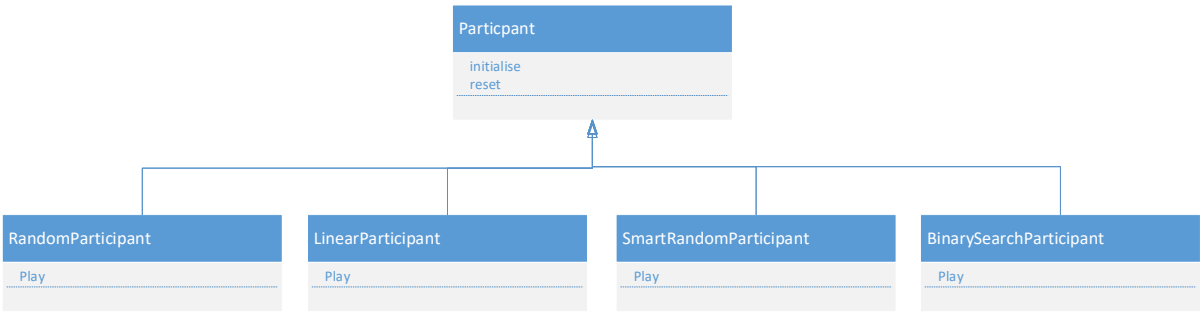
Template Method Pattern and Strategy Pattern – Day 1

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.

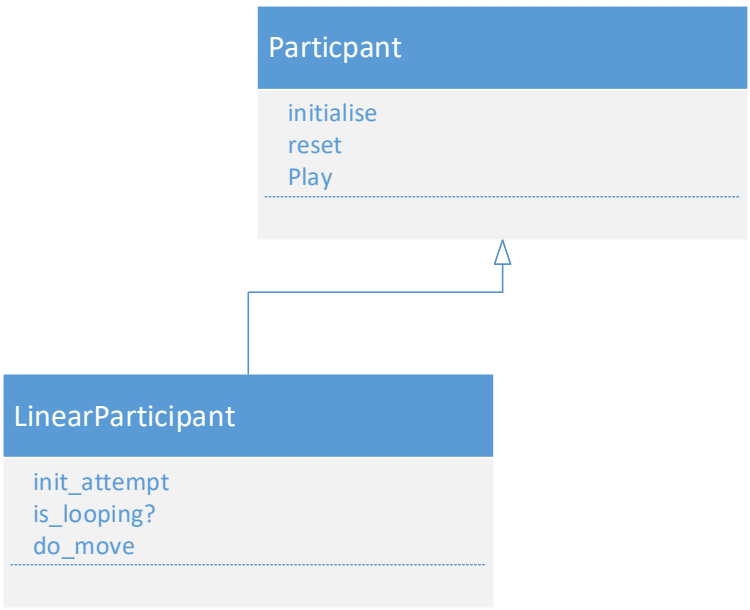
The last piece of work was to refactor the individual strategy implementations to use the Template Method patterns. I struggled a lot with this initially, as the implementations were spread over multiple files making it hard to me to spot stability and instability between them. I identified that the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

Day 3 – Factory Patterns and Singleton

Create the Factory

Initially I created a “ProductFactory” class that would look at creating the individual products and moved the products into that file as I had some issues with circular includes. I know later on I will be breaking them out for a later part of the exercise. I added the three methods as asked for in the exercise and ran the program and saw the output that I expected

Implement Singleton

For stage 2 I simply included Singleton within the factory class to see that it would work and then I added a second new. As I should have expected the first call to factory.new failed as instance is already created. I replaced the call to new with .instance and the program ran as expected.

I then removed the Singleton module and added my own implementation using a class variable @@instance. And the following code;

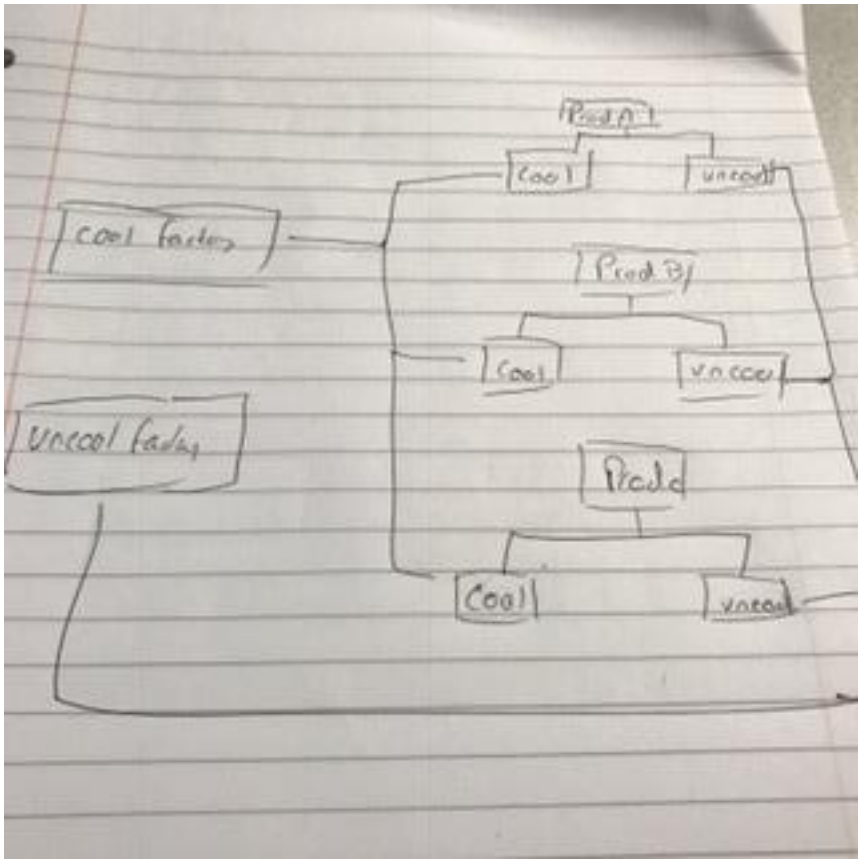
```
puts "Creating the factory"
@@instance=ProductFactory.new

def self.instance
  return @@instance
end
```

Again this worked as expected and I did a bit of playing around in the client to satisfy myself that it was working correctly

Adding the cool and uncool products

I am thinking of the abstract factory method diagram that we saw in class. I will have the following structure;

Add the class diagram from my work book in here



I refactored the products first. That went fine.

Adding deadly products

This was a matter of adding a new subclass to each of the products to do the work and adding a new concrete factory to create each of them. The client simply was passed a new symbol :deadly to call the deadly factory to do the work…

Adding Product D

To do this I created new ProductD and it's 3 sub classes and added the create code to each of the factories. Again I ran this code and check that it worked as expected.

Pause for thought

I fully expect the next part of the exercise to force this design to fail. I reflect on the principles that we heard of today about the "warm, fuzzy feeling" you may expect to get from using a design pattern and in some ways this design I have implemented does this. Both adding a new product and new family had clear places that they fitted into the architecture and were pretty easy to do. I do wonder if we ended up adding "a lot" more products and ended up having a lot more subclasses and also added "a lot" more families resulting in many more factories. For products this feels closer to the actual model we could expect but for families I am not so sure. Whilst I do like the shape of the solution to this point I think it would need to be changed to handle bigger problem spaces.

Refactoring the solution to solve section 6

Initial thoughts are to create a new class “ProductionManager” that will be a singleton itself and will manage a single instance of a factory. As the client asks for something from a new product family the “ProductionManager” will new the appropriate factory throwing the last one away and will remember what production mode the app is in. The individual factories will no longer be singletons themselves meaning a developer could create multiple instances of each of the concrete classes if they wanted so care would need to be taken to make sure that the don’t do this.

This is the solution that I implemented. Each individual part was committed to the my github repository so it should be possible to see previous iterations.

Day 4

State exercise

Taking my understanding of state from the lecture I start with creating an abstract class PersonBehaviour with three methods vote, apply_for_buspass and conscript. I will then create three concrete state classes, ChildBehaviour, AdultBehaviour and PensionerBehaviour to do the right thing depending on the state.

As these are stateless and modelling behaviour, I will create them as singletons…

In the Person class I will switch state object based on @age and call the right behaviour

I added another method (book_retirement_home) which would do nothing in child or adult but would have an implementation in pensioner. As they all follow the same interface I need to add this everywhere and just do nothing in the two cases where it was not appropriate.

############ TODO – Think of a pattern that would make this part more appropriate ##########

Handling changed requirement on subscription

Simply created another singleton state TeenagerBehaviour in the same manner as above and switched state when the user was older than 13.

Adding the medical card

Added the new method to each of the state classes and added a new method to the Person class, it worked as expected.

Adding a state manager

I created a singleton StateManager which makes the decision on which state to use. I like this as it cleans up the Person class and puts state change decisions in one place. If we were to change the definition of a teenager, then we would have one place to do this and it would be in a logical location and all other objects would respect this.

I had solved the last part by making all state a singleton from the outset. As these contain no data you don’t need multiples of these. I also made the StateManager a singleton as you probably only want one of these also.

Decorator Exercise

Thinking of the problem against real world constraints I think of a series of options; coffee type,

Milk type, sugar type, syrup type… Each of these has a description and a price, and may or may not be present. This leads me down the path that the shared interface is description and price. In this way you start with a type of coffee, add a type of milk (or none), add a type of sugar etc.

Each one will probably advertise its options allowing use system to assemble them as necessary.

The interface looks as follows;

```
class Drink
  def description
    'hot water'
  end

  def cost
    1.5
  end

  def declare
    puts "Drink is #{description} and costs #{cost}"
  end
end
```

To drink I would want to apply a type of coffee – regular, decaf, dark_roast, then milk, sugar, syrup. I will use decorators to do this.

On this one I really hit a wall. I know what I want to do but have not the first idea of how to code this in Ruby…

Eventually I created decorators for coffee, milk, sugar, syrup, a nip (say whiskey). At creation time you would add one ingredient and then pass this to the next class which would decorate the class with it's own characteristics. I tested it with the following chain;

Hot water + coffee + milk + sugar + another type of coffee + Syrup + Nip

I then added a specific behaviour to milk. Heat milk, which none of the other classes supported. Nothing else in the chain needs to know about this.

I have lots of issues with this particular pattern that I will need to come back to and make sure that I understand fully.