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

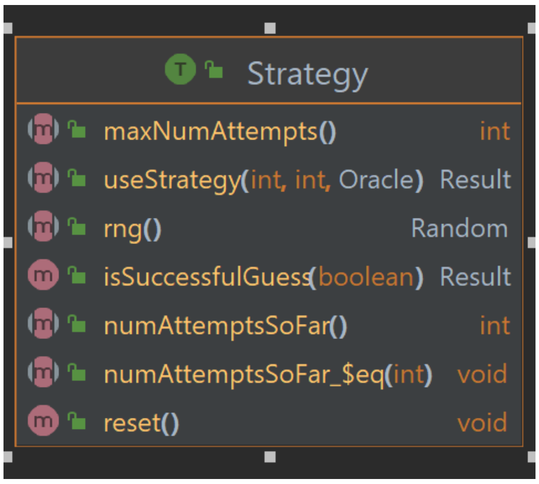
In the code walkthrough, I tried to understand the flow of code. To understand what sorts of method are available and how they are inter-linked. I also tried to identify if there is an existing design pattern in place. Mostly, it was associated as single responsibility principle, as far as I can tell.

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.

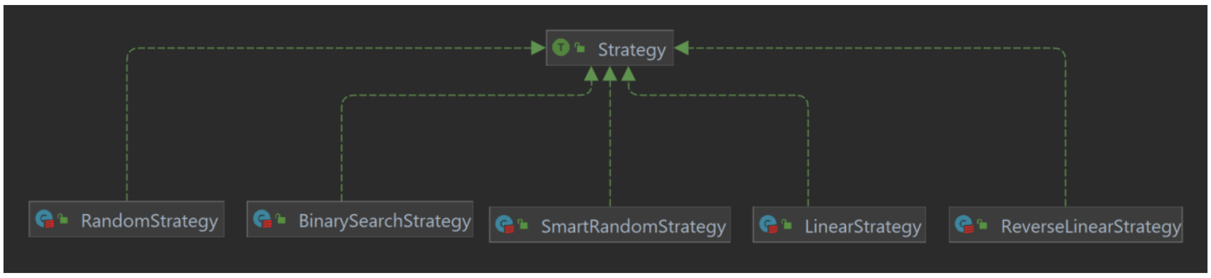
Now, I will discuss the steps I took to redesign the code and reasons behind it.

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.

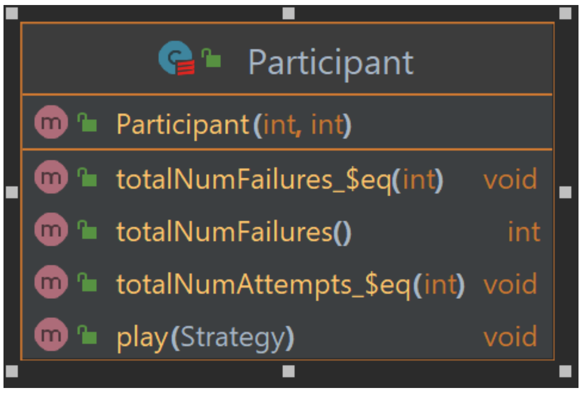


```
Strategy
maxNumAttempts()                      int
useStrategy(int, int, Oracle)      Result
rng()                              Random
isSuccessfulGuess(boolean)         Result
numAttemptsSoFar()                    int
numAttemptsSoFar_$eq(int)            void
reset()                              void
```

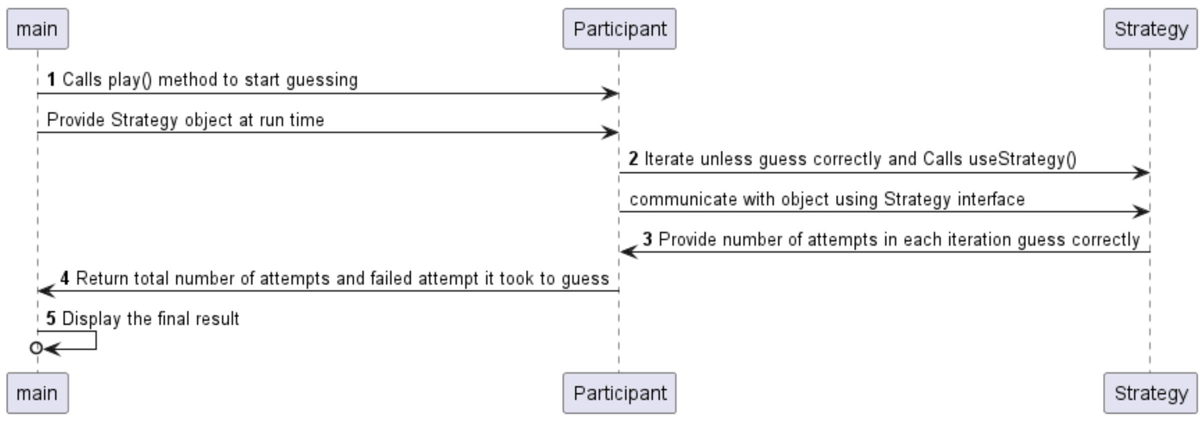
2. The Strategy interface is implemented(extended) by five class as depicted in below diagram.



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.


main
Participant
Strategy
1 Calls play() method to start guessing
Provide Strategy object at run time
2 Iterate unless guess correctly and Calls useStrategy()
communicate with object using Strategy interface
3 Provide number of attempts in each iteration guess correctly
4 Return total number of attempts and failed attempt it took to guess
5 Display the final result

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

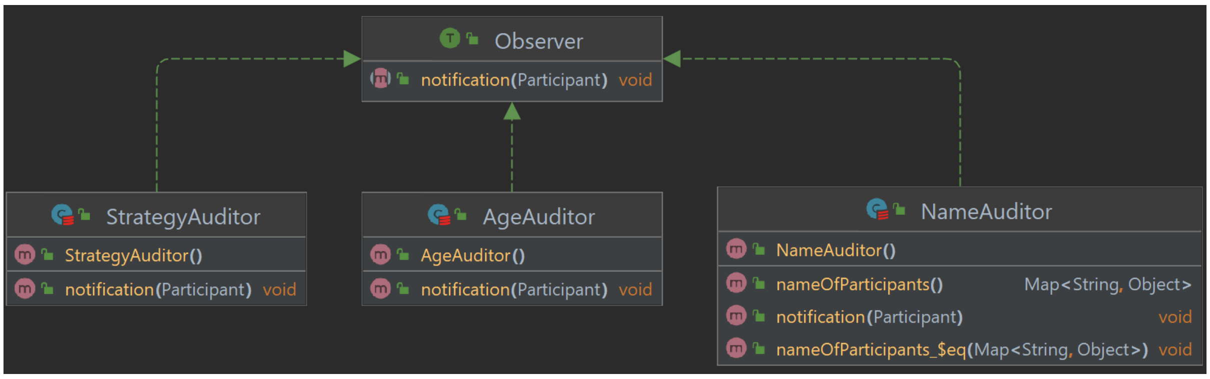
The first step that I decided was to analyse who can be the subject ("the one which is to be observed") and who will be the observer. In our case it was obvious that multiple Auditor extend Observer, but tricky part was which class can extend Observable. Initially, I forced Participant class to extend 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.

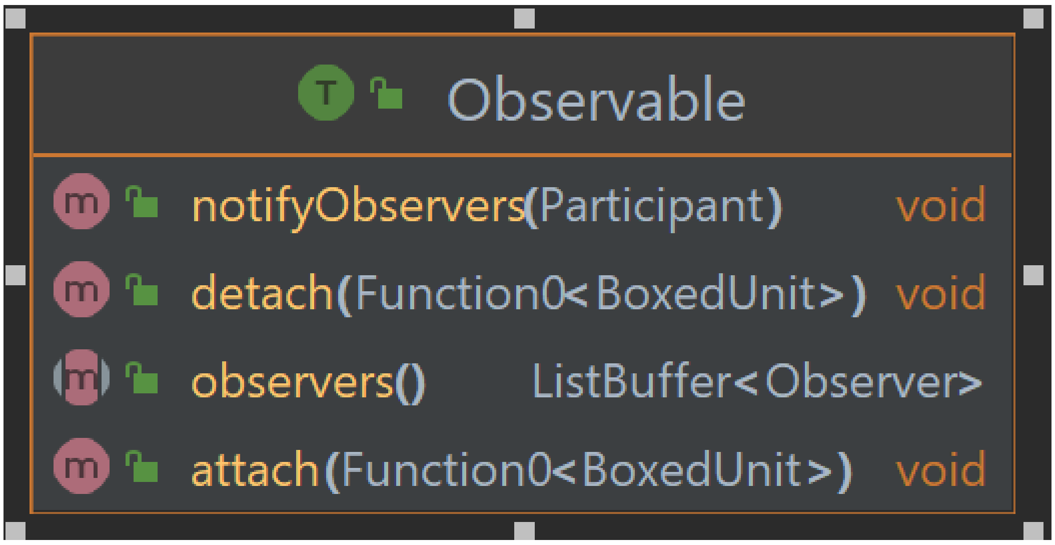

Observer
notification(Participant) void
StrategyAuditor
StrategyAuditor()
notification(Participant) void
AgeAuditor
AgeAuditor()
notification(Participant) void
NameAuditor
NameAuditor()
nameOfParticipants() Map<String, Object>
notification(Participant) void
nameOfParticipants_$eq(Map<String, Object>) void

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
notifyObservers(Participant) void
detach(Function0<BoxedUnit>) void
observers() ListBuffer<Observer>
attach(Function0<BoxedUnit>) void

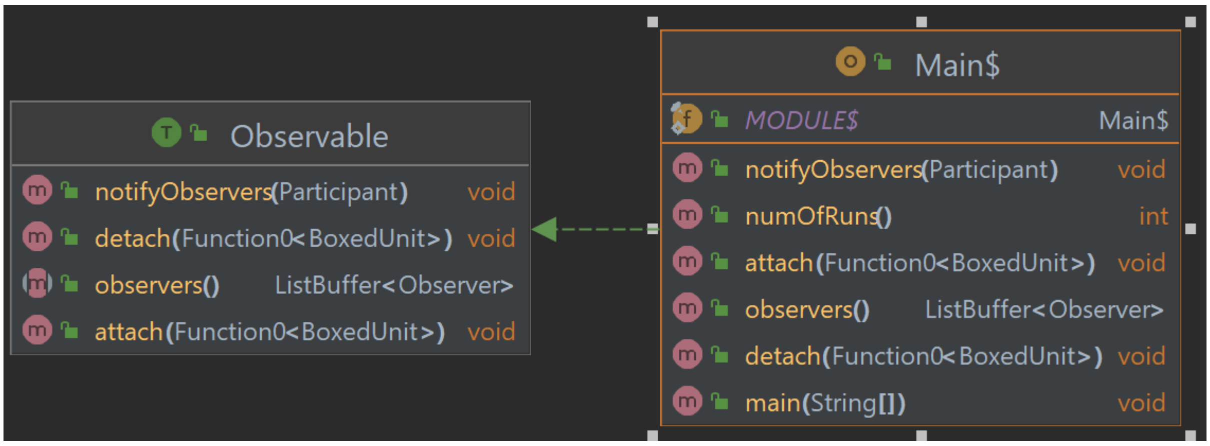
Observable, has one mutable List<Observer>, where it stores the current list of observers which are observing the subject. Observable notifies each observer using this list. It provides three functions for this:

- 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.

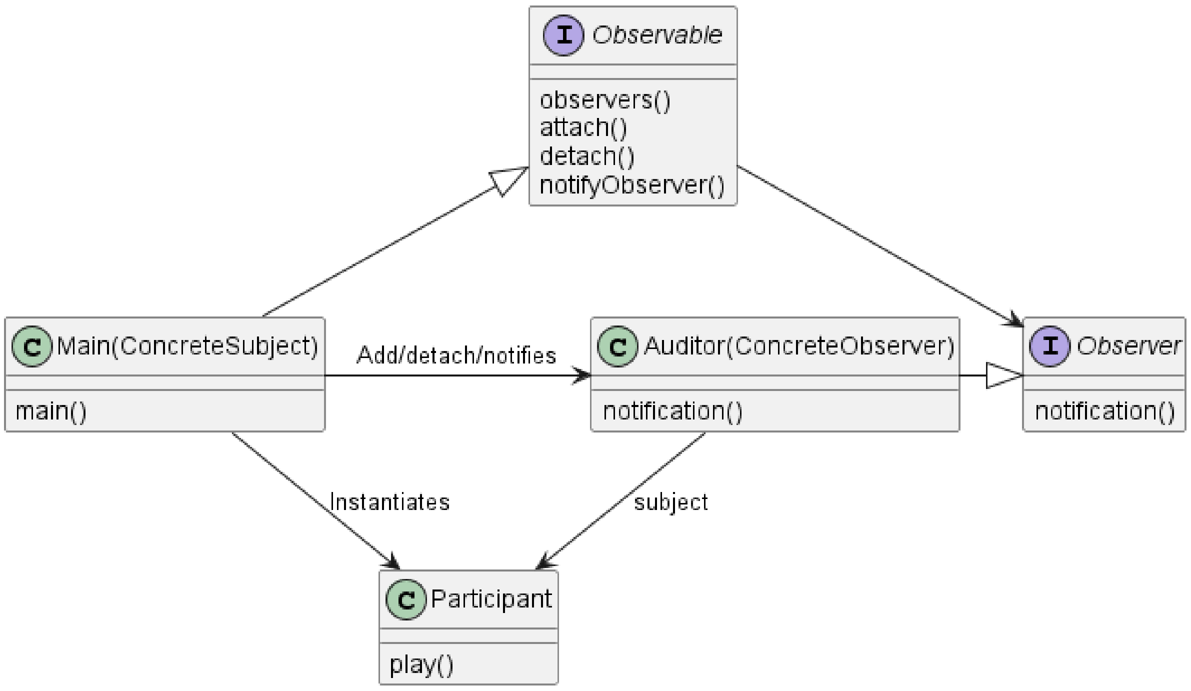
Our code snippet for Observable looks like this:

```
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")
}
```

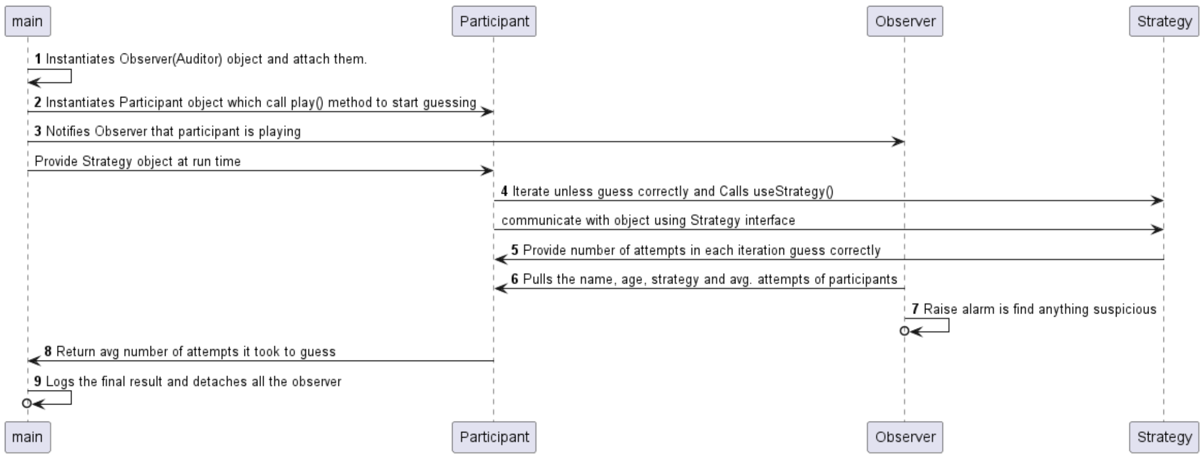
Now, all that is left is to extend this Observable trait. Hence, our client, here Main, extend Observable, and add all the three auditors mentioned earlier, and notifies them, once a participant object is created and has played the game. It is left to the observer to pull the data from subject. Once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


Observable
notifyObservers(Participant) void
detach(Function0<BoxedUnit>) void
observers() ListBuffer<Observer>
attach(Function0<BoxedUnit>) void
Main$
MODULE$ Main$
notifyObservers(Participant) void
numOfRuns() int
attach(Function0<BoxedUnit>) void
observers() ListBuffer<Observer>
detach(Function0<BoxedUnit>) void
main(String[]) void

Class diagram:



Sequence diagram:


main
Participant
Observer
Strategy
1 Instantiates Observer(Auditor) object and attach them.
2 Instantiates Participant object which call play() method to start guessing
3 Notifies Observer that participant is playing
Provide Strategy object at run time
4 Iterate unless guess correctly and Calls useStrategy()
communicate with object using Strategy interface
5 Provide number of attempts in each iteration guess correctly
6 Pulls the name, age, strategy and avg. attempts of participants
7 Raise alarm is find anything suspicious
8 Return avg number of attempts it took to guess
9 Logs the final result and detaches all the observer
main
Participant
Observer
Strategy

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.