COMP 40070 Design Patterns

Lab Journal

Vivek Murarka (22200673)

MSc. in Computer Science (Negotiated Learning)

UCD School of Computer Science

University College Dublin

Sep 2022

Table of Contents

Practical 1: Solid Principle

This practical involved understanding object-oriented principles named by Bob Martin and then rate the heuristics against the 5 principles. These five principles were namely:

1. Single Responsibility Principle (SRP): Every class has single responsibility and that responsibility should be entirely Encapsulated within the class.
2. Open/Close Principle (OCP): Defining what is open for extension via interface/abstract class and closed for modification. It mostly asks designer to define which is the stable module and which isn't.
3. Liskov Substitution Principle (LSP): Emphasis on correct use of inheritance so that an instance of sub-class can be replaced with instance of super class to achieve generalization and thus avoiding code duplication. So, while passing argument if we pass declare super class as argument-type we can achieve more generalization and decide at run-time which sub class type will fit best.
4. Interface Segregation Principle (ISP): As interface belongs to client, the client should only implement functionality which are relevant to them and nothing beyond it. This practice involves segregating functionality into different interface on basis of client need.
5. Dependency Inversion Principle (DIP): In a nutshell, instead of high-level module (the one which uses other's service) should not depend directly on low level module (the one which provide some service) to avoid tight coupling, instead they both should refer to abstraction. Also, abstraction should not be affected by the actual implementation, but the implementation should depend on abstraction. E.g.: declaration of function is defined in abstraction and implementation should use this declaration for defining further details.

1.1 Work Done

I developed a deeper understanding of this principles which is summarized above and then each heuristic were compared against this principle. I first focused on eliminating which principle doesn't necessary apply to heuristics and rated them as 1~2. Second step was to rate the one which are strongly relevant to given heuristics and rated them 4~5. Finally, started cross-examining if the ones which I feel have some relevance against heuristics and tried to establish with some superficial examples and rated them 3~4.

1.2 Reflections

Here, I'm going to talk about the rating against each heuristic and why were they rated so.

1. *All data should be hidden within its class:* Here, this strictly speaks about encapsulation and only principle defined above which speaks about encapsulation is SRP.
   Hence, SRP is rated 5 and rest have no relevance.
2. *Users of a class must be dependent on its public interface, but a class should not be dependent on its users*: OCP is 5 as client is dependent on server but server is not dependent on client hence server can be modified without affecting the client functionality and client can still communicate with server as public interface is stable.
   DI is rated 4 because high level module (User of a class) should not depend on low level module like class(implementation), instead is should depend on abstraction.
   Rest all are not relevant
3. *Minimize the number of messages in the protocol of a class*.: Here the arguments that class requires need to be limited that means precondition should be weaker-LSP. Hence, LSP is rated 5 and others are not relevant.

4. *Implement a minimal public interface which all classes understand (e.g., operations such as copy (deep versus shallow), equality testing, pretty printing, parsing from an ASCII description, etc.).*: SRP is rated 2 because although it talks about interface having minimum features so the class extending it need to implements only relevant methods this limiting the responsibility of the class, but not more than 2 because it doesn't talk about the contents of class.
   OCP is rated 3 because implementing a minimal interface is relatively easy without changing the implementation code thus closely sticking to the principal of OCP.
   LSP is rated 1 because the problem is not looking for inheritance issue.
   ISP is 5, because this is in direct relation to principle "client should not be forced to depend on method it does not use", in this example all class understand or use because it has only generic logic.
   DIP is one because it doesn't talk about who calls whom.
5. *Do not put implementation details such as common-code private functions into the public interface of a class.*: DIP is rated 5 because it speaks about abstraction should not depend on details, instead details should depend on abstraction. Rest has no relevance.
   ISP is rated 4 because this focus on abstraction thus limiting what clients have access to.
6. *Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.*: Again, ISP is 5 because it speaks about client's interest and OCP is 4 because if interface is cluttered client cannot use it without modifying the code which is against the principal of OCP.
7. *Classes should only exhibit nil or export coupling with other classes, i.e., a class should only use operations in the public interface of another class or have nothing to do with that class*.: OCP is 5 because it says instead of using the class itself it should use the public interface of the class.
8. *A class should capture one and only one key abstraction.*: Only SRP is 5 because it only talks about limiting the responsibility of class. Rest has no relevance.
9. *Keep related data and behaviour in one place.*: Closely related to encapsulation, hence SRP is rated 5, rest are not relevant.
10. *Spin off non-related information into another class (i.e., non-communicating behaviour). [If a set of methods operate on a proper subset of the data members of a class (i.e., non-communicating), consider putting them in a class on their own.]:* Not related to any principle, but its more on separation of concern principle.
11. *Be sure the abstractions that you model are classes and not simply the roles objects play.:* ISP is 4. Abstraction of relevant things and not all necessary details of the role of object in program like Hash code in Java should not be abstracted.

Practical 2: Solid Principle-Part 2

This practical I was teamed up with **Ru J** ruyue.jin@ucdconnect.ie, and we together discussed on previous practical. We were supposed to analyse and evaluate each other ratings and come to a conclusion on the differences.

2.1 Work Done

As a first step we shared our workbook and journal among ourself so that we can understand their opinion. Initially, we found that we have difference in views at multiple places. To resolve this Ru and I, together shared our understanding on five principles under solid principle, discussed here. It was observed that our understanding of principle was in-line with each other. It was the heuristics where our understanding was different. So, we tried to explain each other what we can understand from literal meaning and then did our part of research. Most of our confusion were resolved after visiting page on Riel's heuristics [1]. We also referred explanation of Solid Design Principle by Thorben Janssen [2]

2.2 Reflections

After we understood the literal meaning of each heuristic, we again evaluated our rating and following is what we concluded.

1. *All data should be hidden within its class:* Although, we both agreed that this heuristic is nowhere related to OCP, LSP, ISP and DIP. We couldn't reach to common understanding on SRP. In my view this is very much related to encapsulation and SRP closely relates to encapsulation. In Ru's view, data can be encapsulated but responsibility can still be public and according to her opinion, SRP speaks about encapsulation of responsibility and not data, hence we still have major difference in views and thus we have graded it C.
2. *Users of a class must be dependent on its public interface, but a class should not be dependent on its users*: After close evaluation and thorough discussion, I have changed my opinion. This heuristic speaks about Cyclic dependency between components which must be avoided. Ru and I, both are in same opinion that none of the five principals speak about cyclic dependency and thus rating each to 1.
3. *Minimize the number of messages in the protocol of a class.*: Initially, we had difference in opinion. As per Ru, this heuristic was more closely related to OCP, whereas I was inclined towards LSP. But after our in-length discussion, we realised that all it speaks about is – "*don't implement method until you need them*" [3]. We both agreed that this is somewhat related to SRP as this will ultimately reduce the number of methods by reducing the responsibility, but it is not completely related to it as this heuristic also speak about implementing it when we need it. So, we both reached an agreement and rated SRP as 3, and others are not relevant.
4. *Implement a minimal public interface which all classes understand (e.g., operations such as copy (deep versus shallow), equality testing, pretty printing, parsing from an ASCII description, etc.).*: Ru and I we both agree that this heuristic is strongly related with two principal – OCP and ISP.
   OCP because it talks about client dependency on public interface and ISP because it talks about client should not be forced to implement empty methods if it is dependent on some interface. Hence interface must have minimal stuff which are absolutely necessary for client to implement.
5. *Do not put implementation details such as common-code private functions into the public interface of a class.*: After discussion and personal reflection, I have changed my opinion that

it is not dependent on DIP. Instead, it depends on ISP alone. Ru and I have reached a mutual agreement.

6. *Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.*: Again, here I have changed my opinion that this directly related to ISP alone, as it talks about providing interface with minimal and usable function. Ru and I have a mutual agreement.
7. *Classes should only exhibit nil or export coupling with other classes, i.e., a class should only use operations in the public interface of another class or have nothing to do with that class*.: Ru and I had the same opinion on this heuristic that this is in direct relation with OCP. Ru had also come to an understanding that this is not related with ISP.
8. *A class should capture one and only one key abstraction.*: We both believe this heuristic is strong related to SRP. There were no difference and no rating was changed.
9. *Keep related data and behaviour in one place*.: Initially I believed that this is some what related to SRP, but after understanding the heuristic, its quite clear that when talked about related data and behaviour, it symbolises about single responsibility, and all single responsibility shout be kept under one class. Hence, I have changed my rating as it is strongly related to SRP.
10. *Spin off non-related information into another class (i.e., non-communicating behaviour). [If a set of methods operate on a proper subset of the data members of a class (i.e., non-communicating), consider putting them in a class on their own.]:* In my previous understanding I thought that no rule is associated with separation of concern. But SRP also symbolises that related responsibility must be club together and non-related one should be put in separate class. Hence, changed my rating to 5 for SRP.
11. *Be sure the abstractions that you model are classes and not simply the roles objects play*.: Ru and I agree that this heuristic is not related to any given five principles. Instead, it talks about achieving generalization with the help of inheritance. So instead of designing on basis of roles that model will play, we must categorize into what characterises the model and create a super-class on basis of characters and define roles in sub-classes. None of the principle speak very specific on this theory, but are somewhat related.

Practical 3: Template method and Strategy

Firstly, we will start with understanding the concept behind following design pattern:

1. Template Method: This design is usually adapted when there are multiple entities using similar logic to generate different outputs. All involved entities end up looking the same, having similar code base, hence might have similar issue/error. Therefore, they are hard to maintain and adapt to changing needs. It becomes even more manual work when the new but similar entities is involved and needs to be introduced as separate feature.

   The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single template method. *The steps may either be abstract, or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).* [4]

   Steps that are followed to adopt this pattern.
   - Create superclass, and create abstract method that needs to be overridden by all sub-classes. This sub-class will write their own logic here which is very specific to its needs.
   - Identify duplicate codes that can implemented in superclass as default method. Sub-class can override these methods if they need.
   - Implement optional hook methods. These methods are empty and need not be overridden by all sub class. The intention behind this is to give extension for some sub-class which might override it.

2. Strategy: This design is adapted when our class is trying to solve similar business needs for more than one client. If we start adapting the same method for different client it can become quite messy to adapt and error prone. Also, it becomes quiet dependent on client which is not always a pleasant scenario.

   The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called strategies. [5]

   Steps that are followed to adopt this pattern.
   - Create interface which gives the overview of what common function will adopted by implementing classes.
   - Client will call the context class and provide the details at run time which strategy it wants to invoke.
   - The Context class will refer to interface and set the strategy requested by the client.

   This allows dynamic selection of the strategy at run time and avoid tight coupling.

Now, I will discuss on the exercise on hand. Here, we are provided with a Scala application. It models a game where an oracle thinks of a number in a certain range and each participant makes a number of guesses to find out what the number is. After each guess, the oracle tells the participant if they are or not, and if not, whether their guess was too big or too small. Each participant uses different strategy to guess.

3.1 Work Done

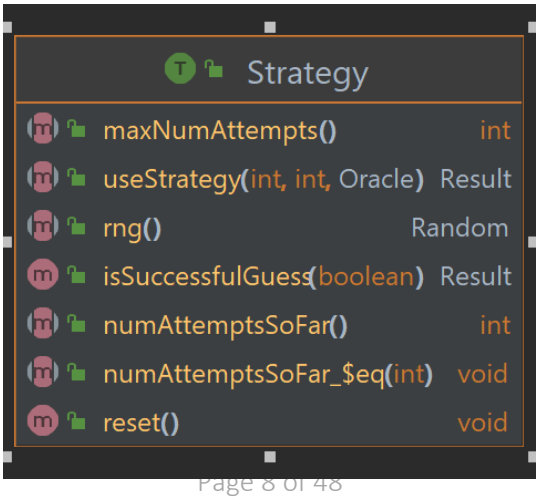
In the code walkthrough, I tried to understand the flow of code. To understand what sorts of method are available and how they are inter-linked. I also tried to identify if there is an existing design pattern in place. Mostly, it was associated as single responsibility principle, as far as I can tell.

Below, I will list down all the issues which were identified in three classes present in code, and steps taken to resolve them.

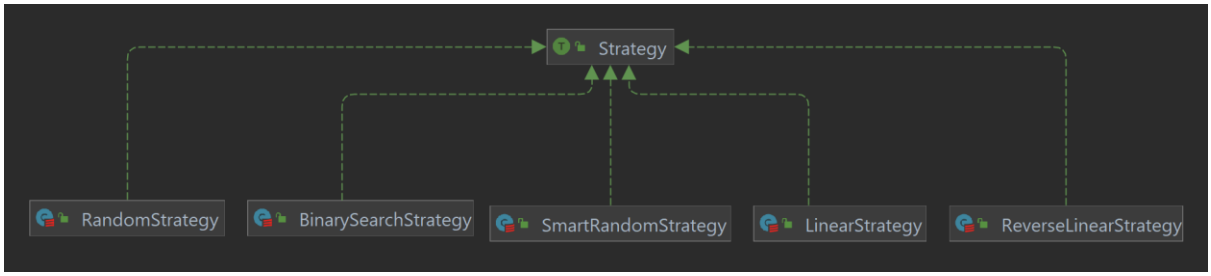
1. main.scala : This class forms the crux of the application. It contains multiple calls to Participant class. The code has repetitive behaviour. Large part of code does exactly the same thing does make it look cumbersome. Adding new participant in a game would require duplication of codes.
2. Oracle.scala: Sort class, with just one implementation and that is to identify the guess of participant was correct or near to it. There is not much change or fix required in this class as this class basically adheres to Single responsibility principle and doesn't have duplicate codes.
3. Participant.scala: Another big class. What makes it really fuzzy, is that all the strategy used by the participant is written in this one giant class. It also definiens some utility method along with other logic. If I have to add a new participant which uses different logic like "Reverse Linear Search" then we need to lot of duplication again.

Now, I will discuss the steps I took to redesign the code and reasons behind it.

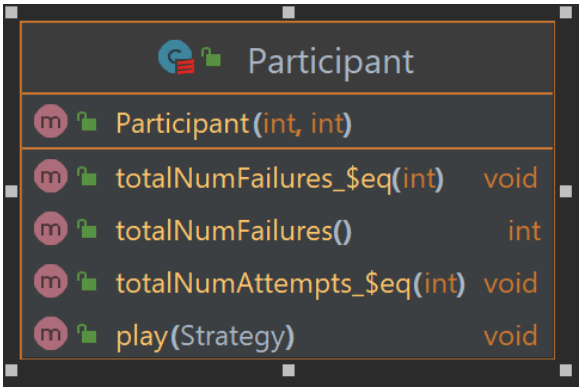
1. The core business logic of this games depends on each participant using different strategy to guess the number in least possible tries. Hence, based on this logic I used the Strategy design pattern to break the class and created a Strategy Interface (*Trait*). This Strategy interface has two *default* methods – *isSuccessfulGuess()* and *reset()* , which is used by all the Implementation class and is not overridden. The *abstract* method which is necessarily over-ridden is *useStrategy()* – this method is where each concrete class defines its own logic on how to guess.
   There is other class variable which is available for concrete class to use.



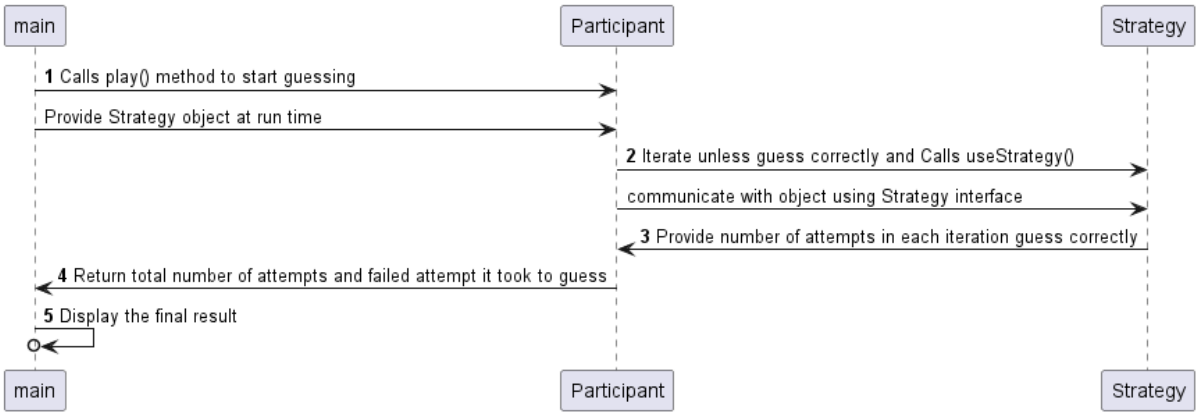
2. The Strategy interface is implemented(extended) by five class as depicted in below diagram.



3. Now since each participant were using the common logic to start guessing, using Template design pattern I brought the duplicate code in Participant class under *play(Strategy)* method. Now this class is not created as abstract class for simple reason that in current implementation we don't need to override or define hook method in sub class, as I feel it will be an overengineering. Instead now I use this as our Context class which defines a *play(Strategy)* method taking Strategy as argument. This Strategy is decided by client at run time. So each participant call play() and provide what sort of strategy it wants to use to guess the number.



4. Now our main method is the one which interact to client directly. Each participant now uses the template provided as play method in Participant class and just provide which strategy they want to use at run time as argument. The flow could be understood from this UML diagram.



5. I added another strategy implementation. That is "*ReverseLinearStrategy*". Basically, it guesses the number order in decrementing order.

3.2 Reflections

So, I started with understanding the need of Template and Strategy design pattern. The use case where they can fit and how to implement such solution in Scala. What was observed that Strategy design pattern has an edge over Template as we usually use interface for implementing Strategy pattern and abstract class for Template. Be it Java or Scala, class can only have one abstract class but class can implement multiple interfaces. Hence "*Traits are more flexible to compose-you can mix in multiple traits, but only extend one class*" [6]. Also, while working with the exercise, creating interface for Strategy made more sense than creating abstract class, as it lays the design that each implementing class should take care.

Now, I also explored to fit Template Strategy, using Participant as abstract class and then creating sub class for each participant (Bart, Lista…etc;) but then this would only make sense if each of this participant are doing something beyond than what Participant class is already doing. Currently they all fit as object of Participant and each such object can be created in main itself. Let's say in future each participant tries to do something beyond after it has got the guess then we could think of creating sub class to fit such use cases.

Practical 4: Observer Pattern

We will start with understanding the Observer Pattern and different implementation associated with it.

In a nutshell, observer pattern is used when we want to inform multiple users, also known as observer, about the change of certain state of certain object which is often called as subject. The observer may or may not take any action based on information passed to them. This pattern closely resembles Publisher-Subscriber model, where publisher is subject, which publishes data to a subscriber interface, to avoid tight coupling with observers, and observer which implement this interface, are called concrete subscribers. Concrete subscriber can anytime register itself to publisher, which maintains the list of all concrete subscriber, and whenever there is an event which trigger change in state, publisher uses this list to disseminate information. This list of information can also be maintained by event manager.

Steps breakdown [7]:

1. The publisher issue event of interest to other objects. These events occur when the publisher change its state or execute some behaviours. Publisher contains a subscription infrastructure that lets new subscriber join and current subscriber leave the list.
2. When new even happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.
3. The subscriber interface declares the notification interface. In most cases, it consists of single update method. The method may have several parameters that let the publisher pass some event details with the update.
4. Concrete subscriber performs some action in response to notification issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.
5. The client creates publisher and subscriber objects separately and then register subscriber for publisher update.

Communication Mechanism: There are two protocols by which the observer interacts with the subject.

1. Push model: To me, the push model is where subject sends the message about change of state, even if observer is not ready. The subject continuously notifies the observer without direct action on the observer's part [8]. Works like how console immediately logs the information even if user is not paying attention to that. This reduces coupling with publisher and subscribers.
2. Pull model: This is when observer is notified about the change in state, and it is up to the observer to find about details of change of state.

Now, we will discuss on Observer pattern exercise. The task was to pick the implementation from practical 3 and apply observer pattern on top of it. The observer, here Auditor, will observe the game flow and raise alarm if it finds something suspicious.

4.1 Work Done

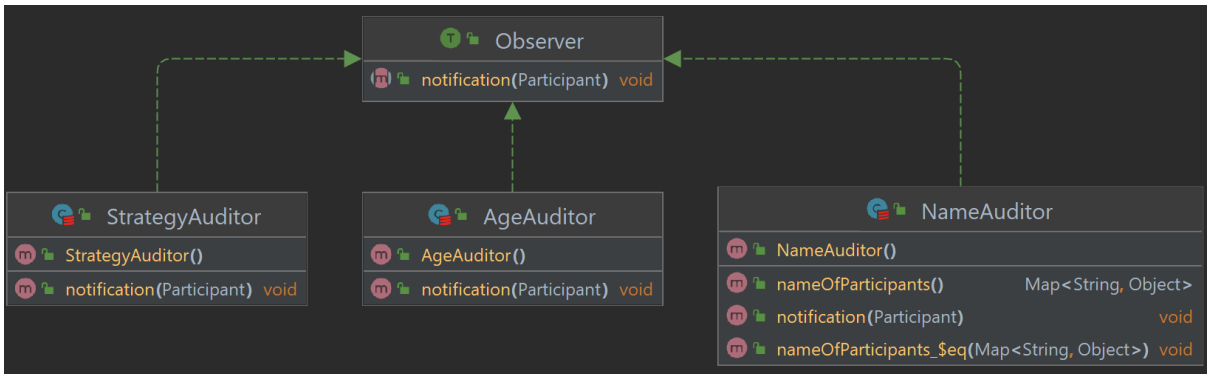
The first step that I decided was to analyse who can be the subject ("the one which is to be observed") and who will be the observer. In our case it was obvious that multiple Auditor extend Observer, but tricky part was which class can extend Observable. Initially, I forced Participant class to extend Observable trait. The problem with this approach was that, Participant will be tightly coupled with Auditors. What if we don't need to have Auditor in future, we need to remove observers and need to remove the notification method. For me, this looked like a big issue in terms of flexibility.

Hence, I made the client, here Main, to extend Observable. The benefit of this is that I can add or remove observer more freely, and if the client decides not to notify the observer, it can either remove the observer or not call *notifyObservers()* method. Also, now active Auditor can monitor when the new participant is added to the game, and easily monitor their game.

Another dilemma that I faced was, whether Auditor should observe every guess that participant makes or it should observe the average number of attempts it took for the participant to guess correctly. I decided to go with latter, as this will allow the auditor to let the game flow naturally and only raise alarm if it finds suspicious activity in the end. For example, if the participant took less than average number of attempts using certain strategy, then that participant might be a possible cheater, and this is what Auditor will check and report.

Now, lets talk about the design.

To implement Observer trait, I created three Auditor class which extends it. The dependency, looks as below.

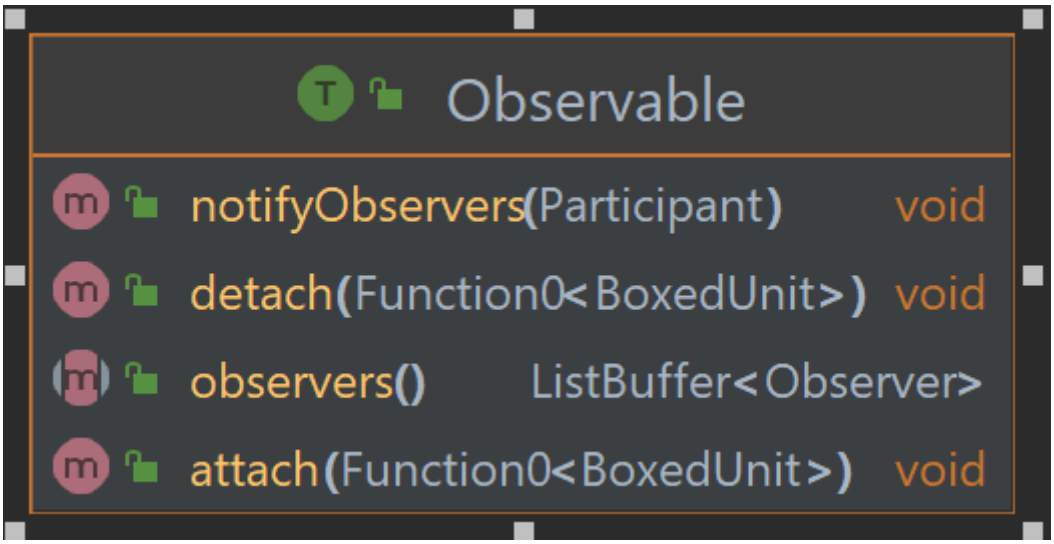


Here, we got three Auditors which are extending Observer trait.

- StrategyAuditor: Observe average number of attempts taken by participant to make a guess. Raise alarm if it is less than the minimum expected number of attempts.
- AgeAuditor: Observes the age of participants. If the age of participant is less than 18 or beyond 60, it raises alarm, by logging that "*legal age is 18-60*"
- NameAuditor: Observes the name of participants. It stores the name of participant and number of times the participant of same name has played, in mutable Map<Sting, Int>. If participant with same names plays more than two times, it raises alarm, that participant of such name has already played the game.

Now, we will see how subject implements Observable trait.

First let's see the structure of Observable trait.



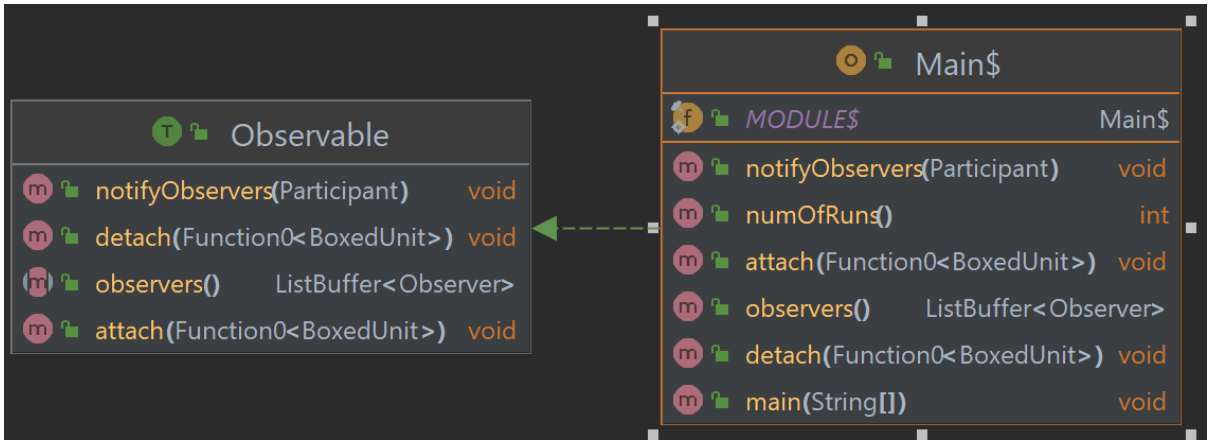
Observable, has one mutable List<Observer>, where it stores the current list of observers which are observing the subject. Observable notifies each observer using this list. It provides three functions for this:

- attach: Accept anonymous function, to add one observer at a time.
- detach: Accept anonymous function, to delete all observer
- notifyObservers: Notify all observes in the list.

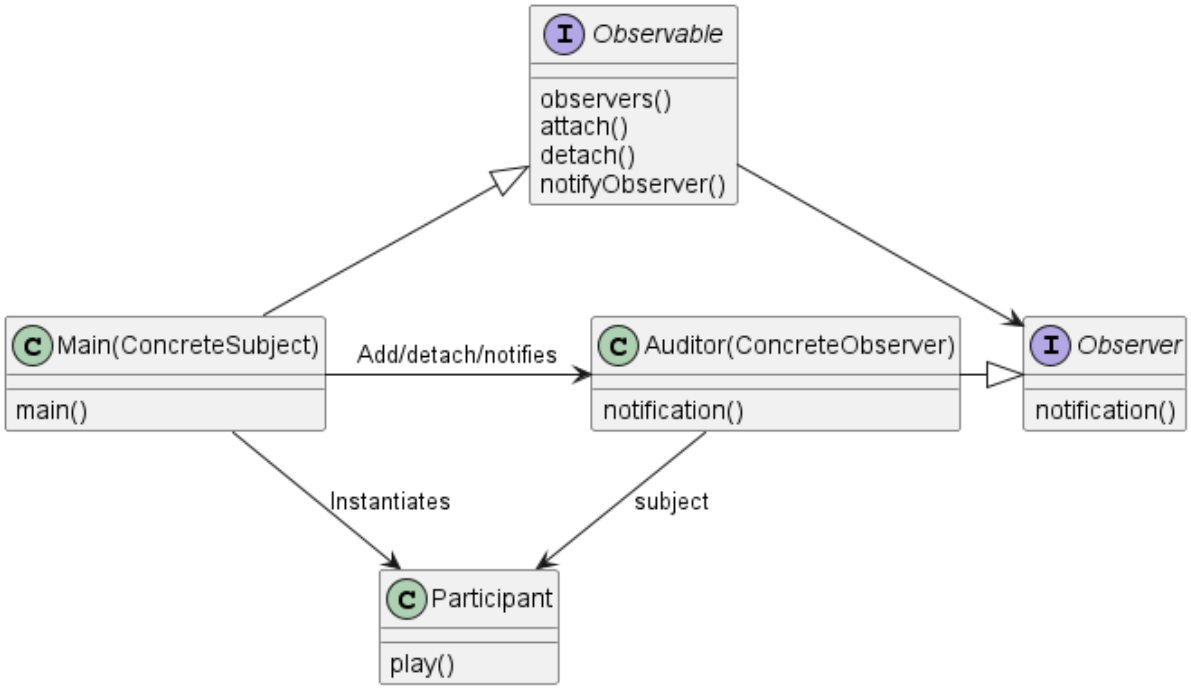
Our code snippet for Observable looks like this:

```
trait Observable {

  val observers = ListBuffer[Observer]()

  def attach(addObserver: () => Unit): Unit = addObserver.apply()

  def notifyObservers(subject : Participant): Unit =
    observers.foreach(_.notification(subject))

  def detach(deleteObserver: () => Unit): Unit =
    deleteObserver.apply()
    println("No one is observing this game")
}
```

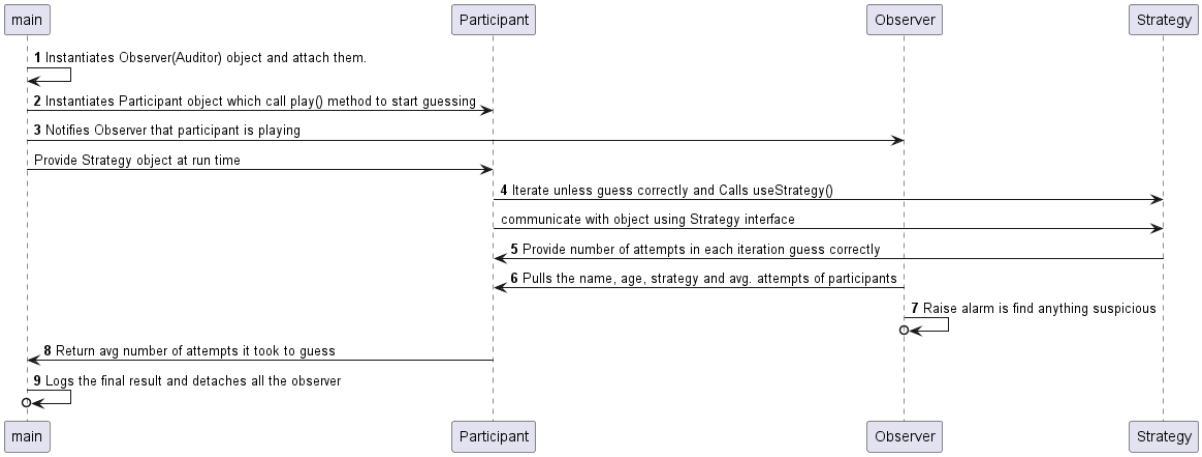
Now, all that is left is to extend this Observable trait. Hence, our client, here Main, extend Observable, and add all the three auditors mentioned earlier, and notifies them, once a participant object is created and has played the game. It is left to the observer to pull the data from subject. Once all participant has played the game, it removes all observer and ends the game. My dependency diagram looks as follows.



Class diagram:



Sequence diagram:



4.2 Reflections

I started this assignment, by first understanding the benefits of observer pattern. The use cases they can fit in. Observer pattern, as discussed earlier, is highly suitable for publisher-subscriber model. The implementation is fairly simply. We use interface to separate, what can be the subject and what can be the observer. Subject is the one who has some information to notify, observers are the ones who have some action to implement on provided information. Observer patterns allows for multi-to-multi relation, where an observer can observe more than one subject and subject also can notify multiple observers at once.

Observer works on push-pull mechanism. Where subject can push the change in status to all observer, or observer can pull the current status of the subject periodically or when needed. In our assignment, I have used push method to notify the Auditors that new participant has joined the game, but its left to the Auditors to pull in the details of participants to fetch more details. To do so, client passes the participant object to the Auditors.

In this process, I have also enhanced my knowledge on Scala. The topics that riddled me were:

- Use of anonymous function. Even though I could pass the function as argument, the function had no effect on data, until I used apply(). [9]
- Use of data structures in Scala. Specially map-mutable and immutable maps. [10]
- Use of switch cases in Scala. [11]

I have used above mentioned items in assignment as well.

Practical 5: Factory Patterns and Singleton

Let's start to understand the motivation behind **Factory Patterns** and the business use cases it resolves along with its drawbacks.

Motivation: Creating tightly coupled object in our program creates an issue of scalability. For example, consider a transport delivery app, which is used by client to deliver their goods. Initially, when the app was established, it was only for delivery via Truck. Our code structure looks like below:

```
class Transport {
  public Deliver () {
    Truck truck = new Truck();
    truck.route(fromDesitnation,toDesination);
  }
}
```

In the above code snippet, Transport is tightly coupled with Truck object. Now, if we want to scale our app to allow client delivery via Aeroplane, we can't use the same Transport class. To make it more flexible and scalable, Factory Pattern suggest, that we replace direct object construction calls (using the new operator) by delegating to a special factory method [12]. The factory pattern consists of following parts:

Creator: Declares the factory method that return new product objects.

Product: An interface, which is common to all object.

Concrete Products: Different implementation depending on products.

Concrete Creators: Override the base factory method so it returns product of specific type.

So, using factory pattern in our problem, our code structure shall look like *Figure1*.

Now, we have freed out Transport glass from being tightly coupled with Truck object, and in future we can always scale our application to deliver with any mode, all we need to do is create new Concrete Products and Concrete Creators.

Every superhero has a weakness. Similarly, the biggest weakness of the factory design pattern is the fact that its implementation leads to a strong increase in the number of integrated classes, because every Concrete Product always require a Concrete Creator [13]. This leads to more elaborate effort from developer perspective.

```
abstract class TransportMode{
  abstract method Vehicle createMode();
}

class Roadways extends TransportMode{
  method Vehicle createMode() {
    return new Truck();
  }
}

class Airways extends TransportMode{
  method Vehicle createMode() {
    return new Aeroplane();
  }
}

trait Vehicle{
  public route(fromDesitnation,toDesination)
}

class Truck implements Vehicle {
  public route(fromDesitnation,toDesination)
}

class Aeroplane implements Vehicle{
  public route(fromDesitnation,toDesination)
}
class Transport(String transportVia) {
  public Vehicle getVehicle(){
    TransportMode transportMode;
    if (transportVia == "Truck") {
      transportMode = new Roadways;
    } else if (transportVia == "Aeroplane") {
      transportMode = new Airways;
    }
    return transportMode.createMode();
  }

  public Deliver () {
    Vehicle vehicle = getVehicle;
    vehicle.route(fromDesitnation,toDesination);
  }
}
```

Figure 1

Now, we will look into **Abstract Factory Pattern.**

If we are dealing with families of products and then there is compatibility issue where one product is compatible only with other product of same family, then to handle such scenario we can use Abstract Factory Patter. For example, we have an online furniture retail app, where we provide two category of product chairs and table. Now each product falls under either of this families - Modern or retro. Now when placing order, client can order all the product together or individually provided they order the products from same family. So, if a client order modern chair then they can buy table from modern family only.

To handle such constraint, we must implement Abstract Factory pattern. Our structure must include following components.

Abstract Factory: An interface that declare method to create different product.

Concrete Factory: Classes that implement Abstract Factory. Each concrete factory class are used to create different product from same family.

Abstract Product: To declare the types of products available.

Concrete Product: To define the different types of abstract product created.

Now to solve the above problem, our code structure must look like *Figure2* and *Figure3*. Here, DesignFactory interface act as Abstract Factory and ModernFctory and RetroFactory class act as Concrete Factory. Chair and Table are Abstract Product and Modern Chair and ModernTable are Concrete Product.

```
trait DesignFactory{
  public Chair createChair();
  public Table createTable();
}
//Similar for RetroFactory
class ModernFactory implements DesignFactory {
  public Chair createChair(){
    return new ModernChair();
  }
  public Table createTable(){
    return new ModernTable();
  }
}
trait Chair {
  public sitHuman();
}
trait Table {
  public placeGoods();
}
//Similar for RetroChair
class ModernChair implements Chair{
  public sitHuman(){
    //DoSomething
  }
}
//Similar for RetroTable
class ModernTable implements Table{
  public placeGoods(){
    //DoSomething
  }
}
```

Figure 2

```
class OrderManagement(String design) {
  main(){
    DesignFactory factory;
    if(design == "Modern"){
      factory = new ModernFactory()
    }else if(design == "Retro"){
      factory = new RetroFactory()
    }
    createOrder(factory);
  }
  public createOrder(DesignFactory factory){
    Chair chair = factory.createChair()
    Table table = factory.createTable()
    List<Product> order = new ArrayList<>();
    order.add(chair);
    order.add(table);
    placeOrder(order);
  }
}
```

Figure 3

Although, this way we ensured that whenever an order is placed it always has same family of products. But it is *Difficult to support new kind of products.* Extending abstract factories to produce new kinds of Product isn't easy. That's because the Abstract Factory interface fixes the set of products that can be created. Supporting new kinds of product require extending the factory interface, which involves changing the Abstract Factory class and all of its subclass [14]. So, if we want to introduce another product say sofa, then we need to change our Design Factory and all its sub classes, also we would need to introduce new interface for sofa and its concrete classes. This adds to complexity and redundancy. Perhaps, Prototype or Builder pattern can be used to resolve such issue.

Factories usually have only a single instance. We can achieve this using Singleton Pattern.

Singleton Pattern is usually implemented by making the default constructor private, to prevent other objects from using the new operator with the Singleton class. Then create a static method that calls private constructor to create an object and save it in static field [15]. So, whenever a static method is called it always return the same object. But this creates an issue of subclasses which we would need in factory pattern. To achieve subclasses in Singleton, we can make the default constructor as protected. And create static register method which assign an instance of singleton object created by subclass. Since, subclass can access protected method from parent class it can now override the static register method and register itself as an instance of Singleton class. So, our code structure would look like *Figure4*.

```
class Singleton {
  protected Singleton(){}
  protected static void register (Singleton s) {
    instance = s;
  }
  public static Singleton getInstance() {
    return instance;
  }
  private static Singleton instance = null;
}
class SingletonSubclass extends Singleton {
  protected SingletonSubclass() {}
  public static void register() {
    Singleton.register(new SingletonSubclass);
  }
}
```

Figure 4

5.1 Work Done

After understanding the pattern, itself, I have to implement these patterns on the exercise in hand. The exercise itself is divided into 6 parts. We will discuss each part individually and action taken to complete such task.

Exercise 1: Implement Abstract Factory on given code to make client loosely coupled with Product classes.

Solution: We implemented a Scala object, which allows creation of three classes. Each class object extends abstract class Product, which provide method that can be overridden by sub-classes. This method is used by client to call for action. Our UML diagram looks like below (*Figure5)* followed by the code snippet(*Figure6)*.

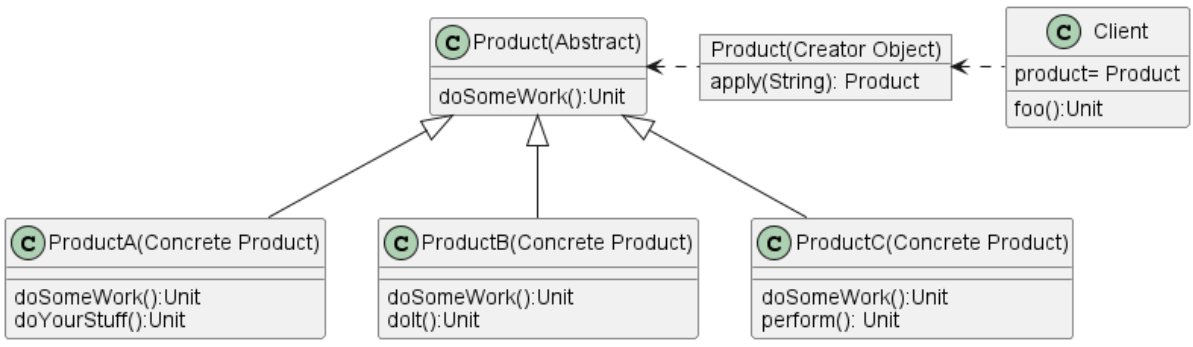


Figure 5

Snippet from the code:

```
abstract class Product{
  def doSomeWork(): Unit
}

object Product {
  private class ProductA extends Product{
    override def doSomeWork(): Unit = doYourStuff()
    def doYourStuff() =
      println("I'm a ProductA, doing my stuff")
  }

  private class ProductB extends Product {
    override def doSomeWork(): Unit = doIt()

    def doIt() =
      println("I'm a ProductB, doing it")
  }

  private class ProductC extends Product {
    override def doSomeWork(): Unit = perform
    def perform =
      println("I'm a ProductC, performing")
  }

  def apply(productName: String):Product = {
    productName match {
      case "ProductA" => new ProductA()
      case "ProductB" => new ProductB()
      case "ProductC" => new ProductC()
    }
  }
}
```

Figure 6

Exercise 2: Creating a family of Products. So, we have three Product namely, ProductA, ProductB and ProductC. Each of this Product falls under either family of Product CoolProduct or NormalProduct.

Solution: To implement this we use Abstract Factory pattern. We have two concrete factory class namely CoolProductFactory and NormalProductFactory, each implement interface ProductFactory. ProductFactory defines three methods to create product and returns either ProductA, ProductB or ProductC. All product ProductA, ProductB or ProductC are abstract class which implements Product. For each of this abstract class we have two concrete sub-class. For example, ProductA has two sub-class CoolProductA and NormalProductA. Similarly, for ProductB and ProductC. Main class take input from user to create which family of product and then calls the respective factory. Then Main method passes this to client. Our client is only dependent on two interface ProductFactory and Product. The UML diagram for this implementation is mentioned in *Figure7.*

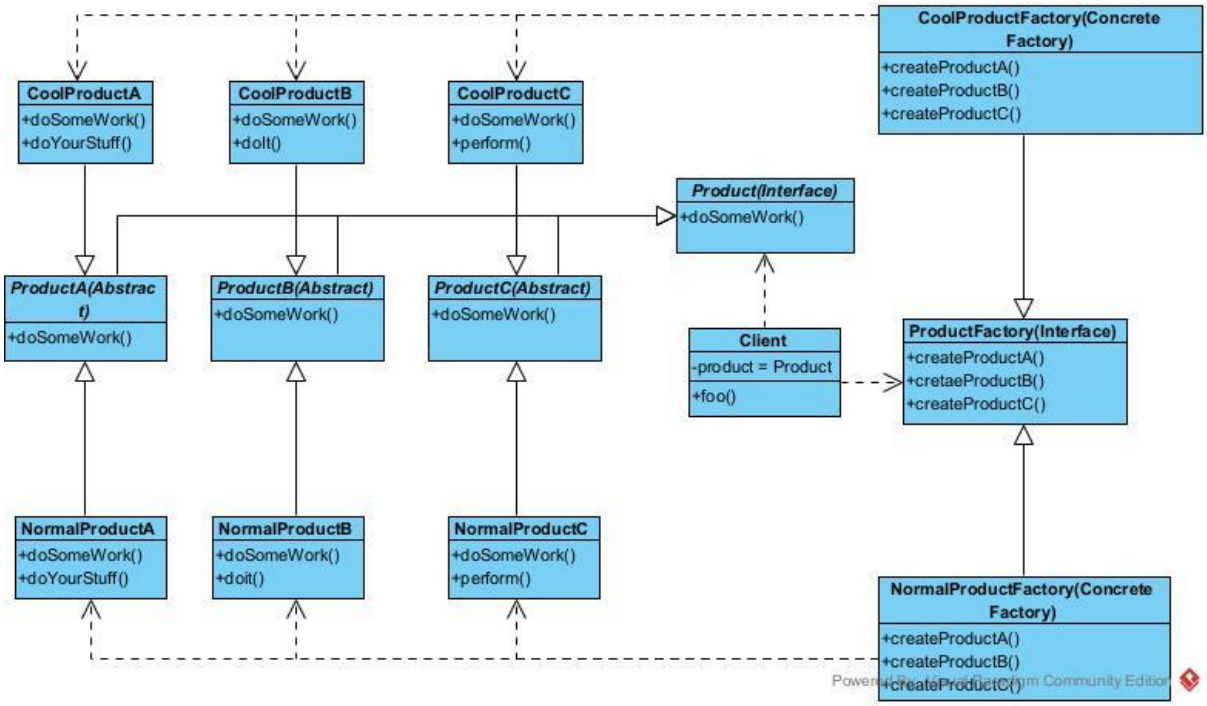


Figure 7

Program output:

```
Enter 1 for Cool and 2 for Normal
1
Creating Cool Products
I'm a ProductA, doing my Cool stuff
I'm a ProductB, doing it cool
I'm a ProductC, performing cool stuff
```

Exercise3: Create a new family of product called DeadlyProduct on top of exercise2.

Solution: To do this all we need to do is create another concrete factory called *DeadlyProductFactory.* And then we need to create concrete classes for each product. These concrete classes are name as *DeadlyProductA, DeadlyProductB and DeadlyProductC.* Final step is to create another placeholder for user input. These changes will work with existing client class. Now our class diagram looks as *Figure8.* The newly introduced class are mentioned in green.

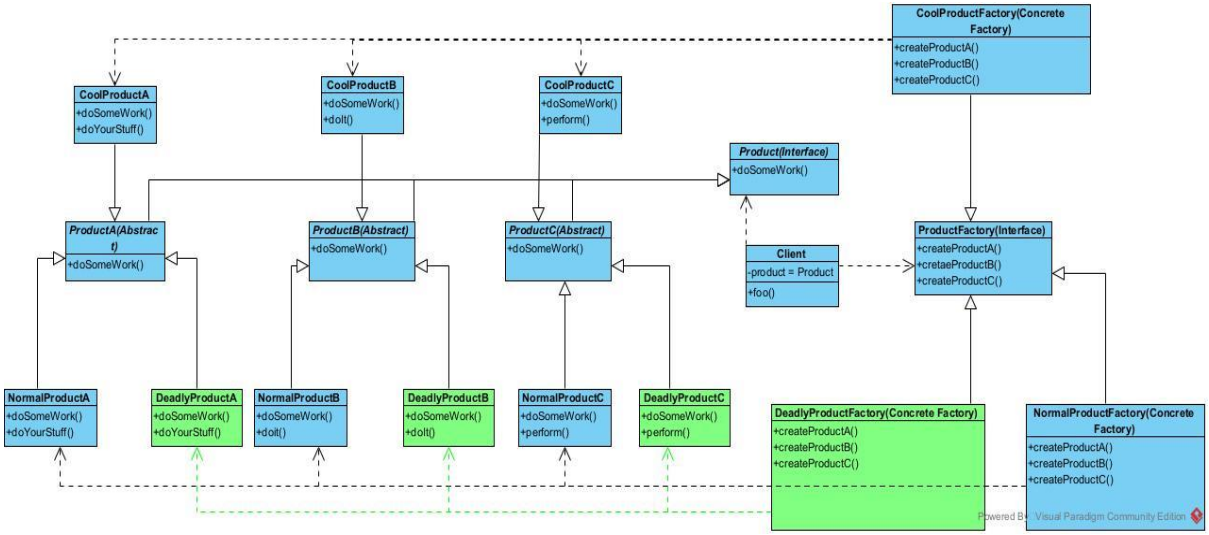


Figure 8

The output for exercise 3 after the change:

```
Enter 1 for Cool, 2 for normal, 3 for deadly
3
Creating Deadly Products
I'm a ProductA, doing my Deadly stuff
I'm a ProductB, doing it deadly
I'm a ProductC, performing deadly stuff
```

Exercise 4: Now we will another Product, called *ProductD*.

Solution: We need to first create an abstract class *ProductD* which implements interface *Product,* then we need create concrete sub-class of ProductD in all *ProductFactory*. Then we need to call *ProductFactory* to give instance of ProductD. So, after doing the change our class diagram looks like *Figure9.* Newly added class are mentioned in green.

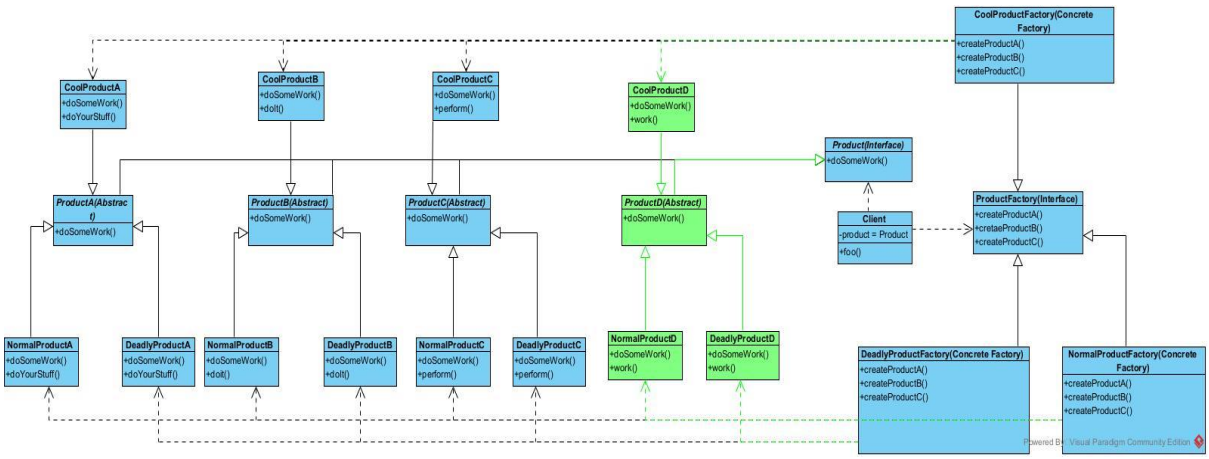


Figure 9

Program output:

```
Enter 1 for Cool, 2 for normal, 3 for deadly
2
Creating Normal Products
I'm a ProductA, doing my Normal stuff
I'm a ProductB, doing it normally
I'm a ProductC, performing normal stuff
I'm a ProductD, performing normally
```

Exercise 5: Applying singleton to concrete factory classes, so any given time there exist one instance of each factory.

Solution: In scala, to create singleton we simply need to convert class to object. Singleton objects effectively implement the singleton pattern in Scala [16]. And when we want to have an instance of this factory, we don't call new operator instead we call the object itself, which always return the same instance. So, to convert our current factory classes like *Figure10*. This change need to be done for all the concrete factory class.

```
class CoolProductFactory extends ProductFactory {
  override def createProductA(): ProductA = ProductA("CoolProductA")
  ...
}

def main(): Unit =
  var productFactory = new CoolProductFactory()
  ...
```

//We convert to:

```
object CoolProductFactory extends ProductFactory {
  override def createProductA(): ProductA = ProductA("CoolProductA")
  ...
}

def main(): Unit =
  var productFactory = CoolProductFactory
  ...
```

Figure 10

Exercise 6: Task here is to dynamically replace an existing factory class with another and replace all existing reference with new.

Solution: We converted our ProductFactory interface to abstract class and then we created companion object in scala for ProductFactory, whose apply method is designed to create concrete singleton factory object on user input. Main method instantiates the ProductFactory. So pseudocode for such design is as follows.

```
abstract class ProductFactory{
  def createProductA(): ProductA
  ...
}
//companion object
object ProductFactory{
  object CoolProductFactory extends ProductFactory {
    //doSomething
  }

  object NormalProductFactory extends ProductFactory {
   //doSomething

  }

  object DeadlyProductFactory extends ProductFactory {
   //doSomething
  }

  def apply(productFactory: Int): ProductFactory = {
    productFactory match {
      case 1 => CoolProductFactory
      case 2 => NormalProductFactory
      case 3 => DeadlyProductFactory
    }
  }
}
//dynamically change ProductFactory sub-type
//but any given time there is only one instance of ProductFactory
//so all existing sub-type will be replaced by new one
def main():
   println("Enter 1 for Cool, 2 for normal, 3 for deadly, 0 to exit ")
   while (input != 0) {
   input = scala.io.StdIn.readInt()
   productFactory = ProductFactory(input)
   val myClient = Client(productFactory)
   myClient.foo()
   }

class Client(productFactory: ProductFactory):
  var product:Product = productFactory.createProductA()
  def foo() =
    product.doSomeWork()
    product = productFactory.createProductB()
    product.doSomeWork()
```

5.2 Reflections

We studied three design patterns namely – Factory Pattern, Abstract Factory Pattern and Singleton. We saw the motive behind creating this pattern and how we step to successfully implement this pattern. We also studied the drawbacks of each pattern. Factory pattern helps in avoiding tight coupling of product creation and client. We saw that how Abstract Factory pattern helps in adhering to *Single Responsibility Principle,* by extracting the product creation code in one place but we also experienced that whenever we need to introduce new Product or a new family of Product, although

our client code saw little to no modification, we ended up creating multiple abstract class, interfaces and then again concrete class. So, it does provide the flexibility of scaling the application but maintainability becomes tedious job and can cause error prone. We also figured that singleton though limited in terms of usability and violates the concept of creating multiple instances of object, it is useful when combined with abstract factory pattern and is very easy to implement in scala. In terms of scala we gained knowledge on companion object, the apply method and its application.

Practical 6: The State Pattern

State is a behavioural design pattern that lets an object alter its behaviour when its internal state changes. It appears as if the object changed its class [17]. We use this pattern when there are finite number of states a program can be in and each state behaves differently than other. When such behaviour has to be implemented, we end up having multiple if-else conditions trying to switch from one behaviour to another thus create complex logic to trace. Also, if we have to add or modify new behaviour, we have to do lot of condition modification. More states we add more difficult it will be to maintain the transition logic.

To solve this state pattern, suggest that we add new class for any possible state of an object and segregate state-specific behaviour into this class. A *context,* store a reference to current state object and delegate all state specific work to this object. During transition, context replace the current state object with different one. States might know about existence of other state but their strategies or behaviour is not known.

To implement we should have an abstract class or interface which represent the set of states. Then we must have concrete class each implementing this state interface. Context class which maintains the current state and call for transition between state when some condition is met and client which calls context with initial state.

6.1 Work Done

Given exercise we have Person class which has multiple stage of life such as children, adult and pensioner. Now we need to apply state pattern on existing code so that our code is leaner and easy to maintain.

Exercise 1: In first exercise we are implementing state pattern. After implementation our code structure looks like *Figure11*.

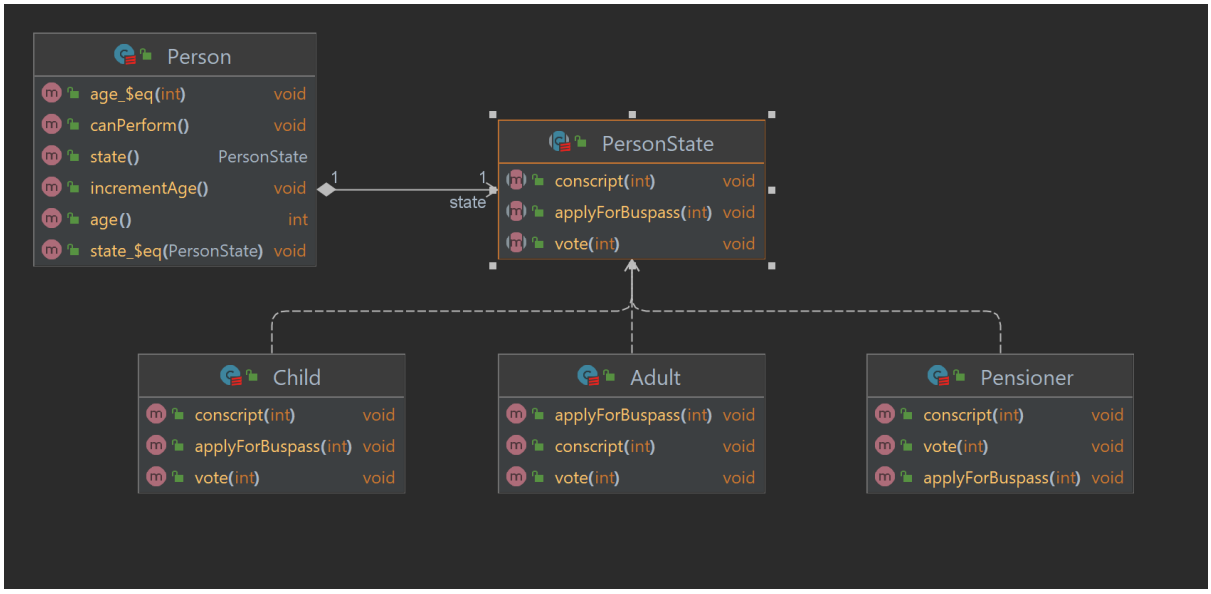


Figure 11

Explanation: PersonState is abstract class which is extended by three sub-classes namely Child, Adult and Pensioner. Person class act as *context* and helps in state transition and assigning work to each state.

Now, initial stage is set to Child, so Person is pointing to Child class, then when age increments to 18 the state is set to Adult, now Person is demanding service from Adult class.

Exercise 2 and 3: We are adding new state Teenager and new behaviour applyForMedicalCard(), which is applicable to only few states. Since we have already modified our code to state pattern, adding new state is as simple as adding a sub class, and new behaviour can be added to each state to see how they react to this behaviour. Now our code structure looks like *Figure12*.

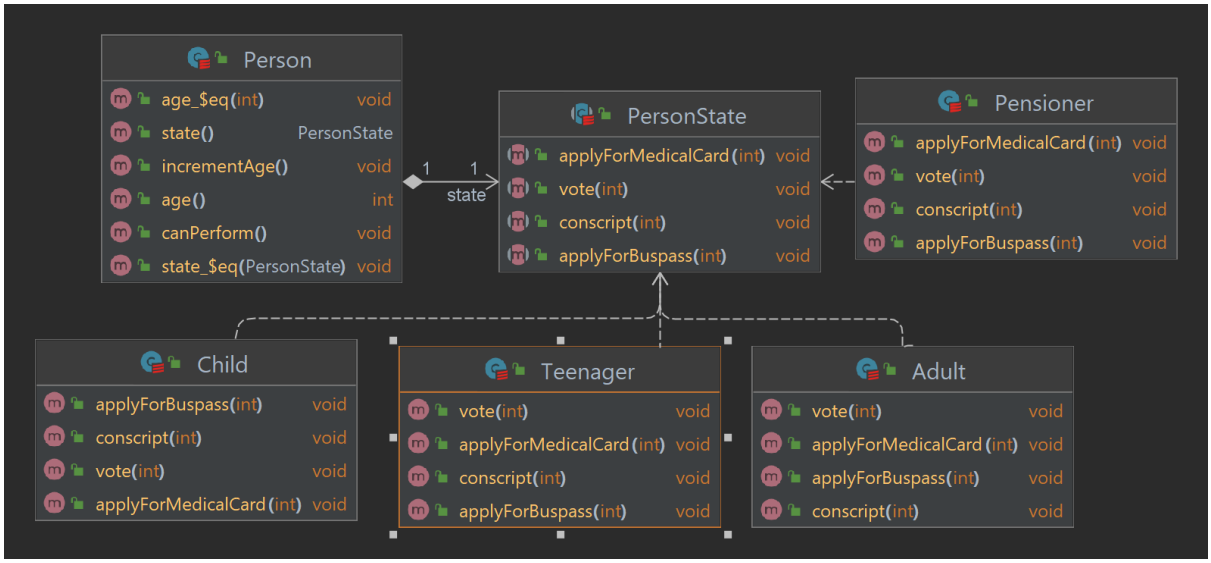


Figure 12

Exercise 4: Up till this point the state transition was happening in Person class, now we will move to state transition to the state class itself. So basically, it will be job of each state to either transition to some another state or remain in same state. We can understand this from the code snippet mentioned below.

```
class Person:
  var age = 0
  var state: PersonState = new Child(this)
  def changeState(state: PersonState) =
    this.state= state

class Child(person: Person) extends PersonState(person){
  override def transitionState(): Unit =
    if (person.age == 13) then
      person.changeState(new Teenager(person))

class Teenager(person: Person) extends PersonState(person){
  override def transitionState(): Unit =
    if (person.age == 18) then
      person.changeState(new Adult(person))
```

Here, we can see that initial state is set to child by Person, but now person doesn't change the state, but its Child, which checks whether its still good to be Child or transfer the action to Teenager, and so does the Teenager class change state to point to Adult Class. Now code diagram looks as displayed in *Figure13.*

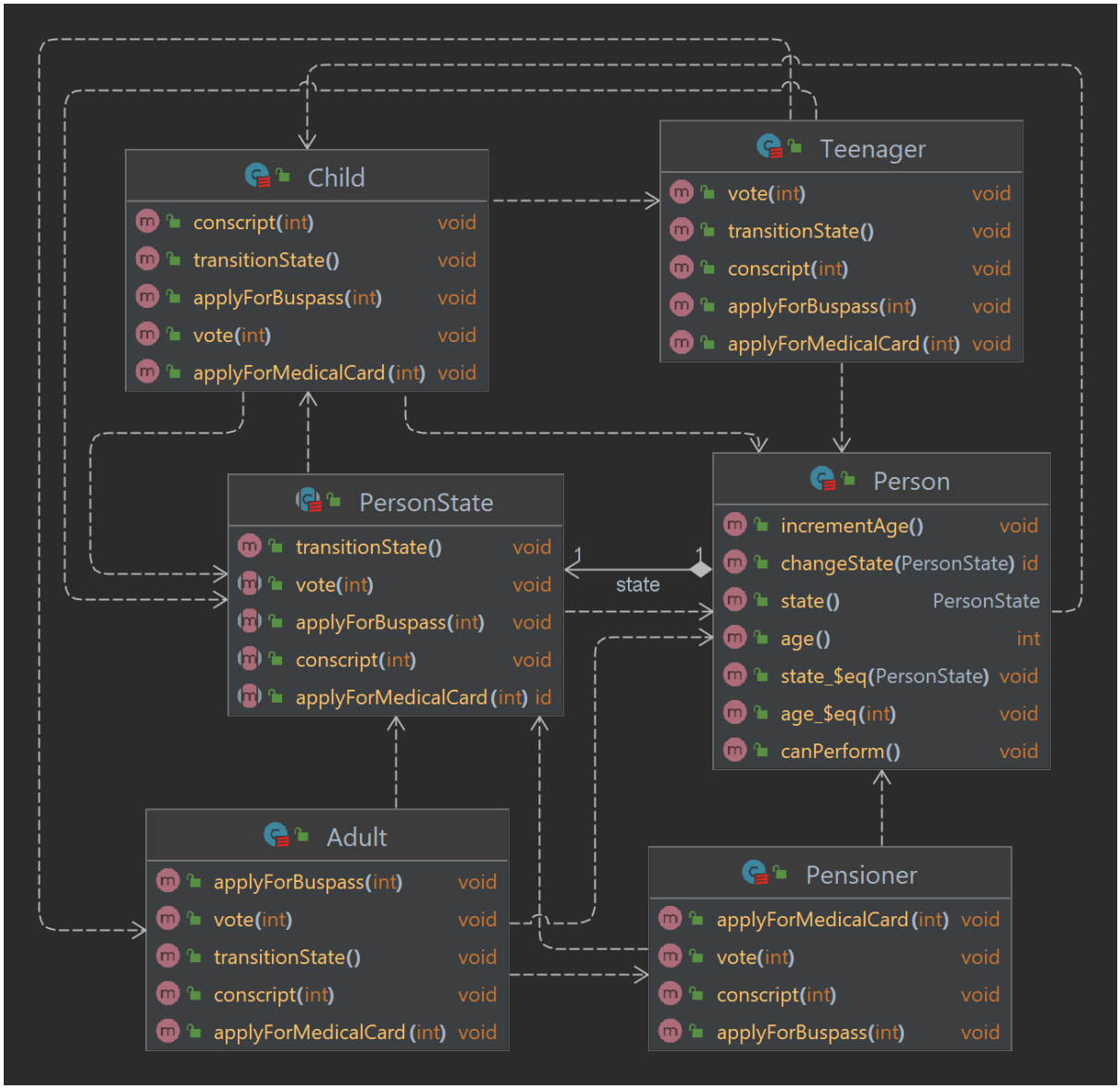


Figure 13

When this will be good idea?

When transformation has to happen on its own, without the interference of the client, or without client actually knowing the current state then we can use this strategy. If we look in this exercise itself. Person as client is only aware that it was born as child and now it can't control when it becomes a teenager, adult or pensioner, it happens naturally in the background.

Exercise 5: Creating single object for each state lazily. To do this we will implement singleton pattern along with state pattern. We convert each state class to object which extends abstract class *PersonState* to implement singleton and rest of the structure remains the same. The code structure looks like *Figure 14.*

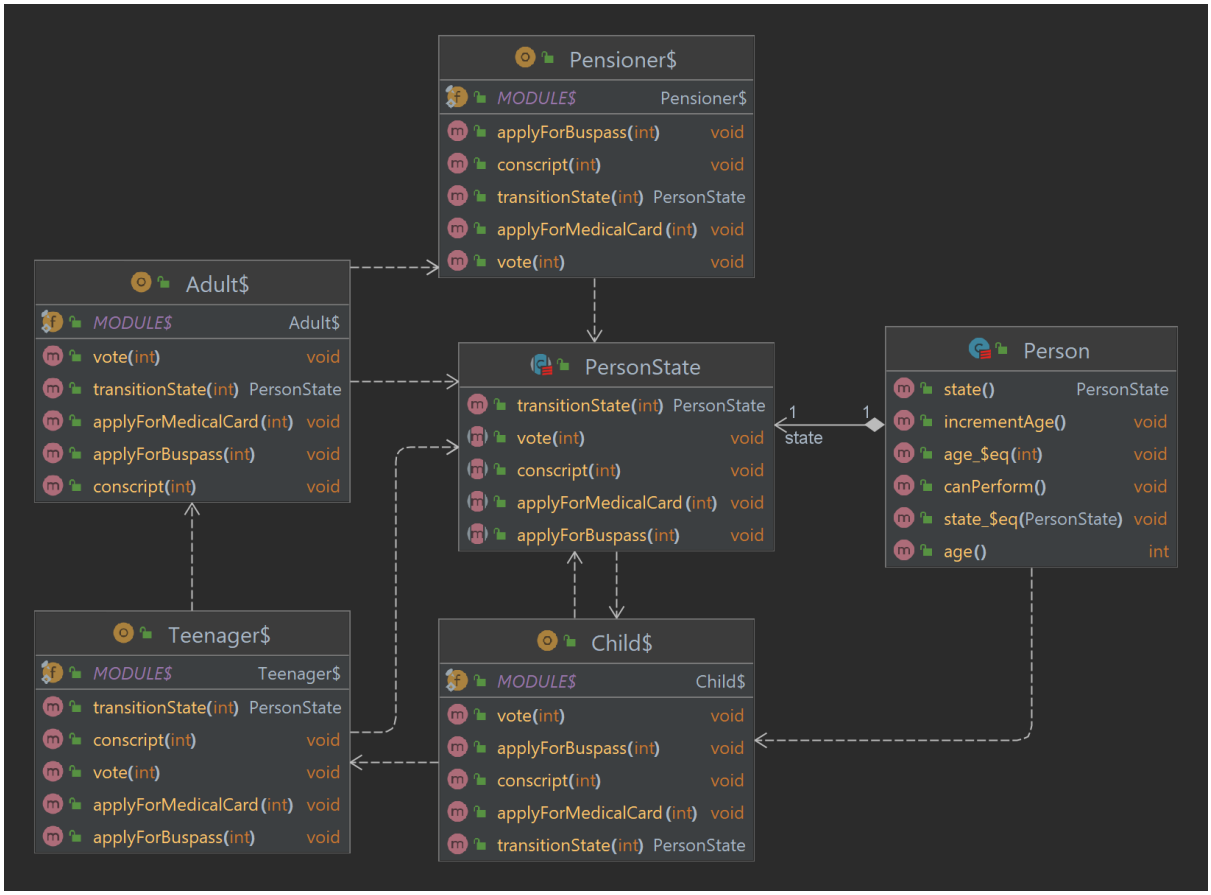


Figure 14

6.2 Reflections

State pattern structure looks very similar to Strategy pattern, but one key difference between them is that in State pattern, one particular state might be aware of each other and initiate transition from one state to another, whereas strategies never know about each other.

State pattern can be used when we know there is a behavioural difference met on some condition. And to avoid complicated enormous if-else conditional blocks. Traceability and maintainability become easier when state pattern is implemented. Due diligence must be applied as it can be easily misused if we are only using for fewer condition and there is no change in class behaviour depending on current value.

Practical 7: The Visitor Pattern

Visitor pattern is behavioural design pattern that lets you separate algorithms from the object on which they operate. Let's see when and how we can use this pattern.

Assuming a use cases, where we have well structured legacy program, where a class is a writer which writes some business data to .xml file after fetching form source and cleaning the data. Now, we have a requirement to write a code to convert the .xml file into .json format for some different business client. To solve this, we can either write the converter logic inside the writer class by adding a method, or implement another writer class which generate the .json file by again fetching the data from the source. But problem with these two approaches is if we add a logic inside our writer class then we risk of introducing bug to our fully functional code and if we apply second approach then we end up maintaining larger code base which was required only as one time activity.

Visitor pattern suggests that you place the new behavior into a sperate class called visitor, instead of trying to integrate it into existing classes. The original object that had to perform the behavior is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object. [18]

Visitor's method uses Double Dispatch technique. In this, instead of letting the client select a proper version of the method to call from visitor class, we delegate this choice to object we 're passing to visitor as argument. This object "accept" a visitor and tell it what visiting method should be executed.

7.1 Work Done

The problem statement: Implement visitor pattern to traverse a binary tree.

Exercise 1 to 4: We are provided with a binary tree which has *Addition* node and *Multiplication* node, now to perform infix or postfix traversal via visitor pattern, we must first decide what will be part of visitor, will it be the label in this node, so that visitor method can return whether label is '+' or '*', or will it be the traversing infix or postfix. To solve this issue, I decided to segregate what is concrete part of class and what part fluctuates. As we can tell that traversing is not fixed as we can perform different type of traversal logic, I decided to put my traversal logic in visitor method. Hence, after implementing visitor pattern my code snippet look like *Figure 15.*

Code-walkthrough: To implement visitor, I first created an interface *Visitor*, which provides two methods, one to traverse and other for accessing leaf. Each traversal (infix or postfix) then implements its own concrete class by extending this interface. Then Node act as abstract element which has accept method. This accept method takes visitor as argument. Now each subclass overrides this accept method by passing themselves to specific visitor method. Client(main) decides at runtime which Concrete Visitor (which traversal) it needs to pass to child nodes, and then each child calls their accept method. This process is repeated in recursion, to evaluate the entire tree.

The code structure looks like *Figure 16*

```
trait Visitor:
  def traverse(node:BinaryOperatorNode): String
  def assignleaf(node: LeafNode): String =
    node.label

class ConcreteInfixVisitor extends Visitor:
  override def traverse(node: BinaryOperatorNode): String=
    val infix: String =  "(" + node.leftNode.traverse + " " +
      node.label + " " + node.rightNode.traverse + ")"
    infix.toString


class AdditionNode(left: Node, right: Node) extends
BinaryOperatorNode(left, right):
  val label = "+"
  override def accept(visitor: Visitor): Unit =
    traverse = visitor.traverse(this)

@main def main(): Unit =
...
var visitor:Visitor = new ConcreteInfixVisitor()
var node1 = AdditionNode(LeafNode("a"), LeafNode("b"))
node1.accept(visitor)
...
```

Figure 15

Code Structure for Exercise 4:

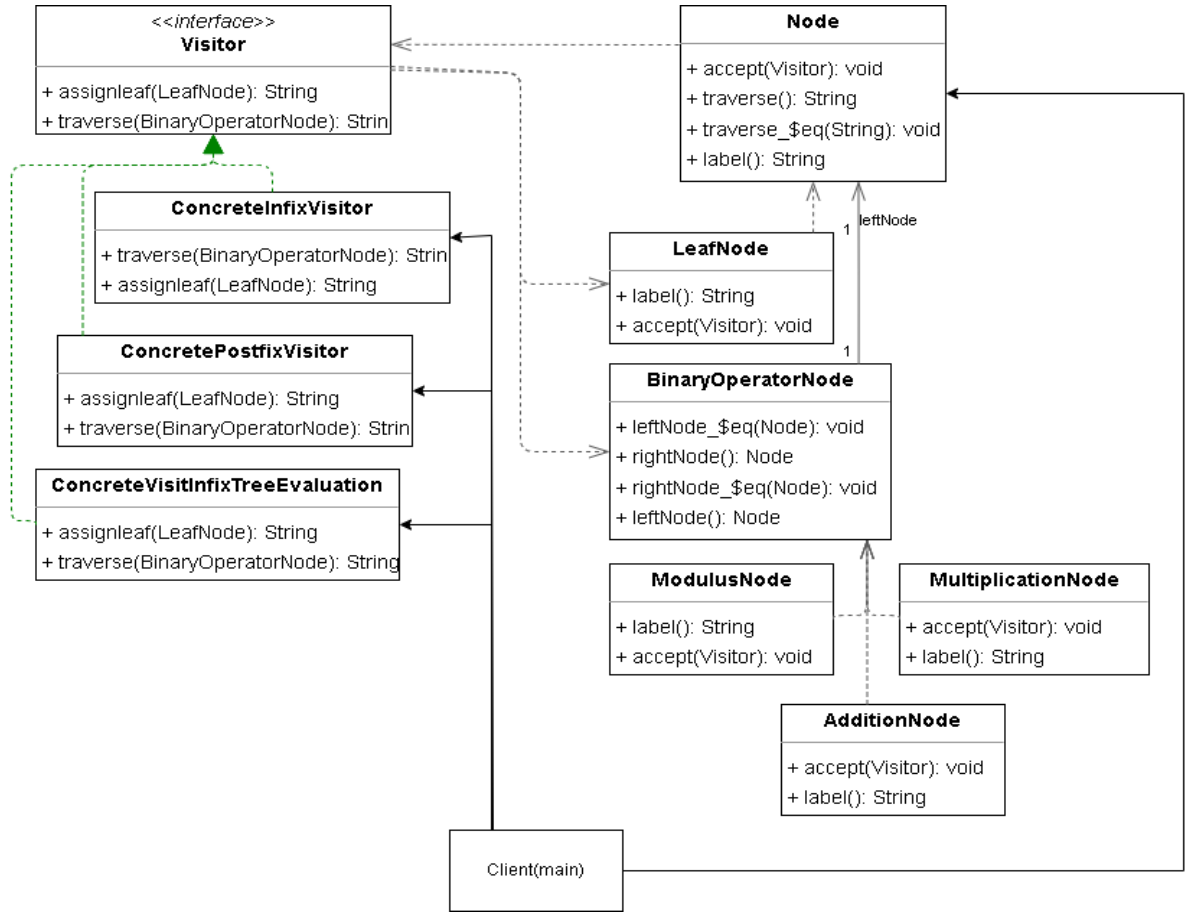


Figure 16

Output of Exercise 4

```
C:\Users\vivek\.jdks\openjdk-18.0.2.1\bin\java.exe "-javaagent:C:\Pr
Infix Traversal:
(((a + b) * c) + 7)
Postfix Traversal:
a b + c * 7 +
Infix Traversal Evaluation:
117
Infix Traversal witH Modulus:
(((a + b) * c) + (13 % 6))
Infix Traversal Evaluation with Modulus:
111

Process finished with exit code 0
```

Exercise 5: The traversal logic was brought back to node class and to exploit the benefits of pattern class we implemented a visitor which counts the number of addition nodes in tree. Code snippet for such implementation is mentioned in Figure 17. This visitor is accept by all Node class.

```
trait Visitor:
  var nodeCount:Int=0
  def count(node:BinaryOperatorNode): Unit

class ConcreteCountVisitor extends Visitor:
  override def count(node: BinaryOperatorNode): Unit=
    if(node.isInstanceOf[AdditionNode] == true)
      nodeCount+=1
```

Figure 17

Output of Exercise 5:

```
C:\Users\vivek\.jdks\openjdk-18.0.2.1\bin\java.exe "
Infix Traversal:
(((a + b) * c) + 7)
Number of Additional Nodes: 2
```

Exercise 6: Implementing Iterator logic externally of Concrete Visitor class. To do this we create an interface *treeIterator* which is implemented by concrete infix and postfix iterator. Visitor method calls this iterator to travers through the tree. The code snippet and structure are in Figure 18 and Figure 19.

```
trait treeIterator:
  def hasChild(node: Node):Boolean
  def getChild(node: Node):String

class ConcreteInfixTreeIterator extends treeIterator:
  override def getChild(node: Node): String =
    var label:String= ""
    if(hasChild(node) && node.isInstanceOf[BinaryOperatorNode]==true)
      label="("
      label=label+getChild(node.asInstanceOf[BinaryOperatorNode].leftNode)
      label=label+node.label
      label=label+getChild(node.asInstanceOf[BinaryOperatorNode].rightNode)
      label= label+")"
    else
      label= label+node.label

    label

  override def hasChild(node: Node): Boolean =
    if(node.isInstanceOf[LeafNode]==true)
      false
    else
      true

class ConcreteInfixVisitor extends Visitor:
  override def traverse(node: BinaryOperatorNode): String=
    val iterator= new ConcreteInfixTreeIterator()
    iterator.getChild(node)

class AdditionNode(left: Node, right: Node) extends BinaryOperatorNode(left, right):
  val label = "+"
  override def accept(visitor: Visitor): Unit =
    traverse = visitor.traverse(this)

@main def main(): Unit =
  var node1 = AdditionNode(LeafNode("a"), LeafNode("b"))
  var visitor:Visitor = new ConcreteInfixVisitor()
  node1.accept(visitor)
```

Figure 18

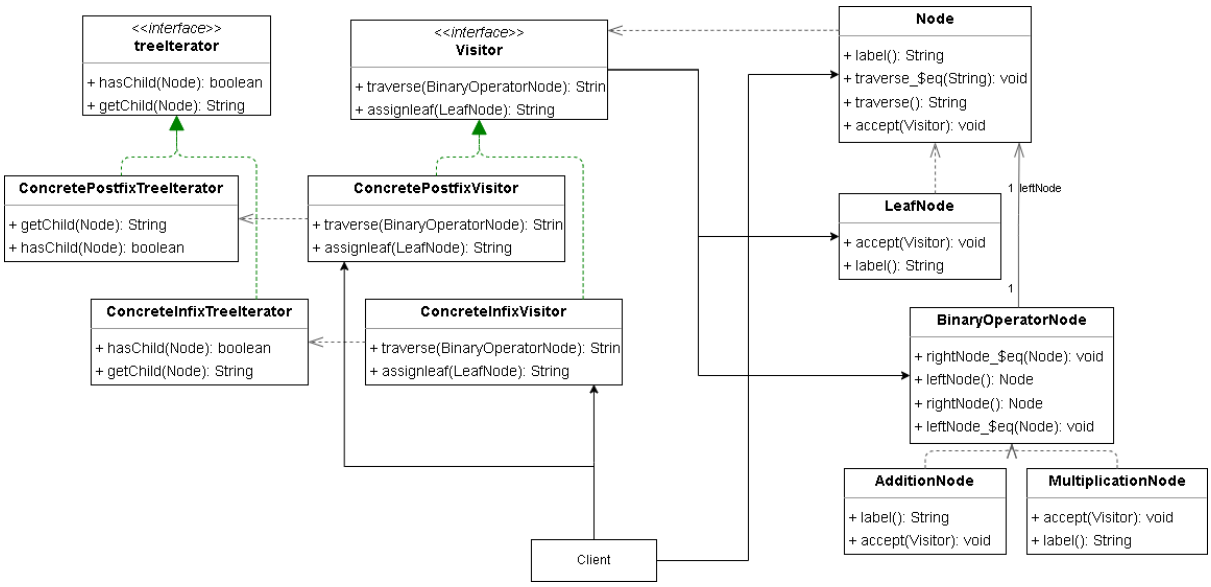


Figure 19

Which is better *internal* or *external* iteration?

In external iteration, we have to introduce another iterator and then call it via visitor. This increases code complexity and can create maintainability issue. The traversal logic itself become complicated as we have to implement method of iterator interface, whereas in internal iteration the traversal logic was part of visitor and could use easy recursion logic. Hence, in this case I can confirm that internal

iteration is better than external iteration but if traversal logic was more complicated perhaps external iteration would have been more useful.

Exercise 6: Implementing Scala case class to use pattern matching for visitor class. For this I have to refactor the entire code, and convert each concrete class of visitor into case class where each such class has only one implementation. Then we implemented accept method in abstract Node class itself so to avoid duplication is sub-classes and inside this accept we implemented pattern matcher against each concrete visitor case class. This visitor object is created by client and pass to accept method of Node class as argument and once it matches with one of concrete visitor the Node class passes itself to visitor to perform tree traversal. The code snippet of this flow is Figure 20.

```
trait Visitor:
  def node:Node
case class ConcreteBinaryInfixVisitor(node: Node) extends Visitor:
  def traverse(): String =
    var infix: String=""
    if(node.isInstanceOf[BinaryOperatorNode])
      infix =  "(" +node.asInstanceOf[BinaryOperatorNode].leftNode.traverse
        + " " + node.label + " " +
       node.asInstanceOf[BinaryOperatorNode].rightNode.traverse + ")"
    else
      infix = node.label
    infix.toString


case class ConcreteBinaryPostfixVisitor(node: Node) extends Visitor:
  def traverse(): String =
    if (node.isInstanceOf[BinaryOperatorNode])
      node.asInstanceOf[BinaryOperatorNode].leftNode.traverse + " " +
       node.asInstanceOf[BinaryOperatorNode].rightNode.traverse + " " +
       node.label
    else
      node.label


abstract class Node:
  def label: String
  var traverse: String=null

  def accept(visitor: Visitor): Unit =
    //Use pattern matching for visitor
    traverse = visitor match
      case infix: ConcreteBinaryInfixVisitor => infix.traverse()
      case postfix: ConcreteBinaryPostfixVisitor => postfix.traverse()

@main def main(): Unit =

  //Create B-tree
  var node1 = AdditionNode(LeafNode("a"), LeafNode("b"))
  var node2 = MultiplicationNode(node1, LeafNode("c"))
  var node3 = AdditionNode(node2, LeafNode("7"))

  var list = List(node1,node2,node3)
  var visitor: Visitor=null

  for(n <- list)
  //Create Visitor at runtime and pass to node
    visitor = new ConcreteBinaryInfixVisitor(n)
    n.accept(visitor)
  //Print the infix traverse
  println(node3.traverse)
```

Figure 20

7.2 Reflections

Visitor pattern is useful in scenarios where we need to perform an operation on all elements of a complex object structure like in our B-tree exercise. It is also useful when we want to keep business logic of auxiliary behavior away from primary class thus keeping it clean. Furthermore, if we want to apply operation on only some classes of class hierarchy but not others, we could implement visitor pattern.

To implement visitor pattern, we use a technique called Double Dispatch which is explained earlier. Visitor pattern helps in adhering to Open/Closed Principle by allowing extension of additional behavior without changing their initial behavior thus closed. Visitor along with iterator can benefit in traversing a complex data structure and execute some operation over these elements, even if the elements of such structure are of different class.

The biggest issue with Visitor pattern is maintainability. We need to update all visitor implementation each time a concrete element class is modified. If the behavior of this class is changed it can make our visitor class completely useless if it is not modified. In addition to this, since visitor cannot access private field of element class it might not be able to perform operation that are dependent on such private field.

Final Exercise: The Happy Pattern

Happy Pattern is not a pattern itself, but an exercise, where we need to apply several design patterns on given use cases that makes our code loosely coupled, flexible, easier to maintain and scalable to changing business needs.

Problem Statement: Apply design pattern on Lego Invoice Creator while preserving the basic functionality. Basic functionality, here, is to create invoice for components order like Block, windows, door etc; and calculate the total amount.

8.1 Work Done

To start with, we analysed the given scala code, on which we need to apply design pattern. The first major problem that was evident was *InvoiceCreator* class is providing different implementation for creating each type of component to get the final amount. Problem with this is whenever we have new kind of component, each time we would need to create new method in original Invoice class. In addition to this what if there is one component whose price depends on the total amount of its child component. Implementing method to construct object for that kind of class would be difficult. For example, house itself is a component, whose price will depend on, individual child component like Blocks, door, window. To get the total price we will end up creating multiple objects of individual type, which again might depend on some other leaf component. The above diagram can be summarized in *Figure21*.

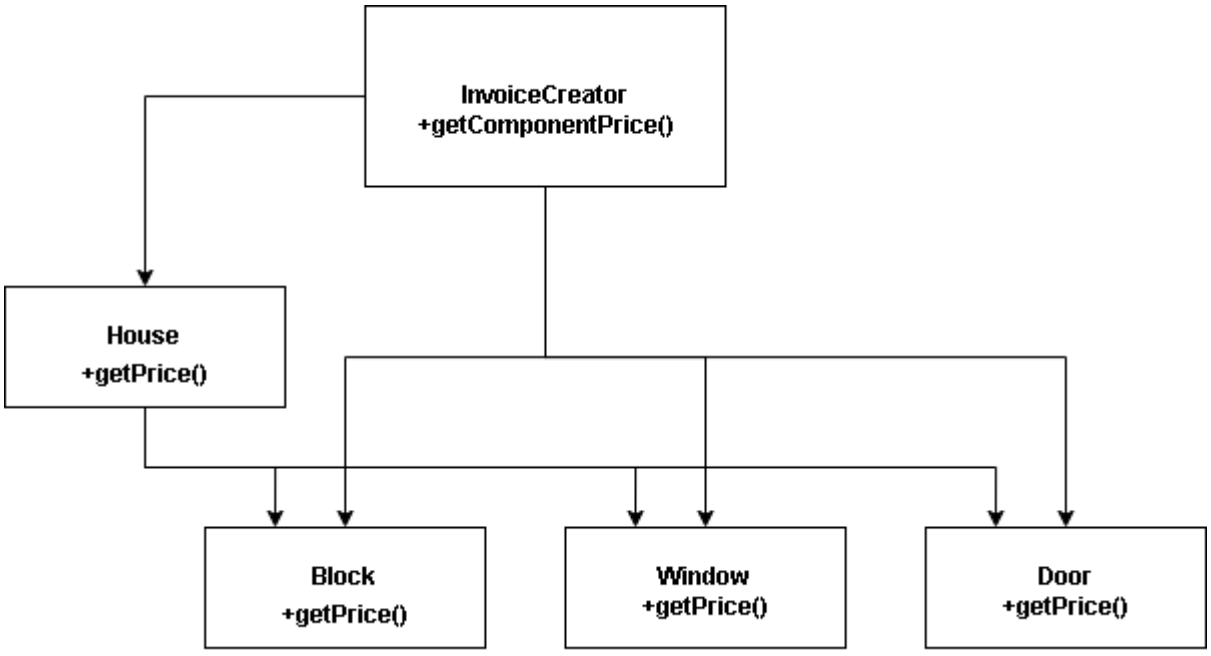


Figure 21

Solution: *Composite Pattern*

To solve above mentioned problem, we will implement Composite Pattern.

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects [19].

To apply composite pattern, we will create Composite (InvoiceComposite) class that will extend the Component interface. It provide implementation for two method, namely add(Component) and remove(Component). This method allows to dynamically add or remove into List *components,* which is declared and maintained in Composite class itself. Since, it implements component class, for each operation of component it will iterate through list of components and call their respective methods, thus eliminating the need to create different method for each type of component, and can reclusively call sub-child component to get to the leaf component. After implementation our code design looks like *Figure22*.

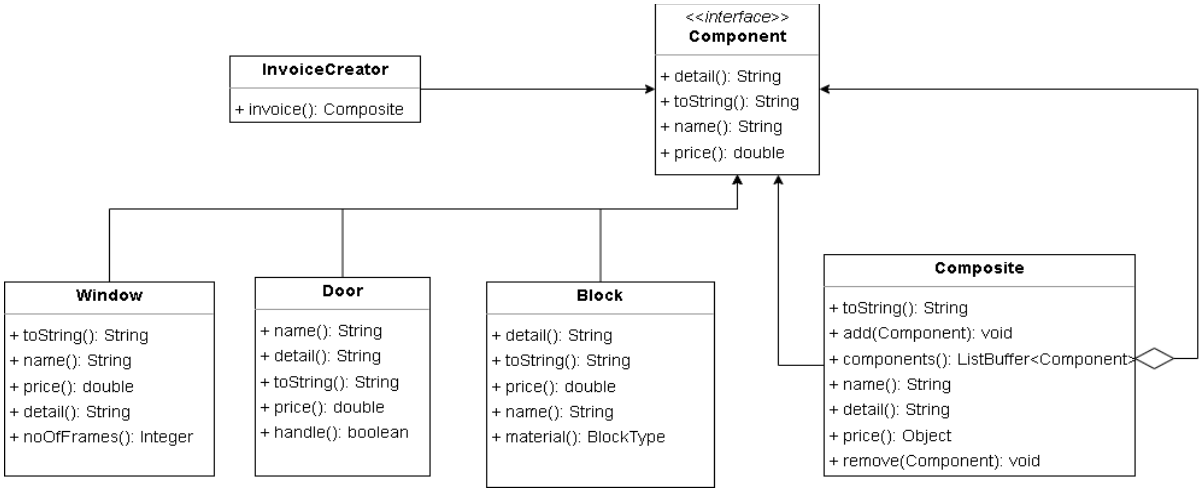


Figure 22

Code snippet for Composite class is mentioned in *Figure23*.

```
case class Composite(name: String,
                     detail: String) extends Component {
  override def price= components.map(_.price).sum
  val components = new ListBuffer[Component]()
  def add(child: Component):Unit = components += child
  def remove(child: Component):Unit =components -= child
  override def toString =
    val compList = components.foldLeft("")((str, c) => str + c.toString +
"\n")
    s"${compList}\nTOTAL PRICE: €${price}"
}
```

Figure 23

Next, InvoiceCreator will call this to add components, to get the final total. Code snippet in *Figure24*.

```
class InvoiceCreator:
  val houseComponent = Composite("House","House contain child component")
  houseComponent.add(Block())
  houseComponent.add(Window())
  houseComponent.add(Door())

  val invoice = Composite("Parent","This is Parent Component")
  invoice.add(Block())
  invoice.add(Block("Block432", 8.34, BlockType.Hollow))
  invoice.add(Window())
  invoice.add(Door())
  invoice.add(houseComponent)
```

Figure 24

Next: This does solve the issue of component creation, but our Invoice Creator looks unorderly when creating composite object. Also, there is no order in which composite methods need to be called. Example, we shouldn't be calling *remove* method before *add* method.

To solve this kind of issue, we could use Builder Pattern, which gives us nice, orderly, easy to read, way of constructing object.

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code [20].

To apply builder pattern, we will create *ConcreteBuilder* class, which takes *InvoiceComposite* object as input, and then implements method as series of steps, where each method directs the *InvoiceComposite* to do some job and then return itself. Finally it implements the *build()* method which build and return the *InvoiceComposite* object.

In Addition to this we implemented Singleton Pattern, on top of bridge pattern so that there exist only one Invoice Builder at any given time. Hence now our design looks like Figure 25.

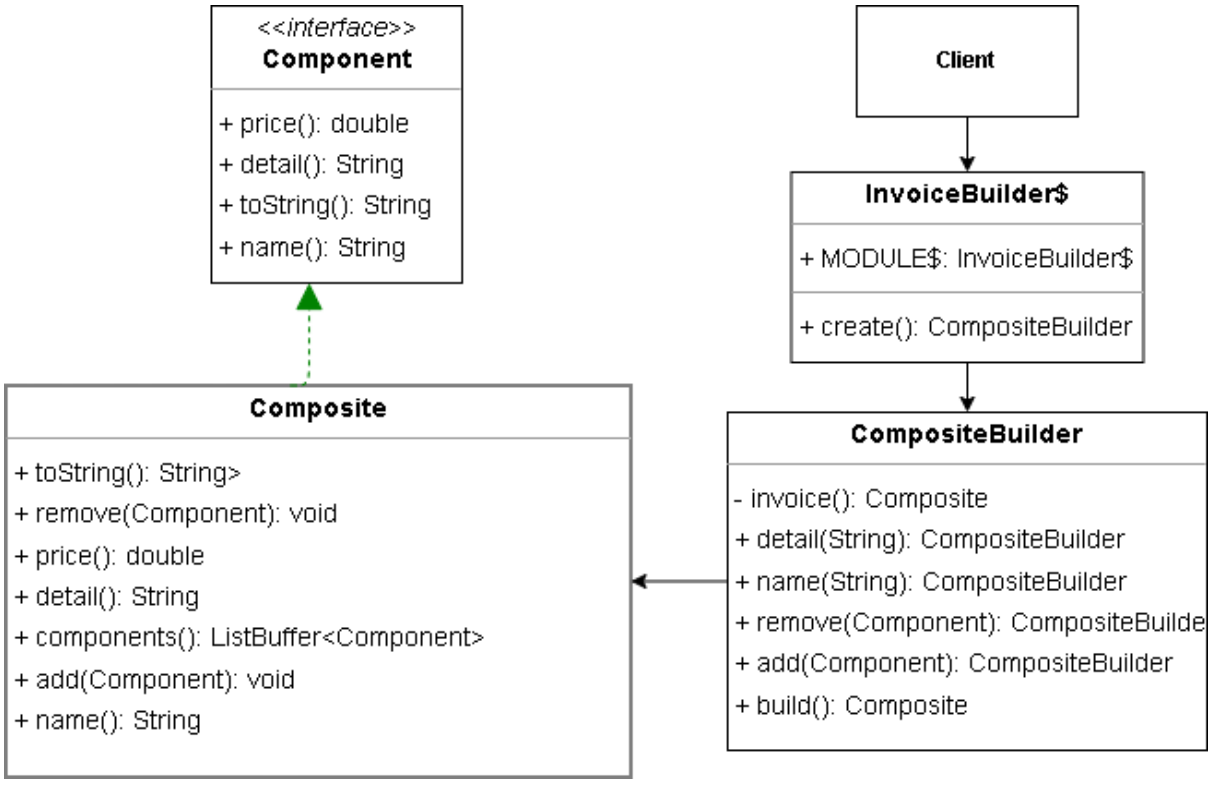


Figure 25

The code snippet for above design would look like *Figure26.*

Now our code is readable of how *InvoiceComposite* needs to be created using builder.

```
class CompositeBuilder:
  def name(compositeName: String): CompositeBuilder={ name = compositeName;
this}
  def detail(compositeDetail: String):CompositeBuilder = {detail =
compositeDetail;this}
  private val invoice = Composite(name,detail)
  def add(child: Component) = { invoice.add(child); this }
  def remove(child: Component)= { invoice.remove(child); this }
  def build() = invoice

object InvoiceBuilder:
  def create() = new CompositeBuilder

@main def main(): Unit =
  val houseComponent = InvoiceBuilder.create()
    .name("House")
    .detail( "House contain child component")
    .add(Block())
    .add(Window())
    .add(Door())
    .build()

  val myInvoiceCreator =  InvoiceBuilder.create()
    .name("Parent")
    .detail("This is Parent Component")
    .add(Block())
    .add(Block("Block432", 8.34, BlockType.Hollow))
    .add(Window())
    .add(Door())
    .add(houseComponent)
    .build()
  println(myInvoiceCreator.toString)
```

Figure 26

Next, we want our invoice creator to have additional feature to handle discount on final bill. The requirement is that we need to update discount dynamically, and the actual implementation of the InvoiceComposite should not change. Discounts are of different type, like festive season discount and if the product is bought Online then additional 5% discount percent must be applied. The percent of discount can also vary.

To solve this, we could have taken discount percentage as input, and check whether the product is bought online or offline. But this would require to change our composite class, which is not an ideal solution. Hence, we will apply Decorator Pattern here.

Decorator *is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors [21].*

To implement decorator patter, we will create an abstract class *Decorator* which extends Composite interface. Our Concrete *InvoiceComposite* now extends two interfaces, first is *Component* and second is *Composite*. Then we will extend the Decorator class to create two sub-child *FestiveDiscountDecorator* and *OnlineRetailDiscountDecorator.* Both this class override the *toString()* method to apply discount. Both this decorator takes Component as input and wrap it to generate a final bill with the discount.

After applying decorator pattern our code looks like *Figure27*

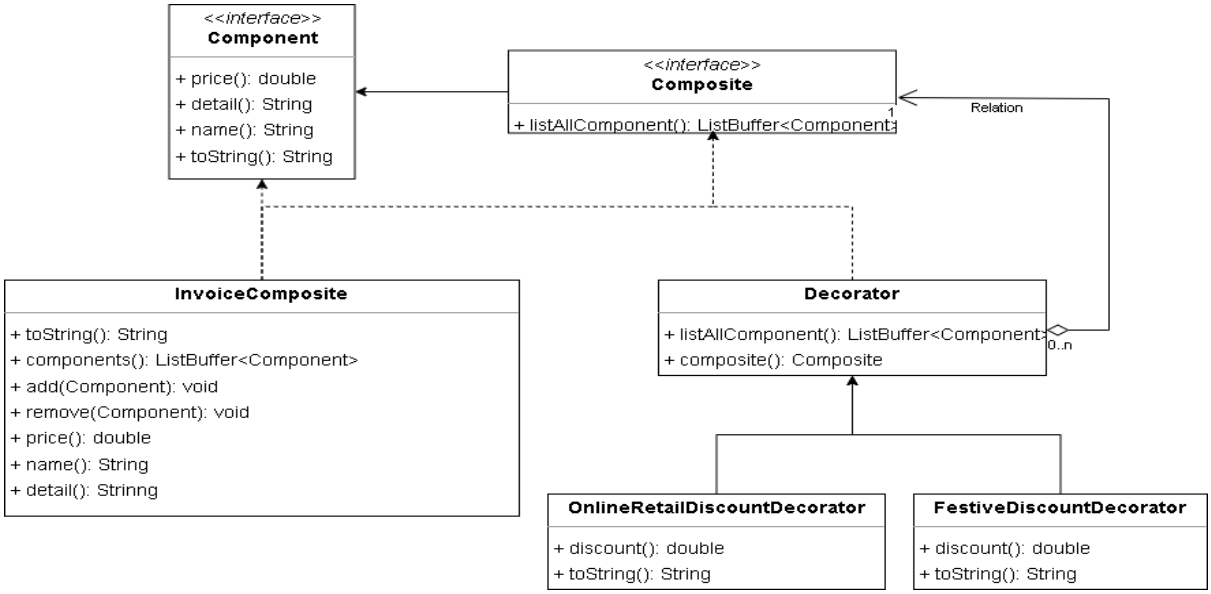


Figure 27

Our code snippet for the implementation of decorator looks like Figure28.

Now, we can add as many types of discounts we want to apply on total bill, without touching the original implementation of *InvoiceComposite* class.

```
trait Composite:
  def toString: String
  def listAllComponent():ListBuffer[Component]

case class InvoiceComposite(name: String,
                    detail: String) extends Component with Composite{..


abstract class Decorator(val composite: Composite) extends Composite {
  def toString: String
  override def listAllComponent(): ListBuffer[Component] = composite.listAllComponent()
}

class FestiveDiscountDecorator(composite: Composite, val discount: Double) extends
Decorator(composite) {
  override def toString: String =
    val totalPrice =listAllComponent().map(_.price).sum
    s"\nTOTAL PRICE:  €${totalPrice}, Final Price after applying Discount of ${discount*100}%
: €${Math.round((totalPrice* (1 - discount))*100.0)/100.0} "
}

class OnlineRetailDiscountDecorator(composite: Composite, val discount: Double) extends
Decorator(composite) {
  override def toString: String =
    val totalPrice =listAllComponent().map(_.price).sum
    s"\nTOTAL PRICE:  €${totalPrice}, Final Price after applying Discount of ${discount*100}%
+ Online Store Discount 5%: €${Math.round((totalPrice* (1 - (discount+0.05)))*100.0)/100.0} "
}

@main def main(): Unit =

  val myInvoiceCreator = InvoiceBuilder.create()
    ..
    .build()

    val decoratedComposite = new FestiveDiscountDecorator(myInvoiceCreator, 0.1)
    println(decoratedComposite.toString())

    var online = true
    if(online) {
      val decoratedComposite = new OnlineRetailDiscountDecorator(myInvoiceCreator, 0.1)
      println(decoratedComposite.toString())
    }
```

Figure 28

One problem with above approach is that we need to pass the ListBuffer[Component] which contains list of all components to decorator class to iterate and get the total price. If for some reason the data structure changes to something else, we would need to change our Decorator class to accommodate such change. Also passing the data structure itself to Decorator class is not a good idea, as it can easily modify the elements in the data structure. All we need in Decorator is to iterate over these elements and operate on them and not modify the list.

To solve this problem, we will implement **Iterator Pattern**.

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.) [22].

We want our **Iterator Pattern** to iterate on all objects of *InvoiceComposite*. To implement this, we will create *InvoiceCompositeIterator* which extends scala.collect.Iterator[E]. Then, we will override *hasNext()* and *next()* methods. *hasNext()* will check if there are more elements in collection to iterate over and *next()* will fetch the element form collection. Now we will modify our Composite interface to have *iterator()* method which will return the Iterator and remove *listAllComponent()* method.

The diagram for above implementation looks like *Figure29.*

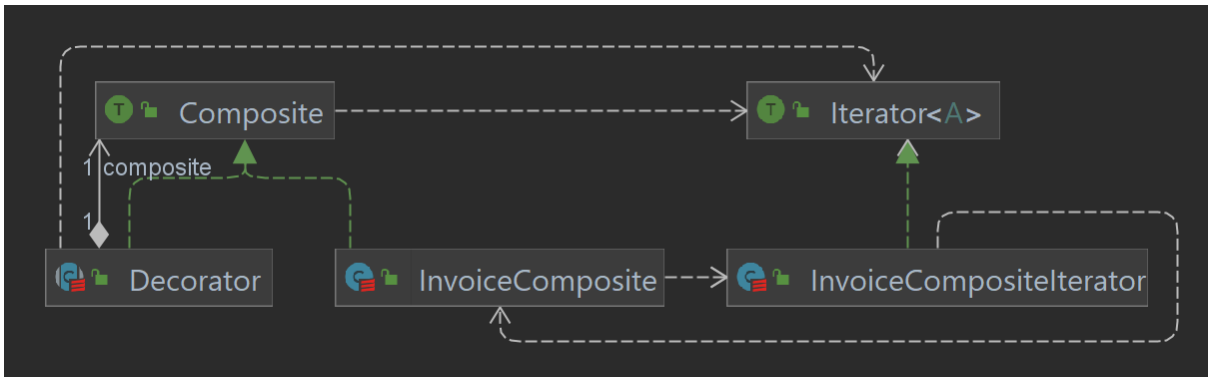


Figure 29

Now, we don't have to expose the underlying representation of collection in *InvoiceComposite.*

Our code snippet for above implementation looks like *Figure30.*

```
class InvoiceCompositeIterator(invoiceComposite: InvoiceComposite) extends
Iterator[Component] {
  private var current = 0

  override def hasNext: Boolean = current <
invoiceComposite.components.length

  override def next(): Component = {
    val result = invoiceComposite.components(current)
    current += 1
    result
  }
}

abstract class Decorator(val composite: Composite) extends Composite {
  def toString: String
  override def iterator(): Iterator[Component] = composite.iterator()
}

class FestiveDiscountDecorator(composite: Composite, val discount: Double)
extends Decorator(composite) {

  override def toString: String =
    var totalPrice:Double =0.0
    val iterator = composite.iterator()
    while (iterator.hasNext) {
      totalPrice= totalPrice + iterator.next().price
    }
    s"\nTOTAL PRICE:  €${totalPrice}, Final Price after applying Discount
of ${discount*100}% : €${Math.round((totalPrice* (1 -
discount))*100.0)/100.0} "
}
```

Figure 30

Next Problem, with our current discount model is that our client (*here main*) is creating what kind of discount it has to apply. But the ideal approach should have been that our client should only add the items in the invoice. Discount should automatically get applied depending on the kind of store the invoice is generated for. That means if invoice is being generated for offline store than only Festive discount should get applied, and if its online store, then Online retail discount should get applied automatically.

We will solve this problem using Observer pattern.

Observer will observe if the client is in *RetailShop* or *ECommerceStore,* and then *notify* the *StoreTypeAuditor* to apply the discount accordingly. To implement this, we will create two interface *Observer* and *Observable*. *Observable* has three main functions, to *attach* and *detach* the *Observer* and then notify all the *Observer. Observer* will be extended by concrete *StoreTypeAuditor* class and on notification on type of store, it will create the corresponding discount decorator class. This implementation could be best understood from *Figure31.*

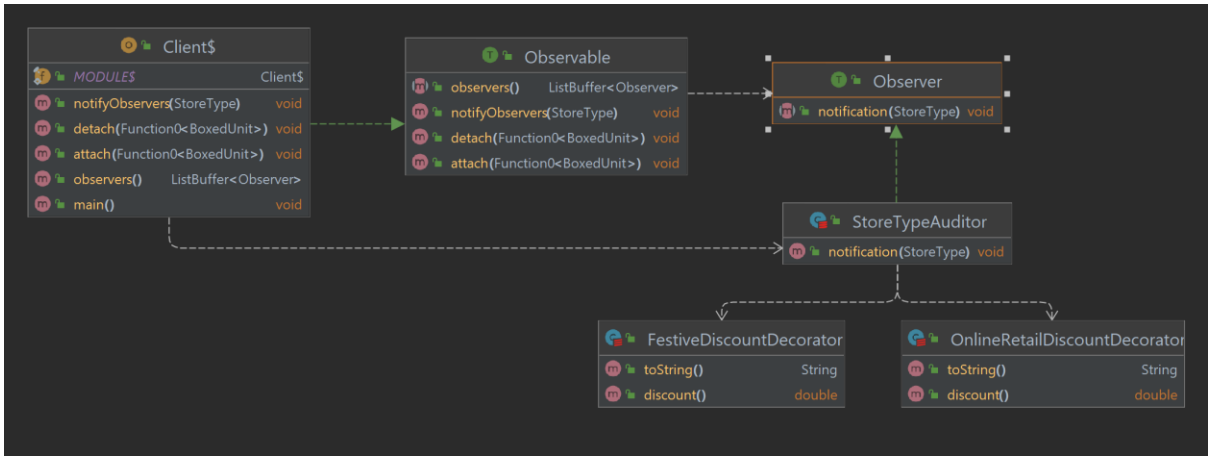


Figure 31

The code snippet for above implementation can be found in *Figure32*.

```
trait Observable {
  val observers = ListBuffer[Observer]()
  def attach(addObserver: () => Unit): Unit = addObserver.apply()
  def notifyObservers(subject : StoreType): Unit =
    observers.foreach(_.notification(subject))
  def detach(deleteObserver: () => Unit): Unit =
    deleteObserver.apply()
}

trait Observer {
  def notification(subject : StoreType): Unit
}

class StoreTypeAuditor(composite: Composite) extends Observer{
  override def notification(subject: StoreType): Unit =
    val decoratedComposite = (subject: @switch) match {
      case StoreType.RetailShop => new FestiveDiscountDecorator(composite, 0.1)
      case StoreType.ECommerceStore => new OnlineRetailDiscountDecorator(composite, 0.1)
      case _ => "Incorrect Store Type"
    }
    println(decoratedComposite.toString())
}
object Client extends  Observable {
  @main def main(): Unit =
    val myInvoiceCreator = InvoiceBuilder.create()
      ..//add components here
      .build()

    val storeTypeAuditor = StoreTypeAuditor(myInvoiceCreator)
    attach(() => observers.addOne(storeTypeAuditor))
    notifyObservers(StoreType.ECommerceStore)
    detach(() => observers.clear())
}
```

Figure 32

Everything looks good with our invoice, except the fact that our price for each object is constant. It can only be set while creating the component. It makes sense to set the price when components are created but what if we want to control this price afterwards. This could be because of many scenarios, like cost of transportation has increased and hence the profit margin is reduced. To keep the profit margin same, we would now need to increase the price, even though the components are already created.

To solve this problem, we would need some kind of remote controller to increase or decrease the price dynamically. We must also make sure that we don't want to modify too much of our existing code.

Hence, we can solve this problem using **Bridge Pattern**.

Bridge *is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other* [23].

Ok, the definition is scary, but the actual implementation is very simple. All we need is to create is create *PriceController* class which provided two method *priceUp()* and *priceDown(),* which accepts component and the amount by which we want to change the price. In the *Component* interface we have added two methods which allow *PriceController* to change the price of already created components. Our price is no more constant. It can be dynamically changed using controller. This creates a bridge between the component and the Price modifier hence it is called Bridge pattern. This can be best understood by *Figure33.*

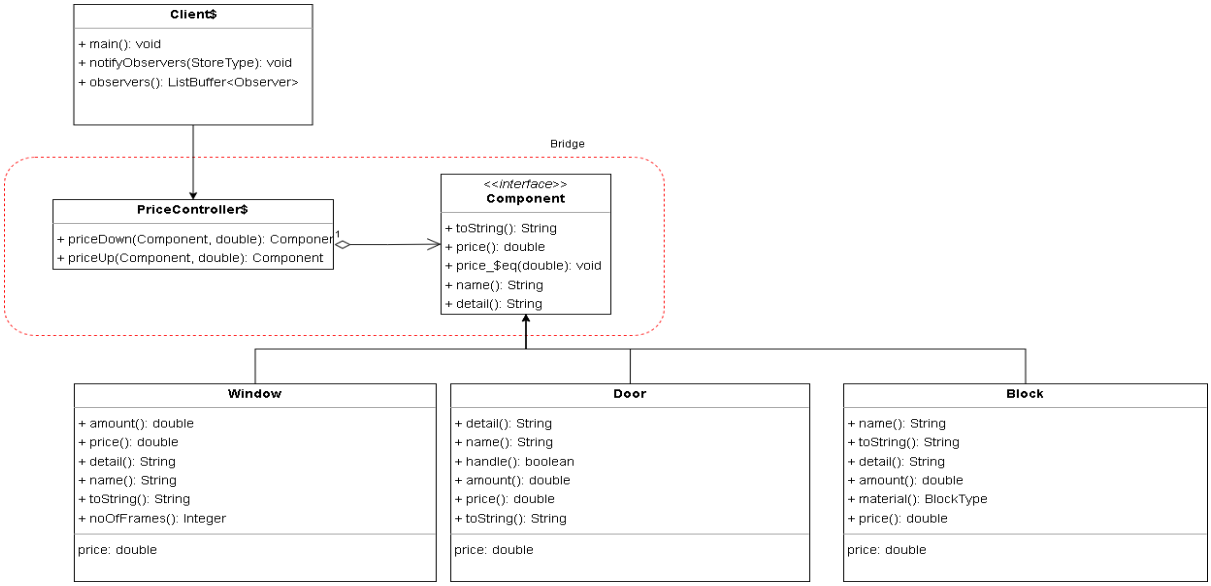


Figure 33

The code for above implementation is in *Figure34.*

```
trait Component:
  var price: Double
  def name: String
  def detail: String
  override def toString = s"${name} $detail €${price}"
  def getPrice():Double = price
  def setPrice(amount: Double):Unit= price = amount

object PriceController:
  def priceUp(component: Component,increaseByAmount: Double): Component =
   component.setPrice(component.getPrice()+increaseByAmount)
   component
  def priceDown(component: Component,decrementByAmount:Double): Component =
   component.setPrice(component.getPrice()-decrementByAmount)
   component


object Client extends  Observable {
  @main def main(): Unit =
    val houseComponent = InvoiceBuilder.create()
      .name("House")
      .detail("House contain child invoiceComposite")
      .add(PriceController.priceUp(Block(),2.0))
      .add(Window())
      .add(PriceController.priceDown(Door(),.49))
      .build()
```

Figure 34

8.2 Reflection

We started with analysis of the problem statement and then went around implementing multiple pattern which best served the business needs.

Composite pattern allowed us to work with complex tree structures more conveniently. It allowed us to add new components without much modification thus adhering to Open/Closed principle.

Builder pattern provided us the mechanism to construct object step-by -step and recursively run those steps. We can use same construction code when building various representation of component. Thus adhering to Single Responsibility principle. In real time StringBuilder class provided by java implements Builder pattern. This enhances our readability and avoid creating multiple String object in java, as they are immutable, because it allows us to concurrently modify the object.

Singleton pattern allow us to have only one object of particular type. This is important when we are dealing with heavy resources and we don't want to overload the system.

Decorator pattern helps us in extending the object's behavior without modifying the current implementation or creating new sub-classes. We can add/remove the behavior at runtime.

A decorator makes it possible to add or alter behavior of an interface at run-time. Alternatively, the Adapter pattern can be used when the wrapper must respect a particular interface and must support polymorphic behavior, and use Façade pattern when an easier or simpler interface to an underlying object is desired. Decorator and Proxy have similar structures, but very different intents. Both patterns are built on the composition principle, where one object is supposed to delegate some of the work to another. The difference is that a Proxy usually manages the life cycle of its service object on its own, whereas the composition of Decorators is always controlled by the client.

Iterator pattern prevents exposing of underlying representation of data structure. It also gives us the flexibility to write our own traversal logic, which might be useful when we are traversing a tree like structure. Hence, we can use the iterator pattern to traverse Composite trees. Iterator is helpful in implementing new types of collection and provide easy iteration over them. Although, we must be careful that our iteration logic doesn't take more time than the one provided by the standard Collection.

Observer pattern, as name suggest allow us to introduce new subscriber classes without having to modify the publisher code. We could use observer pattern to solve many businesses use cases. Like, assume we wanted to observe what product a user has added to the cart and suggest them at checkout to buy the frequent bought together goods or notify customer if the product is still in stock. It could also be used to notify shoppers about price drop. All we need is to add more Auditors/subscribers and call notify method. Observer pattern allow us to modify the relations between object at runtime.

Bridge pattern helps us to bridge between two independent classes so that we don't end up creating multiple combination of each relation. Bridge pattern allow us to switch implementation at runtime. We must be careful with what remote classes could modify. It should not end up changing the entire state of class.

References

[1] COSC427 wiki., "Riel's heuristics," [Online]. Available: https://oowisdom.csse.canterbury.ac.nz/index.php/Riel%27s_heuristics.

[2] T. Janssen, "SOLID Design Principles Explained," [Online]. Available: https://stackify.com/solid-design-principles/.

[3] COSC427 wiki., "Minimize number of methods," [Online]. Available: https://oowisdom.csse.canterbury.ac.nz/index.php/Minimize_number_of_methods.

[4] Refactoring.Guru. , "Template Method," Refactoring.Guru. , [Online]. Available: https://refactoring.guru/design-patterns/template-method.

[5] Refactoring.Guru., "Strategy," Refactoring.Guru., [Online]. Available: https://refactoring.guru/design-patterns/strategy.

[6] Scala Docs, "Scala Book - Abstract class," [Online]. Available: https://docs.scala-lang.org/scala3/book/domain-modeling-tools.html#abstract-classes.

[7] refactoring.guru, "Observer," [Online]. Available: https://refactoring.guru/design-patterns/observer.

[8] J. Chung, "Pull vs. Push," [Online]. Available: https://medium.com/@jchung722/pull-vs-push-b4788a845cce.

[9] stackoverflow.com, "What is the apply function in Scala?," [Online]. Available: https://stackoverflow.com/a/9738862.

[10] A. J., "Scala: How to add, update, and remove elements with a mutable Map," [Online]. Available: https://alvinalexander.com/scala/how-to-add-update-remove-mutable-map-elements-scala-cookbook/.

[11] https://www.baeldung.com, "@switch Annotation in Scala," [Online]. Available: https://www.baeldung.com/scala/switch-annotation.

[12] https://refactoring.guru, "Factory Method," [Online]. Available: https://refactoring.guru/design-patterns/factory-method.

[13] Digital Guide IONOS, "Factory pattern: the key information on the factory method pattern," [Online]. Available: https://www.ionos.com/digitalguide/websites/web-development/what-is-a-

factory-method-pattern/.

[14] GeekForGeeks, “Abstract Factory Pattern,” [Online]. Available: https://www.geeksforgeeks.org/abstract-factory-pattern/.

[15] https://refactoring.guru/, “Singleton,” [Online]. Available: https://refactoring.guru/design-patterns/singleton.

[16] https://www.baeldung.com/, “Creating Singletons in Scala,” [Online]. Available: https://www.baeldung.com/scala/creating-singletons.

[17] https://refactoring.guru/, “State,” [Online]. Available: https://refactoring.guru/design-patterns/state.

[18] https://refactoring.guru, “Visitor,” [Online]. Available: https://refactoring.guru/design-patterns/visitor.

[19] https://refactoring.guru/, “Composite,” [Online]. Available: https://refactoring.guru/design-patterns/composite.

[20] https://refactoring.guru/, “Builder,” [Online]. Available: https://refactoring.guru/design-patterns/builder.

[21] https://refactoring.guru/, “Decorator,” [Online]. Available: https://refactoring.guru/design-patterns/decorator.

[22] https://refactoring.guru/, “Iterator,” [Online]. Available: https://refactoring.guru/design-patterns/iterator.

[23] https://refactoring.guru, “Bridge,” [Online]. Available: https://refactoring.guru/design-patterns/bridge.