2019

Reducing the Large Class Code Smell by Applying Design Patterns

Bayan Turkistani
South Dakota State University

REDUCING THE LARGE CLASS CODE SMELL BY APPLYING DESIGN PATTERNS

BY

BAYAN TURKISTANI

A thesis submitted in partial fulfillment of the requirements for the

Master of Science

Major in Computer Science

South Dakota State University

2019

REDUCING THE LARGE CLASS CODE SMELL

BY APPLYING DESIGN PATTERNS

BAYAN TURKISTANI

This thesis is approved as a creditable and independent investigation by a candidate for the Master of Science in Computer Science degree and is acceptable for meeting the thesis requirements for this degree. Acceptance of this does not imply that the conclusions reached by the candidates are necessarily the conclusions of the major department.

Yi Liu, Ph.D.
Thesis Advisor — Date

George Hamer, Ph.D.
Acting Head, Department of Electrical
Engineering and Computer Science — Date

Dean, Graduate School — Date

ACKNOWLEDGMENTS

I want to take this chance to express the most profound appreciation to the people who supported me towards the achievement of this thesis paper. First of all, I want to thank the government of Saudi Arabia SACM for giving me this chance to achieve a degree in United States, which has presented me to a distinct and enhanced learning system and has guided me towards more significant intelligent development. Secondly, I express my thanks and gratitude to my mentor, Dr. Liu, for her patience with me and supervising me through each difficulty she is one of the best professors who I met.

I want to appreciate the South Dakota State University computer department and faculty members for their motivation, assistance, and support. I would like to thanks Arwa for giving me her tools for detecting the Large Class Code smell that I used it to evaluated my methodology.

I take great pleasure to appreciate my husband, Rami Madani, for his assistance and guidance. He has always been there to support me through challenging moments. He has encouraged me and advised me to overcome barriers during various stages of this study. He is always taking care of my daughters and me. In addition, I thank my families and friends. Their munificent words—advice, helpful feedback, and suggestions—have developed my personality towards improvement.

Each part in life has a direct relation with the people who are near to my heart. To my mother and father, I express my thankfulness for their support and belief in me. They have not only encouraged me since childhood, but additionally given me a direction towards more significant successes and bringing me confidence.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

REDUCING THE LARGE CLASS CODE SMELL

BY APPLYING DESIGN PATTERNS

BAYAN TURKISTANI

2019

Software systems need continuous developing to cope and keep up with ever-changing requirements. Source code quality affects the software development costs. In software refactoring object-oriented systems, Large Class, in particular, hinder the maintenance of a system by letting it difficult for software developers to understand and perform modifications. Also, it is making the development process labor-intensive and time-wasting.

Reducing the Large Class code smell by applying design patterns can make the refactoring process more manageable, ease developing the system and decrease the effort required for the maintaining of software. To guarantee object-oriented software stays clear to read, understand and modify over time, Fowler and Beck claimed that these classes should, therefore, be divided into several classes, or extract the subclasses from the Large Class.

The study presents a methodology designed to reduce the Large Class code smell by understanding the feature of the Large Class then analyzing the causes of the Large Class code smell and depends on two features, complexity and cohesion, then classifying the causes to identical types and proposing a best fit design pattern to address each type and refactor the code to improve the quality of software by reducing the complexity and enhancing cohesion. Our methodology focuses on the Large Class code smell while analyzing the complexity and cohesion; however, the methodology itself can be used wherever the code fits in a category.

Chapter 1 Introduction

1.1 Introduction

Areal-world software product needs to develop and improve with time. As to develop and to meet new requirement, software codes get more complicated and divert away from its original design thus this leads to decrease in its quality. Hence software maintenance cost has a large share in total cost of software development [1].

Software refactoring is intended to improve software throughout the development process.” It enhances the internal structure of the software without changing the external behavior of the code”. The concept of software refactoring assists in creating more flexible and reusable software without altering the required functions or method. Benefits involve increased code readability and decreased complexity; as a result, these can enhance software maintainability and generate a more manageable internal architecture. However, flaws in the complex software refactoring process make maintenance more inefficient [2].

Any character in the program code clearly shows a deeper issue describe as a code smell. Martin Fowler et al. [3] named that weakness code smells. Code smell may introduce to design condition that may adversely affect maintenance of the software and gain a system that is hard to alter, which as a result introduce defects [3]. Technically, these code smells are not supposed to be mistaken and do not stop system from functioning. But these code smell may affect the development process, weaken the sustainability of software and increase the probability of its failures.

A class is known as a large class if “it performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes” [4]. Large class breaks the object-oriented design principle that the knowledge of the system than distributed it equally among the higher level classes [4].

“In general design pattern are reusable solution for a common typical problem or difficulty occurring in the software design [6]”. "It gives a plan to refining the subsystems or parts of a software system, or the relationship between them and defines a usual recurring structure of communicating components that resolve a common design problem within a specific setting". Generally, a design pattern offer the interaction and relationship among classes or object, recommending common solution to many design problems, whereas code smells indicate issues in preventing further maintenance of a software system. Intuitively we assume that both ideas exist mutually exclusive, and the existence of examples identifies with the absence of code smells. Walter, Bartosz, and Tarek Alkhaeir [18] found that the nearness of configuration examples with the absence of code smells in the similar classes. The importance of link between design patterns and code smells differ with respect to particular patterns and smells. The ratio of smelly classes participating, and not in design pattern, shows slightly increased in successive release of two analyzed systems. As a result, the number of smelly classes not in a portion of design patterns increases either proportionally to the number of other smelly classes or marginally faster. We could presume that the occurrence of smells within designed classes remains lower or equivalent during the code evolution than for other classes.

Numerous research works use design patterns to solve some design issues. For example, an approach for refactoring anti-patterns of software systems by applying design patterns application has been introduced by Davide, Arcelli, Daniele Di Pompeo [14]. Their approach aimed on removing likely occurring performance anti-patterns by applying design patterns. A ranking procedure supports the decision of applying a design pattern. They gave initial approval of their methodology, showing how ranking system facilitates the three design pattern for removing semi-trucks performance anti-pattern.

The purpose of using design patterns accelerates the development procedure by giving a tried and confirmed development paradigm. Likewise, the design pattern makes the current design reusable for future usage and allows developers to communicate by recognized and understood names for software communications.

1.2 Motivation

The most frequent code smell occurring during programming is the large class. As a result of considering the large class as one of the most harmful bad smells in software refactoring affecting maintenance and quality, this paper focuses on reducing the large class code smell to enhance software quality. In order to refactor large classes, we analyzed the causes of the large class code smell in terms of features and categorized them into different types. The main features of the Large Class are high complexity and low cohesion. This paper proposed a method to refactor software by analyzing the causes of the large class code smell, classifying causes to identical types, and proposing a design pattern to address and refactor each type. We selected a best-fit design pattern to address each type to improve software quality. Currently, no research paper exists classifying the causes of the large class code smell and applying various design patterns to address each type and reduce the large class code smell.

1.3 Objectives

This paper proposed a clear and effective method to refactor the software system by applying different design patterns to reduce the large class code smell and improve the quality of the software. This methodology may possibly facilitate the process of software development and maintenance. Additionally, this methodology will help reduce some code flaws that need significant consideration to create flexible and reusable software. Hence, the objectives of this study can be summarized as follows:

1. To classify the causes of large class code smell to various types.
2. To propose design patterns to address each type or cause of large class code smells.

1.4 Outline of the Study Paper

This research proposed a method to refactor software by analyzing the causes of the large class code smell, classifying the causes to identical types, and proposing a design pattern to address each type. Furthermore, the goal is not on refactoring a particular type of anti-pattern but on reducing cyclomatic complexity and cohesion. We introduce the classification of the causes based on the complexity and cohesion by addressing each category with design patterns to improve quality. Chapter 2 summarizes the related research work on refactoring anti-pattern applications using a design pattern. Chapter 3 illustrates the details on classifying the causes of large class code smell and proposes design patterns to address each type of cause. Chapter 4 evaluates the work by detecting the large classes in a real system and refactoring it by applying various design patterns. To evaluate our methodology, a specific tool detects the large classes from the chosen system at that time refactors in the large classes by applying our methodology and tests the system after refactoring; to support it by using our methodology, we can reduce the large classes. The last chapter concludes with the restatement of the work and presents suggestions for future work.

Chapter 2 Background

This chapter illustrates some basic concepts associated with the development of software. Furthermore, it provides information about the methods and drawbacks used to address the problem.

2.1 Software Refactoring

Martin Fowler et al. [3] explained that “refactoring is a procedure for restructuring an existing software or system and changing its internal structure without altering its external behavior”. Therefore, the system is kept completely working after each refactoring. There is always a need to improve and enhance software, fix problems, and add new features or functions. However, the quality of the software offers an important difference on how adjustable it is to make these changes.

2.2 Code Smell

Martin Fowler et al. [3] “defined a code smell as a surface indication that usually corresponds to a deeper problem in the system”. A code smell is the attribute within a code that represents the quality or feature of the design. In fact, code smell is not recognized a problem and does not stop the system from functioning, but is considered a defect in the system design that leads to a problem or failure in the future.

Martin Fowler et al. [3] gave a list of found code smells:” long method, duplicated code, large class, long parameter list, divergent change, shotgun surgery, feature envy, data clumps, primitive obsession, switch statements, lazy class, speculative generality, temporary fields, middle man, inappropriate intimacy, alternative classes with different interfaces, message chains, incomplete library class, data class, refused bequest, parallel inheritance hierarchies, and comments” [3].

2.3 Large Class

A class is classified as a large class if it has too many responsibilities compared with the remaining classes in the same system. A large class breaks the object-oriented design principle, that is, the functions of a system should be consistently shared between the level classes [5]. A class sometimes may be hard to be understood, because it may include too many functions or lines of codes. A system that contains a large class has low inner cohesion or high complexity. Usually, classes start small but as the program is developed or modified, the developer may attach more functions to a current class rather than creating a new class; this makes the class larger. It is essential to find ways to divide the large classes so that the software can be more cohesive and less complex, which improves the quality and flexibility of the software.

It is necessary to find a method to reduce a large class. The large class can be reduced by extorting some of its responsibility and placing it into a new class. While the development or the refactoring, the large class changes more than the other classes and may produce a harmful influence on the maintenance of the software.

2.4 Design Pattern

A design pattern is “a general reusable solution to a common problem happening in software design” [6]. “It gives a plan for improving the systems or part of a software system or the connection among them and expresses a generally recurring structure of interacting components that solves a general design problem in a specific context”. In this paper, three design patterns are used to help refactor large class code smell: strategy, abstract factory, and observer.

2.4.1 Strategy Pattern

The strategy design pattern is a type of behavioral software design pattern defining a family of algorithms, encapsulating each algorithm in a separate class (Concrete Strategy D, Concrete Strategy C), and making them interchangeable by selecting one of them at runtime Strategy pattern makes the algorithm vary independently from clients applying it [6].

Writing several algorithms into one class is not useful because it makes classes larger and harder to manage. In addition, when one algorithm has to be chosen at each time or each invocation, we do not need to hold multiple algorithms if we do not use them all. Furthermore, it is complicated to attach new algorithms to the existing class. Building classes that encapsulate numerous algorithms can avoid these issues. An algorithm, encapsulated in this form, is referred to as a strategy.

Use of the strategy pattern is a suitable choice when several associated classes only vary in their behavior, because strategies give a process to configure a class with only one of several behaviors when we need several variants of an algorithm and when an algorithm employs data that users should not know. A class defines several behaviors, and these present in multiple conditional statements. Instead of defining many conditional statements, separate each conditional branch into a strategy class. The structure of the strategy pattern is shown in Figure 1.

Strategy interface defines operations or methods common to the concrete strategy classes. Each concrete strategy implements operations in the strategy interface class. The class that applies the strategy interface to call the algorithm defined by a concrete strategy is the context class.

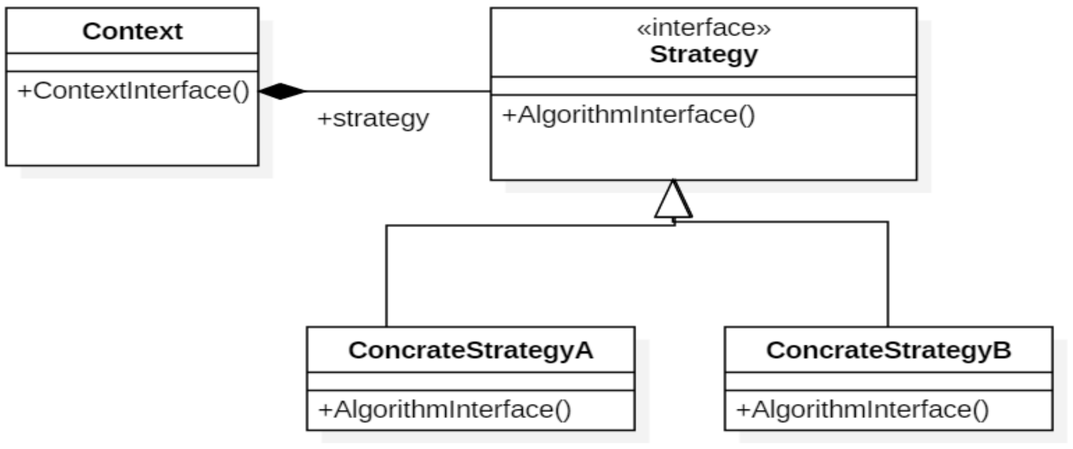


Figure 1 Strategy pattern [6]

2.4.2 Abstract Factory Pattern

The abstract factory pattern is one of the creational patterns that “provides an interface for creating families of related or dependent objects without specifying their concrete classes” [6].

Apply the abstract factory design pattern if a program needs to be unknown of how its products or objects are created, designed, and detailed and in case of a system should choose one of many families of products. Furthermore, when a family of associated product objects is created to be used together and when we need to present a class library of products and we want to show only the interfaces and not the implementations, we need to implement this constraint.

The individual instance of a concrete factory class is executed at execution time. Also, this concrete factory provides product objects that have a special implementation. The user should use a separate concrete factory to create various product objects. Abstract factory delays production of product objects to its concrete factory subclass. Figure 2 illustrates the structure of abstract factory pattern.

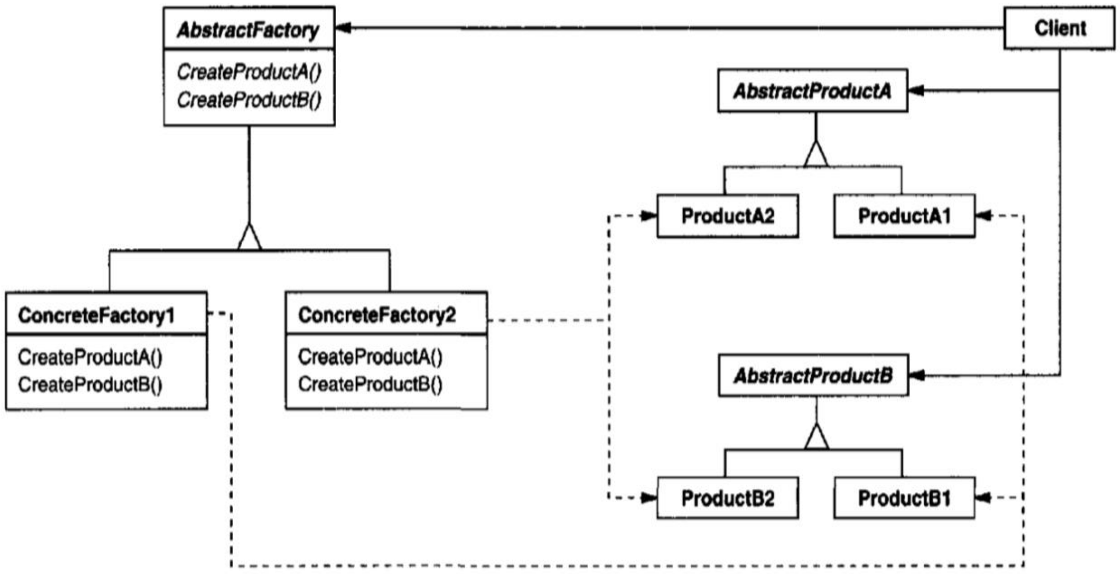


Figure 2 Abstract factory pattern [6]

Abstract factory is an interface for operations creating abstract product objects. Concrete factory (ConcreteFactory1, ConcreteFactory2) performs the operations to produce concrete product objects. Abstract product (Abstract Product A, Abstract Product B) provides an interface for a type of product object. Concrete product (ProductA1, ProductB1) provides a product object to be performed by the corresponding concrete factory and executing the abstract product interface. The client uses only interfaces indicated by the abstract factory and abstract product classes.

2.4.3 Observer Pattern

Observer design pattern “defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically” [6]. The observer pattern allows us to deal with the subjects and observers independently, so we do not need to modify the subject or the other observers when we add a new observer.

Observer pattern can be used if a system has two phases, one reliant on the other. Separating these phases in different classes makes the system reusable and easy to maintain. In addition, the observer pattern is a suitable pattern when a change of one object needs changing other objects that depend on it, and we do not have an idea to know how many objects require to be altered Moreover, using an observer pattern is appropriate when one object need to notify or inform other objects without making assumptions about these objects, which reduces object coupling. The structure of the observer pattern is shown in Figure 3.

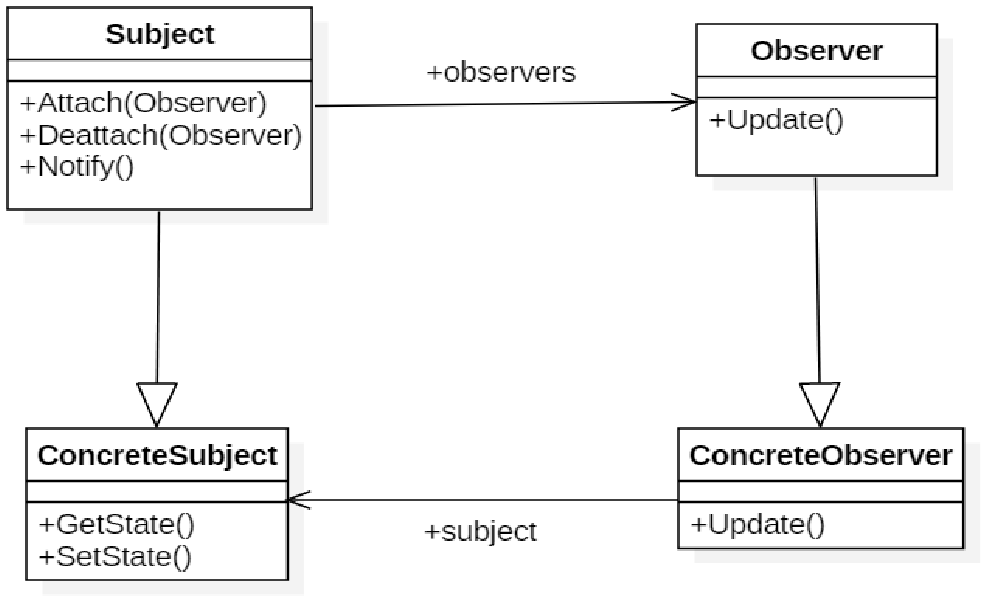


Figure 3 Observer pattern [6]

The observer interface provides an updating interface to the other observers that need to be informed of any changes that happen in a subject. The subject identifies its observers, and it provides an interface for attaching, detaching, and notifying observer objects. ConcreteSubject maintains the state of concern to ConcreteObserver objects and sends a notification to its observers when its state alters.

ConcreteObserver maintains a reference to the ConcreteSubject object, and it performs the Observer updating interface to maintain its state compatible with the subject.

2.5 Related Work

An automated approach presented to refactoring dependent on configuration designs in Java programs. The authors proposed a method involving an inference rule and a refactoring methodology for Java code alteration. The inference relies upon the extraction of system design from a Java code and its portrayal as a lot of prolog-like predicates that are next changed to prolog realities. Inference rules are, further, defined for each target pattern, and then they are converted to prolog rules. The identification of a refactoring procedure happens through the issuing of prolog questions. The technique applied for the process of refactoring applicants to the abstract factory design pattern [9].

Christopoulou et al. [11] have recommended the automated presentation of the strategy pattern and its distinction from the required design pattern. In the proposed refactoring, it includes conditional statements considered by relationships to the strategy design in fact of the selection mode of strategy. This technique, moreover, defines the method of refactoring to strategy that recognized conditional statements. For some exceptional states of these statements, a method is offered for total alteration of conditional logic with method calls of appropriate concrete strategy instances. The refactoring methodology and description algorithm are performed and combined in the JDeodorant Eclipse plug-in.

Gaitani et al. [12] have introduced a technique for automatic refactoring to the null object design pattern. They provided a design clarification for a system that has code duplication and complexity caused by a condition that generated by repeated null-checks in the program. The technique focuses on the avoidance of null-checking conditionals by refactoring the code to null object design pattern. They introduced different cases of conditional statements that can be discharged by applying a null object design pattern. They introduced the transformation process of the code source and set of refactoring preconditions to safely refactor the null-checking conditionals to the null object design pattern. The refactoring procedure is performed and combined with the two systems JDeodorant Eclipse plug-in.

Zafeiris et al. [13] proposed a suggestion moving semi-automated refactoring to the template method. The suggestion indicated by the programmers shows a number of strategies for two different classes that offer a typical abstract class. The developer applied and extended current strategies for clone recognition and extraction to recognize the familiar and various statements of the examined methods. The variety was extorted to a new method such that method have preconditions and must be fulfilled for conduct and new strategies have a typical signature to be polymorphically invoked from the template method. The original method familiar parts are proposed as a single template method to the basic superclass of the modified classes. Also, an extra abstract method is made in the superclass that has a similar sign with the separated strategies. The refactoring strategy of this work has applied as an eclipse plug-in.

Jafaar et al. [15] assessed the effect of the design pattern with reference to variations and faults. "They examined six diffrent design pattern and 10 anti-patterns in 39 releases of JFreeChart, ArgoUML, and XercesJ". They showed that in practically all versions of the three systems, classes having conditions with anti-patterns have a higher risk of defects than other classes; however, this isn't continuously true for classes having dependencies with design patterns. They also demonstrated that most basic changes affecting classes having dependencies with anti-patterns are structural changes

Walter et al. [17] investigated the relationship between the appearance of design patterns and the number of code smells, depended on the result obtained from the examination of two different systems. Their finding shows that in the same classes, the occurrence of design patterns associated with the absence of code smells.

Fontana et al. [20] concentrated on the division of code smell severity by the usage of machine learning techniques in various operations. In fact, code smells with high hardness can be difficult and produce many problems to the maintainability of software a system. "They applied many machine learning models, spanning from multinomial classification to regression, plus a method to apply binary classifiers for ordinal classification". They summarize and differentiate the performance of the designs according to their precision and four various performance standards applied for the evaluation of ordinal classification procedures.

Chapter 3 Methodology

This chapter illustrates a methodology of refactoring the software by analyzing the causes of the large class code smell, classifying the causes to identical types, and proposing a design pattern to address and refactor each type. We selected a best-fit design pattern to address each type to improve the software quality.

3.1 Classify the Causes of Large Class Code Smell

In order to refactor large classes, we need to understand the features, or the causes, of the large class and then classify the causes to different types. “A large class code smell features (1) a high complexity, and (2) a low, inner-class cohesion” [4]. A set of code metrics applied to express those features is listed below.

1) Weighted Method Count (WMC) is” represented as the total of complexities of all methods stated in a class” [7].

Complexity is the quality characteristic developed for most of the metric systems and is analyzed in correlation with other characteristics. Complexity also determines the cost of the applications, as well as the effort to maintain and evolve existing systems [16].

McCabe’s cyclomatic complexity was applied as a complexity measure for the method. “The cyclomatic complexity of source code is the number of linearly independent paths within it, denoted as $V(G) = P + 1$, where P is the number of predicate nodes” [8]. For example, if the program does not include control flow statements which are (conditionals or decision points), the total of the program complexity will be one which means that there is just a individual path within the code. If the source code has one single condition IF statement, then there would be two paths within the source code (one for the If condition to be TRUE and another one for If condition to be FALSE) and the total complexity will be 2. Three nested single-condition IFs, or one IF with three conditions, will generate a complexity that equal 4. The complexity of the source code can be determined by applying the formula "V(G) = E - N + 2, where E - number of edges N - number of nodes or V (G) = P + 1, Where P = number of predicate nodes that contain condition"[8].method. “The cyclomatic complexity of source code is the number of linearly independent paths within it, denoted as V(G) = P + 1, where P is the number of predicate nodes“[8].

2) Tight Class Cohesion (TCC) “is the relative number of direct connections between methods in the class” [9].

If two methods are entering the same variables of the class, these two methods are directly connected. TCC is calculated by NDC/NP, where NDC refers to the number of direct associations between methods and NP is the number of total possible indirect or direct links between methods in each class.“N represents the number of methods in the class, so NP= N*(N-1) / 2” [9]. A class is considered to be large when ((WMC ≥47) ∧ (TCC < 0.3)).

We analyzed the causes of the large class code smell in terms of the features and categorized them into four types:

T1- Large Class with High complexity (WMC ≥47) caused by if-else statements, which belong to the same family.

T2- Large class with high complexity (WMC ≥47) caused by if-else statements, which belong to 2 or more families.

T3- Large class with low, inner-class cohesion (TCC < 0.3) that contains a graphical user interface.

T4- Large class with low, inner-class cohesion (TCC < 0.3) that does not contain a graphical user interface.

3.2 Refactor with Design Patterns

We proposed the most appropriate design pattern to address each type of the large Class code smell (T1-T4) and developed a guideline on applying the design patterns to restructure the code with large class code smell.

3.2.1 Reduce the Complexity

A program with high complexity is most likely difficult to understand, test, maintain, and reuse. There is an essential requisite for a mechanism to decrease complexity by enhancing internal software quality.

McCabe's cyclomatic complexity, applied to calculate WMC, counts the control flow statements, including conditionals and decision points. The strategy pattern or abstract factory pattern can be applied to restructure the conditional statements according to the purpose and selection mode of strategies [6]. We considered the following two cases: (a) conditional branches involve reciprocally or alternately exclusive, corresponding to substitutional algorithm implementations represented by strategies; and (b) branch selection or choice is managed by the users, in proportion to strategy selection in the strategy pattern and abstract factory pattern [6].

3.2.1.1 Applying Strategy Pattern to Address T1 Large Class

The strategy pattern is a behavioral software design pattern that "defines a family of algorithms, encapsulates each one, and makes them interchangeable." "Strategy lets the algorithm vary independently from clients who use it" [6]. The strategy allows the selection of an algorithm at runtime. The program receives a request from a client on which algorithm in the family should be applied and dynamically runs the chosen algorithm. The strategy pattern can be used to address the T1 large class, which is a class with high complexity containing a group of choices belonging to the same family.

The strategy pattern is used for addressing states where there are various paths of logic available, and one of them has to be chosen depended on some condition(s). It is a cleaner extensible alternate to the code with too many if-else statements or switch cases. The strategy pattern basic flow has been introduced as follows:

1. An entry point receives a choice made by a user or system (branch selection controlled by its clients).
2. Conditional branches involve mutually exclusive, one out of many algorithms or paths of logic to be selected to perform.
3. The chosen algorithm is executed.

By replacing the conditional statement with the strategy pattern, we reduce the complexity that causes if-else statements or switch cases [18].

Figure 4 illustrates the code of a console calculator application that provides four operations (addition, subtraction, multiplication, and division) on two numbers. In this example, each calculation strategy is presented in a case statement. Using switch cases or if-else statements is a suitable choice when there are a small number of options. But with huge number of choices or algorithms, the class that contains all these methods and logic becomes complicated. Our purpose was to produce separate classes to calculate each strategy, then adjust the original class to make it more clear. Since algorithms are in the same family, we can abstract them into the abstract strategy interface named Operator shown in Figure 5.

```
switch (user_Choice)
{
case"+":
	result = Add (Num_one, Num_Two)
	break;
case"-":
	result = Sub (Num_one, Num_Two)
case"*":
	result = Multiply (Num_one, Num_Two)
	break;
case"/":
	result = Divide (Num_one, Num_Two)
	break:
}
```

Figure 4 The calculator implemented in switch statement [22]

```
 public interface Operator
{
	int Operation(int a, int b);
}
```

Figure 5 Abstract strategy interface

Each algorithm is now defined in a concrete strategy class that implements the interface operator. Figure 6 shows how a class implements the functionality of "add."

```
public class Op_Add : operator
{
	public int Operation (int a, int b)
	{
		return a+b;
	}
```

Figure 6 Concrete strategy-op add

By applying the strategy pattern, the switch statement can be restructured to the group of classes shown in Figure 7.

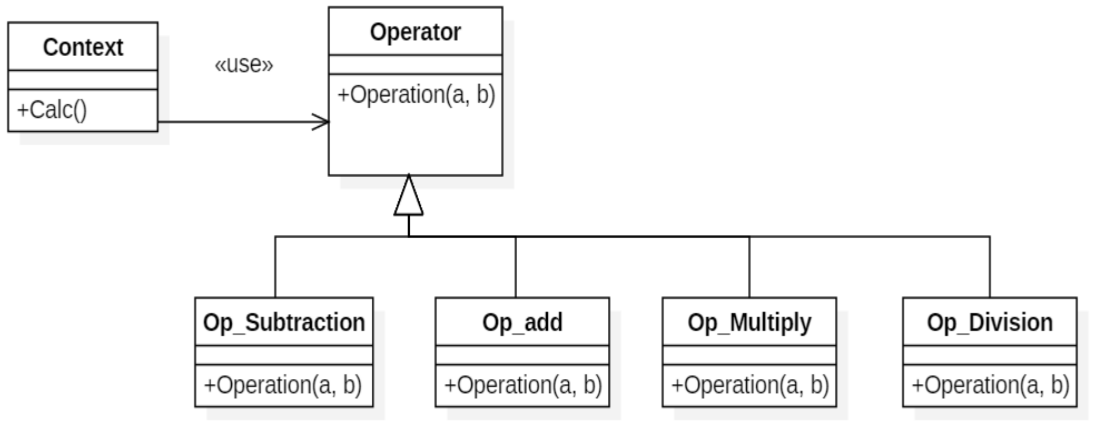


Figure 7 Restructured switch statement by applying strategy pattern

3.2.1.2 Applying Abstract Factory Pattern to Address T2 Large Class

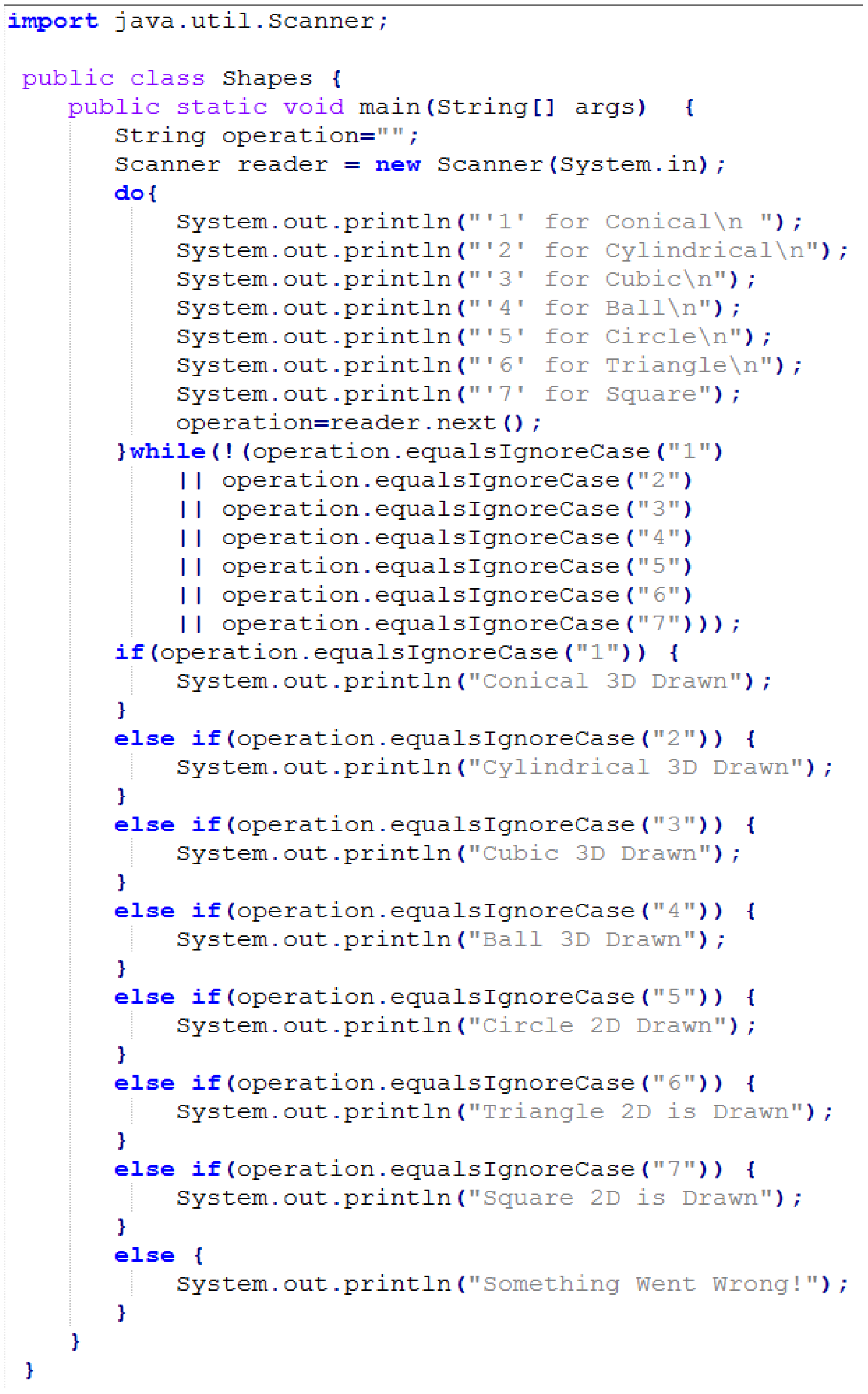
The abstract factory pattern is used to address the T2 large class—a class with high complexity caused by if-else or switch statements that belong to two or more families. The refactoring steps include:

Step1: Categorize the if-else or switch statements to various families of related or dependent products with a common theme.

Step2: Map the families to the components of the abstract factory pattern by creating an abstract product interface for each family and implementing each abstract product with a concrete product class.

Step 3: Create the abstract factory interface and extend it with concrete factory classes that use the concrete product classes from Step 2.

Figure 8 is the partial code of a program that takes a user's input in different shapes. In this example, we can classify the if-else and switch cases to two families: 2D shape and 3D shape. The 2D shape family contains the products circle, tringle, and square, while the 3D shape family includes the products conical, cylindrical, cubic, and ball. As shown in Figure 9, the two families are mapped to classes, factory_2D and factory_3D, extending the interface abstract factory. The products are further organized under product interfaces shape_2D and shape_3D. As a result of applying the abstract factory pattern, the complexity is reduced from V (G) = 48 to V (G) = 13.



```
import java.util.Scanner;

 public class Shapes {
    public static void main(String[] args)  {
        String operation="";
        Scanner reader = new Scanner(System.in);
        do{
            System.out.println("'1' for Conical\n ");
            System.out.println("'2' for Cylindrical\n");
            System.out.println("'3' for Cubic\n");
            System.out.println("'4' for Ball\n");
            System.out.println("'5' for Circle\n");
            System.out.println("'6' for Triangle\n");
            System.out.println("'7' for Square");
            operation=reader.next();
        }while(!(operation.equalsIgnoreCase("1")
            || operation.equalsIgnoreCase("2")
            || operation.equalsIgnoreCase("3")
            || operation.equalsIgnoreCase("4")
            || operation.equalsIgnoreCase("5")
            || operation.equalsIgnoreCase("6")
            || operation.equalsIgnoreCase("7")));
        if(operation.equalsIgnoreCase("1")) {
            System.out.println("Conical 3D Drawn");
        }
        else if(operation.equalsIgnoreCase("2")) {
            System.out.println("Cylindrical 3D Drawn");
        }
        else if(operation.equalsIgnoreCase("3")) {
            System.out.println("Cubic 3D Drawn");
        }
        else if(operation.equalsIgnoreCase("4")) {
            System.out.println("Ball 3D Drawn");
        }
        else if(operation.equalsIgnoreCase("5")) {
            System.out.println("Circle 2D Drawn");
        }
        else if(operation.equalsIgnoreCase("6")) {
            System.out.println("Triangle 2D is Drawn");
        }
        else if(operation.equalsIgnoreCase("7")) {
            System.out.println("Square 2D is Drawn");
        }
        else {
            System.out.println("Something Went Wrong!");
        }
    }
 }
```

Figure 8 Partial code with T2 large class code smell [23]

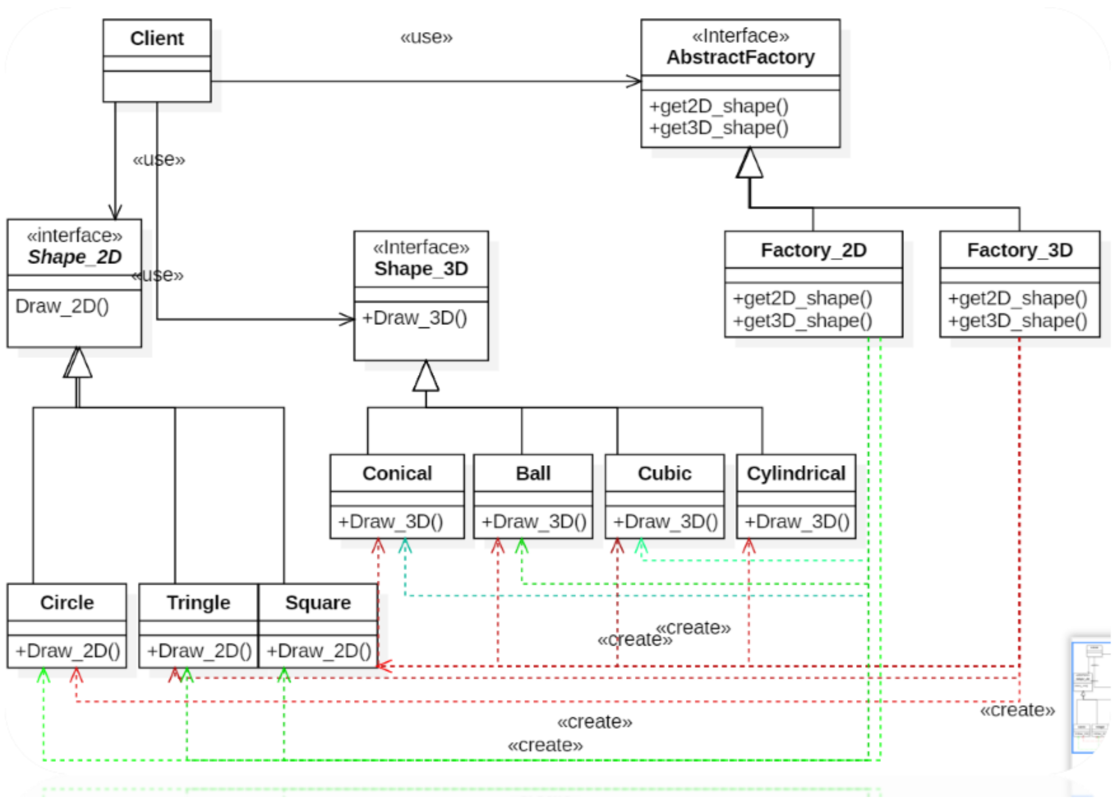


Figure 9 Reducing the complexity by applying an abstract factory pattern

3.2.2 Address Low Inner-Class Cohesion

"Cohesion refers to the degree the elements inside a module relate together" [18]. Cohesion has been classified as one of the essential software quality criteria. A module has the highest cohesion if it expresses exactly single responsibility to perform, and whole its components contribute to this singular task. In opposition, a module with low cohesion leads to unwanted characteristics such as difficulties in modify, develop, read.

3.2.2.1 Applying observer pattern to address T3 Large Class

The traditional approach of GUI programming is to combine the GUI coding and the related operations in the same class. It introduces the inner-class cohesion since the domain data (business logic) and functionalities of user interfaces are bundled together. A large class with low, inner-class cohesion (TCC < 0.3) containing GUI is classified in the T3 large class.

"Observer pattern is a behavioral pattern used when there is one too many dependencies between objects, such as if one object is changed, then its dependent objects are notified automatically". This feature corresponds to one too many dependencies between an operation and GUIs. When an operation causes a change, a related GUI is notified and is changed to be consistent to the changing state.

Observer design pattern applies when an object has a responsibility to inform other objects and no need to make hypotheses about which those objects. In other words, it is to eliminate the tight coupling between classes and improve low inner class cohesion. It is a good candidate to improve the inner cohesion in the T3 large class that contains operations and multiple corresponding GUIs.

The refactoring steps for addressing the T3 large class are specified as follows:

Step 1: Separate the operations from the GUIs and capture them in the subject class.

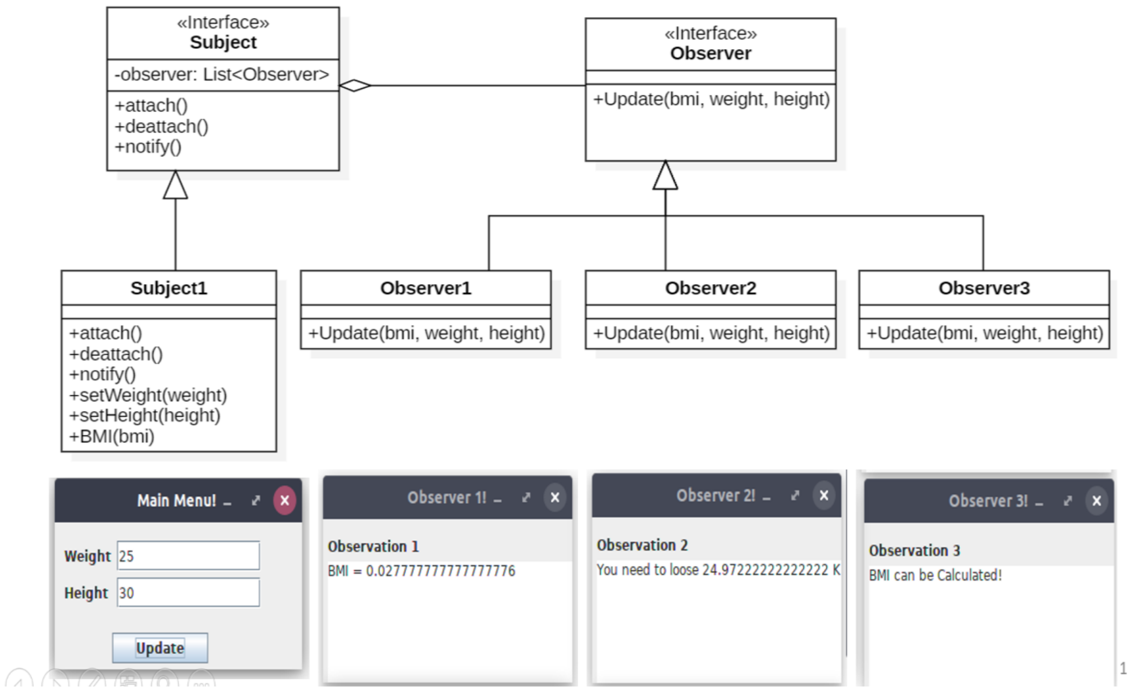
Step 2: Separate the multiple GUIs with each one in a different observer class.

Figure 10 is an example of the partial code with T3 large class code smell. In order to refactor this example code, we distilled the operation for calculating the BMI in the subject. The different displays (observer1, observer2, and observer3) of the BMI results are captured as observer classes. In addition, the subject class is responsible for maintaining a set of observers and notifying them of the alter of the states by calling the update () operation. The observers are responsible for registering and unregistering themselves on a subject to become notified of state changes and to update the state when notified. The design applying observer pattern to the example is shown in Figure 11.

```
 public class BMI
{
        static double weight=0;
        static double height=0;
        static double bmi=0;
        public static void main(String[] args)
        {

                //Observer 1
                JLabel observation1;
                JTextArea jcomp2;
                JFrame frameObserver1;
          frameObserver1 = new JFrame ("Observer 1!");
          frameObserver1.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
          observation1 = new JLabel ("Observation 1");
          observation1.setBounds (5, 10, 100, 25);
          frameObserver1.add(observation1);
          jcomp2 = new JTextArea (5, 5);
          jcomp2.setBounds (5, 35, 255, 125);
          frameObserver1.add(jcomp2);
          frameObserver1.setPreferredSize (new Dimension (270, 165));
          frameObserver1.setLayout (null);
          frameObserver1.pack();
            frameObserver1.setVisible (true);

bUpdate.addActionListener(new ActionListener()
           {
                        public void actionPerformed(ActionEvent ae)
               {
```

```
weight=(Double.parseDouble(jWeightMain.getText()));

height=(Double.parseDouble(jHeightMain.getText()));
                    //observer 1
                    String ob1="";
                    if(height!=0)
                    {
                           bmi=weight/(height*height);
                    }
                    if(weight<1 || height<1)
                    {
                           ob1= "";
                    }
                    else
                    {
                           ob1="BMI = "+String.valueOf(bmi);
                    }
                    jcomp2.setText(ob1);
                    //observation 2
                    String ob2="";
                    if(weight<1 || height<1)
                    {
                           ob2= "";
                    }
```

Figure 10 Partial code with T3 large class code smell [24]



Weight 25
Height 30

Observation 1
BMI = 0.027777777777777776

Observation 2
You need to loose 24.97222222222222 K

Observation 3
BMI can be Calculated!

Figure 11 Applying the observer pattern to reduce the T3 large class

3.2.2.2 Applying Strategy pattern to address T4 Large Class

In this pilot study, we focused on logical cohesion in analyzing the T4 large class. A module has logical cohesion if a logical relation exits between the components of a module, and the components perform a function that is in the same logical class. “In a class with logical cohesion, the elements contribute to activities in the same general category or type” [18].

Defined by Stevens et al. [18], “two processing elements have logical cohesion if at each invocation of the module only one of them is invoked.” Defined by Lakhotia [19], “two variables have logical cohesion if they have a different type of control dependence on the same variable due to the same node.”

Figure 12 shows a module with logical cohesion and its variable dependence graph. The module has logical cohesion. The module calculates the sum of first m integers or the product of first n integers. If the number of the flag equals to 2 then the sum will be calculated, otherwise the product will be calculated.

```
 1 procedure sum_or_product(m,n,flag: integer;
             var sum,prod: integer);
 2 var i,j: integer;
 3 begin
 4   if flag = 1 then begin
 5     i := 1;
 6     sum := 0;
 7     while i <= m do begin
 8       sum := sum + i;
 9       i := i + 1
10     end
11   end
12   else begin
13     j := 1;
14     prod := 1;
15     while j <= n do begin
16       prod := prod * j;
17       j := j + 1
18     end
19   end
20 end
```

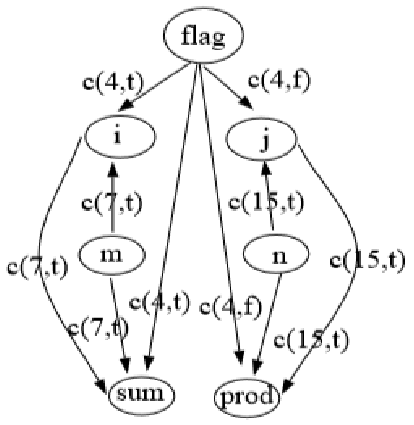
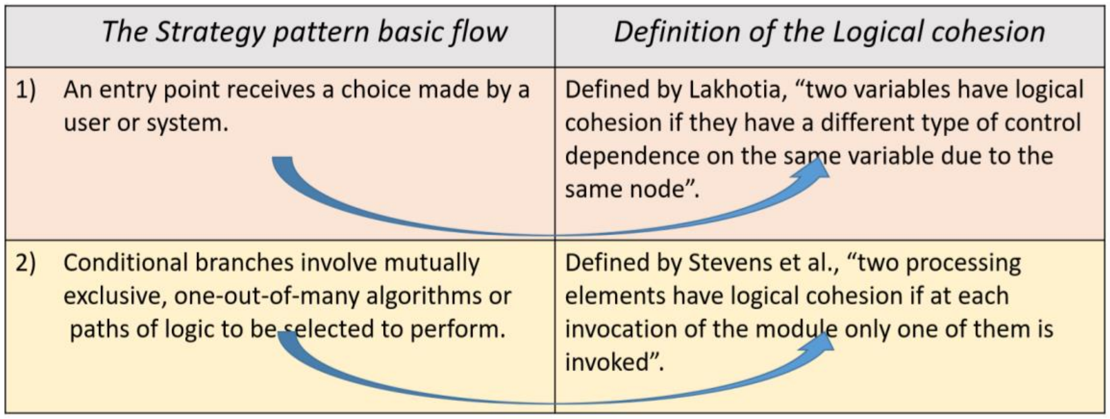


Figure 12 Module with logical cohesion [9]

As described in subsection 1.1, the strategy pattern has (1) an entry point receives a choice made by a user or system, and (2) conditional branches involve mutually exclusive, one out of many algorithms or paths of logic to be selected. The characteristic (1) can address the logical cohesion defined by Lakhotia [19]. The characteristic (2) means at each invocation, one of the concrete strategies will be executed, and it corresponds to the Stevens' definition. The strategy pattern is a good candidate to address the logical cohesion. Figure 13 illustrates the relation between strategy design pattern and logical cohesion. By replacing an entry point that receives a choice with the strategy pattern, we can improve the low inner class cohesion from logical to functional cohesion.



The Strategy pattern basic flow	Definition of the Logical cohesion
1) An entry point receives a choice made by a user or system.	Defined by Lakhotia, “two variables have logical cohesion if they have a different type of control dependence on the same variable due to the same node”.
2) Conditional branches involve mutually exclusive, one-out-of-many algorithms or paths of logic to be selected to perform.	Defined by Stevens et al., “two processing elements have logical cohesion if at each invocation of the module only one of them is invoked”.

Figure 13 The relation between strategy design pattern and logical cohesion

As an example, the strategy pattern is applied to refactor the code in Figure 14. A strategy interface declares an abstract method doOperation (int num1, int num2). The two processes (sum and product) are separated in two ConcreteStategy classes, OperationSum and OperationProduct, and each implements the abstract method in the strategy interface based on its behavior. The context class uses the strategy interface and maintains a reference to the selected concrete strategy class.

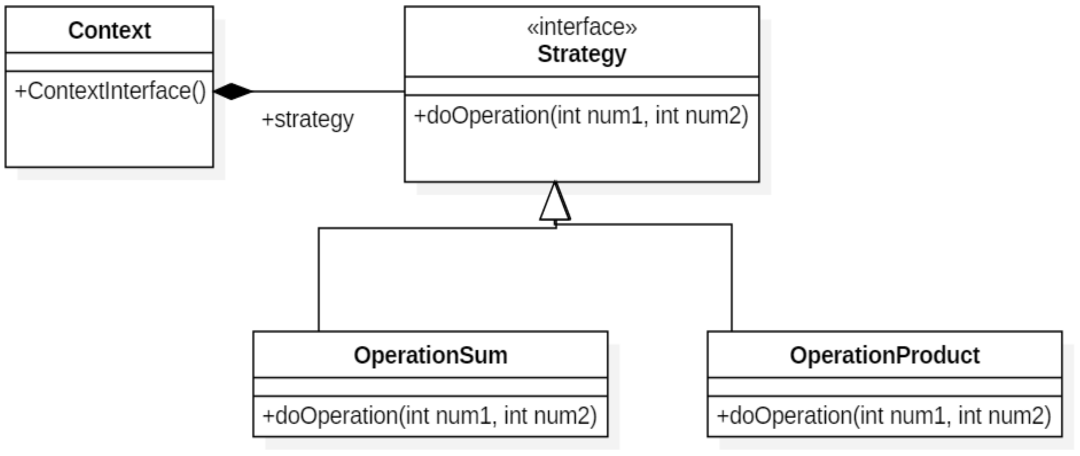

Context
+ContextInterface()
+strategy
«interface»
Strategy
+doOperation(int num1, int num2)
OperationSum
+doOperation(int num1, int num2)
OperationProduct
+doOperation(int num1, int num2)

Figure 14 Applying the strategy pattern to reduce the T4 large class

Chapter 4 Evaluation

The success of this research was evaluated by using an existing system that contained large class code smell and reduces the large classes by applying design patterns suitable for each situation. The significance of our methodology not only focused on reducing a specific type of code smell, but also focused on reducing complexity and enhancing cohesion.

4.1 Evaluate the Performance of the Methodology

The main feature of this methodology was its ability to reduce the large classes from a chosen system by reducing the complexity and enhancing cohesion. This helped in maintenance, reusability, and the quality of the software is improved. A feature of large class code smell is that it focuses on the complexity and cohesion, so we analysed the causes and then classified it into four types (T1 to T4). We proposed T1 and T2, which focused on reducing the complexity, and T3 and T4, which focused on enhancing the cohesion. This research focused on the large class code smell while analysing the complexity and cohesion; however, we can use the methodology itself wherever the code fits in a category.

To evaluate our methodology, we applied a tool to detect the large classes from a chosen system. We randomly selected a system from the Internet. In the first step, we tested the system by using the tool to detect large classes. In the second step we analysed the detected classes to know if it belonged to that classification. In the third step we refactored the large classes by applying the same steps that we proposed in the methodology chapter. In the fourth step, we tested the chosen system again and compared the results before and after applying the design patterns to show the efficiency of our methodology.

The name of the chosen system was Japps-project, and the link is https://www.javatpoint.com/java-application-world-project.“Japps-project system is a Java Application World software where the user can use applications developed in Java such as calculator, notepad+, puzzle game, ip finder, word count tool, source code generator, picture puzzle game, tic tac toe game, and exam system” [21]. The functional requirements of the system are that it can apply all these applications. Figure 15 below shows the main graphical user interface of Japps-project

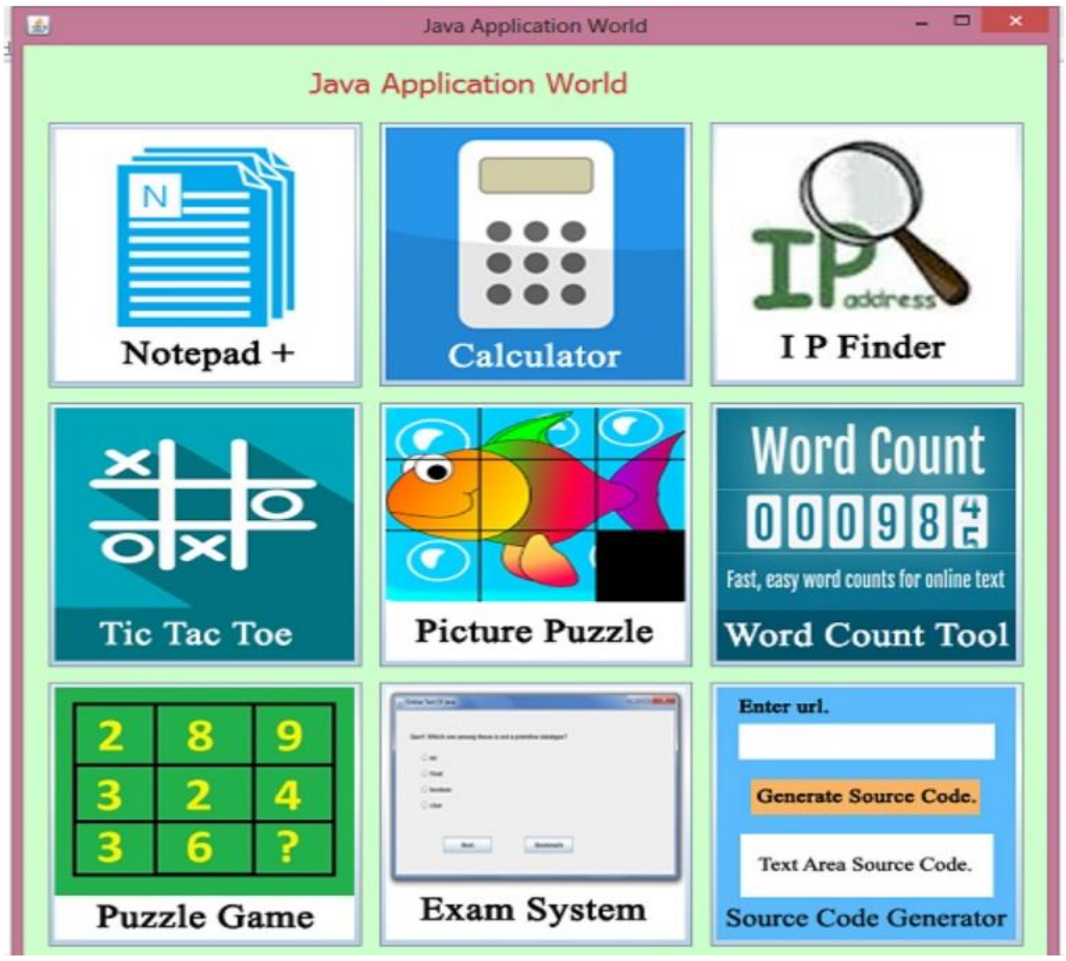
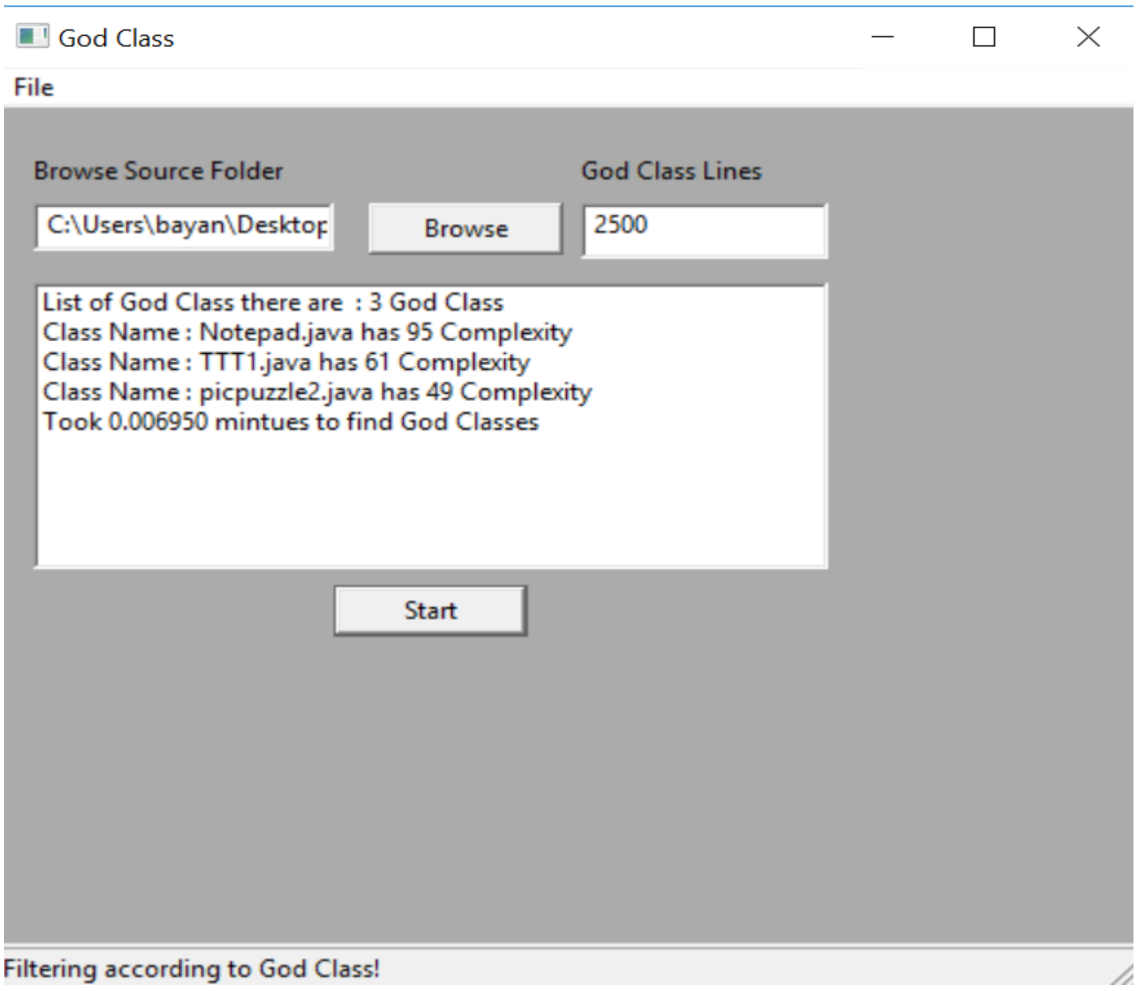


Figure 15 The main graphical user interface of Japps-project system

Step 1. Test the system by using the tool to detect the large classes.

Figure 16 shows three large classes in Japps-project system from 15 classes. The first class is Notepad.java, which has a complexity value = 95. The second class is TTT1.java, which has a complexity value = 61. The third class is picpuzzle2.java, which has a complexity value = 49. According to our classifications of the large class code smell, we needed to understand and analyse the three large classes to categorize them to a suitable type of classification of the large class code smell.



C:\Users\bayan\Desktop

2500

```
List of God Class there are  : 3 God Class
Class Name : Notepad.java has 95 Complexity
Class Name : TTT1.java has 61 Complexity
Class Name : picpuzzle2.java has 49 Complexity
Took 0.006950 mintues to find God Classes
```

Filtering according to God Class!

Figure 16 The large classes in Japps-project system

Each large class belongs to a specific type classified according to features, as explained below:

- The large class Notpade.java belong to the T1 type because it has a high complexity (WMC (95) ≥47) caused by if-else statements, which belong to the same family.
- The large class ShapeDraw.java belong to the T2 type because it has a high complexity
  (WMC (49) ≥47) caused by if-else statements, which belong to two or more families
- The large class TTT1.java belong to the T3 type because it has low, inner-class cohesion
  (TCC < 0.3) that contains a graphical user interface.
- The large class TTT1.java belong to the T4 type because it has low, inner-class cohesion
  (TCC < 0.3) that does not contain a graphical user interface and it has logical cohesion.

4.1.1 T1 Large Class Case

We can use the strategy pattern to address the T1 large class, which is a class with high complexity containing a group of choices, which belong to the same family. “The strategy pattern is a way of addressing situations where different paths of logic are available, and we should choose one of them based on some condition(s)” [3]. It is a cleaner extensible alternate to the code with too many if-else statements or switch cases. According to a preview situation, we found we could apply the strategy pattern to reduce the complexity of the Notpade.java class with too many if-else statements, which belongs to the same family.

The code in Figure 17 illustrates the class of Notpade.java that provides different file operations (Open file, New file, Exit, Print file, Save as file, and Save this file). In this class, we presented each file's operation in if-else statements.

```
if(cmdText.equals(fileNew))
	fileHandler.newFile();
else if(cmdText.equals(fileOpen))
	fileHandler.openFile();
////////////////////////////////////
else if(cmdText.equals(fileSave))
	fileHandler.saveThisFile();
////////////////////////////////////
else if(cmdText.equals(fileSaveAs))
	fileHandler.saveAsFile();
////////////////////////////////////
else if(cmdText.equals(fileExit))
	{if(fileHandler.confirmSave())System.exit(0);}
////////////////////////////////////
else if(cmdText.equals(filePrint))
```

Figure 17 Partial code with T1 Large class code smell [21]

Our purpose was to provide separate classes to implement each operation and then modify the original class to make it more transparent. Since operations are in the same family, we could abstract them into the abstract strategy interface named Strategy shown below in Figure 18.

```
public interface Strategy
{
   public void doOperation(FileOperation fileHandler);
}
```

Figure 18 Abstract Strategy interface – Strategy

We have now defined each operation in a concrete strategy class that implements the interface strategy. The code below shows a class implementing the functionality of "Operation Open file."

```
public class OperationopenFile implements Strategy
{
      @Override
      public void doOperation(FileOperation fileHandler)
      {
            if (!confirmSave(fileHandler))
            {
                  return;
            }
            fileHandler.chooser.setDialogTitle("Open File...");
            fileHandler.chooser.setApproveButtonText("Open this");
            fileHandler.chooser.setApproveButtonMnemonic(KeyEvent.VK_O);
            fileHandler.chooser.setApproveButtonToolTipText("Click me to
open the selected file.!");
```

Figure 19 Concrete strategy – OperationopenFile

Through the application of the strategy pattern, the "if-else" statements could be restructured to the group of classes shown in Figure 20.

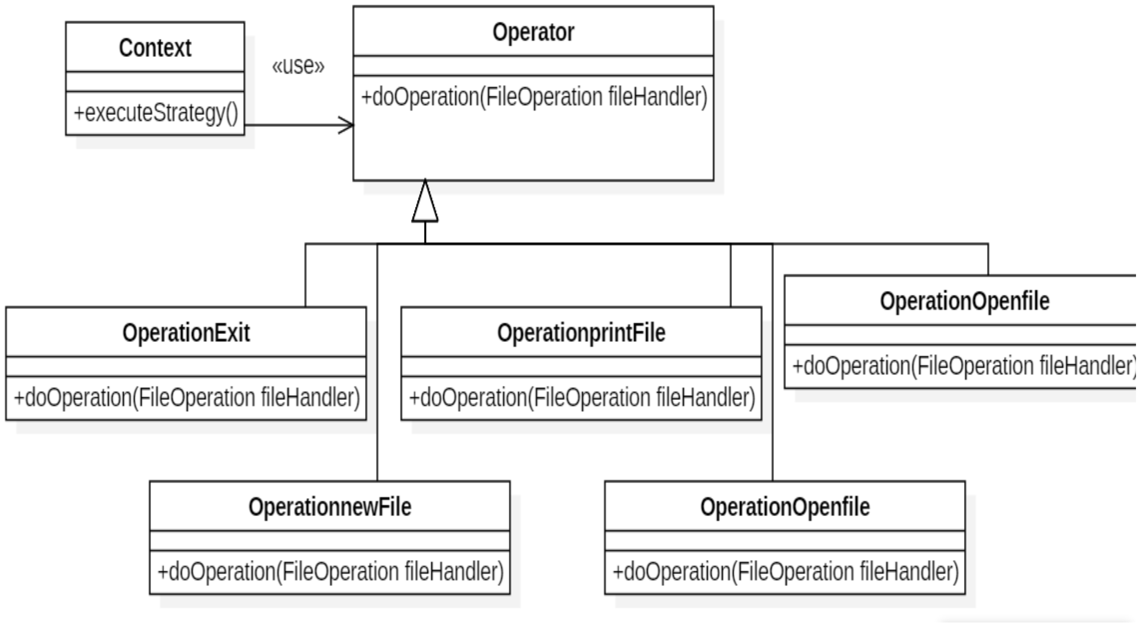
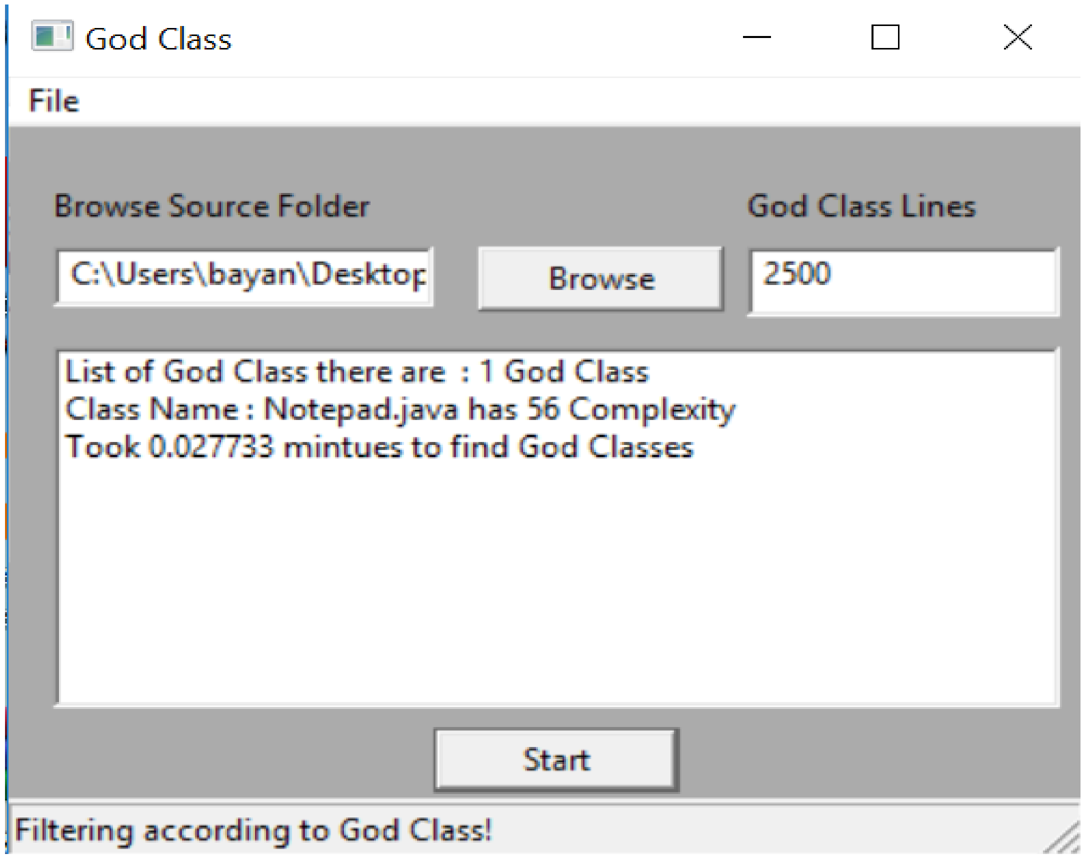

Context
+executeStrategy()
«use»
Operator
+doOperation(FileOperation fileHandler)
OperationExit
+doOperation(FileOperation fileHandler)
OperationprintFile
+doOperation(FileOperation fileHandler)
OperationOpenfile
+doOperation(FileOperation fileHandler)
OperationnewFile
+doOperation(FileOperation fileHandler)
OperationOpenfile
+doOperation(FileOperation fileHandler)

Figure 20 Restructured if-else statements by applying strategy pattern

Because of refactoring the large class by applying the design pattern, the complexity was reduced from 95 to 56; this confirms the benefits of our methodology for reducing the large class code smell. Figure 21 shows the output of the tool after running the refactored Notpade.java class.



C:\Users\bayan\Desktop

2500

```
List of God Class there are  : 1 God Class
Class Name : Notepad.java has 56 Complexity
Took 0.027733 mintues to find God Classes
```

Filtering according to God Class!

Figure 21 The complexity value of Natpade.java after refactoring it

4.1.2 T2 Large Class Case

To address the T2 large class with high complexity caused by if-else/switch statements that belong to two or more families, we used the abstract factory pattern. By applying the same refactoring steps in Chapter 3, we reduced the complexity that occurred in class ShapeDraw.java. The part of the code in Figure 22 shows too many if-else statements, which belong to two families, (Shape family and Color family). In this situation, therefore, we could classify if-else statements into two families to reduce the complexity. The code below shows the part of the Shape.java class that combined to type of if-else statements.

```
case 117:
currentColor = new Color(205, 133, 63);
break;
case 118:
currentColor = new Color(210, 105, 30);
break;
case 119:
currentColor = new Color(139, 69, 19);
break;
case 120:
currentColor = new Color(160, 82, 45);
break;
      }
    }
else
      {
String command = evt.getActionCommand();
if (command.equals("Add"))
      {
if(shapeChoice.getSelectedItem().toString().equals("Rectangle"))
                   {
addShape(new RectShape());
      }
else if(shapeChoice.getSelectedItem().toString().equals("Oval")){
addShape(new OvalShape());
      }
Else if(shapeChoice.getSelectedItem().toString().equals("Round Rectangle"))
```

Figure 22 Partial code with T2 large class code smell [21]

Our purpose was to classify the if-else statements into two families, shapes family and colour family. The shape family contained products such as circle, tringle and square, while the colour family included products such as red, green, and blue. As shown below, the two families are mapped to classes, colourFactory and shapeFactory, which are extending the interface AbstractFatory.

```
public abstract class AbstractFactory {
         abstract Colour getColor(String color);
         abstract Shape getShape(String shape) ;
      }
```

Figure 23 Abstract Factory interface – AbstractFactory

```
public class ShapeFactory extends AbstractFactory {
@Override
public Shape getShape(String shapeType){
if(shapeType == null){
return null;
}
if(shapeType.equalsIgnoreCase("CIRCLE")){
return new Circle();
  } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
```

Figure 24 ShapeFactory.java which extending the interface – AbstractFactory

```
public class ColourFactory extends AbstractFactory
{
      @Override
      public Shape getShape(String shapeType)
      {
            return null;
      }

      @Override
      Colour getColor(String color)
      {
            if(color == null)
            {
                  return null;
            }
            if(color.equalsIgnoreCase("RED"))
            {
```

```
                                return new Red();
                        }
                        else if(color.equalsIgnoreCase("GREEN"))
                        {
                                return new Green();
                        }
                        else if(color.equalsIgnoreCase("BLUE"))
```

Figure 25 ColorFactory.java which extending the interface – AbstractFactory

We further organized the products under product interfaces, colour and shape, as shown below.

```
public interface Colour
{
        Color fill();
  }
```

Figure 26 Color.java product interface

```
public class Ivory implements Colour
{
   @Override
   public Color fill()
   {
             return new Color(255,255,240);
   }
  }
```

Figure 27 Ivory.java product implement color interface

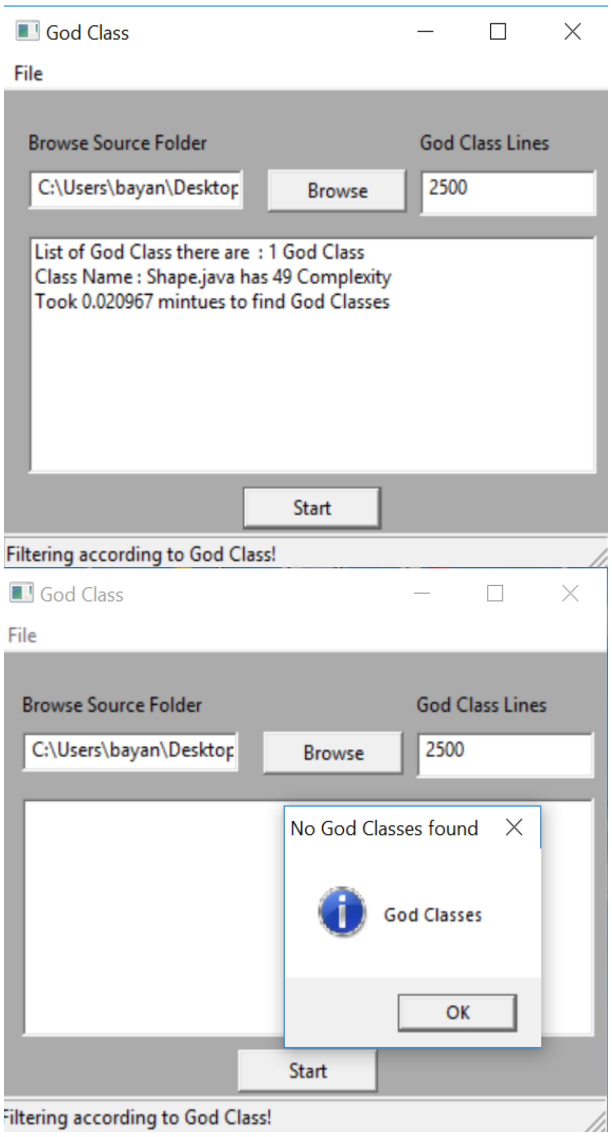
```
public interface Shape {
           void draw();
        }
```

Figure 28 Shape.java product interface

```
public class Square implements Shape {
   @Override
   public void draw() {
```

Figure 29 Square.java product implement shape interface

Since applying the abstract factory pattern, the complexity is reduced from V(G) = 49 to V(G) = < 47. Figure 30 shows the value of the complexity before and after using the tool.



```
List of God Class there are  : 1 God Class
Class Name : Shape.java has 49 Complexity
Took 0.020967 mintues to find God Classes
```

Figure 30 The value of the complexity of T2 before and after using the tool

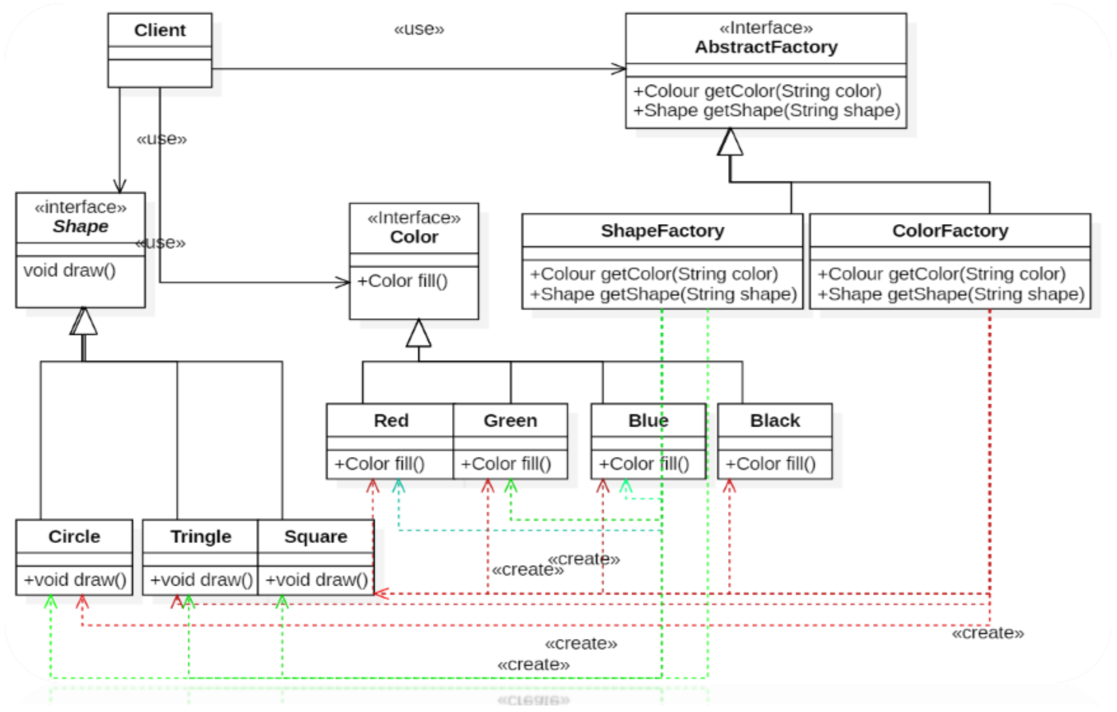
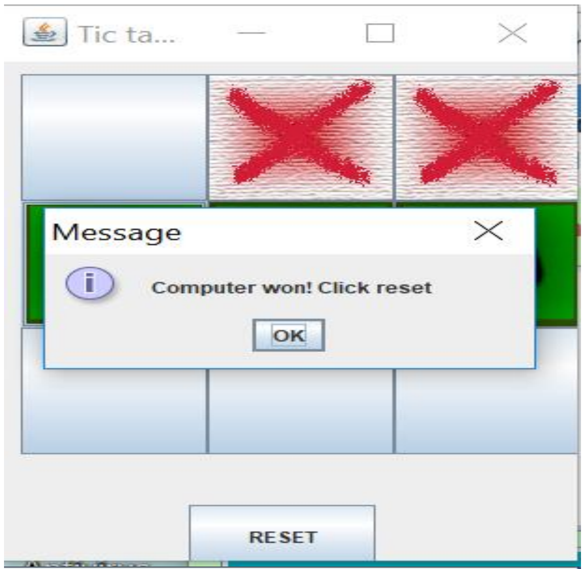


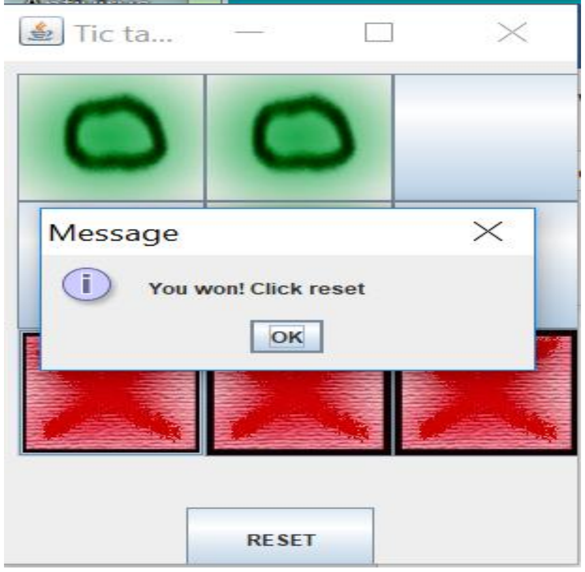
Figure 31 Reducing the complexity by applying an abstract factory pattern

4.1.3 T3 Large Class Case

We classified a large class with low, inner-class cohesion (TCC < 0.3) containing GUI in T3 large class. Observer design pattern reduces the tight coupling between classes and enhances the low, inner-class cohesion. A good candidate improves the inner cohesion in the T3 large class that includes operations and one or multiple corresponding GUIs. In the case of T1, we could use the large class TTT1.java, since it contains two GUIs that depend on the same operation or appear in the same situation. When the user wins, specific GUI appear and when the computer wins, another GUI will appear; thus both GUIs depend on the same situation. Figure 32 shows both GUIs.



Computer won! Click reset



You won! Click reset

Figure 32 GUIs in TTT1.java

The partial code in Figure 33 shows the T3 large class code smell where the GUIs appear in the large class.

```
Icon icon1=b[(a[i][1]-1)].getIcon();
Icon icon2=b[(a[i][2]-1)].getIcon();
Icon icon3=b[(a[i][3]-1)].getIcon();
   if((icon1==icon2)&&(icon2==icon3)&&(icon1!=null)){
               if(icon1==ic1){
                 b[(a[i][1]-1)].setIcon(ic11);
```

```
                b[(a[i][2]-1)].setIcon(ic11);
                b[(a[i][3]-1)].setIcon(ic11);
      JOptionPane.showMessageDialog(TTT1.this,"You won! Click reset");
             break;
                 }
            else if(icon1==ic2){
            b[(a[i][1]-1)].setIcon(ic22);
            b[(a[i][2]-1)].setIcon(ic22);
            b[(a[i][3]-1)].setIcon(ic22);
               JOptionPane.showMessageDialog(TTT1.this,"Computer      won!
Click reset");
```

Figure 33 Partial code with T3 large class code smell [21]

In order to refactor this large class, we distilled the operation for the tic tac toe in subject. We captured the different displays (observer1and observer2) of message box as observer classes. In addition, the subject class was responsible for maintaining a list of observers and notifying them in the change of states by calling the update () operation.

```
abstract class Observer
{
    protected Subject subject;
    public abstract void update();
  }
```

Figure 34 Partial code of Observer.java interface

```
class PlayerObserver extends Observer
{
      Icon ic11;
    public PlayerObserver(Subject subject)
      {
       this.subject = subject;
       this.subject.add( this );
    }
```

```
 public void update()
      {
             int a[][] = subject.getState();
             JButton b[] = subject.getButtons();
             Icon ic1 = subject.getIC1();
```

Figure 35 Partial code of PlayerObserver.java that implement Observer.java interface

```
class ComputerObserver extends Observer
{
       Icon ic22;
    public ComputerObserver(Subject subject)
      {
        this.subject = subject;
        this.subject.add( this );
    }

    public void update()
      {
             int a[][] = subject.getState();
             JButton b[] = subject.getButtons();
             Icon ic2 = subject.getIC2();
             for(int i=0;i<=7;i++)// check for win condition
             {
                    Icon icon1=b[(a[i][1]-1)].getIcon();
```

Figure 36 Partial code of ComputerObserver.java

After refactoring the class by applying the same refactoring steps in Chapter 3, we enhanced the cohesion by separating the operation from the GUIs and capturing the operation in the subject class and the GUIs in the observer classes. The new version of the class, after refactoring it by applying the observer design pattern, is shown in Figure 37. Moreover, it eliminates the complexity. Figure 38 below shows the value of the complexity by using the tool.

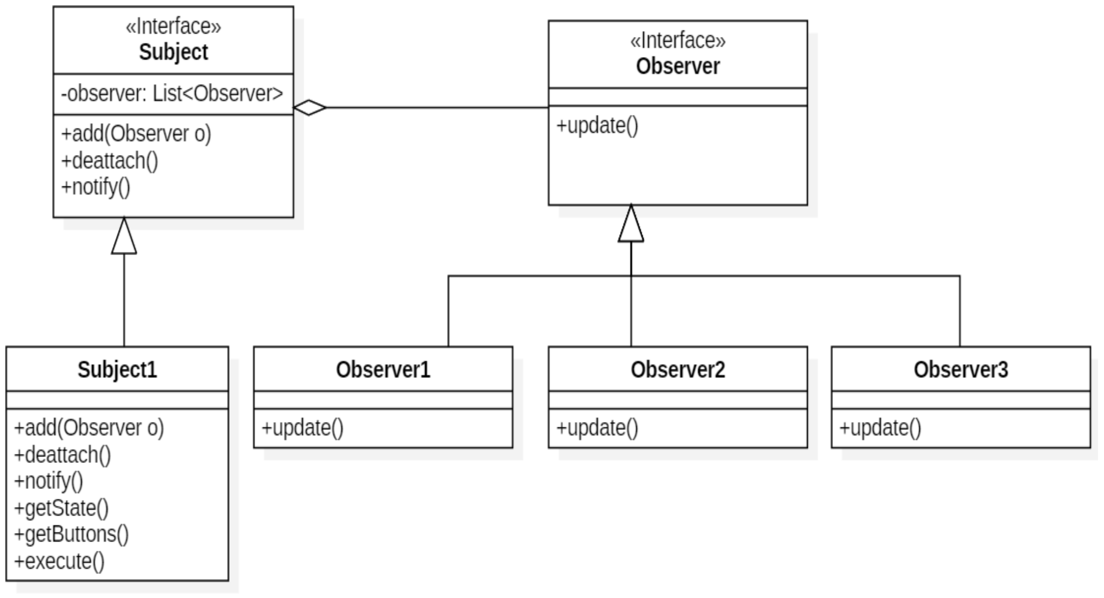
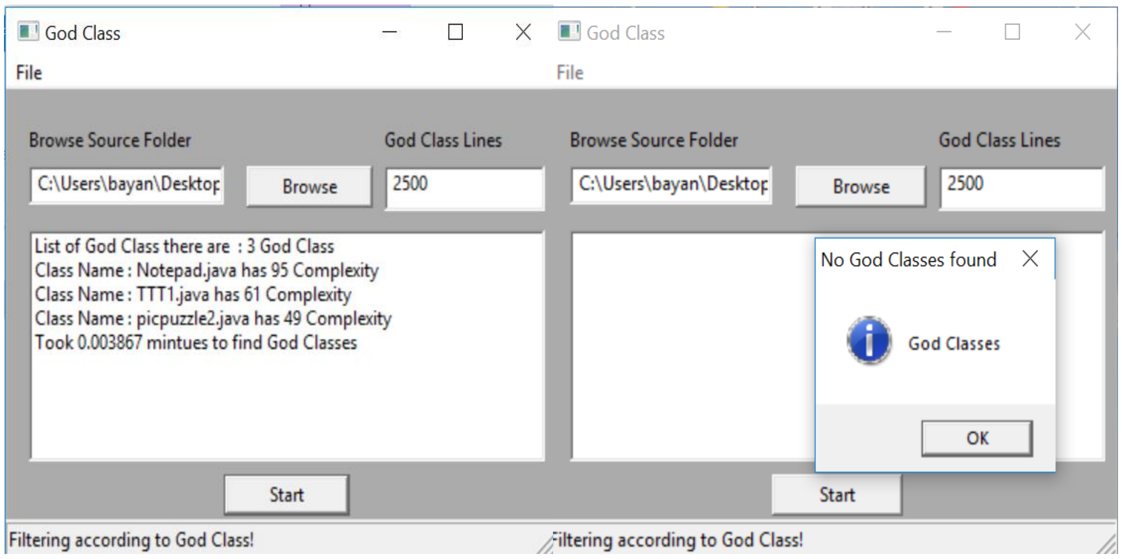


Figure 37 Applying the observer pattern to reduce the T3 large class



C:\Users\bayan\Desktop

2500

List of God Class there are : 3 God Class
Class Name : Notepad.java has 95 Complexity
Class Name : TTT1.java has 61 Complexity
Class Name : picpuzzle2.java has 49 Complexity
Took 0.003867 mintues to find God Classes

Filtering according to God Class!

C:\Users\bayan\Desktop

2500

God Classes

Filtering according to God Class!

Figure 38 The value of the complexity of T3 case by using the tool

4.1.4 T4 Large Class Case

In this case, we focused on logical cohesion in analyzing the T4 large class. The strategy pattern is a good candidate to address the logical cohesion. In the T4 case, we used the large class TTT1.java. "It contains logical cohesion where the elements contribute to activities in the same general category or type" [18]. The class TTT1.java allows the user to choose to play with a friend or the computer. The strategy pattern is a good candidate to address the logical cohesion. By replacing an entry point that receives a choice with the strategy pattern, we could improve the low, inner-class cohesion from a logical to a functional cohesion. The following is part of the class where the logical cohesion occurred:

```
public interface Strategy
{
   public void doOperation(int a1[][],JButton b[], int a[][], ActionEvent
e, boolean state, Icon ic1,Icon ic2,Icon ic11,Icon ic22);
}
```

Figure 39 Strategy.java interface

We have separated the two processes (play with a friend and play with a computer) into ConcreteStategy classes, operationComputer and OperationHuman, where each implements the abstract method in the strategy interface based on its behavior.

```java
public class OperationComputer implements Strategy
{
	@Override
	public void doOperation(int a1[][], JButton b[], int a[][],
	ActionEvent e, boolean state, Icon ic1,Icon ic2,Icon ic11,Icon ic22)
	{
		ImageIcon icon;
		for(int i=0;i<=8;i++)
		{
			if(e.getSource()==b[i])
```

Figure 40 OperationComputer.java that implement Strategy.java interface

```java
public class OperationHuman implements Strategy
{
	@Override
	public void doOperation(int a1[][], JButton b[], int a[][],
ActionEvent e, boolean state,  Icon ic1,Icon ic2,Icon ic11,Icon ic22)
	{
		Icon icon;
		for(int i=0;i<=8;i++)
		{
			if(e.getSource()==b[i])
```

Figure 41 OperationHuman.java that implement Strategy,java interface

The context class uses the strategy interface and maintains a reference to the selected concrete strategy class.

```java
public class Context
{
   private Strategy strategy;

   public Context(Strategy strategy)
   {
      this.strategy = strategy;
   }

   public void executeStrategy(int a1[][], JButton b[], int a[][],
   ActionEvent e, boolean state,  Icon ic1,Icon ic2,Icon ic11,Icon ic22)
   {
      strategy.doOperation(a1, b, a, e, state, ic1, ic2, ic11, ic22);
   }
  }
```

Figure 42 Context.java use Strategy.java interface

Figure 43 shows the UML diagram after refactoring the TTT.java class by applying the strategy pattern. Figure 44 shows the complexity value eliminated after refactoring the large class.

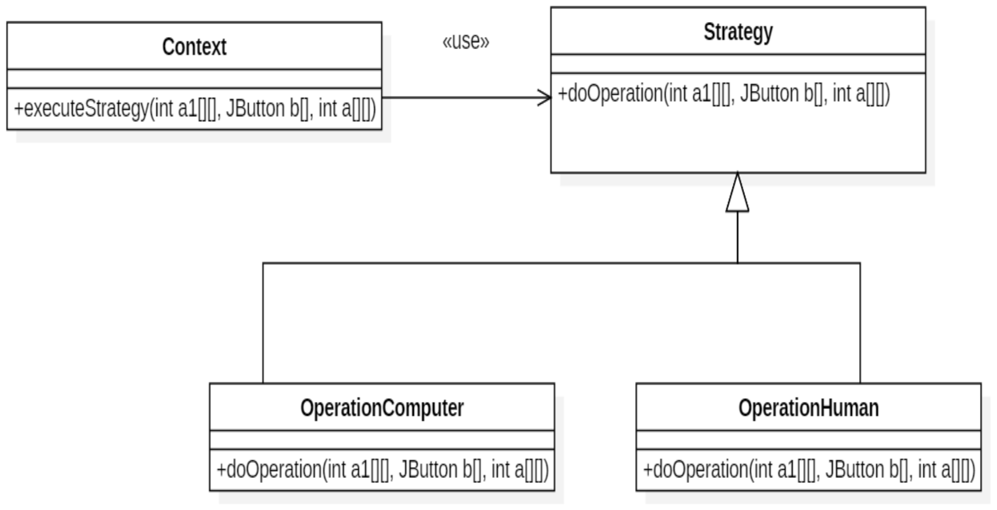
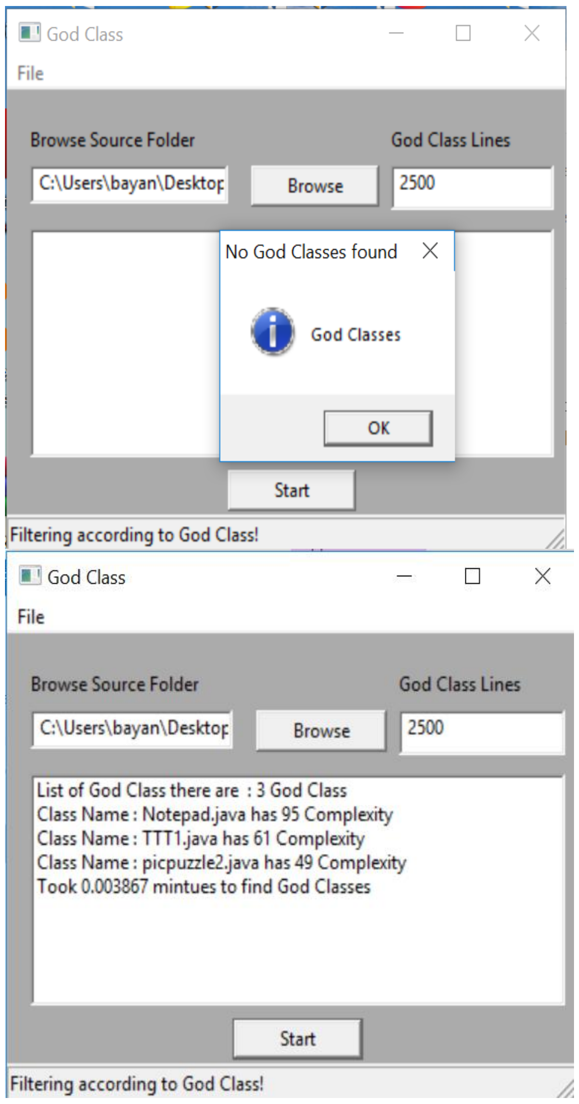

Context
+executeStrategy(int a1[][], JButton b[], int a[][])
«use»
Strategy
+doOperation(int a1[][], JButton b[], int a[][])
OperationComputer
+doOperation(int a1[][], JButton b[], int a[][])
OperationHuman
+doOperation(int a1[][], JButton b[], int a[][])

Figure 43 UML diagram after refactoring the TTT.java class



```
C:\Users\bayan\Desktop
2500
No God Classes found
God Classes
Filtering according to God Class!
```

```
C:\Users\bayan\Desktop
2500
List of God Class there are  : 3 God Class
Class Name : Notepad.java has 95 Complexity
Class Name : TTT1.java has 61 Complexity
Class Name : picpuzzle2.java has 49 Complexity
Took 0.003867 mintues to find God Classes
Filtering according to God Class!
```

Figure 44 The complexity value reduced after refactoring the large class T4

4.2 Summary of the Case Study Results

To conclude the evaluation of our methodology, we summarized the result of the case studies, which is presented in Table 1.

Table 1 Summary of Case Study Results

Case of large class	Class name	Type of large class	Complexity number	Pattern using it	Complexity no. after applying pattern
T1	Notpade.java	Large class with high complexity (WMC ≥ 47) caused by if-else statements, belonging to same family.	WM C= 95	Strategy design	WMC = 56
T2	ShapeDraw.java	Large class with high complexity (WMC ≥ 47) caused by if-else statements, belonging to 2 or more families.	WMC = 49	Abstract design	WMC < 47
T3	TTT1.java	Large class with low, inner-class cohesion (TCC < 0.3) containing graphical user interface.	WMC = 61	Observer design	WMC < 47
T4	TTT1.java	Large class with low, inner-class cohesion (TCC < g.3) not containing graphical user interface.	WMC = 61	Strategy design	WMC < 47

Chapter 5 Conclusion

The methodology for reducing the large class code smell by applying design patterns was produced to improve the quality of the software by reducing the complexity and enhancing the cohesion. The methodology helped to refactor the code to make the maintenance, modification, and reusable easy. The first section demonstrates a summary of the thesis with an overview of our work, and the second section presents future work in fields of reducing the large class code smells.

5.1 Conclusion

A novel dynamic scaling methodology was developed in this research to ease the refactoring process and make the system understandable, reusable, and qualifiable. This methodology proposed a method to refactor the software by analyzing the causes of the large class code smell, classifying the causes to identical types, and proposing a design pattern to address each type. The classification focused on complexity and cohesion.

We classified the causes of the large class code smell to four types (T1 to T4); T1 and T2 focused on reducing the complexity of the code by applying two design patterns. We used the Strategy design pattern to address case T1 large class, which is a large class with high complexity (WMC $\geq$47) caused by if-else statements that belong to the same family. After refactoring type T1 large class by using the strategy pattern, we proved that the complexity is reduced by using our methodology. Furthermore, we used the abstract design pattern to reduce the complexity of case T2 large class, which is a large class with high complexity (WMC $\geq$47) caused by if-else statements, which belong to two or more families. To prove the efficiency of our methodology, we refactored a large class that belongs to the T2 case. As a result of the refactoring process, we could reduce the complexity by using the same steps in the methodology part.

The remaining cases of the large class code smell, T3 and T3, focused on enhancing the cohesion of the code by applying two design patterns. Thee observer design pattern was used to address the T3 large class, which is a large class with low, inner-class cohesion (TCC < 0.3) that contains graphical user interfaces. The methodology chapter details there specific steps the user should follow to refactor. The large class, after refactoring the large class, proved that by applying our methodology, the user could reduce the large class by enhancing the cohesion. Also, we used the strategy design pattern to reduce the large class in case T4, which is a large class with low, inner-class cohesion (TCC < 0.3) that does not contain a graphical user interface. In the T4 case, we focused on the logical cohesion, a program that has logical cohesion if there is a logical relation between the components of a module and the components perform a function that is in the same logical class. After refactoring the large class in case T4 by applying strategy design pattern and running the program in the tool, we could see that we reduced the large class in case T4.

Our methodology focused on the large class code smell while analyzing the complexity and cohesion; however, we could use the methodology itself wherever the code fits in a category. For example, we could use the method to reduce the long method or enhance the duplicate code.

5.2 Future Work

In this research paper we focused on eliminating or reducing Large Class code smell. For future work we will expand our methodology to address other kinds of code smells like" long method, duplicated code, large class, long parameter list, divergent change, shotgun surgery, feature envy, data clumps, primitive obsession, switch statements, lazy class, speculative generality, temporary fields, middle man, inappropriate intimacy, alternative classes with different interfaces, message chains, incomplete library class, data class, refused bequest, parallel inheritance hierarchies, and comments" [3]. By classifying the causes of each code smell and applying suitable design patterns, this will enhance and improve the code smell.

To reduce the large class code smell, we classified the causes of the large class according to its features, which are complexity and cohesion.

- In a complexity case, there are different causes of the complexity, such as like if, while, for, for each, case, default, continue, go to, do, and select. However, in our classification of the large class that depends on the complexity (T1 and T2), we only addressed "if-else" and "switch" statements to reduce the complexity. So, for future improvements, we will focus on reducing the complexity by addressing other causes of the complexity, such as loops or go to, to reduce the complexity.
- In a cohesion case, there are many types of cohesion like coincidental, logical, temporal, procedural, communicational, sequential, and informational. However, we only addressed the logical cohesion in the T4 large class to enhance the cohesion. However, for future improvements, we will expand our methodology to address other types of cohesion.

References

[1] Coleman, D. M., Ash, D., Lowther, B., and Oman, P. W. “Using Metrics to Evaluate Software System Maintainability,” Computer, vol. 27, no. 8, pp. 44-49, Aug. 1994

[2] Mens, T., and Tourwé, T. “A survey of software refactoring.” IEEE Transactions on software engineering 30, no. 2 (2004): 126-139.

[3] Fowler M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. Refactoring – Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

[4] Lanza, M., and Marinescu, R. Object Oriented Metrics in Practice. Springer. 2006.

[5] Riel, A. Object-Oriented Design Heuristics. Addison Wesley.1996.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

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

[8] McCabe, T. (1976). A measure of complexity. IEEE TSE 2(4):308-320, Dec. 1976.

[9] Bieman, J., and Kang, B. (1995). Cohesion and reuse in an object-oriented system. Proceedings of ACM Symposium on Software Reusability (SSR'95), 259-262.

[10] Jeon, S. -U., Lee, J.-S., and Bae, D. H. An automated refactoring approach to Design Pattern-based program transformations in java programs, in: Proc. of Ninth Asia-Pacific Software Engineering Conference (APSEC'02), IEEE Computer Society, 2002, pp. 337–345.

[11] Christopoulou, A., Giakoumakis, E. Zafeiris, V.E., and Soukara, V. Automated refactoring to the Strategy pattern, Inform. Softw. Technol. 54 (2012) 1202–1214.

[12] Gaitani, M. A. G., Zafeiris, V. E., Diamantidis, N., and Giakoumakis, E. Automated refactoring to the null object Design Pattern, Inf. Softw. Technol. 59 (2015) 33–52.

[13] Zafeiris, Vassilis E., Sotiris H. Poulias, N. A. Diamantidis, and Emmanouel A. Giakoumakis. "Automated refactoring of super-class method invocations to the Template Method design pattern." Information and Software Technology 82 (2017): 19-35.

[14] Arcelli, D., and Di Pompeo, D. "Applying Design Patterns to remove software performance Anti-Patterns: a preliminary approach." Procedia Computer Science 109 (2017): 521-528.

[15] Jaafar F., Gu´eh´eneuc Y.G., Hamel S., Khomh F., and Zulkernine M. Evaluating the impact of Design Pattern and anti-pattern dependencies on changes and faults. In: Empirical Software Engineering 2015.21:3.

[16] Enescu, N., Mancas, D. Manole, E., and Udristoiu, S. "Evaluating the correlation between the increasing of the correctness level and McCabe complexity." In Proceedings of the 8th WSEAS International Conference on Applied Computer Science (ACS'08). 2008.

[17] Walter, B., and Alkhaeir, T. “The relationship between design patterns and code smells: An exploratory study.” Information and Software Technology 74 (2016): 127-142.

[18] Stevens, W. P., Myers, G. J., and Constantine, L. L. Structured design. IBM Systems Journal, 13(2):115–139, 1974.

[19] Lakhotia, A. “Rule-based approach to computing module cohesion.” In Proceedings of 1993 15th International Conference on Software Engineering, pp. 35-44. IEEE, 1993.

[20] Fontana, Francesca Arcelli, and Marco Zanoni. "Code smell severity classification using machine learning techniques." Knowledge-Based Systems 128 (2017): 43-58.

[21] “Japps-project”. https://www.javatpoint.com/java-application-world-project

[22] “The calculator implemented in switch statement”. https://stackoverflow.com/questions/36949562/math-program-using-switch-case-statement

[23] “shape_2D and shape_3D” http://forums.codeguru.com/showthread.php?452903-hierarchy-of-shapes

[24] “BMI calculator” http://www.cplusplus.com/forum/general/34447/