COMP40070 Design Patterns

Lab Journal

# Day 1 – 5th December 2016 : Template and Strategy Pattern

## Theory

The Strategy pattern is useful in scenarios where the actor or the doer doesn’t change; the verb/action doesn’t change but the how of the action changes. If one were to model a dancer as a class object, then dance would be one of the methods and dancing in jazz, hip-hop, ballet etc would be different strategies by which the dancer could dance. In software terminology, this dance format can be changed at runtime i.e. once the dancer is on stage, it can dance in jazz style and if the audience wants hip-hop, it can start dancing in hip-hop style. The Template pattern is useful in scenarios where different actions share a common algorithm. The details of the algorithm implementation is however shielded from the superclass.

## Practical explanation – Participant class

The Participant class has 4 play methods corresponding to 4 different strategies. This is a very obvious indication for using Strategy pattern. So a class called PlayHow is created. All the four strategy classes inherit from this. On observation it is also noted, that all the strategies follow a common procedure and only the implementation details of the steps in the procedure vary. This commonality is very clean and the differences clearly distinct. This is the appropriate place to use Template pattern as well. So the class PlayHow will now contain the template method for execution of any strategy method and the subclasses will define the concrete details required. Initially I defined the template method to follow this procedure : i) Generate number ii) Increase number of attempts by participant iii) Check if run is over? (Either when number is correct or max number of attempts reached) iv) Print outcome of run when run is over. I know that the first two lines in every strategy method generate a seed number and increase attempt. I was trying to think of a way to avoid same method call twice. As I wasn’t able to I revised the procedure to now be :i) Generate seed number ii) Increase attempt iii) As long as run is not over do step 1 and 2 iv) Print outcome of run. I defined the functions increase\_attempt, run\_over? and find\_outcome in the PlayHow class. All there are intended to be final methods. I defined the generate\_seed\_number() and generate\_number() for RandomPlay and LinearPlay class. However, I realized for smart random play and binary search, an additional step of manipulating values of upper and lower is required, but this step was same for smart random and binary. So I created another class SmartPlay extending from PlayHow. I added an additional function before\_next\_iteration to the PlayHow class and implemented it in the SmartPlay class. The strategies SmartRandomPlay and BinarySearchPlay extended from SmartPlay. However, later on retrospect I realized as there is just one point of differentiation between PlayHow and SmartPlay, using inheritance is probably an overkill and if the before\_next\_iteration changed for one it would affect both. So I scrapped it , made SmartRandomPlay and BinarySearchPlay extend PlayHow and added the identical method of ‘before\_next\_iteration’ to both these classes. I then added a play\_method attribute to the Participant class. In each of the functions play\_randomly, play\_linear etc I assigned play\_method to the instance of the respective class and passed it the parameters self, upper and lower. Since the play method needed to change the value of no\_of\_attempts attribute, I had to remove it from attr\_reader and add it to attr\_accessor. I also added another strategy for playing ReverseLinearSearch that starts from the upper bound and decrements one by one until it finds success. This class extends PlayHow too and has only two methods implemented generate\_seed\_number and generate\_number.The final template method and relationship between classes can be seen in the following figure.

### Template Method : play

1. Generate seed number *#Abstract*

2. Increment participant attempts *#Final*

3. While step 4 returns false do step 5, 6, and 7. When step 4 returns true move to step 8

4. Check if run is over *#Final*

5. Make any adjustments before next iteration *#Hook*

6. Generate number *#Abstract*

7. Increment attempts *#Final*

8. Find outcome *#Final*



## Practical Explanation : Evaluation class and main.rb

After completing this part , I started making changes to the main.rb file. There is a lot of code that has been repeated and even without using patterns, the code could have been made shorter and cleaner by using case statement. I thought of defining a method that takes strategy type as argument and calls the relevant function. This method could then be used in conjunction with the case statement. The code was still very tedious. As I thought of ways around it, I realized that there was potential for applying a combination of Strategy + Template pattern there as well. If you consider a main class representing our Game for e.g. GuessGame then there are different evaluation strategies. Also there is a clear procedure followed by each strategy. I want to point out that in this case since the actual implementation is very trivial this design pattern here might seem like an overkill, however, it holds for the concept for evaluating the performance of different playing strategies. So I created a GuessGame class which has a method score\_performance and an attribute strategy\_method. The attribute strategy\_method is changed at runtime to instances of subclasses of Evaluation and the method score\_performance calls the evaluate method on it to see the performance of the different playing strategies . Also the older approach was creating multiple instances of Participant class to test performance of different strategies which isn’t right. This strategy creates only one instance of Participant class. The superclass Evaluation has an instance of participant class as an attribute which is created when any instance of Evaluation or its subclass is created.. All the variables like ‘NO\_OF\_RUNS’ and total\_no\_attempts etc are moved to the Evaluation class it’s natural home. So there is no need to pass any data or context from GuessGame class to instances of Evaluation or its subclasses. The commonality is captured in the template method ‘evaluate’

### Template method: Evaluate

1. Reset no of attempts for participant *#Final. As we are using same participant object for all strategies*

2. Set Max attempts for participant *#Abstract. I thought of making this a hook method, as max attempts for three of the strategies is NUM\_OF\_RUNS \* 2 and different for the other two. But I decided against it, because I couldn’t think of a logic for which this should be the default logic.*

3. Repeat step 4, 5, 6 for NUM\_OF\_RUNS times

4. Assign a secret number *#Final*

5. Reset no of attempts for participant *#Final*

6. Participant plays *#Abstract*

7. If outcome of 6 is success, do step 8, else do step 9

8. Increment attempts *#Final.* Move to step 10

9. Increment failures *#Final*.

10. Calculate average no of attempts *#Final*

11. Display *#Hook. I initially made this abstract as the output displays the name of the method. But then I thought of the display method to be analogous to the toString() method in JAVA. It allows you to override but there is a default implementation. So I made it a hook method.*



In both cases, it makes sense to apply the Template pattern first and then the Strategy pattern as that removes a lot of duplicated code.

Only the developer of the template pattern knows the most crucial hook methods to other developers. These should be listed in the documentation for the API.

# Day 2 – 6th December 2016 : Observer Pattern

# Theory

Observer pattern is used where one class encapsulates a state. When the state changes you want to notify all the classes that are dependent on this state

## Practical Explanation

I decided to address the problem in the following particular order and managed to get it working.

1. Get it working for one participant and one observer(auditor) with push model
2. Get it working for one participant and one observer(auditor) with pull model
3. Get it working for multiple participants and one observer(auditor) with pull model
4. Get it working for multiple participants and multiple observers(police , auditor) with pull model
5. Add both built-in and custom observable module to the participant class
6. Attach one of the observer using the built-in module and one using the custom observable module and see how it affected the system
7. Get the custom observable module working with a proc
8. Create a simulation with a few participants and run the process 100 times.
9. Post the simulation, do some analyses on the data stored by the three observers (auditor , police and proc)

After adding name, gender , age and strategy to the Participant class, I got rid of the various play\_xxx methods. The single play method now based on the value of the strategy method, calls the appropriate play method. I thought to use symbols as values of available strategies and declared them as a constant array in Strategies module. I created the Auditor class with the pull model, added an update function that would take outcome of the run and no of attempts as inputs. If it was success and no of attempts were less than 3 it would print “CHEAT”.To test this, I added another playing strategy cheat that knows the answer and tells it in 1-3 steps. After this, to get the pull model, I passed the self object to the update method. Then I thought that the update method should support both pull and push. This is because there will be a thousand observers but a single observable and it’s the observable’s discretion whether to use pull and push. So using a check I modified my update method to support both pull and push. To now test this with multiple participants I needed to create dummy data. I created 6 participants with the help of integer.times and arrays, each one of which plays using one of the 6 strategies. To add multiple auditors I initially implemented a RegularAuditor and a SeniorAuditor but I could find much value for them, so I implemented a RegularAuditor and PoliceOfficer class, two auditors with totally different functions. However, both of them needed to check if a particular run by a participant was to be considered cheating or not. This seemed like a right use of module to me so I created a module Supervisory with the method to check cheating and included it in both RegularAuditor and PoliceOfficer class. The RegularAuditor’s update method checks for cheating and then stores the strategy used for cheating. This seemed like a good metric to analyse on. The PoliceOfficer’s class checks for cheating and stores the participant object in a hashmap of offenders with their offence count. Again this record can be used for analyzing multiple runs and multiple participants’ performance. Right about now, it looked to me that an entire simulation process was coming together. So I created a Simulation class with the methods prepare\_data, simulate, analyse\_by\_strategy and analyse\_by\_age which would make all the participants play multiple times and generate statistics on the information collected by both the observers. At this point, I didn’t think it was a good idea to simply delete all of my old code and replace it with a new custom observable module. So I thought to explore what would happen if I were to use both the original and custom observable module. I did that and just changed the method names slightly in the custom observable module. I let the RegularAuditor attach with the built-in observable and attached PoliceOfficer with the custom observable and it worked. This means that if multiple observers for a given object can be grouped into different sets, then the given object can include observable modules for each of these sets, which will have different implementations of notify. I created a Proc object and passed it to my custom observable module’s attach\_method. That worked as long as I had a check for @observer.class == Proc in my notify method. However, I couldn’t pass a simple block to the same method, as it required a different argument type. However, it’s never a good idea for an interface to support only entities of one type. So I added separate methods for adding and removing block code. I was also curious to check the true closure property of Ruby as we were discussing in class today. So I created a block that would simply update the gender count for each run. Then in the final step of simulation, I used these gender counts to display stats of gender breakdown and it worked.