Mehran University Research Journal of Engineering and Technology
Vol. 39, No. 2, 342 - 352, April 2020
p-ISSN: 0254-7821, e-ISSN: 2413-7219
DOI: 10.22581/muet1982.2002.10

Impact of Design Patterns on Software Complexity and Size

Nosheen Qamar[1] Ali Afzal Malik[2]

RECEIVED ON 19.02.2019, ACCEPTED ON 03.05.2019

ABSTRACT

Many different factors influence the quality of software. Among the most important of these factors is software complexity. One way to improve software quality, therefore, is to minimize its complexity making it more understandable and maintainable. The design phase of the software development life cycle plays an instrumental role in fostering quality in software. Seasoned designers often use past design best practices codified in the form of design patterns to make their designs and the resultant code more elegant, robust, and resilient to change. Little work, however, has been done to empirically assess the quantitative impact of design patterns on software complexity. This research is an attempt to fill this gap. A comparative analysis of before and after versions of program pairs written without and with design patterns was performed for all twenty-three GoF (Gang of Four) design patterns. These program pairs were collected (or, in some cases, developed) and compared with respect to their complexity and size. The results of this comparative analysis reveal that the cyclomatic complexity of the programs written using design patterns was less for most of the design patterns as compared to the programs written without using design patterns. However, the values of CK metrics, number of classes, and software size SLOC (Source Lines of Code) increased when design patterns were used.

Key Words: **CK Metrics, Design Patterns, Gang of Four, Software Complexity, Software Quality, Software Size.**

1. INTRODUCTION

Software has become an invisible driving force of our individual and collective existence. It is now playing an indispensable role in our international trade, our stock exchanges, our educational institutions, our healthcare, and our entertainment. Yet, despite the fact that software has penetrated different aspects of our lives, it is still engineered following a set of steps which together constitute the SDLC. Some of these steps focus on the problem domain while others on the solution domain. The first step, and probably the most important, in the solution domain is design.

Design is the step in which the software engineer starts to think about solving the problem which has been understood in the upstream SDLC steps like requirements engineering and analysis. Designers focus on different aspects of the solution e.g. design of interfaces, design of data and data storage, design of software configuration, etc. Needless to say, most of the times it is impossible to get the design right the first time. Usually, multiple iterations are required and for most medium and large scale software-intensive systems design is a challenging exercise.

Software designers, however, often face recurring problems. Such problems can be solved using similar solutions called design patterns. In simple terms, a design pattern can be defined as a template or description for solving a recurring design problem [1]. Gamma *et al.* [1], commonly referred to as the GoF, were the first to catalog a collection of 23 commonly used design patterns. Apart from enabling the reuse of

[1] Department of Computer Science and Information Technology, University of Lahore, Lahore, Pakistan
Email: nqz786@gmail.com (Corresponding Author)
[2] Department of Computer Science, National University of Computer & Emerging Sciences, Lahore, Pakistan
Email: ali.afzal@nu.edu.pk

design concepts, these design patterns were considered useful for making designs more flexible and elegant.

Past research has shown that these design patterns have a positive impact on different software quality factors e.g. reusability, understandability, maintainability, *etc.* [2]. Software quality, however, can also be improved by reducing its complexity. Naturally, code that is less complex is easy to understand, maintain, modify, and reuse. It is also less error prone.

One of the commonly used metrics for quantitatively measuring code complexity is the cyclomatic complexity metric proposed by McCabe [3]. Given a program, it identifies the number of linearly independent paths through that program [3]. Generally speaking, the more the number of conditional statements (representing branching logic) in a program, the more the value of its cyclomatic complexity. Apart from the number of conditional statements (e.g. if-else, switch-case, etc.), the size of the software itself has an impact on its complexity. SLOC (Source Lines of Code) is one of the mostly commonly used metric for measuring the physical size of a program [4].

For object-oriented programs (including those written using design patterns), the well-known suite of OO-metrics proposed by Chidamber and Kemerer [5] can be used to measure complexity. This suite consists of six metrics which are commonly referred to as the CK metrics. Table 1 includes a brief description of each of these six CK metrics.

Even though researchers have conducted different studies to explore the impact of design patterns on different aspects of software quality [2,6], little work has been done to empirically examine the impact of design patterns on software complexity – one of the most important quality factors [6]. To the best of our knowledge, no past research has systematically assessed the impact of all 23 GoF design patterns on cyclomatic complexity, values of CK metrics, number of classes, and size measured using SLOC. This research is an attempt to fill this gap.

TABLE 1. CK METRICS [5]

Name	Description
WMC (Weighted Methods per Class)	Aggregated value of weights for the methods defined in a class
DIT (Depth of Inheritance Tree)	The length of the longest inheritance path from a root class to the current class
NOC (Number of Children)	Number of sub-classes which inherit directly from the current class
CBO (Coupling Between Objects)	The number of other classes which are coupled to the current class
RFC (Response For a Class)	Sum of the number of methods in a class and other remote methods those directly be called by that class
LCOM (Lack of Cohesion of Methods)	Lack of cohesion among the methods of a class

We gathered before and after versions of programs written to solve the same problem. The only difference between the before and after versions was that the before version was written without using any design pattern whereas the after version used one of the 23 GoF design patterns. The before/after program pairs were then analyzed to quantitatively assess the impact of using design patterns on software complexity and size.

The next section briefly summarizes the relevant work in this area. Section three provides the details of our research methodology while section four contains a discussion on the main results of our research. Finally, section five concludes this paper by summarizing our main findings and providing directions for future work in this area.

2. LITERATURE REVIEW

After design patterns were first introduced by Gamma *et al.* [1], people have shown immense interest in the use of design patterns. Lange and Nakamura, for instance, looked at how design patterns improve program understandability [7]. Their study, however, focused on only one quality attribute and was applicable on only a few design patterns.

Wydaeghe *et al.* [8] built an OMT (Osteopathic Manipulative Treatment*)*, editor and presented a detailed study on the use of six different design patterns. They described the impact of these patterns on modularity, reusability, understandability, and flexibility. They concluded that, while design patterns have various advantages not all design patterns have a positive impact on software quality. This study also was limited to authors' own experiences and the evaluations made and conclusions drawn may not be applicable to all design patterns.

McNatte and Bieman [9] evaluated the coupling between design patterns and their impact on quality attributes. Their results reveal that maintainability, reusability, and performance can be greatly improved when design patterns are abstracted and loosely coupled. Subburaj *et al.* [10] also assessed the effect of design patterns on software reusability. Their results reveal that design patterns improve architecture-level reusability of software.

Prechelt *et al.* [11] carried out an experiment on four systems to analyze the impact of five design patterns (i.e. Abstract, Factory, Composite, Decorator, Observer and Visitor) on maintainability. They concluded that design patterns are highly preferable even for simple design solutions. Hegedus *et al.* [12] also assessed the impact of design patterns on software maintainability. They took into account a total of three hundred revisions of the J Hot Draw software system. Their results revealed that the system's maintainability showed great improvement after using design patterns. This conclusion is corroborated by another experiment, conducted by Abdullah [13], to study the impact of design patterns on maintainability and performance. The results of this experiment also show that applying design patterns helps in attaining fairly good maintainability.

Rudzki [14] chose a slightly different approach. He assessed the impact of two distinct design patterns (i.e. Command and Façade) on the same software to see how the two differed in their respective impact on the level of performance of the software. While conducting this study, he ran the software in nine different test cases and reached the conclusion that the Command design pattern worked better than Façade and had a positive impact on software performance. Jeanmart *et al.* [15] assessed how the Visitor design pattern affects quality factors like understandability and maintainability. They concluded that the Visitor pattern is more time consuming in understanding and handling of tasks that require adjustments to be made. However, it was also revealed that when the Visitor design pattern is used in its canonical form, much less work and effort is required for adjustment tasks.

Aydinoz [16] applied refactoring on object-oriented programs with design patterns and found that complexity in terms of CBO (Coupling Between Objects), WMC (Weighted Method Per Class), and RFC (Response for a Class) had reduced. Huston [17] theoretically evaluated the impact of design patterns on class metric scores. He compared the complexity score of three design patterns (i.e. Bridge, Mediator, and Visitor) with non-pattern solutions. His results indicated that quality could improve by reducing the NOC (Number of Children) value.

Hsueh *et al.* [18] theoretically analyzed the impact of design patterns using QMOOD (Quality Model for Object Oriented Design). Their results showed that polymorphism and abstraction can be improved by using design patterns and Singleton design pattern does not contribute to quality improvement. Yu and Ramaswamy [19] analyzed the impact of 13 design patterns on class structure quality extracted from five different open source projects. Their results revealed that the use of design patterns can increase class complexity.

3. RESEARCH METHODOLOGY

Fig. 1 shows the main steps of our research process. The first step was the selection of design patterns for this experiment. We selected only GoF design patterns since they are widely used in the software industry. These 23 patterns are divided into three purposes (i.e. creational, behavioral and structural) and two scopes (i.e. object and class). The choice of appropriate object-oriented programming languages was the next step. C++ and Java were selected as they are popular object-oriented languages and a lot of applications patterns.

The third step was the collection of programs written with and without design patterns for the same problem. For most of the GoF design patterns, we found such programs with before and after versions at SourceMaking.com [20]. Before versions of just four design patterns were missing there. These were written by the first author herself [21]. Fig. 2 depicts the before/after program pair for the Builder design pattern.

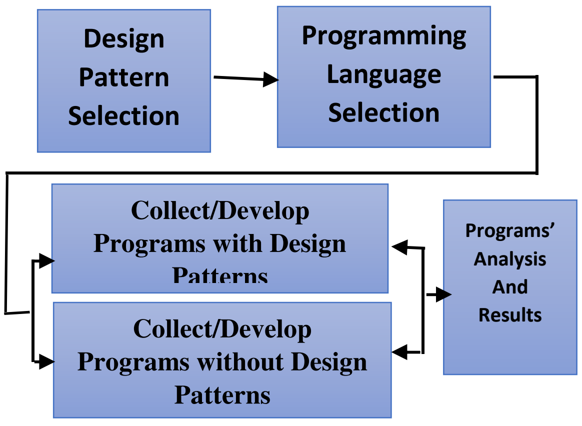
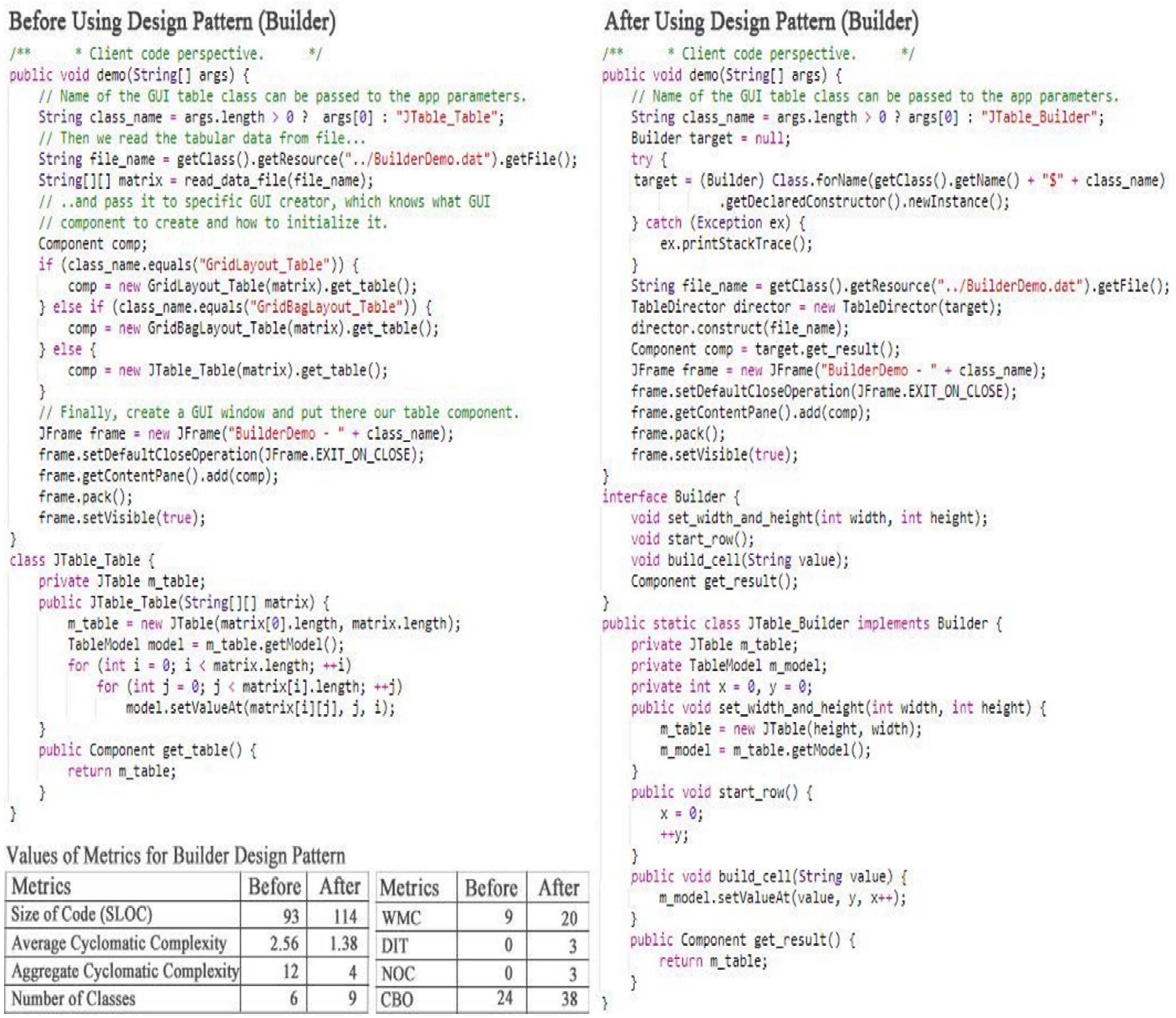

Design Pattern Selection
Programming Language Selection
Collect/Develop Programs with Design Patterns
Collect/Develop Programs without Design Patterns
Programs' Analysis And Results

FIG 1: RESEARCH STEPS



Before Using Design Pattern (Builder)

```
/**      * Client code perspective.      */
public void demo(String[] args) {
    // Name of the GUI table class can be passed to the app parameters.
    String class_name = args.length > 0 ?  args[0] : "JTable_Table";
    // Then we read the tabular data from file...
    String file_name = getClass().getResource("../BuilderDemo.dat").getFile();
    String[][] matrix = read_data_file(file_name);
    // ..and pass it to specific GUI creator, which knows what GUI
    // component to create and how to initialize it.
    Component comp;
    if (class_name.equals("GridLayout_Table")) {
        comp = new GridLayout_Table(matrix).get_table();
    } else if (class_name.equals("GridBagLayout_Table")) {
        comp = new GridBagLayout_Table(matrix).get_table();
    } else {
        comp = new JTable_Table(matrix).get_table();
    }
    // Finally, create a GUI window and put there our table component.
    JFrame frame = new JFrame("BuilderDemo - " + class_name);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(comp);
    frame.pack();
    frame.setVisible(true);
}
class JTable_Table {
    private JTable m_table;
    public JTable_Table(String[][] matrix) {
        m_table = new JTable(matrix[0].length, matrix.length);
        TableModel model = m_table.getModel();
        for (int i = 0; i < matrix.length; ++i)
            for (int j = 0; j < matrix[i].length; ++j)
                model.setValueAt(matrix[i][j], j, i);
    }
    public Component get_table() {
        return m_table;
    }
}
```

Values of Metrics for Builder Design Pattern

Metrics	Before	After
Size of Code (SLOC)	93	114
Average Cyclomatic Complexity	2.56	1.38
Aggregate Cyclomatic Complexity	12	4
Number of Classes	6	9

Metrics	Before	After
WMC	9	20
DIT	0	3
NOC	0	3
CBO	24	38

After Using Design Pattern (Builder)

```
/**      * Client code perspective.      */
public void demo(String[] args) {
    // Name of the GUI table class can be passed to the app parameters.
    String class_name = args.length > 0 ? args[0] : "JTable_Builder";
    Builder target = null;
    try {
     target = (Builder) Class.forName(getClass().getName() + "$" + class_name)
                .getDeclaredConstructor().newInstance();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    String file_name = getClass().getResource("../BuilderDemo.dat").getFile();
    TableDirector director = new TableDirector(target);
    director.construct(file_name);
    Component comp = target.get_result();
    JFrame frame = new JFrame("BuilderDemo - " + class_name);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(comp);
    frame.pack();
    frame.setVisible(true);
}
interface Builder {
    void set_width_and_height(int width, int height);
    void start_row();
    void build_cell(String value);
    Component get_result();
}
public static class JTable_Builder implements Builder {
    private JTable m_table;
    private TableModel m_model;
    private int x = 0, y = 0;
    public void set_width_and_height(int width, int height) {
        m_table = new JTable(height, width);
        m_model = m_table.getModel();
    }
    public void start_row() {
        x = 0;
        ++y;
    }
    public void build_cell(String value) {
        m_model.setValueAt(value, y, x++);
    }
    public Component get_result() {
        return m_table;
    }
}
```

FIG. 2. PART OF BEFORE/AFTER PROGRAM PAIR FOR BUILDER (CODE TAKEN FROM SOURCEMAKING.COM [20])

In the fourth step, we used "Source Monitor" [22] and "CCCC" [23] - two measurement/assessment tools - to compare the programs written using design patterns with those written without using design patterns. These tools were selected because they can measure values of selected metrics for both C++ and Java programming languages. The program pairs were compared using average and aggregate cyclomatic complexity and four CK metrics (i.e. WMC, DIT, NOC and CBO) since "CCCC" provides values for just these four metrics. Size was measured by counting the non-blank and non-comment SLOC. Table 2 summarizes the choices made at each step of our research process.

TABLE 2. PARAMETER VALUES

Parameter	Value(s)
Design Patterns	GoF Design Patterns
Programming Language	C++, Java
Tools Used for Analysis	SourceMonitor, C & C++ Code Counter (CCCC)
Comparison Attributes	Average & Aggregate Cyclomatic Complexity, Size of Code (SLOC), Number of Classes, CK Metrics (WMC, DIT, NOC and CBO)

4. RESULTS AND ANALYSIS

Table 3 shows the detailed comparison of program pairs with respect to size and cyclomatic complexity. It contains the values of SLOC, number of classes, average cyclomatic complexity, and aggregate cyclomatic complexity for programs written before and after using design patterns. Note that 24 program pairs (instead of 23) are selected and analyzed since the Adapter design pattern has two variants – one for class scope and one for object scope. The data in Table 3 reveals that SLOC increases for most of the design patterns except Composite, Command, Template Method, and Visitor. The number of classes increases or remains the same for all the design patterns. It can be seen that the average cyclomatic complexity of the programs either decreases or remains the same for most of the design patterns except Flyweight, Chain of Responsibility and Observer. The aggregate cyclomatic complexity increases for Abstract Factory, Factory Method, Prototype, Flyweight, Proxy, Chain of Responsibility, and Mediator. It remains the same or decreases for rest of the design patterns.

Fig. 3 shows, pictorially, the impact of using different design patterns on program average cyclomatic complexity. In the case of Builder, Prototype, Adapter (Class), Interpreter, Mediator, State, and Template Method the complexity decreases by at least 25% and by at most 53%. In the case of Abstract Factory, Decorator, and Strategy there is no change in average cyclomatic complexity. For Flyweight, Chain of Responsibility and Observer, there is only a slight (around 10%) increase in complexity.

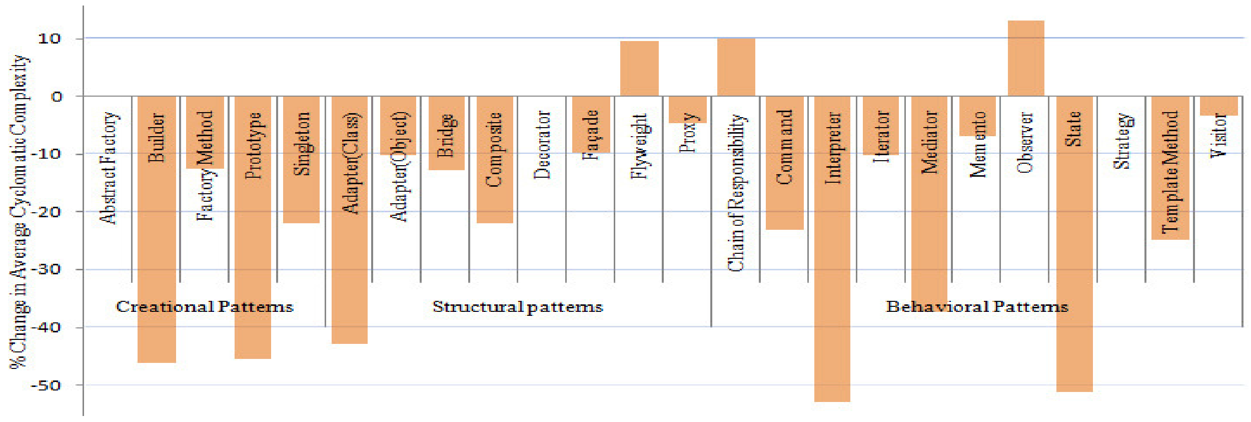


FIG. 3. IMPACT OF DESIGN PATTERNS ON PROGRAM AVERAGE CYCLOMATIC COMPLEXITY

TABLE 3. BEFORE AND AFTER USING DESIGN PATTERNS – SIZE & CYCLOMATIC COMPLEXITY COMPARISON

No.	Purpose	Scope	PL	Design Pattern	SLOC		No. of Classes		Average Cyclomatic Complexity		Aggregate Cyclomatic Complexity	
					B	A	B	A	B	A	B	A
1.	Creational Patterns	Object	C++	Abstract Factory	58	80	6	9	1.00	1.00	1	4
2.		Object	Java	Builder	93	114	4	5	2.56	1.38	12	4
3.		Class	C++	Factory Method	50	55	4	4	2.75	2.40	7	10
4.		Object	C++	Prototype	50	51	4	5	2.75	1.50	7	9
5.		Object	C++	Singleton	39	40	1	1	1.50	1.17	4	3
6.	Structural Patterns	Class	Java	Adapter	25	46	3	5	2.00	1.14	3	1
7.		Object	C++		33	39	5	5	1.76	1.58	1	1
8.		Object	C++	Bridge	68	72	5	6	2.00	1.42	4	4
9.		Object	Java	Composite	31	28	3	3	1.50	1.17	2	1
10.		Object	C++	Decorator	66	74	6	6	1.00	1.00	1	1
11.		Object	C++	Façade	135	143	4	4	2.22	2.00	10	10
12.		Object	C++	Flyweight	29	54	1	2	1.67	1.83	2	6
13.		Object	C++	Proxy	31	52	1	2	1.50	1.43	3	4
14.	Behavioral Patterns	Object	Java	Chain of Responsibility	27	41	2	2	2.50	2.75	3	4
15.		Object	C++	Command	77	35	2	3	1.56	1.20	3	3
16.		Class	Java	Interpreter	54	67	1	4	5.50	2.58	10	5
17.		Object	Java	Iterator	26	52	2	3	1.67	1.50	1	1
18.		Object	C++	Mediator	49	55	1	2	2.40	1.50	4	5
19.		Object	C++	Memento	43	44	2	3	1.70	1.58	3	3
20.		Object	C++	Observer	48	60	3	4	1.00	1.13	1	1
21.		Object	Java	State	40	58	2	6	2.50	1.22	4	1
22.		Object	C++	Strategy	97	102	5	5	1.50	1.50	4	4
23.		Class	C++	Template Method	60	58	2	3	3.20	2.40	11	9
24.		Object	C++	Visitor	85	85	3	6	1.13	1.09	1	1

*PL= Programming Language, B = Before, A = After

Fig. 4 depicts how program aggregate cyclomatic complexity is influenced by design patterns. The aggregate complexity decreases between 18-75% for Builder, Singleton, Adapter (Class), Composite, Interpreter, State, and Template Method. In the case of Adapter (Object), Bridge, Decorator, Façade, Command, Iterator, Memento, Observer, Strategy, and Visitor there is no change in aggregate complexity. For Abstract Factory and Flyweight, the increase in aggregate complexity is more prominent as compared to the increase in case of Factory Method, Prototype, Proxy, Chain of Responsibility, and Mediator. This decrease in the average and aggregate cyclomatic complexity of the programs examined in this research implies that using design patterns may lead to an improvement in program understandability and, thereafter, an enhancement in program flexibility and maintainability.

Fig. 5 shows, graphically, the impact of using different design patterns on program size (SLOC). Fig. 5 clearly reveals that the size in SLOC decreases for Composite, Command, and Template Method only. In the case of Visitor, there is no change in size. The increase in size is very nominal for Prototype, Singleton, Bridge, Façade, Memento, and Strategy design patterns. Change is more prominent (at least 40%) for some patterns namely Adapter (Class), Flyweight, Proxy, Chain of Responsibility, Iterator, and State. This increase in program size may simply be due to the fact

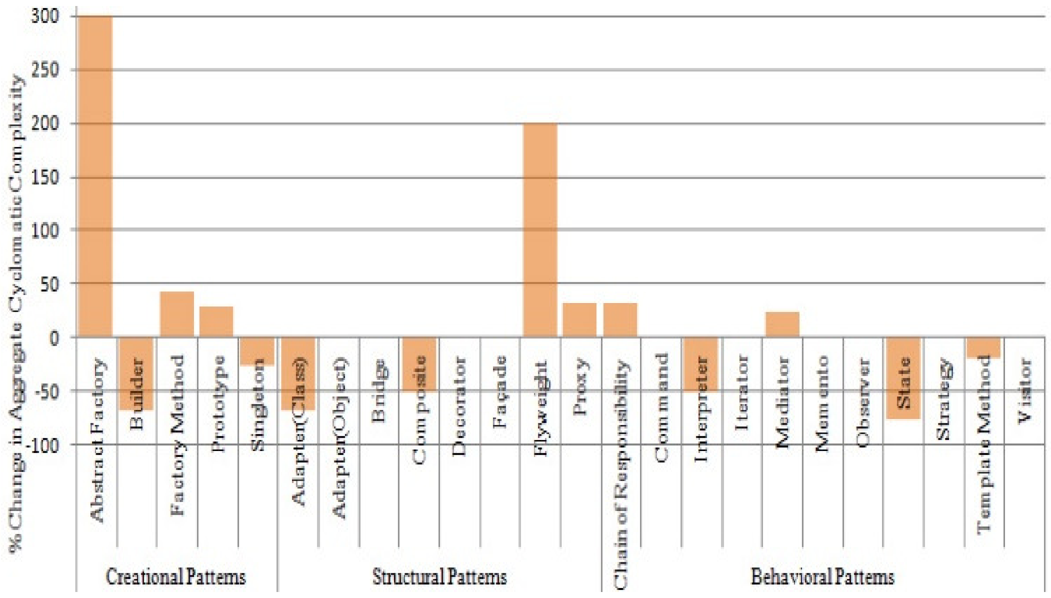


FIG. 4. IMPACT OF DESIGN PATTERNS ON PROGRAM AGGREGATE CYCLOMATIC COMPLEXITY

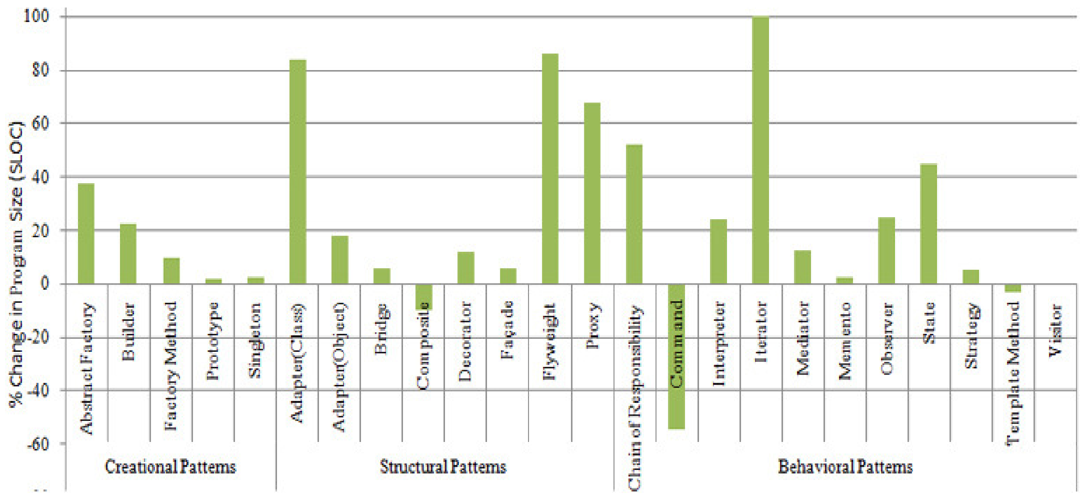


FIG 5: .IMPACT OF DESIGN PATTERNS ON PROGRAM SIZE (SLOC)

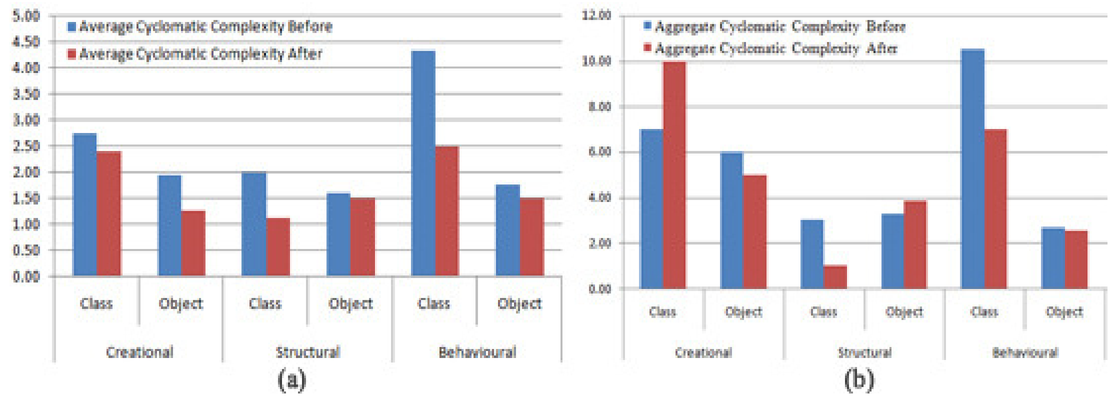


FIG. 6. IMPACT ON CYCLOMATIC COMPLEXITY FOR DIFFERENT CATEGORIES OF DESIGN PATTERNS (A) AVERAGE CYCLOMATIC COMPLEXITY (B) AGGREGATE CYCLOMATIC COMPLEXITY

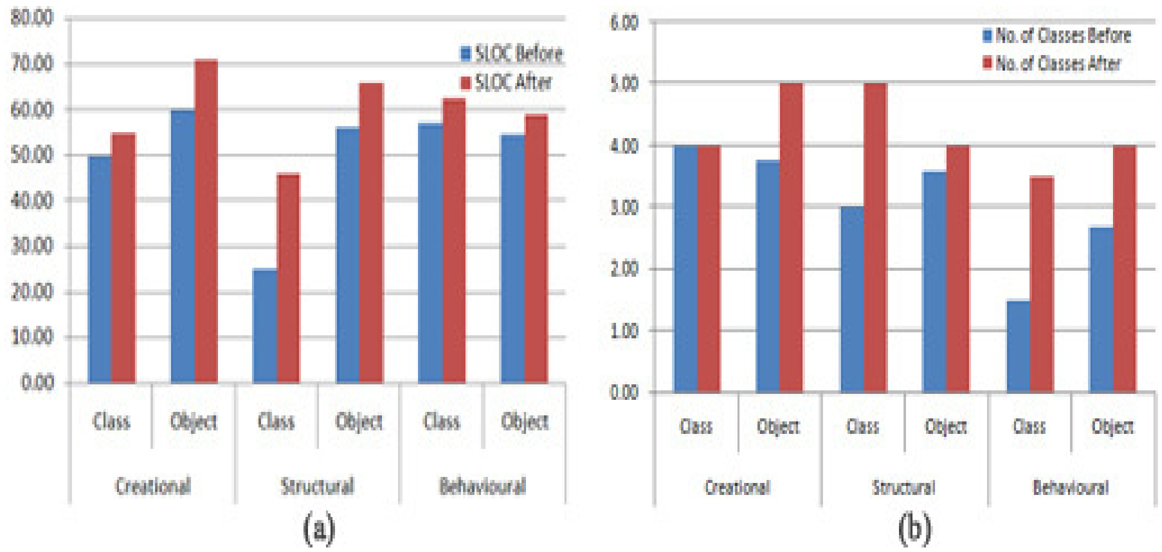


FIG.7. IMPACT ON SIZE AND NUMBER OF CLASSES FOR DIFFERENT CATEGORIES OF DESIGN PATTERNS (A) SIZE (SLOC) (B) NUMBER OF CLASSES

that design patterns add additional classes and layers of indirection to achieve design elegance and flexibility.

Fig. 6(a-b) depicts the impact on average and aggregate cyclomatic complexity, respectively, for different categories of design patterns. The average cyclomatic complexity decreases for all combination of scopes and purposes. This decrease in complexity is most prominent for class behavioral design patterns. The aggregate cyclomatic complexity decreases for object creational, class structural, and both combinations of behavioral design patterns only. This decline, however, is most prominent for class structural and class behavioral design patterns.

Fig. 7(a-b) illustrates the impact on program size (SLOC) and number of classes, respectively, for different categories of design patterns. It is evident that, after using design patterns, the SLOC and number of classes increase for all categories of design patterns. The only exception to this is class creational design patterns for which the number of classes remain the same.

This slight increase in the program size and the number of classes is the necessary by-product of using design patterns which achieve design reusability by introducing additional classes and hence program statements.

TABLE 4. BEFORE AND AFTER USING DESIGN PATTERNS – CK METRICS COMPARISON

No.	Purpose	Scope	PL	Design Pattern	WMC			DIT			NOC			CBO		
					B	A	%	B	A	%	B	A	%	B	A	%
1.	Creational Patterns	Object	C++	Abstract Factory	9	16	77.8	4	6	50.0	4	6	50.0	8	14	75.0
2.		Object	Java	Builder	9	20	122.2	0	3	-	0	3	-	24	38	58.3
3.		Class	C++	Factory Method	5	6	20.0	3	3	0.0	3	3	0.0	6	6	0.0
4.		Object	C++	Prototype	5	10	100.0	3	3	0.0	3	3	0.0	6	8	33.3
5.		Object	C++	Singleton	6	7	16.7	0	0	-	0	0	-	0	0	-
6.	Structural Patterns	Class	Java	Adapter	3	8	166.7	0	2	-	0	2	-	2	10	400.0
7.		Object	C++		10	11	10.0	1	1	0.0	1	1	0.0	8	11	37.5
8.		Object	C++	Bridge	8	12	50.0	0	4	-	0	4	-	2	10	400.0
9.		Object	Java	Composite	5	6	20.0	0	2	-	0	2	-	8	12	50.0
10.		Object	C++	Decorator	10	17	70.0	7	8	14.3	8	5	-37.5	16	16	0.0
11.		Object	C++	Façade	9	11	22.2	0	0	-	0	0	-	6	6	0.0
12.		Object	C++	Flyweight	3	6	100.0	0	0	-	0	0	-	0	2	-
13.		Object	C++	Proxy	4	7	75.0	0	0	-	0	0	-	0	2	-
14.	Behavioral Patterns	Object	Java	Chain of Responsibility	2	4	100.0	0	0	-	0	0	-	4	4	0.0
15.		Object	C++	Command	5	7	40.0	0	0	-	0	0	-	0	2	-
16.		Class	Java	Interpreter	2	12	500.0	0	3	-	0	3	-	2	12	500.0
17.		Object	Java	Iterator	0	0	-	0	0	-	0	0	-	0	0	-
18.		Object	C++	Mediator	5	6	20.0	0	0	-	0	0	-	0	4	-
19.		Object	C++	Memento	5	7	40.0	0	0	-	0	0	-	0	2	-
20.		Object	C++	Observer	8	9	12.5	0	2	-	0	2	-	4	12	200.0
21.		Object	Java	State	4	10	150.0	0	4	-	0	4	-	4	24	500.0
22.		Object	C++	Strategy	13	13	0.0	3	3	0.0	3	3	0.0	6	6	0.0
23.		Class	C++	Template Method	5	6	20.0	0	2	-	0	2	-	0	4	-
24.		Object	C++	Visitor	14	14	0.0	4	4	0.0	4	4	0.0	26	26	0.0
	*PL=Programming Language, B = Before, A = After, %=Percentage Change															

Table 4 shows the detailed comparison with respect to the values of four CK metrics. It can be seen from the data of Table 4 that the values of WMC, DIT (Depth of Inheritance Tree), NOC, and CBO increase or remain the same for all design patterns. The only exception is the value of NOC for Decorator. The values of WMC, DIT, NOC, and CBO remain the same for Iterator, Strategy, and Visitor.

Fig. 8(a-d) shows the impact of design patterns on values of four CK metrics for all combinations of scopes and purposes. As is evident from Fig. 8(a-d), after using design patterns, the values of WMC, DIT, NOC, and CBO increase for almost all categories of design patterns. The values of DIT, NOC, and CBO remain the same for the class creational (i.e. Factory Method) design patterns only. The increase in the values of these four CK metrics for most design

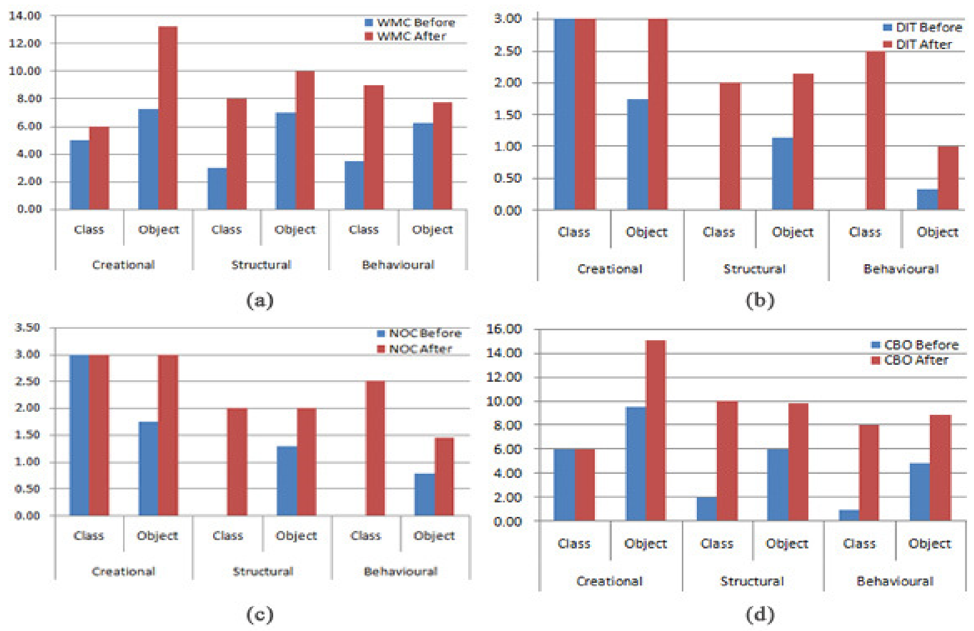


FIG.8. IMPACT ON VALUES OF CK METRICS FOR DIFFERENT CATEGORIES OF DESIGN PATTERNS (A) IMPACT ON WMC (B) IMPACT ON DIT (C) IMPACT ON NOC (D) IMPACT ON CBO

patterns may be attributed to the fact that almost all design patterns use inheritance thereby adding additional parent and child classes and their associated methods.

5. CONCLUSIONS

The aim of this research was to evaluate, quantitatively, the impact of all 23 GoF design patterns on software size and complexity. Program pairs written with and without these design patterns were selected and analyzed for this purpose. The results of this study reveal that design patterns have a positive impact on the average cyclomatic complexity of the system i.e. average complexity decreases for most of the patterns. The impact on the values of CK metrics, number of classes, and size (SLOC) of software, however, is mostly negative.

This research is the first attempt in studying the quantitative impact of all 23 GoF design patterns on software complexity and size. Even though the results seem promising, there is lots of room for further exploration and experimentation. For instance, so far we have looked at programs with medium complexity level. It would be interesting to determine the impact of design patterns on more complex industrial and open-source projects (obtained, for instance, from GitHub). Another beneficial exercise would be to study the impact of using a combination of related design patterns (e.g. Abstract Factory and Factory Method) on program size and complexity. Different assessment tools (other than the ones we selected) could be used for this purpose. Similarly, it also seems worthwhile to replicate this experiment for different programming languages, project sizes, and application domains. Broadening the scope of this study may help us in drawing further insights regarding the impact of design patterns.

ACKNOWLEDGEMENT

The authors acknowledge the Department of Computer Science and Information Technology, University of Lahore, and Department of Computer Science, National University of Computer and Emerging Sciences Lahore, Pakistan for providing facilities to do this research.

REFERENCES

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994.

[2] Riaz, M.N., "Impact of Software Design Patterns on the Quality of Software: A Comparative Study", Proceedings of International Conference on Computing, Mathematics and Engineering Technologies, pp. 1-6, 2018.

[3] McCabe, T J., "A Complexity Measure", IEEE Transactions on Software Engineering, Volume. 2, pp. 308-320, 1976.

[4] Nguyen, V., Deeds-rubin, S., Tan, T., Boehm, B., "A SLOC Counting Standard", COCOMO-II Forum, 2007.

[5] Chidamber, S.R., and Kemerer, C.F., "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, Volume 20, pp. 476–493, 1994.

[6] West, A., "NASA Study on Flight Software Complexity", Technical Report, NASA, 2009.

[7] Lange, D.B., and Nakamura, Y., "Interactive Visualization of Design Patterns Can Help in Framework Understanding", Proceedings of 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 342 -357, 1995.

[8] Wydaeghe, B., Verschaeve, K., Michiels, B., Damme, B.V., Arckens, E., and Jonckers, V., "Building an OMT-Editor Using Design Patterns: An Experience Report", Proceedings of Technology of Object-Oriented Languages, 1998.

[9] McNatt, W.B., and Bieman, J.M., "Coupling of Design Patterns: Common Practices and their Benefits", Proceedings of 25th Conference on Computer Software and Applications, pp. 574-579, 2001.

[10] Subburaj, R., Jekese, G., Hwata, C., "Impact of Object Oriented Design Patterns on Software Development", International Journal of Scientific & Engineering Research, Volume 6, pp. 961-967, 2015.

[11] Prechelt, L., Unger, B., Tichy, W.F., Brossler, P., and Votta, L.G., "A Controlled Experiment in Maintenance: Comparing Design Patterns to Simpler Solutions", IEEE Transactions on Software Engineering, Volume 27, pp. 1134-1144, 2001.

[12] Hegedűs, P., Dénes, B., Rudolf, F., and Tibor, G., "Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability", Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity, pp. 138-145, 2012.

[13] Abdullah, F., "Evaluating Impact of Design Patterns on Software Maintainability and Performance", Thesis, Institute of Informatics Faculty of Mathematics and Natural Sciences, University of Oslo, 2017.

[14] Rudzki, J., "How Design Patterns Affect Application Performance–a Case of a Multi-tier J2EE Application", Scientific Engineering of Distributed Java Applications, pp. 12-23, 2005.

[15] Jeanmart, S., Gueheneuc, Y.G., Sahraoui, H., and Habra, N., "Impact of the Visitor Pattern on Program Comprehension and Maintenance", Proceedings of 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 69-78, 2009.

[16] Aydınoz, B., "The Effect of Design Patterns on Object–Oriented Metrics and Software Error–Proneness", Thesis, Middle East Technological University, Turkey, 2006.

[17] Huston, B., "The Effects of Design Pattern Application on Metric Scores", Journal of Systems and Software, Volume 58, pp. 261-269, 2001.

[18] Hsueh, N.L., Chu, P.H., and Chu, W., "A Quantitative Approach for Evaluating the Quality of Design Patterns", Journal of Systems and Software, Volume 81, pp. 1430-1439, 2008.

[19] Yu, L., and Ramaswamy, S., "An Empirical Study of the Effect of Design Patterns on Class Structural Quality", Handbook of Research on Emerging Advancements and Technologies in Software Engineering, pp. 106-125, 2014.

[20] SourceMaking: https://sourcemaking.com/ [17 December, 2018]

[21] Design Pattern Programs: http://zelogix.com/programs/ [27 November, 2018]

[22] Source Monitor: http://www.campwoodsw.com/sourcemonitor.html [15 September, 2017]

[23] CCCC – C & C++ Code Counter: http://cccc.sourceforge.net/ [02 November, 2018]

[24] GitHub Inc.: https://github.com/ [01 January, 2019]