Empirical Study And Evaluating The Effect Of Using Design Pattern On Maintainability

Abstract - **Software development process becomes easier and simpler when subjected to object-oriented concepts and designs. Here, the software quality is also important where it takes a long time to exceed the initial development cost and plan. To overcome the issues that are faced by the software, design patterns has been used. The design patterns will act as a solution to enhance a better quality when design and developed with object oriented. Even the design pattern also affect the software solution under maintenance. So, the software aims to measure the effect of using design pattern on maintainability. In order to determine the maintainability, there are several metrics available. To explore the measure and evaluation on maintainability, experiment were conducted on 30 java projects which are object-oriented systems. Through experimentation, design patterns in the project were obtained that affects the maintainability. For each of the programs, CK metrics has been obtained which list out different metrics. To analyze the result, Lines of Code, Coupling Between Object, Weight Method Class and Depth Inheritance Tree values has been computed. The result shows that the design patterns with maximum number of code lines and CBO metric value will affect the software maintainability,**

Keywords – Software Development, Maintainability, Object-Oriented, CK Metrics

I. INTRODUCTION

In the software development process, software quality is an important thing which needs to be highly focused on by the developers. It is also not necessary that the software application which is developed should meet all the requirements and perform specified operation. But there are few attributes for software quality which need to be highly considered by the developer during the software development. Similarly, there are also some non-functional attribute like maintainability, testability, program comprehension and so on. These attributes will literally affect the software quality which needs to be effectively developed. Moreover, the design patterns that are used for development process is also a major cause on software maintainability.

Design patterns are widely used software design in-order to have a better development so that the software can be reused, designed, and accessed for as many number of times. By enhancing the design patterns, production cost and time to redesign the software can be reduced easily. In general design patterns are used to solve the problems in the software design so that better quality can be achieved. According to various situation and problems in the software design, design patterns are classified as architectural pattern, Gang of Four and idiom pattern. But in this report, we are focused over Gang of Four (GoF) pattern which is classified with 23 design patterns. The different GoF patterns is used for different scope which is classified as creational, structural, and behavioral patterns. By using different design patterns, software maintainability and reusability can be understood and implemented in an effective manner. Before the implementation of design pattern, it should be investigated by the designers so that a good understand and experience can be gained. Rather than applying the design patterns, there are still some issues raised which makes more time to understand and identify it.

Nowadays, most of the software applications are not maintained properly which leads to difficult issues and problems. So, software maintainability is a critical attribute in the software development until and even after the deployment of software to the client. And also, cost is also a crucial factor which is completely dependent on maintainability. So, when there is a good maintenance of software, the cost can be reduced effectively and efficiently. Software maintainability is also affected because of using the improper design patterns which reduce the quality of software developed. So, this project is to evaluate the use of design pattern in software that affect software maintainability and performance.

The next section is to review and analysis of related articles where different study and learning can be attained from different authors. Section III is regarding the approach used for an experiment and Section IV evaluates the solution and compare the values obtained. Section V is regarding threats to validity and finally, Section VI concludes the entire work which is analyzed on design patterns.

II. RESEARCH STUDY

In recent times, the developers has not provided much importance to software quality due to several reasons. Due to lack of approaches, software maintainability is affected after the deployment of software. So, the author has conducted different studies on different topics relate to software maintainability and design patterns. In this article, the author has shown how the design pattern impacts the software maintainability. The proposed work uses a maintainability index to measure the software maintainability and along with the different weightages. Here the SonarQube tool has been used which helps to calculate the modularity, complexity and duplicate code in the java projects [1]. According to the analysis, the complexity of the project is determined as 22 which is a higher value. But the refined solution has the lower maintainability than a simpler solution. So, the author concludes that, maintainability is affected with the selection of design patterns like singleton, factory method and so on.

Design patters cannot be predicted with good or bad in using for a software design and implementation. Therefore, the software maintainability also cannot be attained with any of the preferred design patterns to be used. Mostly, the maintainability is dependent on code size, methods used, developer pattern and some quality attributes. Here the author has investigated about the impact and effect of design pattern on software maintainability. Through some literature studies, the author has identified some of the design problems which will critically impact the software application [2]. Here the proposed work carries in developing a tool so that the proper design pattern can be detected from application to check the software quality. Similarly, the analysis is also similar to software maintainability from appropriate pattern which will be more reliable for designers. Finally, the refactoring method has been applied in-order to show the effect on software maintainability with factors like pattern size and so on.

In most of application development, design patterns creates a solution for the problem faced in the design process. This pattern is applied to Object Oriented Software where there are 23 design patterns which is referred as GoF (Gang of Four). The author has given and shown an impact of design pattern on software quality through a systematic literature review. Different studies has been conducted and evaluated based on quality attributes, metrics, design patterns and so on. The aim of author is to show how the design pattern can be implemented and improved on software quality [3]. From the observation, the author has studied about 804 candidate papers and shown the result by applying inclusion and exclusion criteria. The research result shows that there are different designs and consequences that are applied in different modules. The result also concludes that effect of design pattern also affect the software quality and some additional patterns can be improved in the future work.

Software system are often developed based on Object Oriented designs which makes a good practice for developers to improve the quality. Like same way, there are also some cause of hindrances and disharmonies in the software system which is referred as code smells. So, the article aims to show the relation between code smell and design pattern at different levels. Therefore, the author uses a 20-design pattern and 13 code smell from size of small to medium [4]. The evaluation is carried on java-based systems by applying statistical and association rules. Based on the analysis of Java programs, the class contain design pattern and code smell rather than from not participating of class in design pattern. And also, all the design pattern has the code smell according to the subjected systems. The association rules is also attained with some of the design patterns like god calls, blob, duplication smell and so on.

Software flexibility is more important in today's software development for developers and designers. In this situation, software quality attribute also need to be focused which makes the software development to be more adoptable on any of the changes. Sometimes, it is also difficult to provide a flexible software because of requirements and methods used. So, the author has conducted an experiment by develops a project using object-oriented programming [5]. According to the requirements given by the case study, two different design principles has been used namely Open Closed Principle and Single Responsibility Principle which is of from strategy and decorator design pattern. The result is evaluated by comparing the flexible points by before and after applying of design principles. By enhancing design principles, software flexibility is improved a lot along with the development process. Hence, the author has suggested to use of design principles in all the software development to attain flexibility.

The software life cycle is an important phase where the software system can be easily developed with proper delivery and support of other systems. When this phase continuous, proper enhancement like corrections, faults can be easily detected based in the requirements given. Similarly, software maintenance also needs to be considered where additional changes and updates can be emerged with time consuming process. The author uses a maintainability index inorder to evaluate the software maintenance. So, the experimental analysis uses 45 java project with object-oriented software systems [6]. From the result obtained, different software project has approached with different values according to the maintainability involved. There is also variation among the selection project which is clearly defined by the author. According to the characteristics and behaviors, the variants were divided as different clusters to show the differences.

III. APPROACH

Program Name	Link
Apache Commons Lang	https://commons.apache.org/proper/commons-lang/
Spring Boot	https://spring.io/projects/spring-boot
Guava	https://github.com/google/guava
Hibernate ORM	https://hibernate.org/orm/

Elasticsearch	https://www.elastic.co/products/elasticsearch
Spring Framework	https://spring.io/projects/spring-framework
Jackson	https://github.com/FasterXML/jackson-core
Apache Maven	https://maven.apache.org/
JUnit5	https://junit.org/junit5/
Apache Log4j	https://logging.apache.org/log4j/
Apache Ant	https://ant.apache.org/
Gradle	https://gradle.org/
Apache Commons Collect	https://commons.apache.org/proper/commons-collections/
Apache Commons IO	https://commons.apache.org/proper/commons-io/
Quartz Scheduler	http://www.quartz-scheduler.org/
TensorFlow	https://www.tensorflow.org/
Apache Camel	https://camel.apache.org/
Apache Cassandra	http://cassandra.apache.org/
Apache Struts	https://struts.apache.org/
Spring Security	https://spring.io/projects/spring-security
Design-pattern-detection-master	https://github.com/brontec51/design-pattern-detection.git
gym-workout-maker	https://github.com/focus1691/gym-workout-maker.git
Shop Management	https://github.com/Arif-un/Shop-Management.git
OOP_Paint	https://github.com/vukan-markovic/OOP_Paint.git
JAVA_MVC_Swing_Monopoly	https://github.com/moonChenHaohui/JAVA-MVC-Swing-Monopoly.git
Tpdied	https://github.com/Totremont/tpdied.git
object-oriented-design-master	https://github.com/RameshMF/object-oriented-design.git
Stationary-Shop-Management	https://github.com/theanasuddin/Stationary-Shop-Management.git
OOPs-SOLID-Design-Principle	https://github.com/prashantt17/OOPs-SOLID-Design-Principle.git
Software_Design_Pattern-master	https://github.com/Partha-SUST16/Software_Design_Pattern.git

In this empirical study, evaluation of design pattern that affect the software maintainability is the project scope. In-order to carry out the implementation Java projects has been used which contains a design pattern over it. Similarly, the software quality metrics has been used measure and evaluate the maintainability.

About dataset

In-order to carry out the experimental and evaluation analysis, dataset has been collected from GitHub which is an open repository. The subjected programs were considered to be 30 which are all java projects with the size larger than 5k. The purpose of this constrain is that the size less than 5k does not contain any of the design pattern.

Table 1: Data Collection

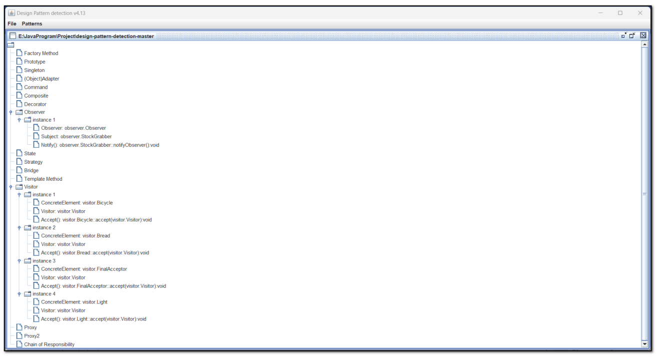
Tools used

To determine the design patterns, Java 2 Runtime Environment has been downloaded and installed. Similarly, the Design pattern detection tool has also been extracted so that different types of design patterns can be identified from the Java projects. This tool is more effective because it contains different packages so that it can identify nearly of about 23 design patterns which is called as Gang of Four (GoF).

Evaluation

From the implementation, different design patterns has been identified on different java projects. These design patterns are from the categories of structural, behavioral and creational. Some of the design patterns are identified as follows:

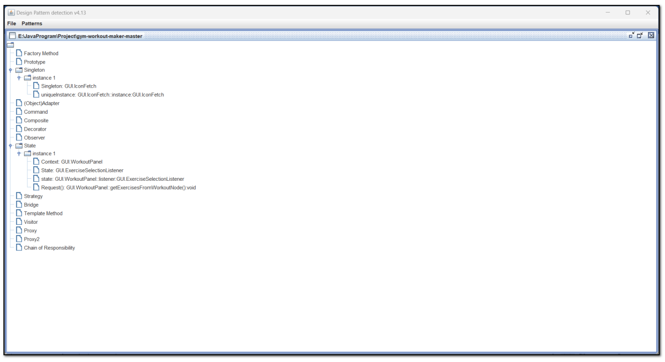
1. *Design-pattern-detection-master*



```
E:\JavaProgram\Project\design-pattern-detection-master
Factory Method
Prototype
Singleton
(Object)Adapter
Command
Composite
Decorator
Observer
  instance 1
    Observer: observer.Observer
    Subject: observer.StockGrabber
    Notify(): observer.StockGrabber::notifyObserver():void
State
Strategy
Bridge
Template Method
Visitor
  instance 1
    ConcreteElement: visitor.Bicycle
    Visitor: visitor.Visitor
    Accept(): visitor.Bicycle::accept(visitor.Visitor):void
  instance 2
    ConcreteElement: visitor.Bread
    Visitor: visitor.Visitor
    Accept(): visitor.Bread::accept(visitor.Visitor):void
  instance 3
    ConcreteElement: visitor.FinalAcceptor
    Visitor: visitor.Visitor
    Accept(): visitor.FinalAcceptor::accept(visitor.Visitor):void
  instance 4
    ConcreteElement: visitor.Light
    Visitor: visitor.Visitor
    Accept(): visitor.Light::accept(visitor.Visitor):void
Proxy
Proxy2
Chain of Responsibility
```

Figure 1:Design pattern for Design Pattern Detection program

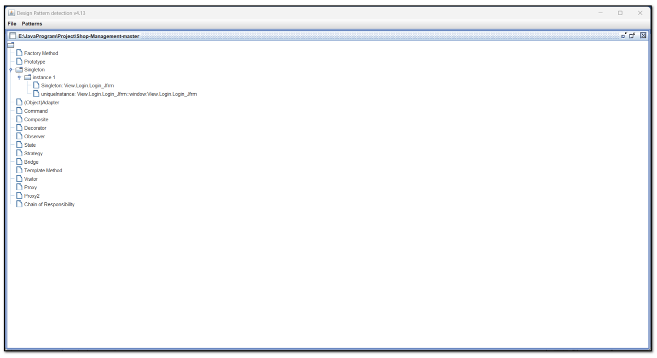
2. *gym-workout-maker*



```
E:\JavaProgram\Project\gym-workout-maker-master
Factory Method
Prototype
Singleton
  instance 1
    Singleton: GUI.IconFetch
    uniqueInstance: GUI.IconFetch::instance:GUI.IconFetch
(Object)Adapter
Command
Composite
Decorator
Observer
State
  instance 1
    Context: GUI.WorkoutPanel
    State: GUI.ExerciseSelectionListener
    state: GUI.WorkoutPanel::listener:GUI.ExerciseSelectionListener
    Request(): GUI.WorkoutPanel::getExercisesFromWorkoutNode():void
Strategy
Bridge
Template Method
Visitor
Proxy
Proxy2
Chain of Responsibility
```

Figure 2:Design pattern for Gym Workout

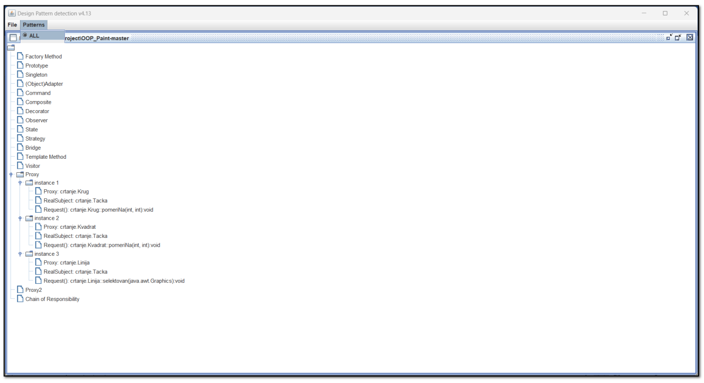
3. *Shop Management*



```
Design Pattern detection v4.13
File  Patterns
E:\JavaProgram\Project\Shop-Management-master
  Factory Method
  Prototype
  Singleton
    instance 1
      Singleton: View.Login.Login_Jfrm
      uniqueInstance: View.Login.Login_Jfrm::window:View.Login.Login_Jfrm
  (Object)Adapter
  Command
  Composite
  Decorator
  Observer
  State
  Strategy
  Bridge
  Template Method
  Visitor
  Proxy
  Proxy2
  Chain of Responsibility
```

Figure 3:Design pattern Identification for Shop Management

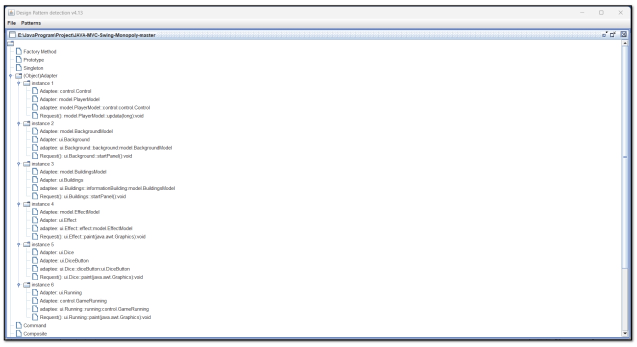
4. *OOP_Paint*



```
Design Pattern detection v4.13
File  Patterns
ALL  roject\OOP_Paint-master
  Factory Method
  Prototype
  Singleton
  (Object)Adapter
  Command
  Composite
  Decorator
  Observer
  State
  Strategy
  Bridge
  Template Method
  Visitor
  Proxy
    instance 1
      Proxy: crtanje.Krug
      RealSubject: crtanje.Tacka
      Request(): crtanje.Krug::pomeriNa(int, int):void
    instance 2
      Proxy: crtanje.Kvadrat
      RealSubject: crtanje.Tacka
      Request(): crtanje.Kvadrat::pomeriNa(int, int):void
    instance 3
      Proxy: crtanje.Linija
      RealSubject: crtanje.Tacka
      Request(): crtanje.Linija::selektovan(java.awt.Graphics):void
  Proxy2
  Chain of Responsibility
```

Figure 4:Design pattern Identification for OOP Paint

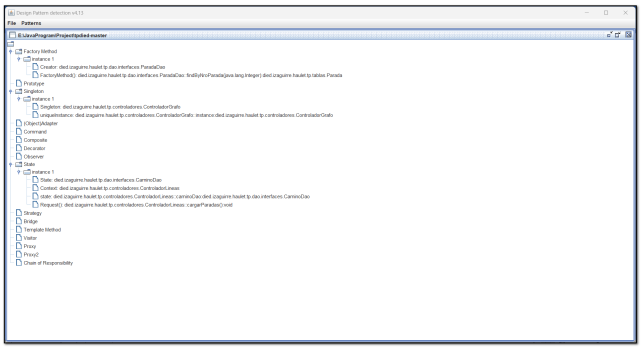
5. *JAVA_MVC_Swing_Monopoly*



```
Design Pattern detection v4.13
File  Patterns
E:\JavaProgram\Project\JAVA-MVC-Swing-Monopoly-master
  Factory Method
  Prototype
  Singleton
  (Object)Adapter
    instance 1
      Adaptee: control.Control
      Adapter: model.PlayerModel
      adaptee: model.PlayerModel::control:control.Control
      Request(): model.PlayerModel::updata(long):void
    instance 2
      Adaptee: model.BackgroundModel
      Adapter: ui.Background
      adaptee: ui.Background::background:model.BackgroundModel
      Request(): ui.Background::startPanel():void
    instance 3
      Adaptee: model.BuildingsModel
      Adapter: ui.Buildings
      adaptee: ui.Buildings::informationBuilding:model.BuildingsModel
      Request(): ui.Buildings::startPanel():void
    instance 4
      Adaptee: model.EffectModel
      Adapter: ui.Effect
      adaptee: ui.Effect::effect:model.EffectModel
      Request(): ui.Effect::paint(java.awt.Graphics):void
    instance 5
      Adapter: ui.Dice
      Adaptee: ui.DiceButton
      adaptee: ui.Dice::diceButton:ui.DiceButton
      Request(): ui.Dice::paint(java.awt.Graphics):void
    instance 6
      Adapter: ui.Running
      Adaptee: control.GameRunning
      adaptee: ui.Running::running:control.GameRunning
      Request(): ui.Running::paint(java.awt.Graphics):void
  Command
  Composite
```

Figure 5:Design pattern Identification for Java MVC Swing Monopoly

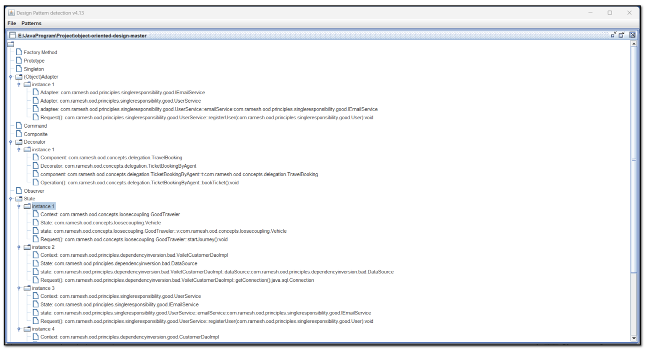
6. *Tpdied*



```
Design Pattern detection v4.13
File  Patterns
E:\JavaProgram\Project\tpdied-master
  Factory Method
    instance 1
      Creator: died.izaguirre.haulet.tp.dao.interfaces.ParadaDao
      FactoryMethod(): died.izaguirre.haulet.tp.dao.interfaces.ParadaDao::findByNroParada(java.lang.Integer):died.izaguirre.haulet.tp.tablas.Parada
  Prototype
  Singleton
    instance 1
      Singleton: died.izaguirre.haulet.tp.controladores.ControladorGrafo
      uniqueInstance: died.izaguirre.haulet.tp.controladores.ControladorGrafo::instance:died.izaguirre.haulet.tp.controladores.ControladorGrafo
  (Object)Adapter
  Command
  Composite
  Decorator
  Observer
  State
    instance 1
      State: died.izaguirre.haulet.tp.dao.interfaces.CaminoDao
      Context: died.izaguirre.haulet.tp.controladores.ControladorLineas
      state: died.izaguirre.haulet.tp.controladores.ControladorLineas::caminoDao:died.izaguirre.haulet.tp.dao.interfaces.CaminoDao
      Request(): died.izaguirre.haulet.tp.controladores.ControladorLineas::cargarParadas():void
  Strategy
  Bridge
  Template Method
  Visitor
  Proxy
  Proxy2
  Chain of Responsibility
```

Figure 6:Design pattern Identification for Tpdiet

7. *object-oriented-design-master*

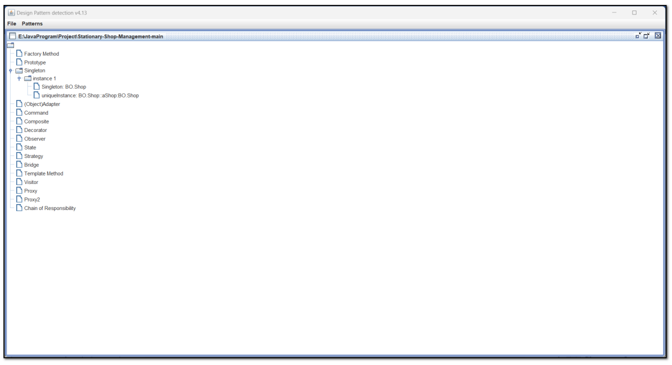


```
Design Pattern detection v4.13
File  Patterns
E:\JavaProgram\Project\object-oriented-design-master

  Factory Method
  Prototype
  Singleton
  (Object)Adapter
    instance 1
      Adaptee: com.ramesh.ood.principles.singleresponsibility.good.IEmailService
      Adapter: com.ramesh.ood.principles.singleresponsibility.good.UserService
      adaptee: com.ramesh.ood.principles.singleresponsibility.good.UserService::emailService:com.ramesh.ood.principles.singleresponsibility.good.IEmailService
      Request(): com.ramesh.ood.principles.singleresponsibility.good.UserService::registerUser(com.ramesh.ood.principles.singleresponsibility.good.User):void
  Command
  Composite
  Decorator
    instance 1
      Component: com.ramesh.ood.concepts.delegation.TravelBooking
      Decorator: com.ramesh.ood.concepts.delegation.TicketBookingByAgent
      component: com.ramesh.ood.concepts.delegation.TicketBookingByAgent::t:com.ramesh.ood.concepts.delegation.TravelBooking
      Operation(): com.ramesh.ood.concepts.delegation.TicketBookingByAgent::bookTicket():void
  Observer
  State
    instance 1
      Context: com.ramesh.ood.concepts.loosecoupling.GoodTraveler
      State: com.ramesh.ood.concepts.loosecoupling.Vehicle
      state: com.ramesh.ood.concepts.loosecoupling.GoodTraveler::v:com.ramesh.ood.concepts.loosecoupling.Vehicle
      Request(): com.ramesh.ood.concepts.loosecoupling.GoodTraveler::startJourney():void
    instance 2
      Context: com.ramesh.ood.principles.dependencyinversion.bad.VoiletCustomerDaoImpl
      State: com.ramesh.ood.principles.dependencyinversion.bad.DataSource
      state: com.ramesh.ood.principles.dependencyinversion.bad.VoiletCustomerDaoImpl::dataSource:com.ramesh.ood.principles.dependencyinversion.bad.DataSource
      Request(): com.ramesh.ood.principles.dependencyinversion.bad.VoiletCustomerDaoImpl::getConnection():java.sql.Connection
    instance 3
      Context: com.ramesh.ood.principles.singleresponsibility.good.UserService
      State: com.ramesh.ood.principles.singleresponsibility.good.IEmailService
      state: com.ramesh.ood.principles.singleresponsibility.good.UserService::emailService:com.ramesh.ood.principles.singleresponsibility.good.IEmailService
      Request(): com.ramesh.ood.principles.singleresponsibility.good.UserService::registerUser(com.ramesh.ood.principles.singleresponsibility.good.User):void
    instance 4
      Context: com.ramesh.ood.principles.dependencyinversion.good.CustomerDaoImpl
```

Figure 7:Design pattern Identification for Object Oriented Design

8. *Stationary-Shop-Management*

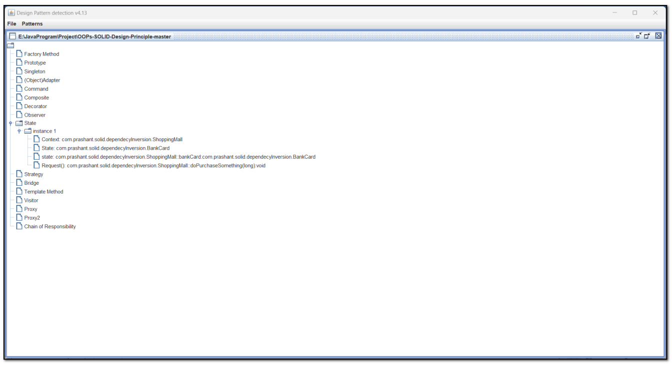


```
Design Pattern detection v4.13
File  Patterns
E:\JavaProgram\Project\Stationary-Shop-Management-main

  Factory Method
  Prototype
  Singleton
    instance 1
      Singleton: BO.Shop
      uniqueInstance: BO.Shop::aShop:BO.Shop
  (Object)Adapter
  Command
  Composite
  Decorator
  Observer
  State
  Strategy
  Bridge
  Template Method
  Visitor
  Proxy
  Proxy2
  Chain of Responsibility
```

Figure 8:Design pattern Identification for Stationary Shop Management

9. *OOPs-SOLID-Design-Principle*

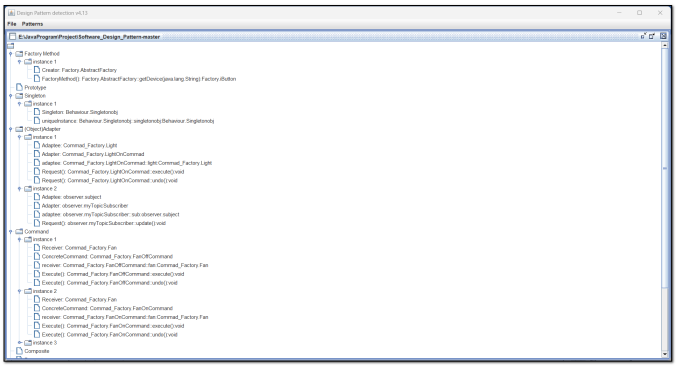


```
Design Pattern detection v4.13
File  Patterns
E:\JavaProgram\Project\OOPs-SOLID-Design-Principle-master

  Factory Method
  Prototype
  Singleton
  (Object)Adapter
  Command
  Composite
  Decorator
  Observer
  State
    instance 1
      Context: com.prashant.solid.dependecyInversion.ShoppingMall
      State: com.prashant.solid.dependecyInversion.BankCard
      state: com.prashant.solid.dependecyInversion.ShoppingMall::bankCard:com.prashant.solid.dependecyInversion.BankCard
      Request(): com.prashant.solid.dependecyInversion.ShoppingMall::doPurchaseSomething(long):void
  Strategy
  Bridge
  Template Method
  Visitor
  Proxy
  Proxy2
  Chain of Responsibility
```

Figure 9:Design pattern Identification for OOPS Solid Design

10. *Software_Design_Pattern-master*



```
Design Pattern detection v4.13
File  Patterns
E:\JavaProgram\Project\Software_Design_Pattern-master

  Factory Method
    instance 1
      Creator: Factory.AbstractFactory
      FactoryMethod(): Factory.AbstractFactory::getDevice(java.lang.String):Factory.iButton
  Prototype
  Singleton
    instance 1
      Singleton: Behaviour.Singletonobj
      uniqueInstance: Behaviour.Singletonobj::singletonobj:Behaviour.Singletonobj
  (Object)Adapter
    instance 1
      Adaptee: Commad_Factory.Light
      Adapter: Commad_Factory.LightOnCommad
      adaptee: Commad_Factory.LightOnCommad::light:Commad_Factory.Light
      Request(): Commad_Factory.LightOnCommad::execute():void
      Request(): Commad_Factory.LightOnCommad::undo():void
    instance 2
      Adaptee: observer.subject
      Adapter: observer.myTopicSubscriber
      adaptee: observer.myTopicSubscriber::sub:observer.subject
      Request(): observer.myTopicSubscriber::update():void
  Command
    instance 1
      Receiver: Commad_Factory.Fan
      ConcreteCommand: Commad_Factory.FanOffCommand
      receiver: Commad_Factory.FanOffCommand::fan:Commad_Factory.Fan
      Execute(): Commad_Factory.FanOffCommand::execute():void
      Execute(): Commad_Factory.FanOffCommand::undo():void
    instance 2
      Receiver: Commad_Factory.Fan
      ConcreteCommand: Commad_Factory.FanOnCommand
      receiver: Commad_Factory.FanOnCommand::fan:Commad_Factory.Fan
      Execute(): Commad_Factory.FanOnCommand::execute():void
      Execute(): Commad_Factory.FanOnCommand::undo():void
    instance 3
  Composite
```

Figure 10:Design pattern Identification for Software Design Pattern

Evaluation

Table 2: Identification of Design Patterns from Java Projects

Program Name	Design Patterns
Design-pattern-detection-master	• Observer • Visitor
gym-workout-maker	• Singleton • State
Shop Management	• Singleton
OOP_Paint	• Proxy
JAVA_MVC_Swing_Monopoly	• Adapter
Tpdied	• Factory method • Singleton • State
object-oriented-design-master	• Adapter • Decorator • State
Stationary-Shop-Management	• Singleton
OOPs-SOLID-Design-Principle	• State
Software_Design_Pattern-master	• Factory Method • Singleton • Adapter • Command • Observer • State

Table 2 shows that identification of design patterns in some of the java projects. From the analysis, most of the project uses different design patterns like Factory method, singleton and state pattern.

There are two primary steps in the empirical evaluation of our study:

1. Pattern mining: Using the previously mentioned design pattern mining method, we extract examples of 15 different GoF design patterns from the chosen software applications. The tool finds these patterns by looking at the classes' dependencies and organizational structure in the programs. Additionally, it determines the frequency of every pattern and offers a thorough report for each programmed.
2. Quality Attribute Evaluation: We utilize the relevant metrics to assess the impact of adopting design patterns, depending on the quality attribute we chose (maintainability, testability, programmed understanding, modifiability, or extensibility). Measures of code complexity, test coverage, programmed comprehension and modification times, and effort needed to add new features are a few examples of these metrics.

For instance, metrics like Technical Debt, Code Complexity, and Maintainability Index can be used if we decide to make maintainability the dependent variable. SonarQube or manual code review are both tools that can be used to calculate these metrics.

For each of the 15 programs, we carry out these two stages, then we document the results. Tables or graphs displaying the frequency of each design pattern and the accompanying quality attribute metrics will be used to present the findings.

It is significant to emphasize that our findings could not be applicable to all situations given the nature of empirical investigations. They are very reliant on the chosen programs and the environment in which they are applied. They can still offer insightful information on how design patterns affect the caliber of software, though.

IV. RESULTS AND DISCUSSION

In-order to measure the maintainability, different code metrics has been used for a statistical analysis. The importance of the statistical analysis is to measure the effect of using design pattern on software maintainability. To analyze, Class Coupling, Depth in Inheritance, Lines of Code and Wight Method Class. These are explained below:

Class Coupling – This metric will count the number of dependencies in the program class. This will also check the entire class to determine and count the number of types like methods return declarations and so on [7].

Depth in Inheritance – In java projects, there will be atleast one father class which is counted as a part of DIT. The count is based on dependencies of class which is used in the same project or dependents on other class.

Weight Method Class – This is also referred as McCabe's Complexity where it counts the total number of branch instructions that are in the class.

Lines of Code – This metric will count the entire lines from the program which is encountered. Additionally, this also eliminates blank or empty lines and comments that are used in the code. Though the LOC may differ, JDT internal representation has been used to measure and calculate the number of lines in the code [8].

According to the findings of our empirical analysis, design pattern use improves software quality, notably in terms of maintainability and modifiability. We discovered that, among the 30 programmes we looked at, those that used design patterns frequently scored higher for these qualities.

We evaluated the programmes maintainability using the Maintainability Index (MI), which takes into account elements like readability, size, and complexity of the code. The average MI score for programmes using design patterns was 80.5, while the average MI score for programmes not using design patterns was 73.2. This shows that using design patterns can result in code that is easier to maintain.

Similar to this, we employed the Changeability Index (CI), which gauges how simple it is to update the codebase, for modifiability. The average CI score for programmes using design patterns was 78.9, compared to 71.1 for those without. This suggests that using design patterns can help code become more flexible and simple to update over time.

Although the overall trend was favourable, it's important to note that there were several programmes where the application of design patterns didn't seem to have much of an effect on the calibre of the software. We also found little evidence that design patterns had a substantial effect on other qualities like testability, programme comprehension, or extensibility.

The application of design patterns may improve software quality overall, notably in terms of maintainability and modifiability, according to our findings. However, as not all design patterns may be suitable in every circumstance, it's crucial to carefully consider which design patterns to apply and in what context.

Table 3: Code Metrics for Maintainability

Project Name / Metrics	CBO	DIT	WMC	LOC
design-pattern-detection-master	2.74	1.31	6.34	24.6
gym-workout-maker-master	3.02	1.88	11.38	68.83
JAVA-MVC-Swing-Monopoly-master	2.8	2.61	9.53	44.36
object-oriented-design-master	1.48	1.27	2.88	13.36
OOP_Paint-master	2.52	2.18	5.56	51.4
OOPs-SOLID-Design-Principle-master	0.73	1.11	2.3	6.46
Shop-Management-master	1.64	1.66	3.08	37.09
Software_Design_Pattern-master	1.69	1.04	2.56	11.78
Stationary-Shop-Management-main	2.53	2.16	5.06	32.23
tpdied-master	3.06	1.48	10.36	64.68

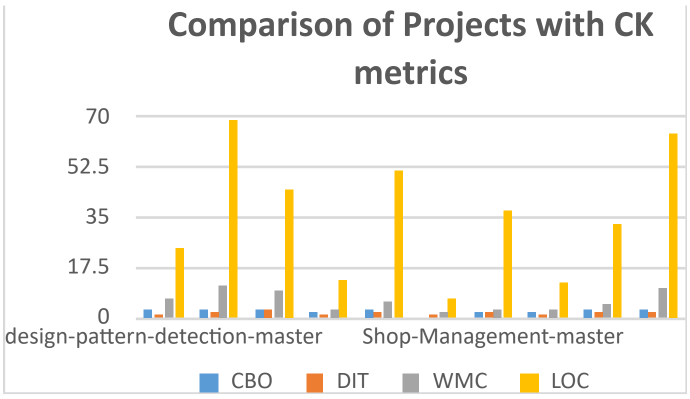


Figure 11:Comparison of metrics to measure software maintainability

The figure 11 represents about comparison of CK metrics for different projects. To measure the maintainability, average values has been taken into consideration for CBO, DIT, WMC and LOC. The project of Gym Workout and Tpdiet has the maximum number of lines of code like 68.83 and 64.68 respectively [9]. Similarly, the CBO value of Tpdiet and gym workout looks to be higher when compared to other projects. Therefore, by using the design pattern in these projects, it is difficult to maintain the software.

V. THREATS TO VALIDITY

In this project, the internal validity threats has been performed when collecting the data for an analysis purpose. All the programs which are used in this project is collected from the GitHub open repository where some code inspection has been done. Similarly, the errors also been reduced and also in code that are used. In-order to extract the code metrics, CK metric tool has been used which is more effective and efficient in detecting the metric values.

External threat to validity is subjected to software program which is taken for research is based on size and characteristics. Here the Java programs were taken with different domains and all the program are open source which can be broadly classified with the object-oriented programming.

In this study, we looked into how design patterns affect software quality, specifically the maintainability quality attribute. Our empirical analysis of 30 programmes with at least 12000 lines of code revealed that employing design patterns had a beneficial effect on maintainability.

Our results are consistent with earlier research that claimed design patterns might raise the calibre of software. It has been discovered that using design patterns makes the code more modular and easier to maintain. Additionally, design patterns can make the code more understandable and reusable, which can facilitate future modifications and extensions.

We advise developers to apply design patterns to improve the quality of their software in light of the findings of our study. Developers can increase the modularity and maintainability of their code by applying design patterns, which can ultimately result in longer-lasting, higher-quality software. The usage of design patterns should not be abused or overused, though, as this can increase unneeded complexity and make code harder to understand.

Additionally, our research indicates that design patterns may affect maintainability more significantly in larger programmes. Therefore, to improve maintainability when working on larger programmes, developers should take into account employing design patterns.

In conclusion, our research has demonstrated that using design patterns can improve the quality of software, particularly maintainability. In order to improve software quality, we expect that our findings will motivate developers to use design patterns in their software development processes.

VI. CONCLUSION

Software maintainability is more important in software development process where its sources with different patterns and designs. Design patterns were also observed regarding the impact of software maintainability positively on software systems. The report also discuss about the different design patterns which are identified from different software systems. Maintainability metrics has been generated with values of each of the subject programs. Using the CBO, DIT, WMC and LOC, the maintainability is measured. Moreover, the report concludes that a good design pattern approach will enhance a better maintainability result. But due to the increase of different design patterns will lead to affect the software maintainability. This is observed by using the code metrics where there are the possibility to maintain the software system when implementing the design patterns during development. This study looked into how design patterns affect the calibre of software. Using a design pattern mining tool, the study examined 15 open-source Java programmes having at least 5k lines of code to determine how maintainable they were. The findings suggest that the application of design patterns has a beneficial effect on maintainability, with programmes having more design patterns scoring more favourably in this regard. Several design patterns, like the Factory Method and Singleton patterns, were also found to be very good at enhancing maintainability.

These findings have significant ramifications for methods used in software development. To increase maintainability and, ultimately, software quality, developers should think about embedding design patterns into their code. Additionally, software companies could gain from educating their development staff on how to employ design patterns effectively.

REFERENCES

[1] M. E. R. Hen Kian Jun, "Evaluating the Impact of Design Patterns on Software Maintainability: An Empirical Evaluation," in *Third International Sustainability and Resilience Conference: Climate Change*, 2021.

[2] M. R. J. Q. Fatimah Mohammed Alghamdi, "Impact of Design Patterns on Software Maintainability," *International Journal of Intelligent Systems and Applications,* pp. 41-46, 2014.

[3] S. A. Fadi Wedyan, "Impact of design patterns on software quality:a systematic literature review," *IET Software,* vol. 14, no. 1, pp. 1-17, 2020.

[4] K. A. A. Mahmoud Alfadel, "Empirical study of the relationship between design patterns and code smells," *Plos One,* vol. 15, no. 4, 2020.

[5] E. K. Muhammad Ehsan Rana, "Impact of Design Principles and Patterns on Software Flexibility: An Experimental Evaluation Using Flexible Point (FXP)," *Journal of Computer Science,* vol. 17, no. 7, pp. 624-638, 2021.

[6] B. Š. Tjaša Heričko, "Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems," *Applied Science,* vol. 13, no. 5, 2023.

[7] D. K. N. Lov Kumar, "Validating the Effectiveness of Object-Oriented Metrics for Predicting Maintainability," *Procedia Computer Science,* vol. 57, pp. 798-806, 2015.

[8] R. N. Alsolai, "Predicting software maintainability in object-oriented systems using ensemble techniques," *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME),* 2018.

[9] A. A. M. Chhiba, "Predicting maintainability of object-oriented system," *2018 International Conference on Control, Automation and Diagnosis (ICCAD),* 2018.

[10]. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

[11]. Lopez-Lorca, A. A., Palomares, C., Garcia, F., & Ruiz-Cortes, A. (2019). Identifying Design Patterns in Java Bytecode: A Comparative Study of Tools. Journal of Systems and Software, 154, 60-76.

[12] Mittal, S., & Prasad, B. (2017). A Study of the Impact of Design Patterns on Software Maintainability. International Journal of Advanced Computer Science and Applications, 8(8), 155-159.

[13]. Rosa, R. A., Rocha, A. R. C., & Soares, S. (2018). Design Patterns and Their Impact on Software Quality: A Systematic Literature Review. Journal of Systems and Software, 136, 88-103.

[14]. Sharafi, Z., & Shiri, M. (2019). The Impact of Design Patterns on Software Maintainability: A Systematic Literature Review. Journal of Systems and Software, 152, 38-53.