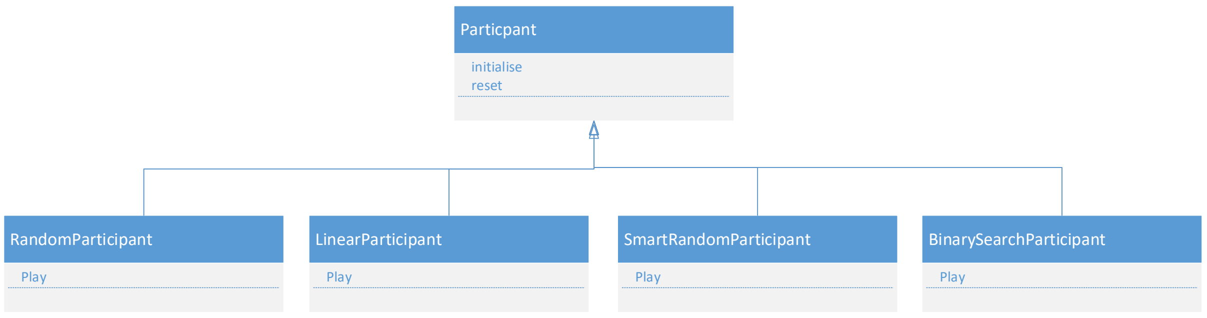
Template Method Pattern and Strategy Pattern – Day 1

Initially I found this quite difficult. This was probably a lot to do with the fact that I was rusty with code and there was a reasonable bit of detail here from my perspective. The code initially looked quite messy and I needed to step through it a bit to get the “gist” of what was going on.

Reasonably quickly I could see the repeated code in the Participant.rb file and the main.rb file also. There were slight differences in each implementation but lots with lots of common code also. From a conceptual level I could see that each “mini-game” looked like a candidate for the strategy pattern.

As the exercise called for attention in the participant class I focused my initial attention there and discussed with my partner. We quickly identified the approach to apply the strategy pattern to the participant class in the following manner



After completing this I struggled a little trying to think about how to use the Template Method Pattern to reduce the amount of repeated code.

This was easier for me to do in the main.rb class. I created a method run_game and passed the strategy as a parameter. The method would then instantiate the appropriate participant class and run that objects play method to run the exercise. This action really simplified the main.rb file making the code a lot more manageable.

The work in cleaning up this main class was very similar to the work in the Participant class in that the strategy was passed as a parameter and that common code was compressed into one method that all strategies would call. This would support the requirement of changing strategy at run time.

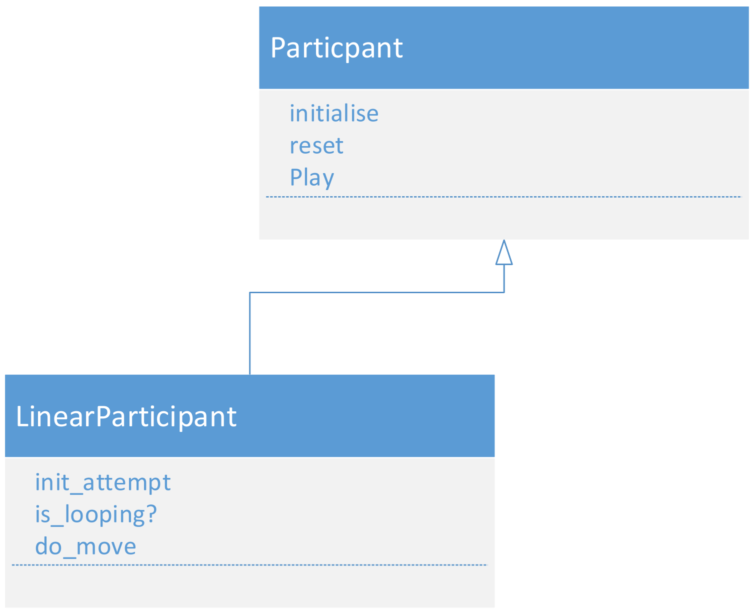
The last piece of work was to refactor the individual strategy implementations to use the Template Method patterns. I struggled a lot with this initially, as the implementations were spread over multiple files making it hard to me to spot stability and instability between them. I identified that the base structure was common

1. Initialise the starting state
2. Determine when to loop and when to break
3. Make a move
4. Return a success or failure.

All strategies performed these 4 steps but the specifics of 1..3 were different.

I refactored the play method to be the template, and each subclass would evaluate parts 1..3 specific to the type of game they were modelling.

This new structure looked as follows



All three methods are abstract methods but I chose not to throw an exception in the superclass following guidance in the morning.

I added a descending game also to extend the problem to more game types. This proved pretty easy to do compared to the other work in the exercise…

Overall in this exercise I think I was able to spot the need for each of the patterns from a quick look at the 2 files (main.rb and participant.rb) but found myself making them more complex by focusing on how to make the two work together in an appropriate manner.

The end result was code that certainly looked a lot better, would read much cleaner, and was easier to debug through but one consequence was that code was split over more locations, that caused the diffing to individual algorithms more complex.

I naturally identified the Strategy pattern first and this was the pattern that I kept seeing as a good help to the problem and as such I implemented that first. With hindsight I do wonder if leading with the Template Method Pattern would have gotten me to where I ended up in a faster manner.

Day 2: Observer Pattern

Today I started with the output from yesterday's exercise. I planned to do the work in the following order;

1. Use the Ruby built in Observable module for Auditor
2. Add a new Strategy (cheater) to simulate someone getting it right first time every time
3. Implement my own Observable module
4. Investigate using blocks and / or procs

Using the built in Ruby Observable module

This turned out to be reasonably straight forward. Due to the way I had factored the Template Method pattern from yesterday I had all the data that I needed in the Participant superclass, indicating that my application of the template yesterday was not as full as I thought. As opposed to factoring further I decided to concentrate on todays exercise so I added a register_result method that would notify the observers in the following fashion;

```
def register_result(result)
  @result=result
  changed
  notify_observers(self)
end
```

I then created an Auditor class that would get notified and used the pull method pulling the data from the self object passed above. To use the push method I would simply expect to add the result and number of guesses from the Subject, however I don't intend to try this unless I have time to spare at the end.

I then created a new object in the main client of type Auditor and used participant.add_observer to connect up the observer. I finally added a report_out method to the auditor to report its progress…

Running the main.rb gave me results that I would expect…

Adding the Cheater strategy

The strategy pattern from yesterday allowed this to be added pretty easily. I added a new method to oracle to allow me to steal the magic number and used that to guess right the first time. Alternatively I could have just changed the cheater strategy to pretend to be right even when wrong and this would have avoided adding the cheat method to the Oracle class, although this does not materially affect the purpose of the evercise.

Using the following code in main I was able to run the game with that strategy also…

```
when :play_cheater
  player = CheatingParticipant.new(oracle, max_num_attempts:NUM_OF_RUNS*5)
```

Again I ran the program and saw output that I would expect…

It also struck me that the code in main that created each game type looked a lot like a hardcoded or inline class factory pattern… In due course I will do some additional factoring here also.

```
case strategy
  when :play_random
    player = RandomParticipant.new(oracle, max_num_attempts: NUM_OF_RUNS*2)
  when :play_linear
    player = LinearParticipant.new(oracle, max_num_attempts:NUM_OF_RUNS*2)
  when :play_smart_random
    player = SmartRandomParticipant.new(oracle, max_num_attempts:NUM_OF_RUNS*5)
  when :play_binary_search
    player = BinarySearchParticipant.new(oracle, max_num_attempts:NUM_OF_RUNS*5)
  when :play_descending
    player = DescendingParticipant.new(oracle, max_num_attempts:NUM_OF_RUNS*2)
  when :play_cheater
    player = CheatingParticipant.new(oracle, max_num_attempts:NUM_OF_RUNS*5)
end
```

Writing my own Observable module

In general if there is capability provided by the language, run-time or framework I would use that for a huge number of reasons, so using the built in Observable module above would be my default approach but for the purposes of the exercise I next implemented my own Observable module.

This proved relatively trivial to implement. I added an array of observers and then added methods to add and observer, delete an observer an a notify_observers method that would iterate through the array and call the update method for each observer.

Whist this was intresting to see the underlying structure it did re-enforce my original choice that if the platform, language, framework or environment has implemented a particular pattern that this should be the port of first call, unless it is shown not to support the specific context that you need to try it in.

Investigate passing blocks or procs to the observer

First thing that came to mind as I started to think about this is that it would make the code simpler for very rudimentary observers, my auditor class already has functionality that would make the code block approach more complex…

I have added logic to my auditor to remember the number of guesses taken for each successful attempt. I plan to use this to check for patterns.

I added the code block to the observer to print out the results but using a class allowed me to do much more so I reverted the code back to use the Auditor class instead;

```
player.add_observer do |game_run|
  puts "Particpant result: #{game_run.result}"
  puts "Number of runs: #{game_run.num_attempts}"
end
```

I forgot to change the notify_observers to use proc.call as opposed to observer.update and this got me stuck for a while but when I resolved that it worked as expected.

Key Learnings from the lab today

The observer pattern is really powerful and easy to use by simply including Observable. There are lots of ways that it can be used influenced by the factoring and design of the rest of the application. My preference was to use the power of having a class that can be further enhanced to do deeper levels of auditing than the code block mechanism but for simple observers code blocks are also reasonable to use. As I mention a few time I do like using system provided functionality when possible so it is really nice that this pattern is supported natively in Ruby.