Empirical Quantitive Investigation of the Effect of GoF Design Patterns on the Quality of Software Systems

Somia Abufakher
Department of Software Engineering-College of Information Technology,
The World Islamic Sciences and Education University, Amman, Jordan

Abstract—**Design patterns are universal, reusable fixes for common issues in software design. They are supposed to encourage better design choices by rep-urposing already-established successful solutions, saving cost and time. The purpose of the current study is to present quantitive evidence about the expected implications on software quality by employing GoF design patterns; 10 commonly applied patterns have been empirically assessed for their impact on software quality. The evaluated patterns are: Factory Method, Prototype, Singleton, Adapter, Composite, Decorator, Observer, State, Template Method, and Proxy. The study considers software quality attributes that match the intents of the subject design patterns; these attributes are: software maintainability, testability, reusability, simple design, sensitivity to change, and error-proneness. The empirical evaluation of patterns is performed by computing 10 software quality metrics for pattern classes that have been detected in 10 real open-source projects implemented with Java. The findings reveal that the evaluated patterns promote software quality, except for the classes of Prototype, Composite, and Singleton, which were often found not cohesive.**

Keywords—Gang-of-four patterns; empirical quantitive investigation; software systems quality

I. Introduction

A design pattern is a generic, reusable procedure for an issue that often comes up at the software design phase, coding, and implementation levels [1]. In 1977, Alexander et al. introduced patterns in the discipline of architecture [2]. Gamma et al. suggested methods for using architecture patterns on software projects with object orientation in the middle of 1990s. Twenty-three design patterns were suggested; these are known as the "Gang-of-Four (GoF): Gamma, Helms, Johnson, and Vlisides". Three categories: creational, structural, and behavioral are used to group these patterns [1].

Design patterns are anticipated to offer many advantages, including: 1) encouraging better design choices, 2) facilitating communication among developers, 3) enhancing software maintainability, 4) assisting in meeting the system's non-functional needs, and 5) reducing costs, time, and effort by reusing already-proven solutions [1].

Since the proposal of GoF design patterns, researchers, empirically and analytically, have investigated the consequences of using them on several quality characteristics of software systems. According to a few of the previous results, design patterns make it more difficult to produce software that meets the necessary standards. However, according to other research, developers should employ design patterns to enhance the quality of their programs.

In this work, an empirical investigation is established and carried out to assess the impact of using 10 of the widely adopted GoF design patterns on the quality attributes of software systems. Ten (10) open-source software projects have been chosen. Next, design patterns recognition methods were used to identify the classes that are involved in each of the examined patterns in the subject programs. Finally, 10 software quality metrics were calculated for the purpose of assessing the quality characteristics of the classes included in the design patterns under consideration.

Mainly, the goal of the current study is to answer the following question:

What are the implications of GoF design patterns on the maintainability, testability, reusability, simple design, sensitivity to change, and error-proneness of software systems?

The obtained results show that in general, the classes involved in Factory Method, Adapter, Decorator, Observer, State, Template Method, and Proxy are maintainable, testable, simple, reusable, less error-prone, and less sensitive to changes occurring in other modules of the system. This conclusion is stated because the means of all computed metrics for classes involved in these patterns were found in their optimal ranges.

In contrast, the means of RFC of Prototype and Composite were found to be higher than the optimal; this indicates that a class of these two patterns will run a significant number of methods in response to an action received. Consequently, these patterns may impede achieving the expected level of software quality.

Similarly, the mean of LCOM of Singleton classes was found to be higher than the optimal range, which means that classes involved in Singleton are less cohesive, which may impede the desirable software quality.

This is how the remainder of the study is structured: Related works are presented in Section II. The study's methodology and procedures are shown in Section III. The results collected are presented in Section IV. Section V wraps up the work and provides a summary and the recommendations.

II. Related Work

Despite being proposed and published in the middle of the 1990s, GoF design patterns continue to serve as a guide for software developers and designers. Therefore, researchers have evaluated the consequences of applying design patterns as reusable strategies on software quality characteristics. In the following, some of the works in this domain are briefly mentioned:

Sousa et al. (2019) present an empirical study to check whether the deployment of design patterns may reduce bad smells in code. According to the results, using design patterns does not always stop code smells from occurring. Some patterns, like Adapter, Proxy, and State, commonly co-occur with code smells and must therefore be implemented carefully. While others, like Composite, Factory Method, and Singleton, can support high-quality system design. Moreover, the researchers state that the main reasons why issues could arise in the code are poor planning and improper usage of design patterns [3].

In addition, Feitosa et al. (2019) investigate whether using patterns ensures acceptable coding practices. The researchers use static analysis to study the connection between 12 of GoF design patterns and infringements of best practices pertaining to source code accuracy, performance, and security. The results indicate that classes that don't follow patterns are more likely to break security, performance, and accuracy guidelines. In addition, the study concludes that patterns with more intricate structures (like Decorator) and pattern instances that are more prone to change (like subclasses) are more likely to be linked to a higher number of violations [4].

Further, Bontchev and Milanova (2020) investigate the usability of GoF patterns on the testability, maintainability, extendibility, readability, reliability, and performance efficiency of programs. The researchers use a survey with 82 object-oriented professionals. According to the findings, Memento and Flyweight are the least well-known and least practical patterns, whereas Singleton and Factory Method are the most well-known and practical. Also, the responses indicate that the least common design patterns are behavioral ones. Even so, seasoned responders think that design patterns greatly enhance software quality, with differing impacts on different quality criteria [5].

Likewise, Qamar and Malik (2020) investigate the implications of GoF patterns on the complexity and size of software projects. The researchers conducted a comparison between the before and after editions of program sets developed with and without utilizing design patterns. According to the comparison's findings, programs created with design patterns exhibited lower cyclomatic complexity than programs produced without design patterns for the majority of GoF design patterns. However, the study found that usage of design patterns resulted in an increase in the software size, number of classes, and CK metrics [6].

As well, Kurmangali et al. (2022) present an empirical investigation to assess the effectiveness of Decorator and Abstract Factory on software maintainability. The following C&K metrics were computed: WMC, DIT, NOC, CBO, RFC, and LCOM, and compared for patterns' classes. According to the study, deployment design patterns can boost the maintainability of software. Moreover, Decorator showed a little superior performance than other patterns [7].

More recently, Hsu et al. (2024) studied the outcomes of design patterns on the quality of software systems during their evolution by analyzing the complexity of systems and the changes in dependency relationships among components. The study considers three of GoF patterns: i.e., Factory Method, Decorator, and Observer. According to the findings, design patterns successfully lessen dependencies created as a system evolves, reducing system complexity and improving overall quality [8].

Furthermore, Asaad and Avksentieva (2025) assess GoF design patterns that are detected in GitHub repositories on software engineering procedures. The research's findings demonstrate that GoF patterns are actively applied in actual software development, and they significantly raise software quality characteristics [9].

Additionally, Elish (2025) examines the potential effects of design patterns on software maintainability in a quantifiable way. The Maintainability Index (MI), which is a quantitative indicator that aggregates size, complexity, and other code parameters into a single number, was measured in order to achieve the goal. The study's conclusion shows that using design patterns can simplify and strengthen a system's structure, boosting software maintainability. However, patterns may differ from one to another; in comparison to behavioral design patterns, creational and structural design patterns have a better maintainability score. Moreover, according to the maintainability index, the top three patterns are Builder, State, and Adapter [10].

Finally, Noureddine and Goaer (2025) conducted experimental research to assess the influence of design patterns on the energy consumption of software. The study considers the 23 GoF patterns and multiple platforms, operating systems, programming languages, and compilers. The researchers found that all GoF design patterns are highly energy-efficient [11].

Despite the widespread recognition of Gang of Four (GoF) design patterns as a means of enhancing software design quality, the majority of current research concentrates on theoretical advantages and qualitative evaluations rather than offering empirical, quantitative proof of their actual influence on software quality features.

The majority of earlier research lacks thorough analysis across many systems or contexts and instead looks at design patterns in isolation or relies on small-scale case studies. Furthermore, little research has been done on how particular GoF design patterns affect quantifiable software quality criteria, including maintainability, testability, reusability, simple design, sensitivity to change, and error-proneness. Moreover, there is no consensus on the findings for the same pattern and the same quality attribute (but using different data sets).

Additionally, the literature currently in publication does not adequately address the comparative effects of various GoF pattern categories (structural, behavioral, and creational) on software quality.

The current work contributes to the empirical evaluation of the implications of design patterns on the quality of software systems by performing a comprehensive investigation of 10 of the most commonly used design patterns and all related quality attributes (i.e., quality attributes that are, in theory, consequents of using the evaluated patterns).

III. Methodology

In order to conduct the empirical evaluation of design patterns on software quality, this section outlines the methodology and the procedures used to collect and evaluate the data. First, the most popular GoF design patterns were chosen for assessment. In the second step, the quality attributes that match the intents of the structure of the selected patterns were determined. Then, to measure the quality attributes of pattern classes, 10 software quality metrics were determined, and how their values affect software quality was analyzed. After that, 10 open source Java projects were selected, and the classes of the considered design patterns were detected using design pattern mining tools. Then the software quality metrics were computed for each of the detected pattern classes. Finally, the results were studied and analyzed to conclude the implications of the considered patterns on the related quality attributes of object-oriented software. These steps are illustrated in more detail in the upcoming subsections.

A. Design Patterns

In this study, 10 of the GoF design patterns are evaluated, these are: Factory Method, Prototype, Singleton, Adapter, Composite, Decorator, Observer, State, Template Method, and Proxy. These patterns are selected because they have the following characteristics: 1) they belong to the three categories classified in GoF: creational, structural and behavioral, 2) they are widely used in software design, 3) they are from the most evaluated patterns in the previous empirical studies 4) their instances are the most detected by the design patterns mining tools used in this study. In the following, the main intent of the selected patterns is briefly described:

Creational patterns are used to create objects and abstract the creation process to facilitate the implementation process and create an independent system [1]. In this study, the following creational patterns are evaluated:

- Factory Method: A pattern for building instances that let subclasses choose which class to generate at runtime, by specifying a common interface [1].
- Prototype: An optimization pattern that uses a prototypical implementation to identify the kind of created instances [1].
- Singleton: A pattern used to restrict a class to only one instance, making this instance a shared resource in the system by providing global access to it [1].

Structural patterns are used to create simple communication among classes and their instances; they are concerned with how the modules of the system relate to each other to form larger structures [1]. In this study, the following structural patterns are evaluated:

- Adapter: A pattern used when two classes belong to distinct interfaces wanted to work with each other without modifying any of their code, but by using inheritance to get the needed implementation [1].
- Composite: A pattern creates a tree structure to describe part-whole hierarchies by treating a collection of related instances as a single one [1].
- Decorator: A pattern that optimizes a class instance by dynamically adding new responsibilities without changing how other instances in the same class behave [1].
- Proxy: A pattern used to control the access of a class instance by providing an alternative way [1].

Behavioral patterns are communication patterns; they are concerned with algorithms and determine the functions of the objects in the system [1]. In this study, the following behavioral patterns are evaluated:

- Observer: In this pattern, a class instance is created called "subject" to define its dependents, so that when a change occurs on the subject, it automatically updates and informs all of its dependents [1].
- State: A pattern that enables an entity to alter how it acts in response to changes in its internal state [1].
- Template Method: A pattern used to define the algorithm of the system in one method called "Template Method"; it delays some steps to subclasses without changing the structure of the program [1].

B. Software Quality Attributes

According to previous studies, software quality has been greatly affected by GoF design patterns. In this study, six quality attributes are considered; these are: maintainability, testability, reusability, simple design, sensitivity to change, and error-proneness. These attributes are considered because they match the intents of the structure of the assessed design patterns, and they are among the consequences of using these patterns on object-oriented software. In the following, each software quality attribute is briefly described:

- Software maintainability: The average of effort and time required to correct, adapt, or improve the software through its life cycle until postmortem [12].
- Software testability: The effort required to complete the testing process for every module in the system [12].
- Software reusability: Reusable classes can be used in other parts of the system or in another one [12].
- Software simplicity: In this study, the pattern classes are evaluated to determine whether their design is complex; there are numerous complex inter-relationships between pattern classes and the other modules of the system.
- Sensitivity to change: A class is considered sensitive if it is affected by changes that occur in other classes of the system, and vice versa [12].
- Error-proneness: The number of faults may be discovered in the classes of a system [13].

C. Software Quality Metrics

To numerically evaluate design patterns, 10 software quality metrics were computed for each class that is involved in any of the considered patterns. In the following, the used metrics are illustrated, in addition to the optimal range of each one, and how these metrics may affect the software quality attributes of software systems.

The Chidamber and Kemerer measures set, or C&K measures, is an approach used to assess the quality of object-oriented software systems [13]. C&K set includes six types of metrics that could be computed for a single class in a software project. In the following, each metric is briefly illustrated:

WMC, which is "Weighted Methods per Class," is measured by counting how many methods are defined in a class. A class should have the lowest possible count of methods. A lot of methods in one class have a negative impact on software quality. Chidamber and Kemerer mentioned that the good class has 20 to 50 methods [13].

DIT, which is "Depth of Inheritance Tree," represents the longest sequence of inheritance between a parent class and its offspring. It calculates a class's place in the inheritance tree [12]. Misra and Bhavsar found that high DIT promotes method reusability, but on the other hand, it makes the class design more complex and may increase the faults in the system [23]. Chidamber and Kemerer mentioned that DIT should not be more than 5 [13].

NOC, which is "Number of Children," measures the count of subclasses directly derived from a base class. There is a close relation between NOC and DIT; more levels of inheritance may increase DIT and reduce NOC. NOC promotes reusability, and unlike the other five metrics, it reduces the number of faults in classes; however, the base classes may need more testing effort [12]. There is no optimal NOC determined in the literature, but researchers agreed that it should not be high (e.g., Chidamber and Kemerer [13]).

CBO, which is "Coupling between Objects," indicates how many classes a class is associated with. CBO should be kept down, because it impedes reusability and maintainability, and it produces more faulty classes [13]. Sahraoui et al. recommended that the CBO should be no more than fourteen [14].

RFC, which is "Response set of a Class," is the collection of methods that are likely to be used when a class sends an action. High RFC is undesirable [12]. Misra and Bhavsar found that it increases the design complexity and the faults in the system, making it less understandable, and thus the system will require more testing effort [15].

LCOM, which is "Lack of Cohesion of a class," is a cohesion metric; it measures whether a class includes any related methods performing one function. Cohesive design is desirable because it promotes reusability and makes the system easier to maintain. Therefore, LCOM should be as low as possible; when it reaches 0, the class is considered very cohesive [13].

However, researchers suggested optimal ranges only for WMC, DIT, and CBO. For NOC, RFC, and LCOM, they just stated that the values of these metrics should not be high. To minimize this conflict, a study conducted by NASA in 2003 is considered; the study reports the optimal means for C&K metrics for a high-quality Java system. It stated that the optimal mean of RFC is less than or equal to 43.84 and the optimal mean of LCOM is less than or equal to 78.34 [16]. So, these two values are used as references in the current evaluation. Table I summarizes the desirable values of C&K metrics in object-oriented design according to [13] and [16].

TABLE I. THE OPTIMAL USE OF C&K METRICS

Metric	Optimal Use
WMC	< 50
DIT	<= 5
NOC	Not high
CBO	<= 14
RFC	<= 43.84
LCOM	<= 78.34

For more clarification about the impact of C&K metrics on object-oriented software, Table II presents some empirical studies from the literature that evaluate the impact of the increasing values of C&K metrics on quality attributes of software.

Along with the C&K metrics, the following software quality metrics are also calculated to improve the correctness of the results about the evaluation of design patterns:

LOC, which is "Number of Lines of Code," is a software metric used to determine the size of a whole system or just one module in it. Actual LOC is measured by computing the number of lines of code that contain executable or declarative statements; comments and whitespaces should be excluded.

TABLE II. THE IMPACT OF HIGH VALUES OF C&K METRICS ON SOME SOFTWARE QUALITY ATTRIBUTES

Increasing metric	Software maintainability	# Faults	Software testability	Design complexity	Software reusability	Sensitivity to change
WMC	Impede	Enhance	Enhance	Enhance	Impede	Enhance
DIT	Impede	Enhance	Enhance	Enhance	Enhance	Enhance
NOC	Impede	Impede	Impede	Enhance	Enhance	Enhance
CBO	Impede	Enhance	Enhance	Enhance	Impede	Enhance
RFC	Impede	Enhance	Enhance	Enhance	Impede	Enhance
LCOM	Impede	Enhance	Enhance	Enhance	Impede	Enhance
Reference	[13] [15] [17] [18]	[13] [15] [19] [20] [21] [22] [23]	[15] [24] [25] [26]	[15] [27]	[13] [15] [27]	[27]

Researchers empirically studied LOC in several ways; for example, Gyimothy et al. [19] and Yu et al. [28] evaluated the relationship between LOC and the count of defects discovered in the system; they concluded that high LOC increases the number of faults in classes. Li and Henry [17] evaluate the impact of high LOC on software maintainability; they found that as LOC increases, developers will need more effort to maintain the system. In addition, Bruntink and Van Deursen [24] concluded that large systems require more testing effort. All those researchers stated that LOC can be used as a reliable measure of software complexity. It is suggested that a good Java class should not have more than 1500 lines of code [28].

NIM, which is "Number of Instance Methods": instance methods are the set of methods that are created in a class, and only objects in that class can communicate with them [29].

NIV, which is "Number of Instance Variables": instance variables are the set of variables that are created in a class, and only objects in that class can communicate with them [29].

Lorenz and Kidd suggested NIM and NIV as measures of class size [29]. Several researchers evaluated the effectiveness of these two metrics in object-oriented design. For example, it was concluded that there is a strong correlation between NIM and NIV with software maintainability, where a large size class impedes achieving a high level of maintainability [30].

McCabe Cyclomatic Complexity is a software metric that Thomas J. McCabe developed to assess how complex a software component's design is. Cyclomatic complexity may be computed for the whole system using a control flow graph or for one class of the system by counting the independent paths in it; when this number is high, it means that the program should be restructured into simpler modules. As software complexity increases, the testability decreases, and the program will be more difficult to understand [31]. In this study, McCabe complexity of pattern classes is computed by computing the average of all cyclomatic complexities of nested methods of the classes that are involved in any of the considered patterns.

To compute the previously mentioned metrics for each class involved in each of the considered patterns, the Understand^{TM} tool is used, which is a commercial reverse engineering tool that statistically extracts information from Java source code.

D. Benchmarks

In order to perform the current evaluation, 10 open source software projects were selected. All the considered projects are implemented with the Java programming language, applying object-oriented principles. Many GoF design patterns are used in their code design. The size of the selected projects is medium to large. Table III presents the selected projects and summarizes their main characteristics.

As shown in the table, Apache Tomcat and Fitnesse have the most detected pattern classes, where 487 pattern classes are detected in Apache Tomcat and 333 pattern classes are detected in Fitnesse. The lowest number of pattern classes is detected in Java Mail API; only 71 classes.

Fig. 1 presents the number of pattern classes found in each software project.

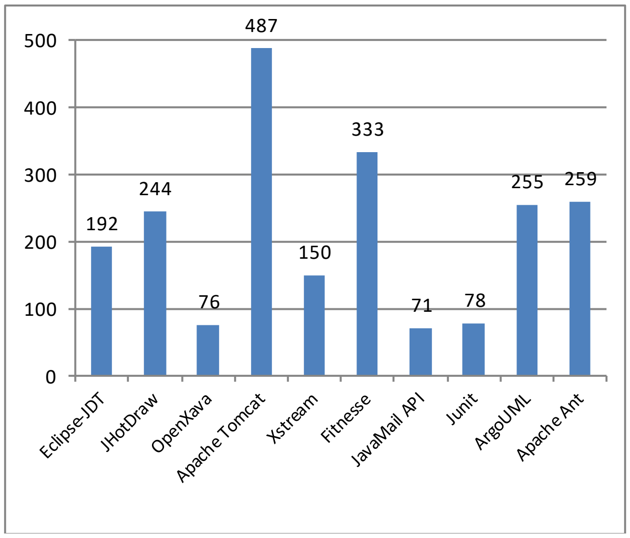


Fig. 1. Number of pattern classes detected in each benchmark.

TABLE III. CHARACTERISTICS OF THE BENCHMARKS USED IN THE STUDY

Software Projects	# Classes	# Files	# Units	# LOC	# Patterns Classes	Patterns classes ÷ All classes
Eclipse-JDT	1105	793	11535	241326	192	17%
JHotDraw	480	309	3781	56990	244	51%
OpenXava	1587	1542	12850	154239	76	5%
Apache Tomcat	3009	1807	23430	464352	487	16%
Xstream	1406	653	4878	71150	150	11%
Fitnesse	1487	1035	9002	90149	333	22%
JMail API	265	192	1983	52142	71	27%
JUnit	181	89	788	7230	78	43%
ArgoUML	1430	1226	10563	240755	255	18%
Apache Ant	1300	929	10112	200178	259	20%

On the other hand, JHotDraw has the largest percentage of design pattern classes; 51% of its classes have pattern instances in their design. JUnit comes after; 43% of its classes contain pattern instances. Fig. 2 presents the percentage of pattern classes to all the project's classes.

The classes of design patterns were detected in the selected software projects using Tsantalis approach [32] and PINOT [33] detection tools. These tools detected many instances of patterns in the source code of each of the software projects used, as presented in Table IV.

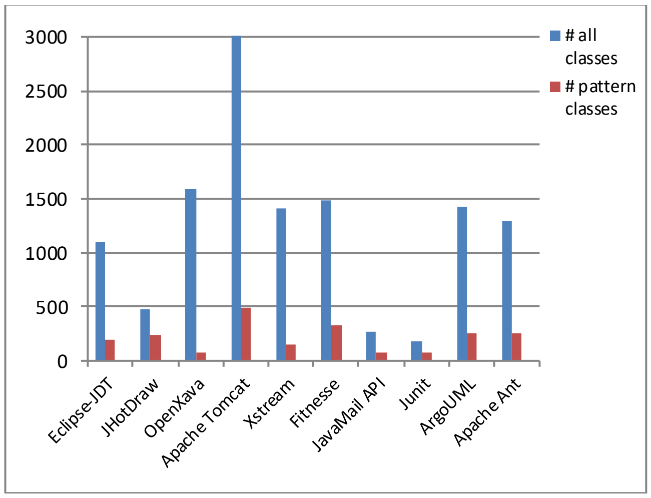


Fig. 2. The percentage of pattern classes to all classes in each benchmark.

TABLE IV. NUMBER OF CLASSES INVOLVED IN THE INSTANCES OF DESIGN PATTERNS: FACTORY METHOD (FM), PROTOTYPE (P), SINGLETON (S), ADAPTER (A), COMPOSITE (C), DECORATOR (D), PROXY (X), OBSERVER (O), STATE (ST), AND TEMPLATE METHOD (TM)

Software Projects	**# detected classes that are involved in design patterns**									
	FM	**P**	**S**	**A**	**C**	**D**	**X**	**O**	**St**	**TM**
Eclipse-JDT	2	4	10	111	0	0	0	19	31	15
JHotDraw	1	14	9	77	5	8	0	3	114	13
OpenXava	1	3	25	16	0	1	4	0	13	13
Apache Tomcat	8	1	58	181	5	19	5	17	142	50
Xstream	2	0	2	71	0	10	5	0	50	10
Fitnesse	12	0	36	125	3	17	8	0	107	25
JMail API	4	6	9	27	0	2	0	2	12	9
JUnit	0	0	0	18	2	2	0	3	8	45
ArgoUML	5	5	149	27	2	7	4	1	28	27
Apache Ant	22	15	3	71	30	8	28	21	40	21
Total	**57**	**48**	**301**	**724**	**47**	**74**	**54**	**66**	**545**	**228**

As the table shows, some patterns are used more than others in the design of the software projects; Adapter and State are the most used patterns, while Composite and Prototype are the least used patterns.

IV. RESULTS

After detecting the classes that are involved in design patterns in the benchmarks used in this study, the extent to which the detected classes promote software quality is determined by computing 10 metrics for each class. The obtained values are presented by scatter diagrams in the Appendix. After that, the means of the obtained values for each metric are computed and analyzed to determine whether this mean is optimal and how it may affect the considered quality attributes of software. Table V summarizes the means of all the obtained results.

By analyzing the results presented in Table V and the figures in the Appendix, the following can be summarized about the evaluated design patterns:

First: The impact of creational design patterns on software quality

The means of the 10 computed metrics for the 57 classes of the Factory Method pattern are in the optimal ranges. This is expected because this pattern is proposed to define and maintain the complex relationships among objects, which facilitate the missions of subclasses, making the process of creating objects inside a class more flexible [1].

The means of the computed software metrics for the 48 classes of Prototype are in the optimal ranges, except the mean of RFC, which is 65, i.e., higher than the optimal. This indicates that a Prototype class will most likely run a large number of methods, which makes this pattern less cohesive. This negative result is not expected; Prototype is considered an optimization pattern proposed to let developers create new objects by cloning the prototype instance [1]; it is anticipated that it will result in fewer classes inside the system by creating new objects without using inheritance. To the best of the researcher's knowledge, there is no clear justification for the high RFC of Prototype!

The means of computed software metrics for the 301 classes of Singleton are in the optimal ranges, except the mean of LCOM, which is 82, i.e., higher than the optimal. This means that Singleton classes are not cohesive. This is expected because this pattern is used as a shared resource object that has global access to it [1].

TABLE V. SUMMARY OF THE OBTAINED MEANS OF THE COMPUTED SOFTWARE QUALITY METRICS FOR DESIGN PATTERNS CLASSES WHERE: FACTORY METHOD (FM), PROTOTYPE (P), SINGLETON (S), ADAPTER (A), COMPOSITE (C), DECORATOR (D), PROXY (X), OBSERVER (O), STATE (ST) AND TEMPLATE METHOD (TM)

Metrics	**The means of metrics for pattern classes**									
	FM	**P**	**S**	**A**	**C**	**D**	**X**	**O**	**St**	**TM**
WMC	21	32	9	17	18	18	16	22	18	18
DIT	2	2	2	2	3	2	2	2	2	2
NOC	1	4	0	1	5	3	0	4	1	4
CBO	7	11	4	8	7	6	9	9	10	6
RFC	30	**65**	25	38	**49**	24	35	30	32	30
LCOM	68	71	**82**	48	66	49	66	55	59	6
LOC	176	395	87	193	205	146	176	255	205	164
NIM	20	31	8	16	17	18	16	21	18	17
NIV	6	8	2	5	7	4	7	7	6	5
Complexity	2	2	2	2	2	1	2	2	2	2

In short, Factory Method can be safely considered maintainable, less error-prone, testable, promotes simple design, reusable, and less sensitive to changes occurring in the system. But the other creational patterns, i.e., Prototype and Singleton classes, may be difficult to maintain, less reusable, less testable, complex in design, error-prone, and sensitive to changes occurring in the system.

Second: The impact of structural design patterns on software quality.

The means of the 10 computed software metrics for the 724 classes of Adapter are in the optimal ranges. An adapter is proposed to be used when two classes belonging to incompatible interfaces want to work with each other without modifying any of their code, but by inheriting the needed interface and implementation [1]. The positive impact of Adapter classes will be achieved if it works with a reasonable number of Adaptees. Otherwise, high inheritance may cause problems in the system.

The means of the 10 computed software metrics for the 47 classes of Composite are in the optimal ranges, except the mean of RFC, which is 49, i.e., higher than the optimal. Since this pattern deals with a collection of related objects handled as one instance of an object [1], the negative outcome is to be expected. Each object in this group mostly belongs to a different class; when an action occurs on one, the rest will respond in the same way, which expands the count of executed methods in response to a Composite class.

The means of the 10 computed software metrics for the 74 classes of Decorator are in the optimal range. This is to be expected since this pattern is used to improve an object without changing how other objects behave [1]. So Decorator is more flexible than subclasses for adding new functionalities; it avoids using static inheritance, which causes many problems in the program, achieving the principle of "open for extension, closed for modification".

The means of the 10 computed software metrics for the 54 classes of Proxy are in the optimal ranges. This is expected because a proxy is used to control the access of an object by providing a virtual alternative to the target object [1].

Thus, the structural patterns: Adapter, Decorator, and proxy may be safely considered maintainable, less error-prone, testable, promote simple design, reusable, and less sensitive to changes occurring in the system. While about Composite, the opposite can be stated; this pattern may impede software quality unless users pay attention to prevent the possible high coupling of its classes.

Third: The impact of behavioral design patterns on software quality

The means of the 10 computed software metrics for the 66 classes of Observer are in the optimal ranges. The results about Observer classes are not expected, because in Observer, when a change occurs on the subject, all of its dependents are informed and modified automatically [1]. Observers do not know about each other's behavior, so unexpected updates may occur in the system, which means that Observer classes are expected to be sensitive to changes that negatively affect the quality attributes of the software systems.

The means of the 10 computed software metrics for the 545 classes of State are in the optimal ranges. Given that this pattern enables an object to automatically adjust its behavior when its internal state changes—the object just changes its class [1]—this is to be expected. Thus, the State does not have consequences on the other modules of the system.

The means of the 10 computed software metrics for the 228 classes of Template Method are in the optimal range. This is expected because this pattern is used to define the structure of the program in one method without changing the structure of the program [1]. Thus, Template Method supports reusing the code and makes full control over the code structure without modifying it and without using inheritance relations.

Thus, the behavioral patterns which are evaluated in this study are maintainable, less error-prone, testable, promote simple design, reusable, and less sensitive to changes occurring in the system.

The results obtained in this study are highly consistent with the previous ones in the domain of evaluating the implications of applying design patterns on the quality of software systems; where the previous studies found that GoF design patterns are significantly supportive of software quality [3], [4], [5], [7], [8], [9], [10], [11]. However, the study of Qamar and Malik (2020) found that for the majority of the 23 GoF design patterns, deployment of design patterns may expand the size of software, number of classes, and CK metrics, while programs created with design patterns exhibited lower cyclomatic complexity [6].

V. Conclusion

In this research, the consequences of 10 of GoF design patterns on the quality of software are empirically investigated. The evaluated patterns are: Factory Method, Prototype, Singleton, Adapter, Composite, Decorator, Observer, State, Template Method, and Proxy. The study is performed by detecting the patterns' classes in 10 real software projects, then computing 10 software quality metrics for each detected class using a reverse engineering tool, and determining how the abnormal values of the computed metrics may affect software quality.

The results do show that, except for three patterns (Prototype, Composite, and Singleton), design patterns in general promote the quality attributes considered in this study, and to use them safely, developers are recommended to take into consideration the following:

When using Prototype or Composite in software design, developers should ensure that the number of executed methods resulting from an action taken by a class of any of these patterns is as low as possible. In other words, developers should pay attention to designing a cohesive Prototype or Composite class to limit the possible negative impact of these two patterns on software quality.

When designing a Singleton class, developers should pay attention to limiting the general access to the single object of it as much as possible. Consequently, limit the low cohesion of Singleton classes, which may impede achieving the required level of software quality.

The results obtained about the evaluated design patterns, whether they are positive or negative, are not surprising because they match the structure and the intent of them, expect the results obtained about Observer and Prototype; where, based on the structure and intent of these two patterns, Observer is expected to impede software quality and Prototype is expected to improve it, but the opposite is obtained.

In this work, the internal validity is about the extent to which the increase or decrease in software quality attributes is caused by design pattern classes, not by other factors, e.g., other classes of the projects or the experience of developers who designed the projects.

To minimize, as much as possible, this threat: 1) Only the classes that contain any of the considered patterns in their design are evaluated, eliminating the classes that do not. 2) The evaluation includes a large number of classes, i.e., 2198 have different sizes and selected from 10 Java software projects.

The findings of the current work are not claimed to be generalized unless the following circumstances are taken into account: 1) The results are obtained from software projects written only in the Java programming language. 2) The classes of design patterns were detected using Tsantalis and PINOT detection tools. 3) The software quality metrics for design patterns were obtained using the Understand^{TM} tool. The obtained values are related to the precision of the used tools.

In the future, further investigation is required to evaluate all GoF design patterns; where only 10 of them are evaluated in this research due to the difficulty of detecting all patterns in the software projects because of the limitations of the existing design patterns detection tools, and due to patterns frequency; where some of patterns are widely used by developers more than others in software projects. Additionally, the evaluation of design patterns may be applied following other possible empirical approaches. Finally, Prototype, Singleton, Composite, and Observer have to be evaluated with more concentration in order to state a safe conclusion about their effect on software quality.

REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "Design Patterns: Elements of Reusable Object Oriented Software", Addison-Wesley, 1st edition, 1994.

[2] Alexander, C., Ishikawa, S., and Silverstein, M., "A Pattern language: Town, Buildings, Construction". Oxford University Press, New York 1977.

[3] Sousa, B. L., Bigonha, M. A., & Ferreira, K. A., "An exploratory study on cooccurrence of design patterns and bad smells using software metrics", Software: Practice and Experience, vol. 49(7), pp. 1079-1113, 2019.

[4] Feitosa, D., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A., & Nakagawa, E. Y., "What can violations of good practices tell about the relationship between GoF patterns and run-time quality attributes?", Information and Software Technology, vol. 105, pp. 1-16, 2019.

[5] Bontchev, B., & Milanova, E., "On the usability of object-oriented design patterns for a better software quality", Cybernetics and Information Technologies, vol. 20(4), pp. 36-54, 2020.

[6] Qamar, N., & Malik, A. A., "Impact of design patterns on software complexity and size", Mehran University Research Journal Of Engineering & Technology, vol. 39(2), pp. 342-352, 2020.

[7] Kurmangali, A., Rana, M. E., & Ab Rahman, W. N. W., "Impact of Abstract Factory and Decorator Design Patterns on Software Maintainability: Empirical Evaluation using CK Metrics". In International Conference on Decision Aid Sciences and Applications (dasa), IEEE, pp. 517-522, 2022.

[8] Hsu, K. H., Szu-Tu, H. C., & Tsai, C. H., "Assessing System Quality Changes during Software Evolution: The Impact of Design Patterns Explored via Dependency Analysis", Electronics, vol. 13(8), pp. 1444, 2024.

[9] Asaad, J., & Avksentieva, E., "Assessing the Impact of Gof Design Patterns on Software Engineering Practices". In International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), IEEE, pp. 930-934, 2025.

[10] Elish, M. O., "An Empirical Study on Maintainability Index of Software Design Patterns", In IEEE/ACIS 23rd International Conference on Software Engineering Research, Management and Applications (SERA), IEEE, pp. 438-443, 2025.

[11] Noureddine, A., & Le Goaer, O., "Investigating the Impact of Software Design Patterns on Energy Consumption", In 22nd International Conference on Software Architecture (ICSA), IEEE, pp. 153-163, 2025.

[12] Bourque, P. and Fairley, R. E., "Guide to the Software Engineering Body of Knowledge (SWEBOK (R)", Version 3.0., IEEE Computer Society Press, 2014.

[13] Chidamber, S. R. and Kemerer, C. F., "Towards a metrics suite for object oriented design", In Conference Proceedings for Object Oriented Programming: Systems, Languages and Applications, OOPSLA'91, ACM, vol. 26 (11), pp. 197-211, 1991.

[14] Sahraoui, H., Godin, R. and Miceli, T., "Can metrics help to bridge the gap between the improvement of OO design quality and its automation?". In International Conference Proceedings on Software Maintenance, IEEE, pp. 154-162, 2000.

[15] Misra, S. C. and Bhavsar, V. C., "Relationships between selected software measures and latent bug-density: Guidelines for improving quality", In Computational Science and Its Applications—ICCSA, Springer Berlin Heidelberg, pp. 724-732, 2003.

[16] Laing, V. and Coleman, C., "Principal Components of Orthogonal Object Oriented Metrics", White Paper Analyzing Results of NASA Object Oriented Data, 29 May 2003.

[17] Li, W. and Henry, S., "Object oriented metrics that predict maintainability", Journal of systems and software, vol. 23(2), pp. 111-122, 1993.

[18] Chidamber, S. R., Darcy, D. P. and Kemerer, C. F., "Managerial use of metrics for object oriented software: An exploratory analysis", IEEE Transactions on Software Engineering, vol. 24(8), pp. 629-639, 1998.

[19] Gyimothy, T., Ferenc, R. and Siket, I., "Empirical validation of object oriented metrics on open source software for fault prediction", IEEE Transactions on Software Engineering, vol. 31(10), pp. 897-910, 2005.

[20] El Emam, K., Melo, W. and Machado, J. C., "The prediction of faulty classes using object oriented design metrics", Journal of Systems and Software, vol. 56(1), pp. 63-75, 2001.

[21] Basili, V. R., Briand, L. C. and Melo, W. L., "A validation of object oriented design metrics as quality indicators", IEEE Transactions on Software Engineering, vol. 22(10), pp. 751-761, 1996.

[22] Olague, H. M., Etzkorn, L. H., Gholston, S. and Quattlebaum, S., "Empirical validation of three software metrics suites to predict fault-proneness of object oriented classes developed using highly iterative or agile software development processes", IEEE Transactions on Software Engineering, vol, 33(6), pp. 402-419, 2007.

[23] Subramanyam, R. and Krishnan, M. S., "Empirical analysis of C&K metrics for object oriented design complexity: Implications for software defects", IEEE Transactions on Software Engineering, vol. 29(4), pp. 297-310, 2003.

[24] Bruntink, M. and Van Deursen, A., "Predicting class testability using object oriented metrics", In Fourth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 136-145, 2004.

[25] Badri, L., Badri, M. and Toure, F., "An empirical analysis of lack of cohesion metrics for predicting testability of classes", International Journal of Software Engineering and Its Applications, vol. 5(2), pp. 69-85, 2011.

[26] Badri, M. and Toure, F., "Empirical analysis of object oriented design metrics for predicting unit testing effort of classes", Journal of Software Engineering and Applications, vol. 5, pp. 513-526, 2012.

[27] Chidamber, S. R. and Kemerer, C. F., "A metrics suite for object oriented design", IEEE Transactions on Software Engineering, vol. 20(6), pp. 476-493, 1994.

[28] Yu, P., Systa, T. and Müller, H., "Predicting fault-proneness using OO metrics: An industrial case study", In Sixth IEEE European Conference Proceedings on Software Maintenance and Reengineering, pp. 99-107, 2002.

[29] Lorenz, M. and Kidd, J., "Object oriented software metrics: a practical guide", Prentice Hall Object Oriented Series, Englewood Cliffs. N.J, 1994.

[30] Dagpinar, M. and Jahnke, J. H., "Predicting maintainability with object oriented metrics-an empirical comparison", In Proceedings of the 10th Working Conference on Reverse Engineering (WCRE), IEEE Computer Society, pp. 155-164, 2003.

[31] McCabe, T. J., "A complexity measure", IEEE Transactions on Software Engineering, vol. 4, pp. 308-320, 1976.

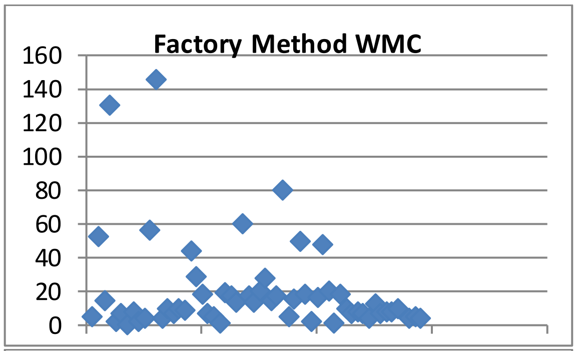
[32] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. and Halkidis, S. T., "Design pattern detection using similarity scoring", IEEE Transactions on Software Engineering, vol. 32(11), pp. 896-909, 2006.

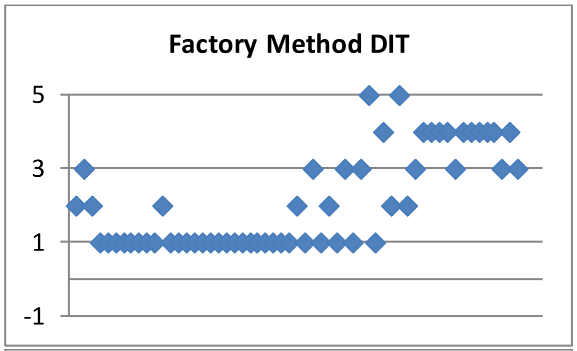
[33] Shi, N. and Olsson, R., "Reverse engineering of design patterns from java source code", In 21st IEEE/ACM International Conference on Automated Software Engineering, ASE'06, pp. 123-134, 2006.

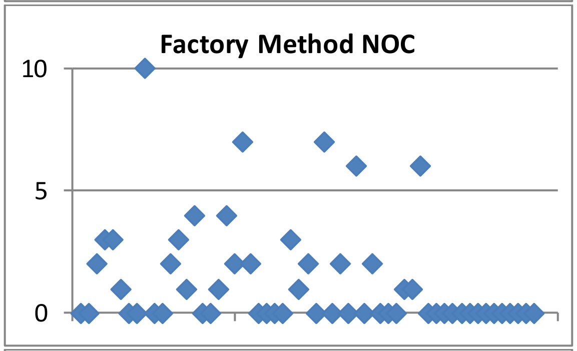
Appendix

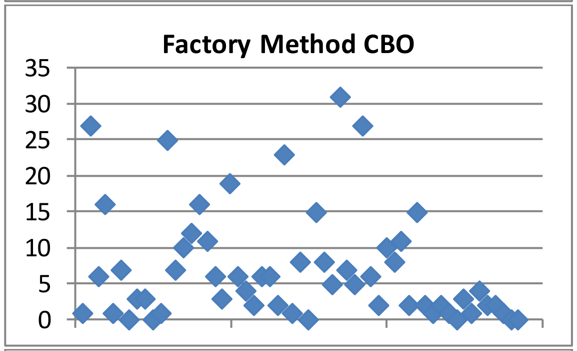
Scatter diagrams of the software quality metrics computed for design patterns' classes are given below:

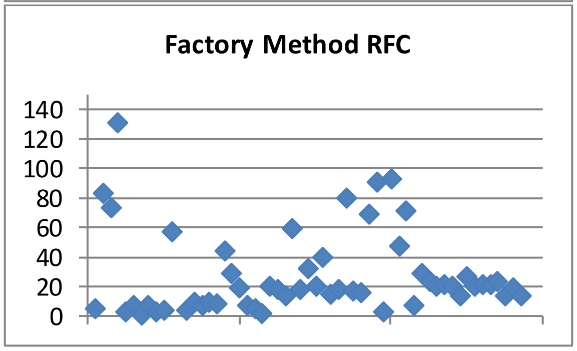
Note: The x-axis presents the number of detected pattern classes, and the y-axis presents the computed metrics for these classes.

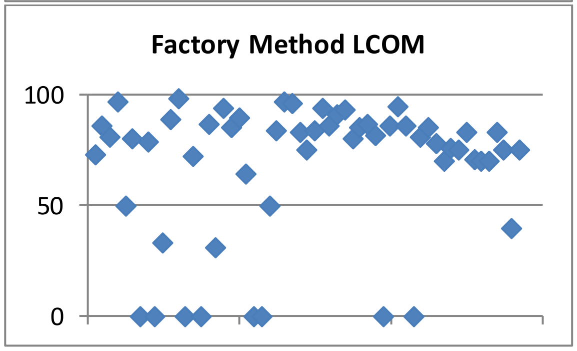


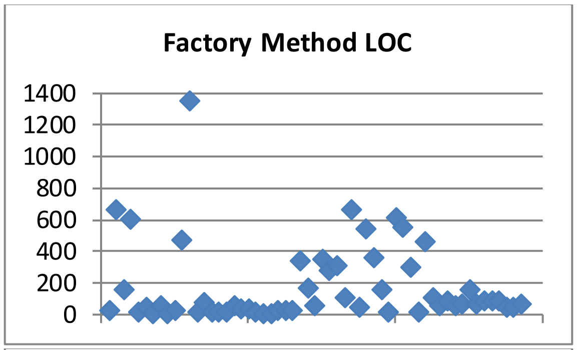


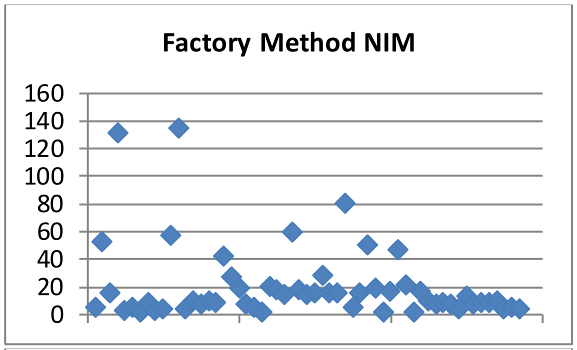


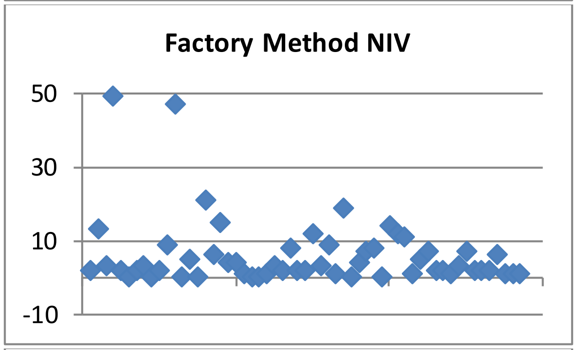


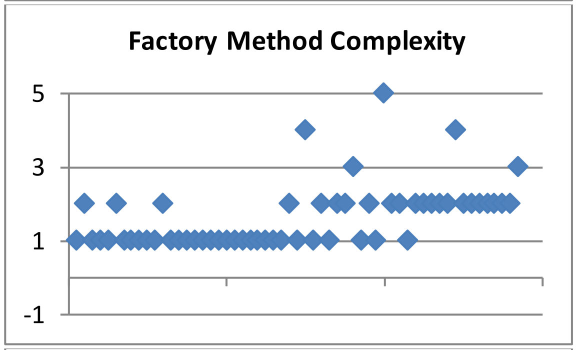


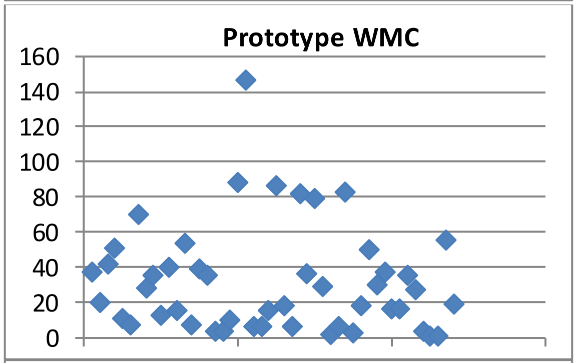


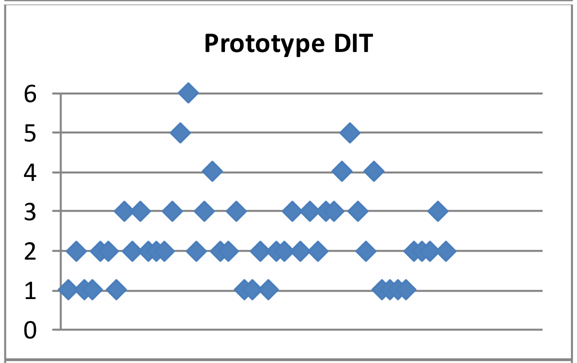

Factory Method LOC
1400
1200
1000
800
600
400
200
0

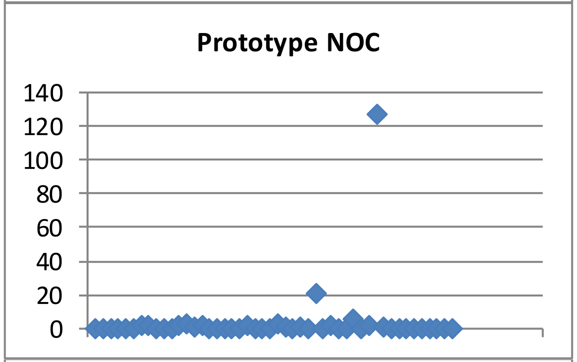

Factory Method NIM
160
140
120
100
80
60
40
20
0

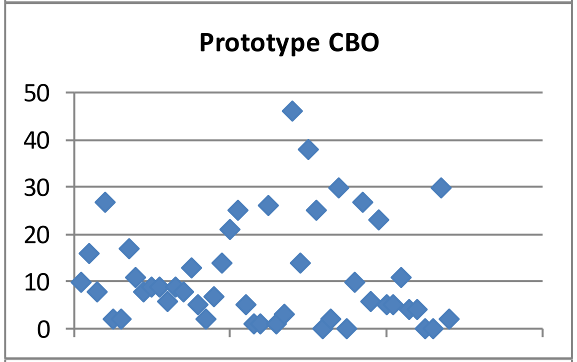

Factory Method NIV
50
30
10
-10

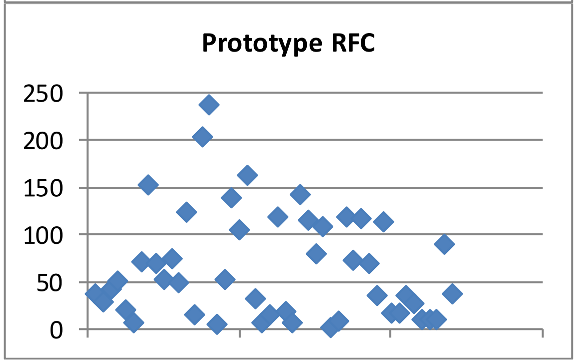

Factory Method Complexity
5
3
1
-1

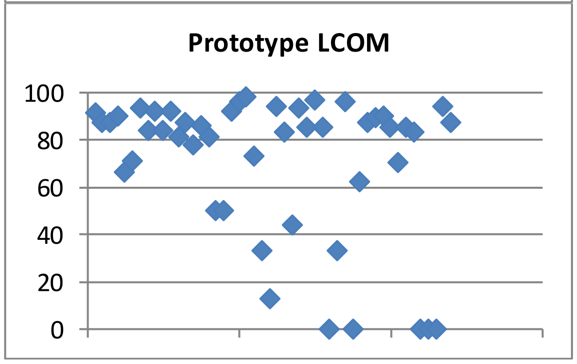

Prototype WMC
160
140
120
100
80
60
40
20
0

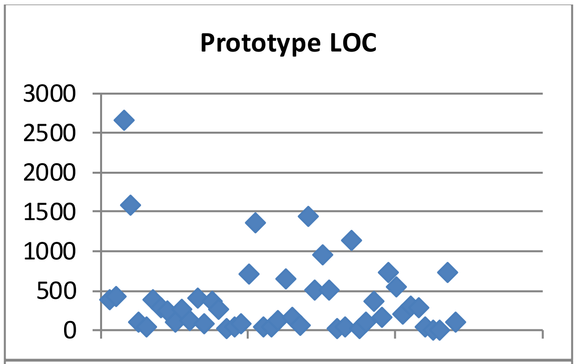

Prototype DIT
6
5
4
3
2
1
0

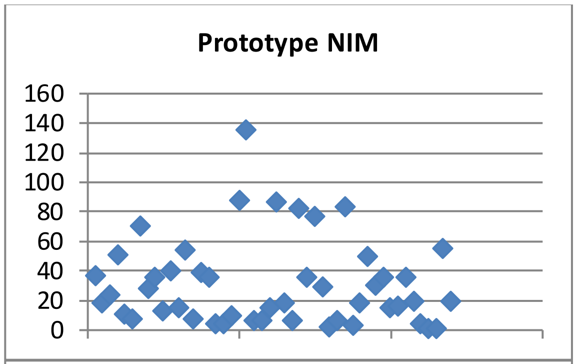

Prototype NOC
140
120
100
80
60
40
20
0

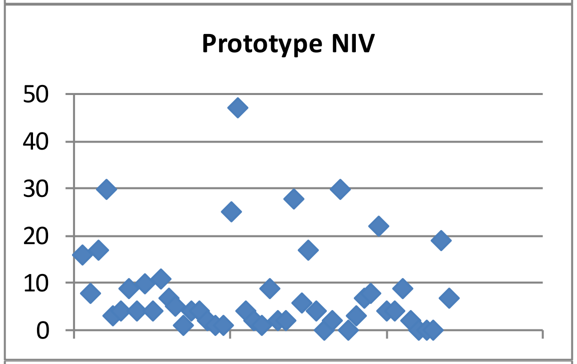

Prototype CBO
50
40
30
20
10
0

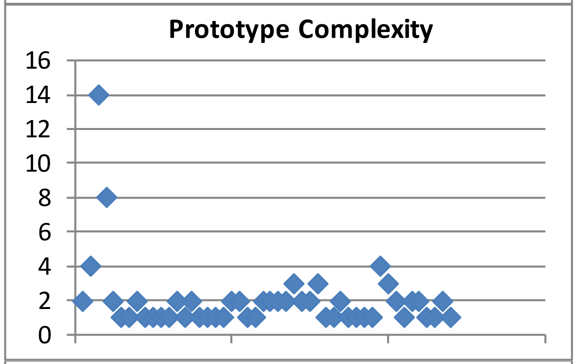

Prototype RFC
250
200
150
100
50
0

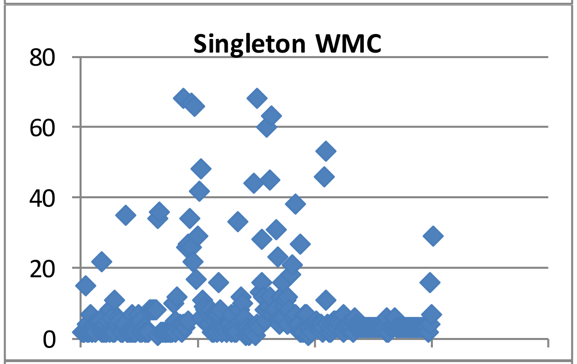

Prototype LCOM
100
80
60
40
20
0

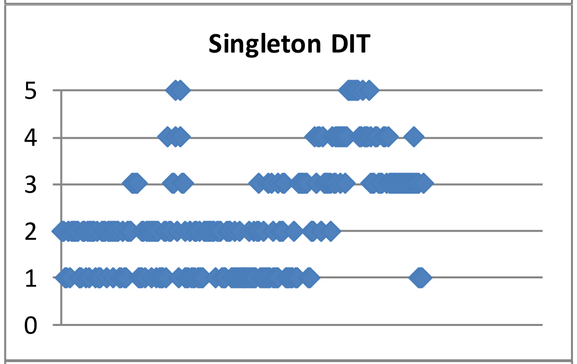

Prototype LOC
3000
2500
2000
1500
1000
500
0

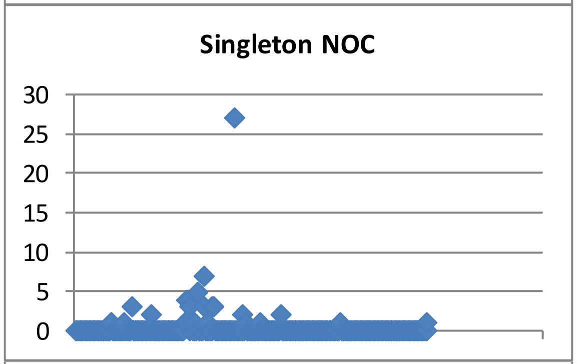

Prototype NIM
160
140
120
100
80
60
40
20
0

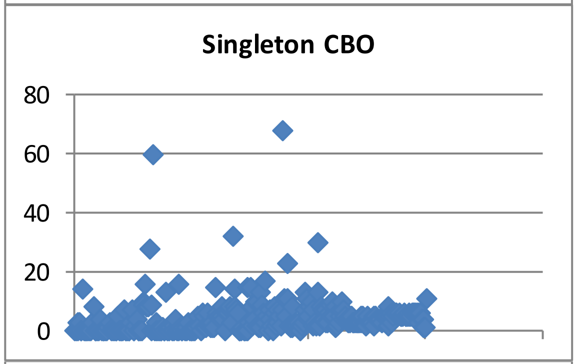

Prototype NIV
50
40
30
20
10
0

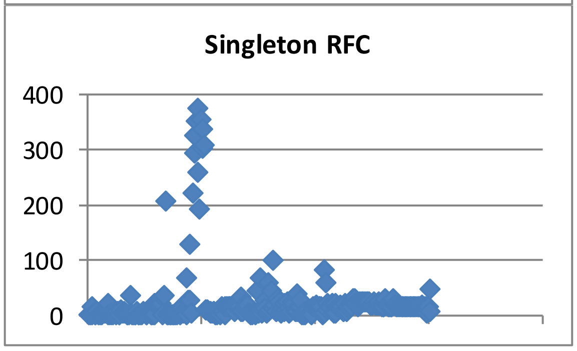

Prototype Complexity
16
14
12
10
8
6
4
2
0

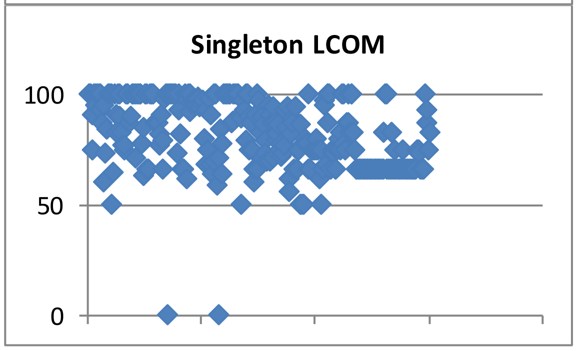

Singleton WMC
80
60
40
20
0

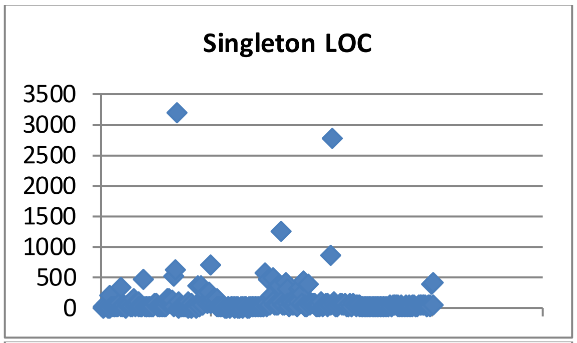

Singleton DIT
5
4
3
2
1
0

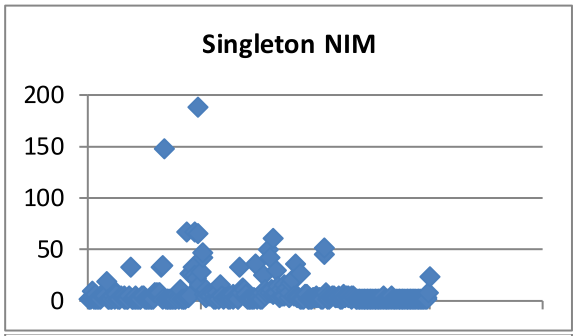

Singleton NOC
30
25
20
15
10
5
0

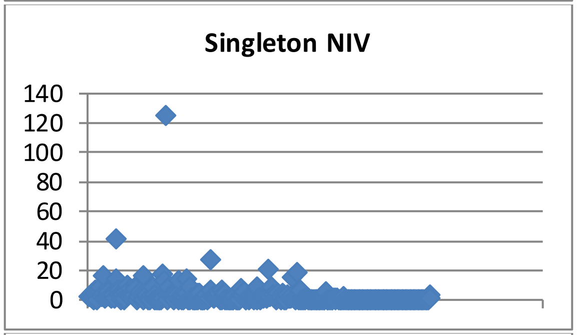

Singleton CBO
80
60
40
20
0

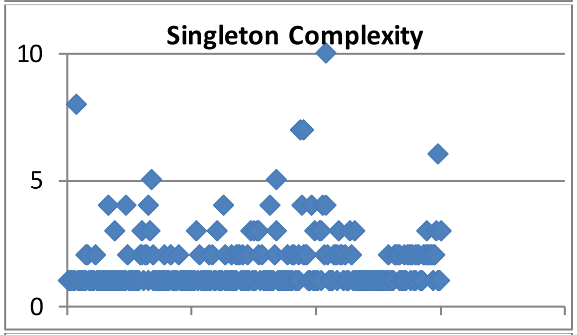

Singleton RFC
400
300
200
100
0

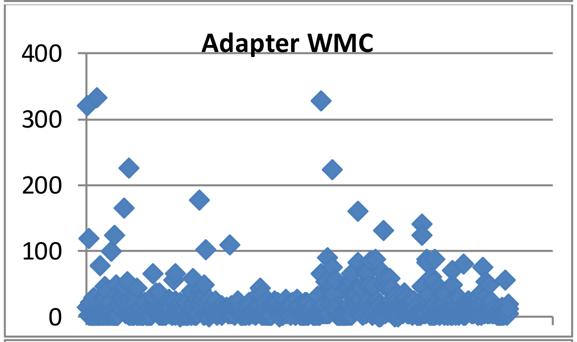

Singleton LCOM
100
50
0

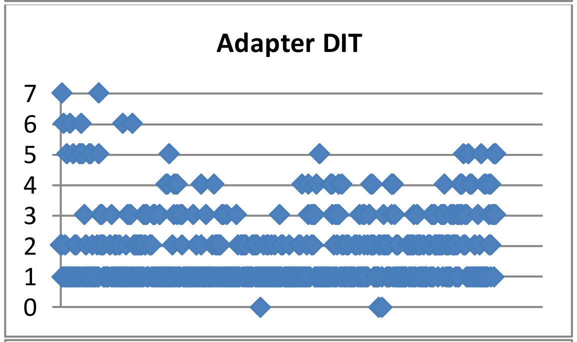

Singleton LOC
3500
3000
2500
2000
1500
1000
500
0

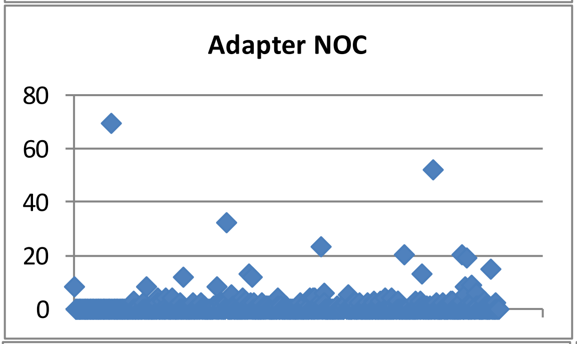

Singleton NIM
200
150
100
50
0

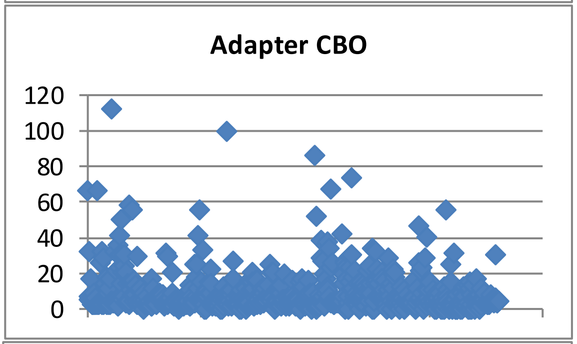

Singleton NIV
140
120
100
80
60
40
20
0

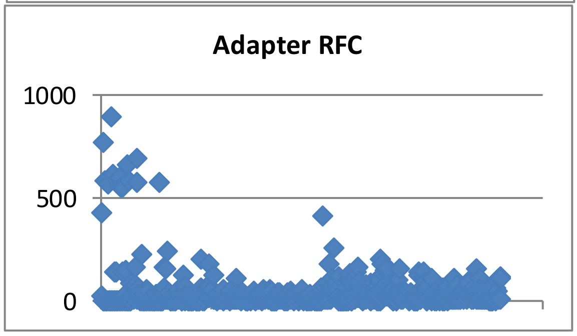

Singleton Complexity
10
5
0

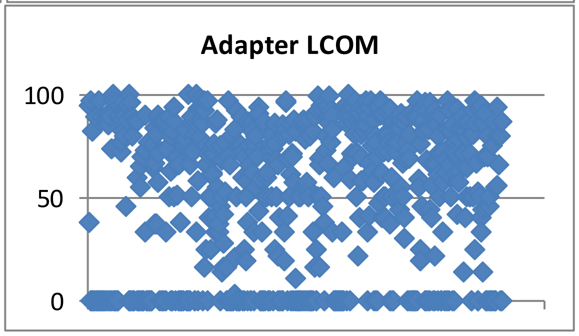

Adapter WMC
400
300
200
100
0

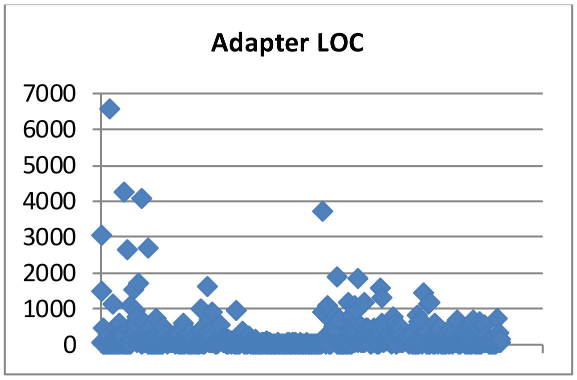

Adapter DIT
7
6
5
4
3
2
1
0

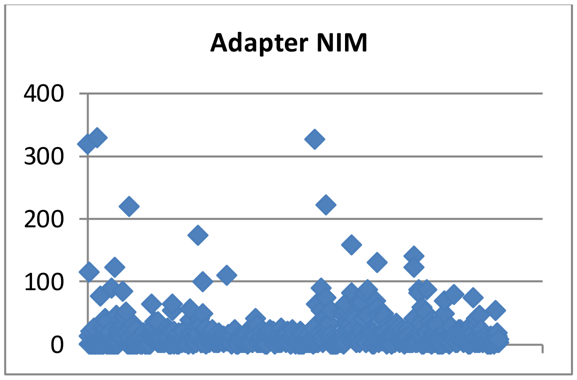

Adapter NOC
80
60
40
20
0

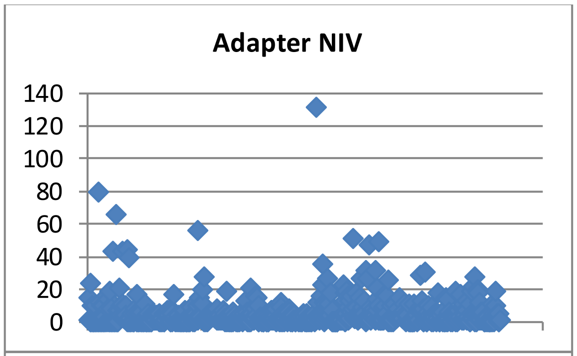

Adapter CBO
120
100
80
60
40
20
0

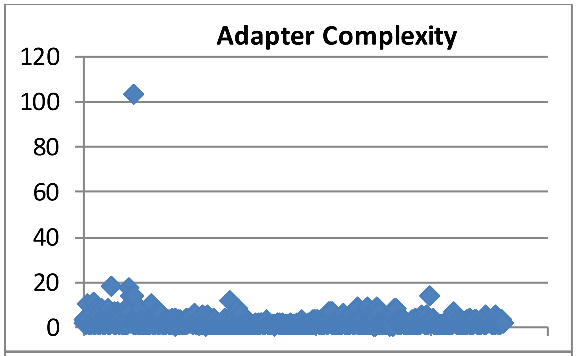

Adapter RFC
1000
500
0

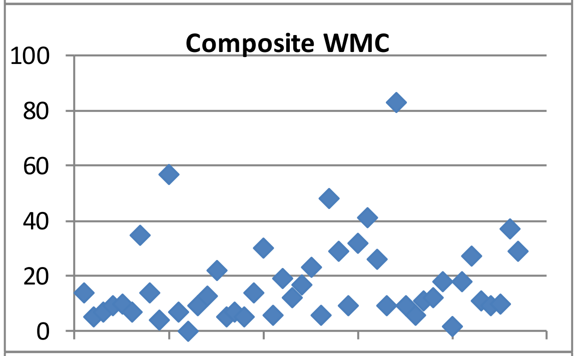

Adapter LCOM
100
50
0

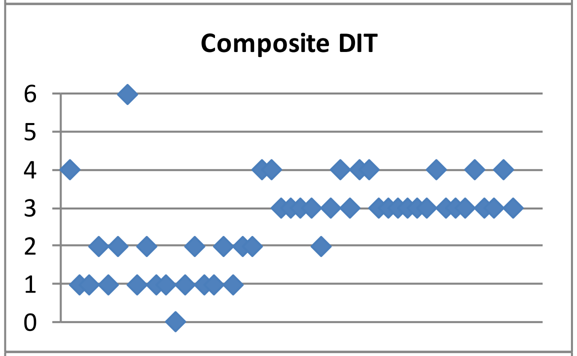

Adapter LOC
7000
6000
5000
4000
3000
2000
1000
0

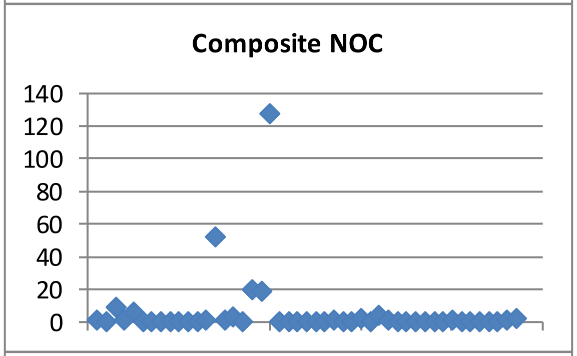

Adapter NIM
400
300
200
100
0

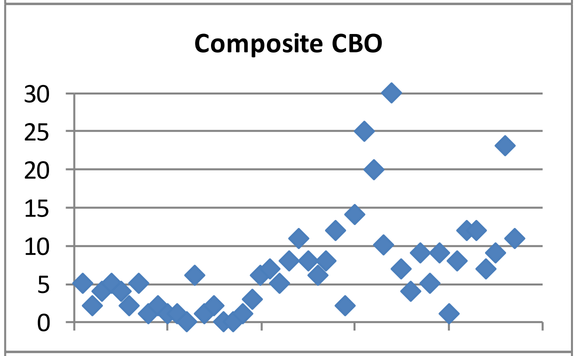

Adapter NIV
140
120
100
80
60
40
20
0

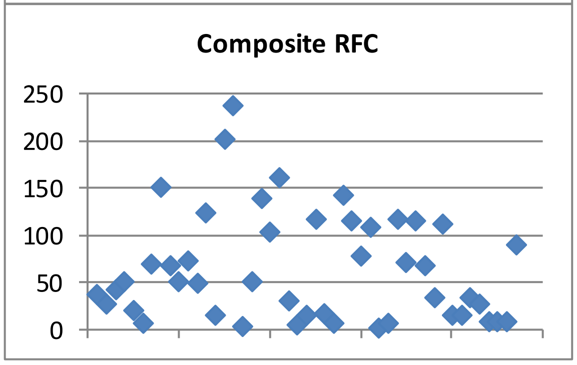

Adapter Complexity
120
100
80
60
40
20
0

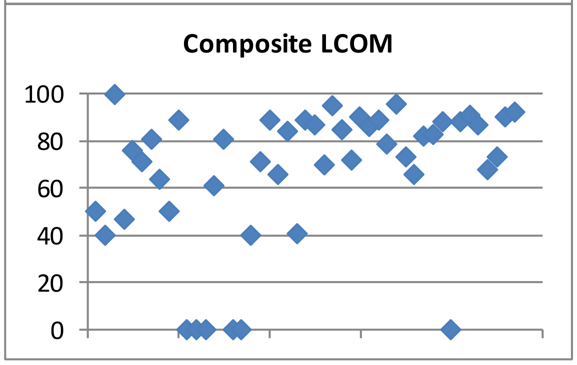

Composite WMC
100
80
60
40
20
0

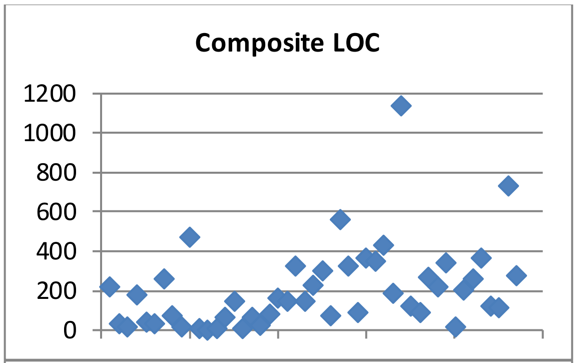

Composite DIT
6
5
4
3
2
1
0

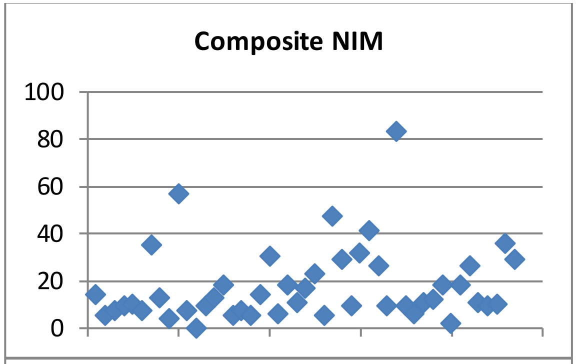

Composite NOC
140
120
100
80
60
40
20
0

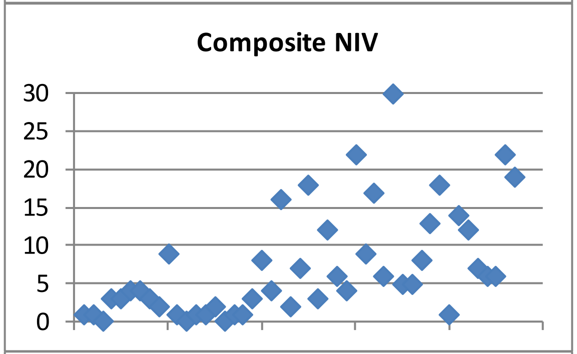

Composite CBO
30
25
20
15
10
5
0

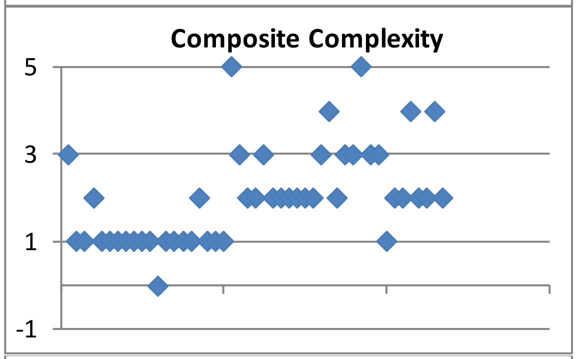

Composite RFC
250
200
150
100
50
0

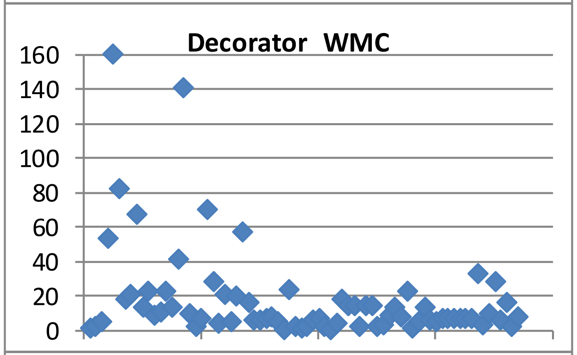

Composite LCOM
100
80
60
40
20
0

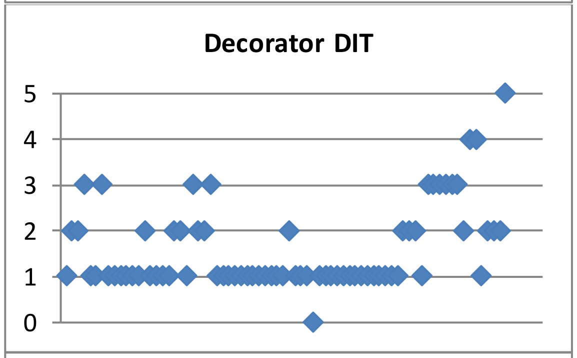

Composite LOC
1200
1000
800
600
400
200
0

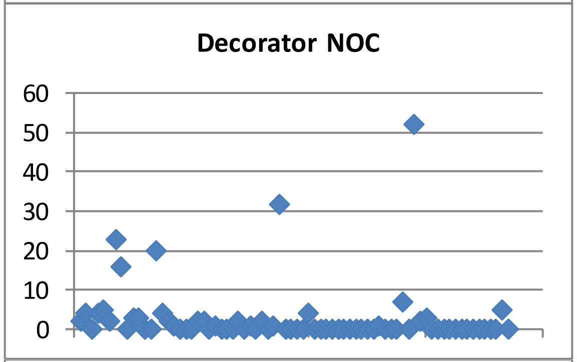

Composite NIM
100
80
60
40
20
0

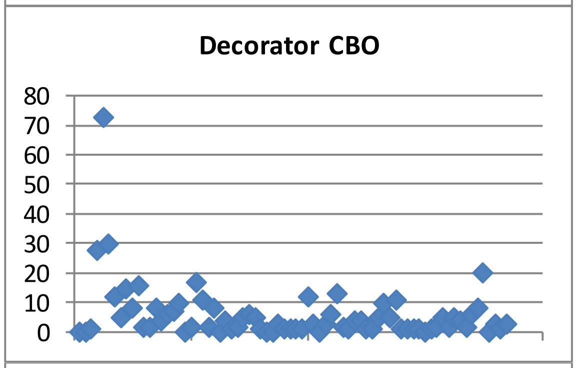

Composite NIV
30
25
20
15
10
5
0

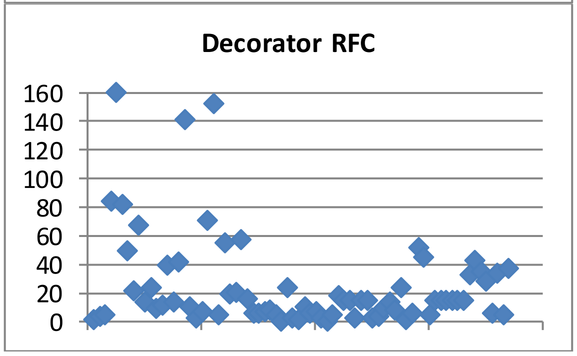

Composite Complexity
5
3
1
-1

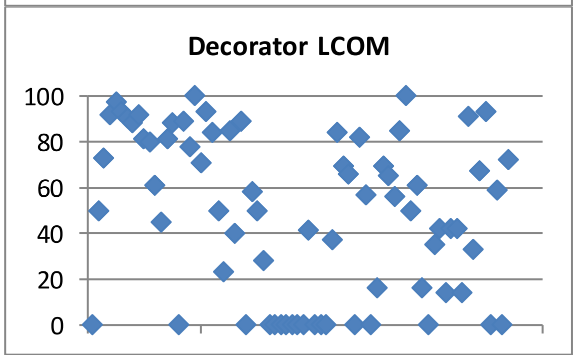

Decorator WMC
160
140
120
100
80
60
40
20
0

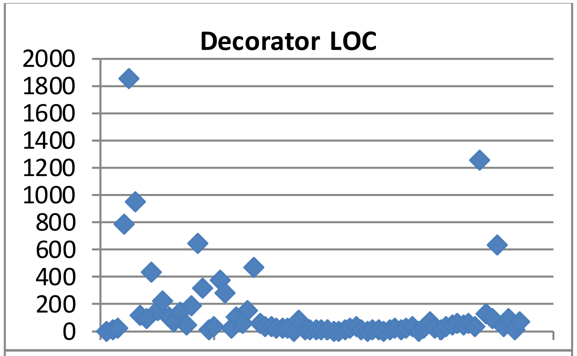

Decorator DIT
5
4
3
2
1
0

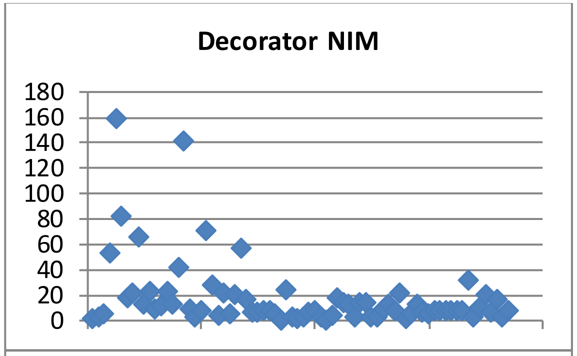

Decorator NOC
60
50
40
30
20
10
0

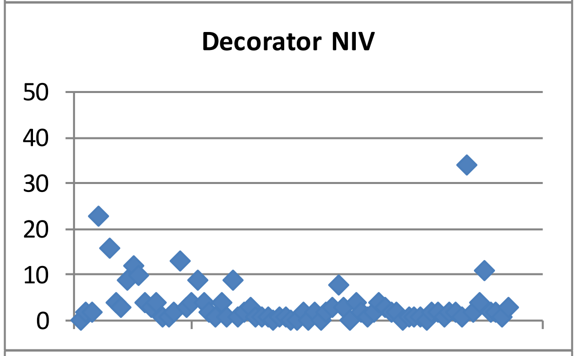

Decorator CBO
80
70
60
50
40
30
20
10
0

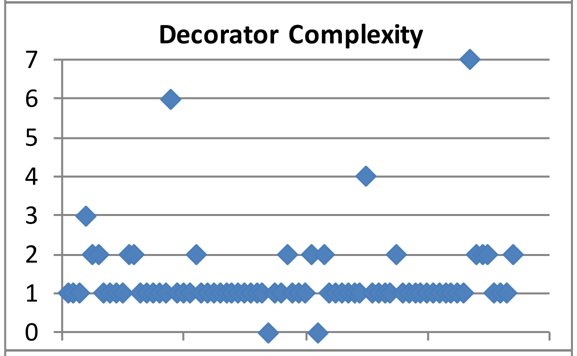

Decorator RFC
160
140
120
100
80
60
40
20
0

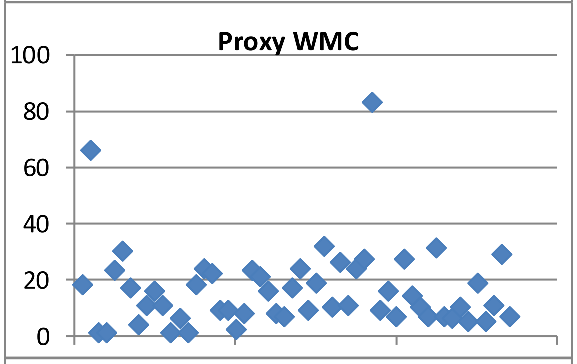

Decorator LCOM
100
80
60
40
20
0

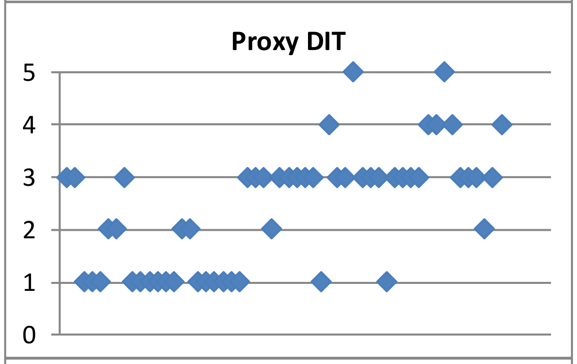

Decorator LOC
2000
1800
1600
1400
1200
1000
800
600
400
200
0

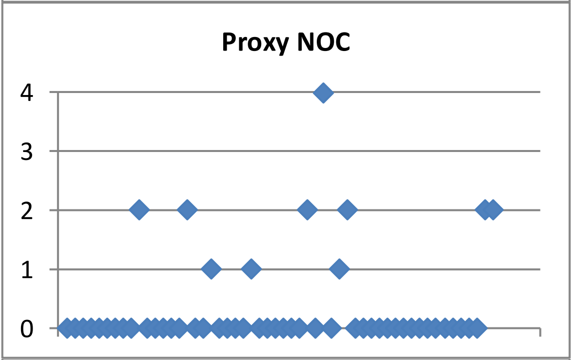

Decorator NIM
180
160
140
120
100
80
60
40
20
0

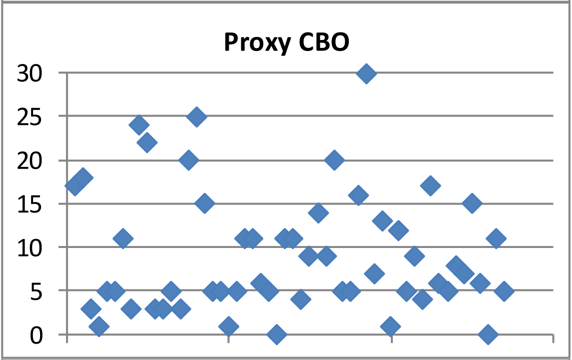

Decorator NIV
50
40
30
20
10
0

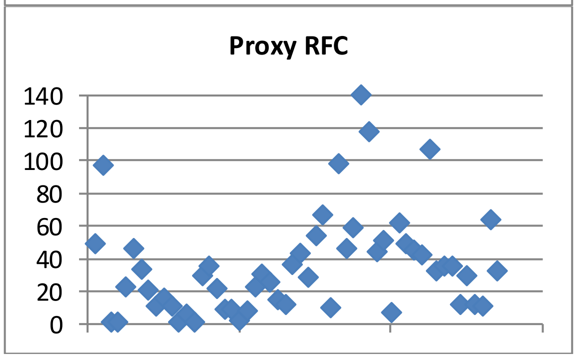

Decorator Complexity
7
6
5
4
3
2
1
0

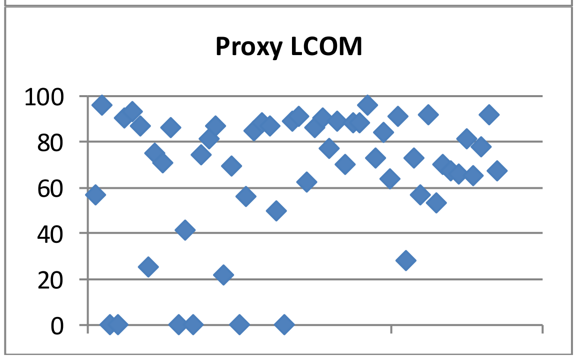

Proxy WMC
100
80
60
40
20
0

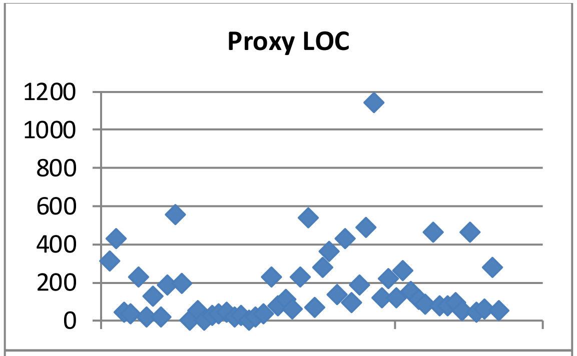

Proxy DIT
5
4
3
2
1
0

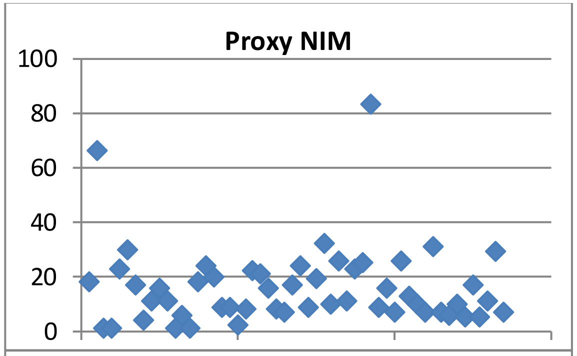

Proxy NOC
4
3
2
1
0

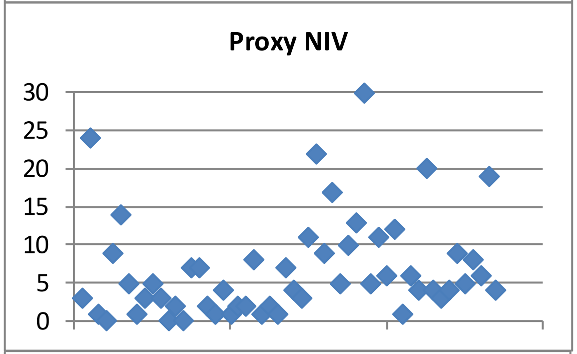

Proxy CBO
30
25
20
15
10
5
0

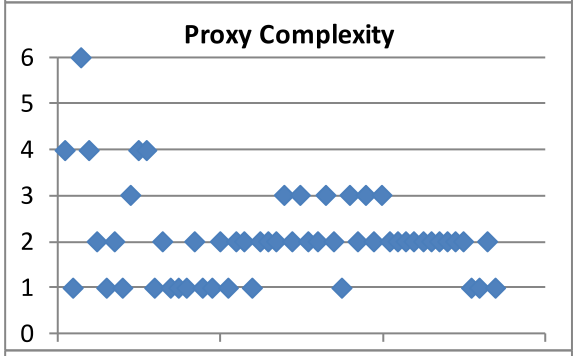

Proxy RFC
140
120
100
80
60
40
20
0

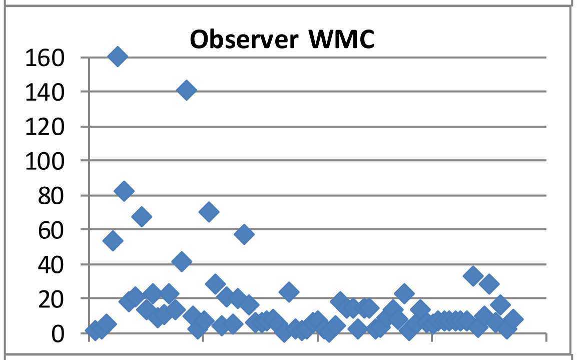

Proxy LCOM
100
80
60
40
20
0

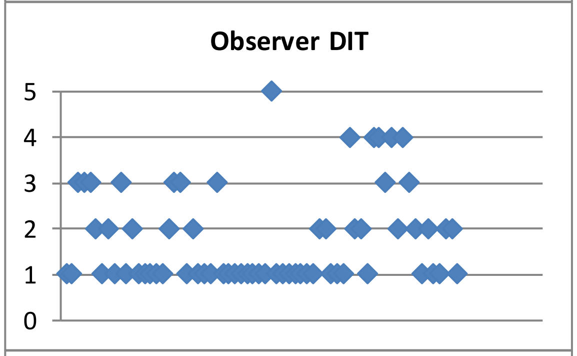

Proxy LOC
1200
1000
800
600
400
200
0

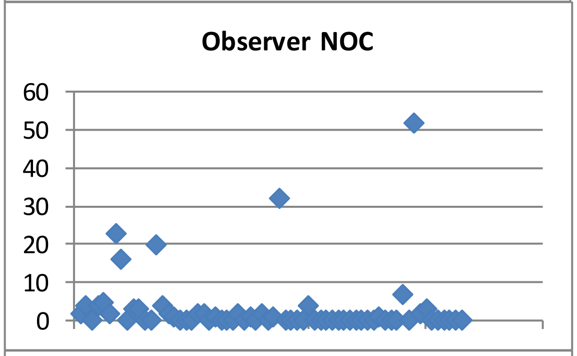

Proxy NIM
100
80
60
40
20
0

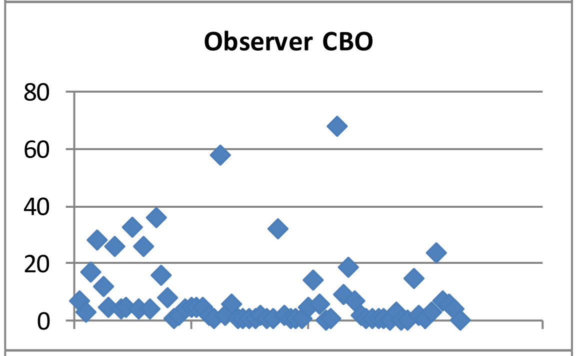

Proxy NIV
30
25
20
15
10
5
0

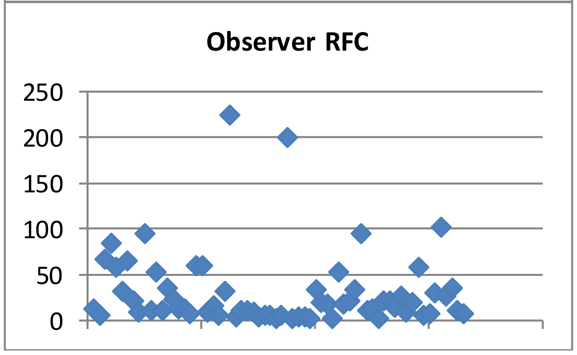

Proxy Complexity
6
5
4
3
2
1
0

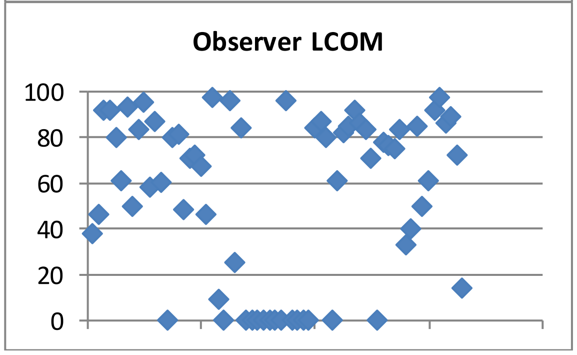

Observer WMC
160
140
120
100
80
60
40
20
0

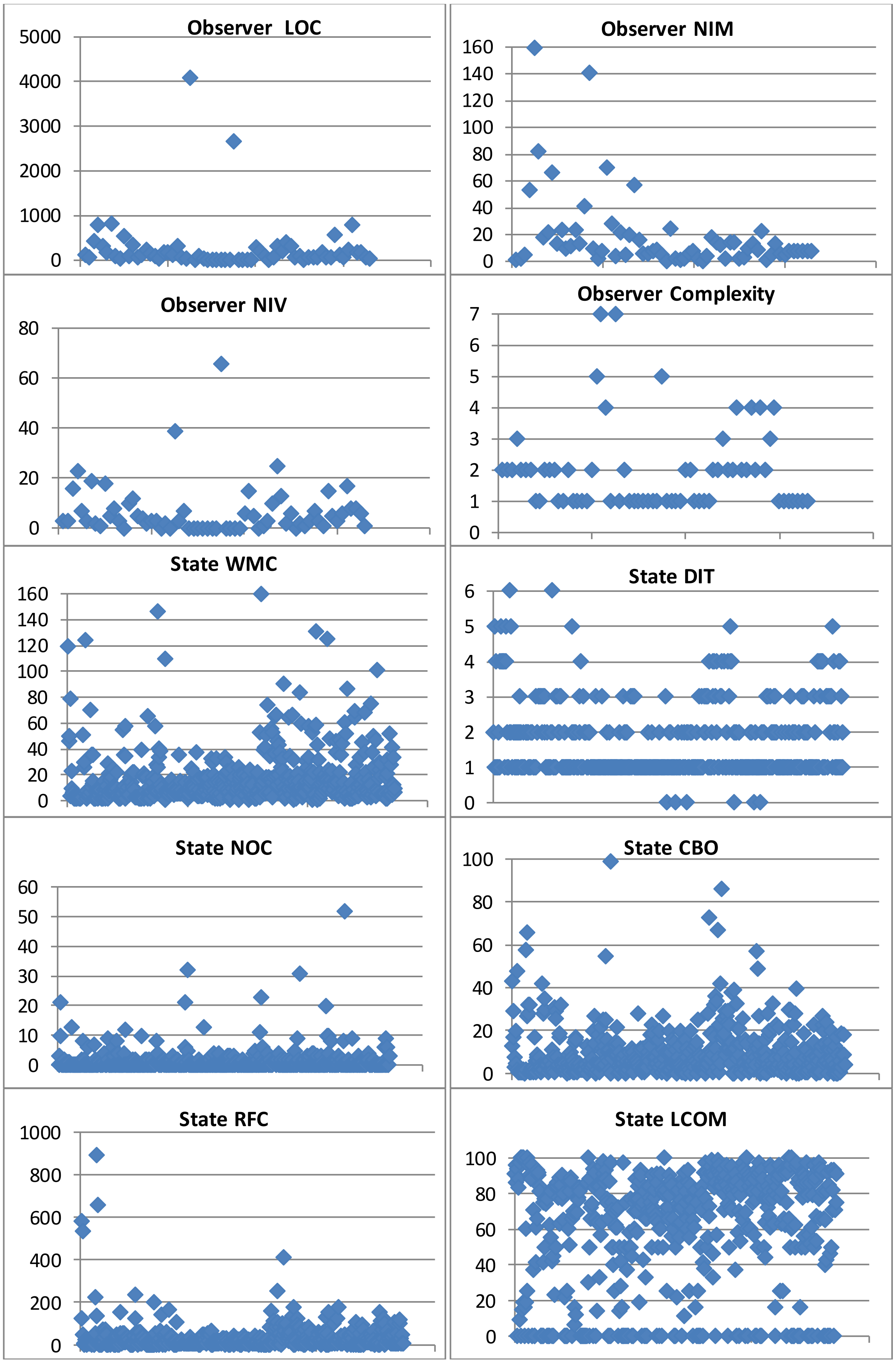

Observer DIT
5
4
3
2
1
0

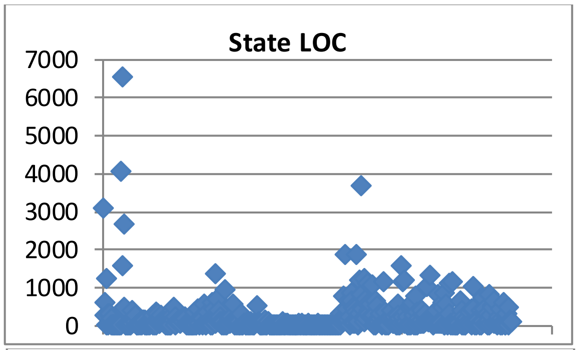

Observer NOC
60
50
40
30
20
10
0

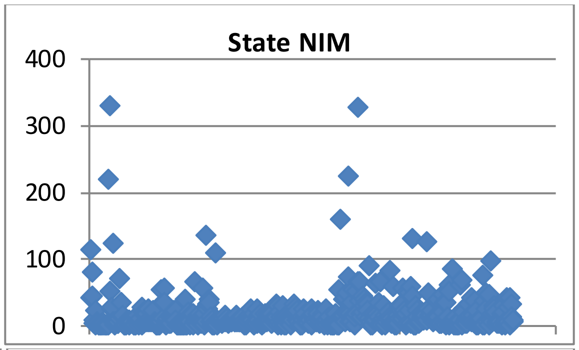

Observer CBO
80
60
40
20
0

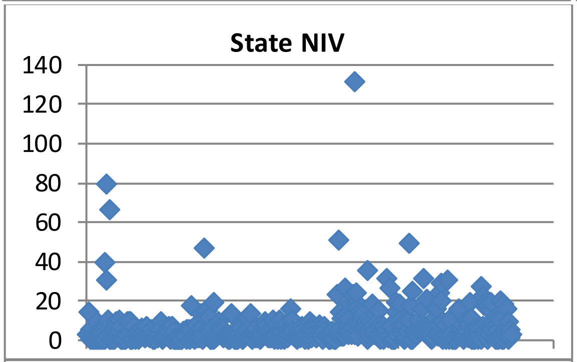

Observer RFC
250
200
150
100
50
0

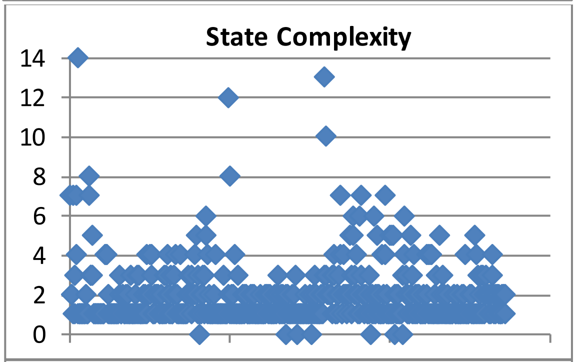

Observer LCOM
100
80
60
40
20
0

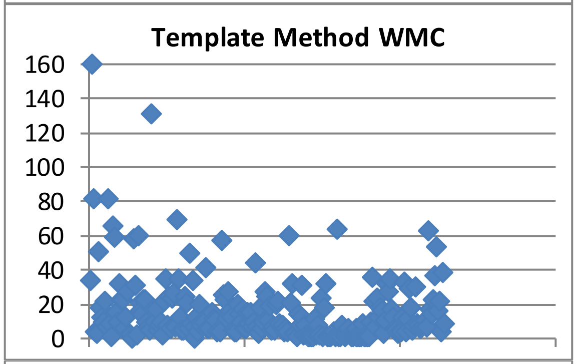

Observer LOC
0
1000
2000
3000
4000
5000
Observer NIM
0
20
40
60
80
100
120
140
160
Observer NIV
0
20
40
60
80
Observer Complexity
0
1
2
3
4
5
6
7
State WMC
0
20
40
60
80
100
120
140
160
State DIT
0
1
2
3
4
5
6
State NOC
0
10
20
30
40
50
60
State CBO
0
20
40
60
80
100
State RFC
0
200
400
600
800
1000
State LCOM
0
20
40
60
80
100

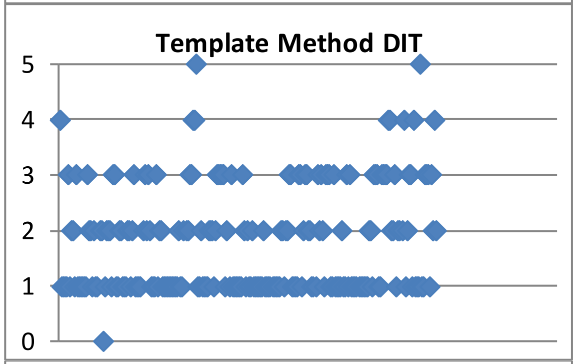

State LOC
7000
6000
5000
4000
3000
2000
1000
0

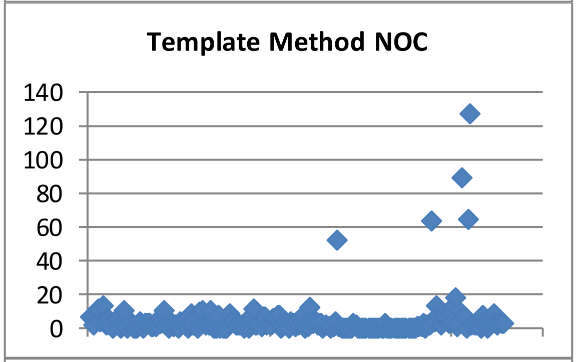

State NIM
400
300
200
100
0

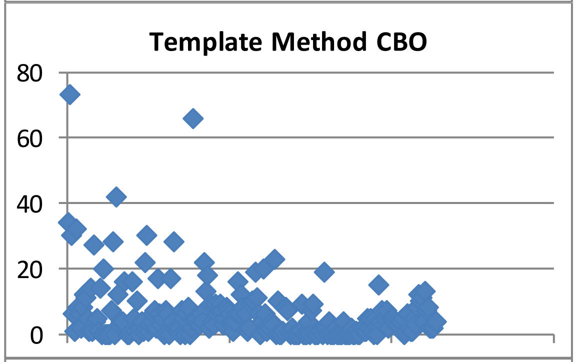

State NIV
140
120
100
80
60
40
20
0

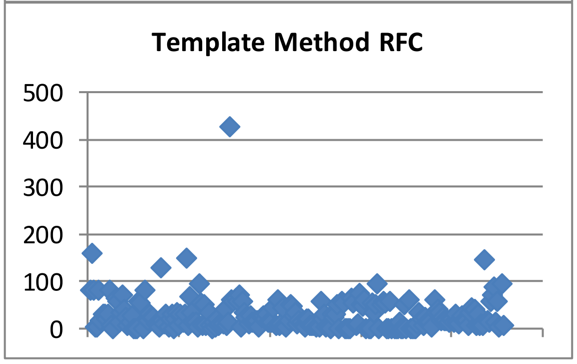

State Complexity
14
12
10
8
6
4
2
0

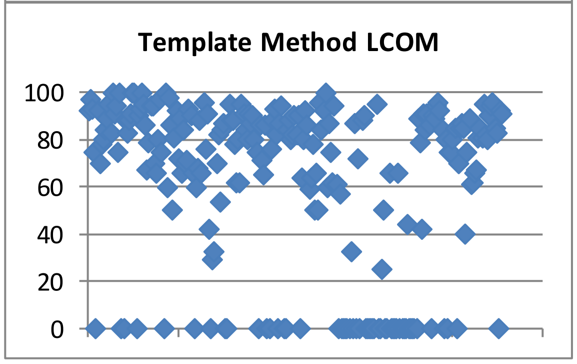

Template Method WMC
160
140
120
100
80
60
40
20
0

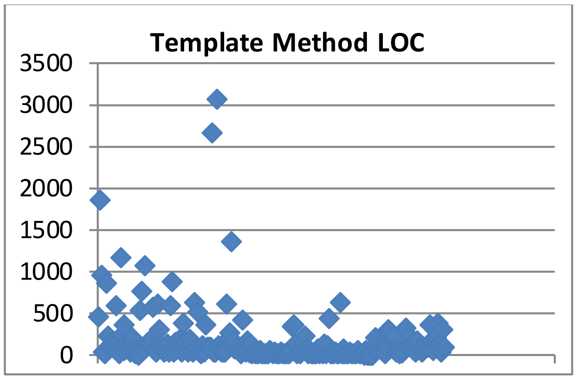

Template Method DIT
5
4
3
2
1
0

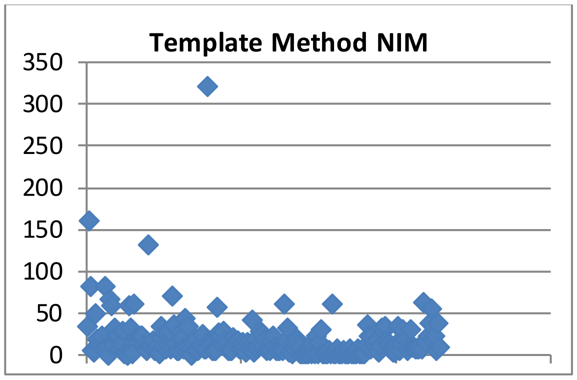

Template Method NOC
140
120
100
80
60
40
20
0

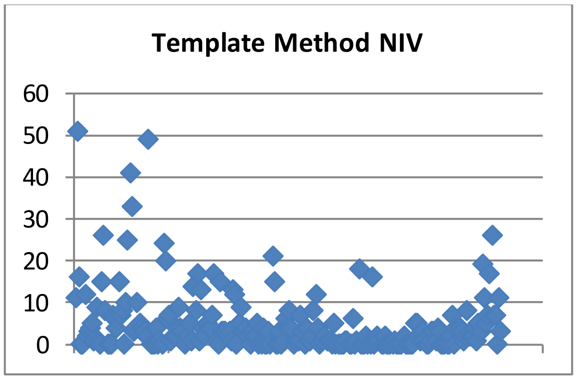

Template Method CBO
80
60
40
20
0

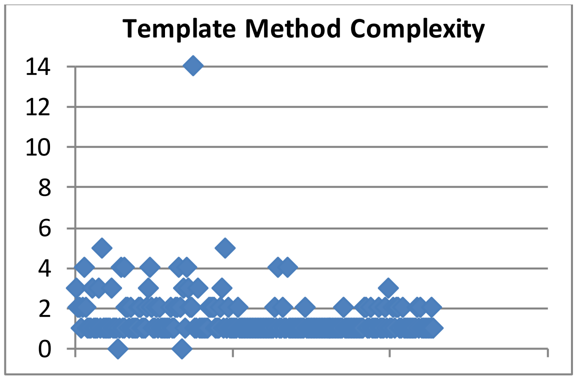

Template Method RFC
500
400
300
200
100
0


Template Method LCOM
100
80
60
40
20
0


Template Method LOC
3500
3000
2500
2000
1500
1000
500
0


Template Method NIM
350
300
250
200
150
100
50
0


Template Method NIV
60
50
40
30
20
10
0


Template Method Complexity
14
12
10
8
6
4
2
0