A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study”, Technical Report ISERN-98-29, Int'l Software Eng. Research Network, 1998.

[BWDP00]: L. Briand, J. Wuest, J. Daly, V. Porter, “Exploring the Relationships Between Design Measures and Software Quality in Object-Oriented Systems”, J. Systems and Software, Vol. 51, 2000.

[CA77]: Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel, “A Pattern Language”, Oxford University Press, New York, 1977.

[CK94]: S. Chidamber, C. Kemerer, “A Metrics Suite for Object-Oriented Design”, IEEE Trans. Software Eng., Vol. 20, No. 6, June 1994.

[EBGR01]: Khaled El Emam, Saida Benlarbi, Nishith Goel, Shesh N. Rai, “The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics”, IEEE Trans. Software Eng., Vol. 27, No. 7, July 2001.

[FPB87]: Frederick P. Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering”, Computer, Vol. 20, No. 4, April 1987, pp. 10 – 19.

[GOF98]: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1998.

[IEEESG90]: ANSI/IEEE, “IEEE standard glossary of software engineering terminology”, IEEE Std. 729 - 1983, 1983.

[IS04]: I. Sommerville, “Software Engineering”, Addison-Wesley, 2004.

[LK94]: M. Lorenz, J. Kidd, “Object-Oriented Software Metrics”, Prentice-Hall, 1994.

[MC96]: Mel Ó Cinnéide, “Towards Automating the Introduction of the Decorator Pattern to Avoid Subclass Explosion”, Technical Report TR-97-7, University College Dublin, 1996.

[NC01]: “NetClass Project”, detailed information available at: http://sourceforge.net/projects/netclass, February 2006

[PG02]: “Power Grab Project”, detailed information available at: http://sourceforge.net/projects/power-grab, March 2006

[PSK90]: David L. Parnas, A. John van Schouwen, Shu Po Kwan, “Evaluation of Safety-Critical Software”, Communications of the ACM, Vol. 33, No. 6, June 1990, pp. 636 – 648.

[RCM00]: Robert C. Martin, “Design Principles and Design Patterns”, 2000, available at: http://www.objectmentor.com, April 2005

[RJB99]: James Rumbaugh, Ivar Jacobson, Grady Booch, “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1999.

[TKC99]: Mei-Huei Tang, Ming-Hung Kao, Mei-Hwa Chen, “An Empirical Study on Object Oriented Metrics”, Proc. Sixth Int'l Software Metrics Symposium, 1999, pp. 242 – 249.

[TTL00]: “Tau Telelogic Logiscope”, detailed information available at: http://www.telelogic.com/corp/products/logiscope/index.cfm, December 2005

[VP95]: Frans Ververs, Cornelis Pronk, “On the Interaction between Metrics and Patterns”, OOIS, 1995, pp. 303 – 314.