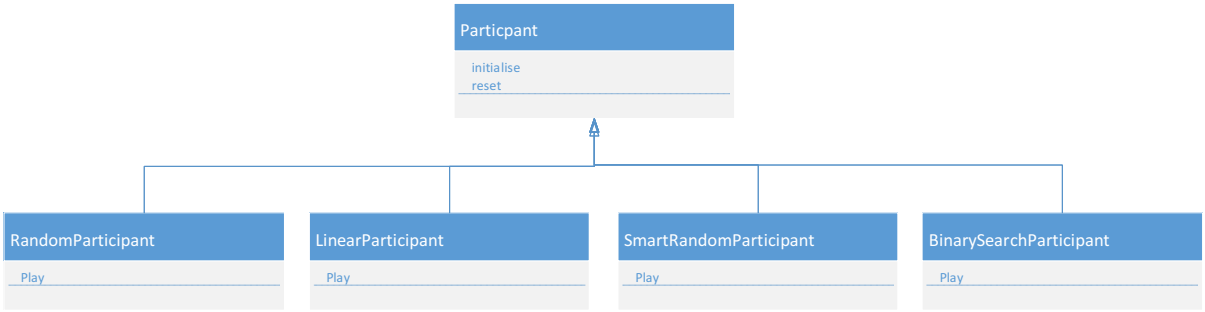
Template Method Pattern and Strategy Pattern – Day 1

The code initially looked quite messy and I needed to step through it a bit to get a handle on how the logic was flowing. Ruby was also relatively unfamiliar territory so it took a small bit of time to get settled with the language also.

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

As the exercise called for attention in the participant class it warranted initial attention and this was discussed with my partner. We quickly identified the approach to apply the strategy pattern to the participant class in the following manner



Applying the strategies were logical but following up I struggled a little trying to think about how to use the Template Method Pattern to reduce the amount of repeated code for each of the strategies.

This was easier for 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.

Having done the previous step, 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.

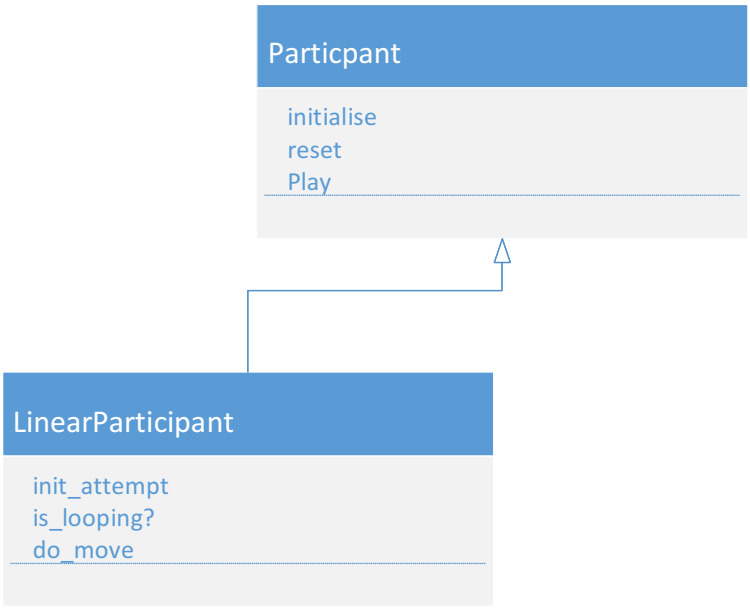
The last piece of work was to refactor the individual strategy implementations to use the Template Method patterns. I struggled a lot with this initially, as the implementations were spread over multiple files making it hard to me to spot stability and instability between them. The base structure was 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, not hook, but I chose not to throw an exception in the superclass following guidance in the morning's lecture.

To develop and test the design I added a descending game strategy 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 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 combination of the two patterns were confusing the purpose of each individual one.

The end result was code that certainly looked a lot better, would read much cleaner, was more extensible 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.

Day 2: Observer Pattern

The starting point for Day 2's lab was 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 factoring of the Template Method pattern from yesterday, all the data that needed was 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 one would simply expect to add the result and number of guesses from the Subject.

Next, a new object was added 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 that were expected…

Adding the Cheater strategy

The strategy pattern from yesterday allowed this to be added pretty easily. This was done by added a new method to oracle to allow someone to steal the magic number and used that to guess right the first time.
Alternatively one 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 exercise.

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

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

Again running the program and saw output that was expected…

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…

```
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 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. By adding an array of observers and then addind methods to add an observer, delete an observer and 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;

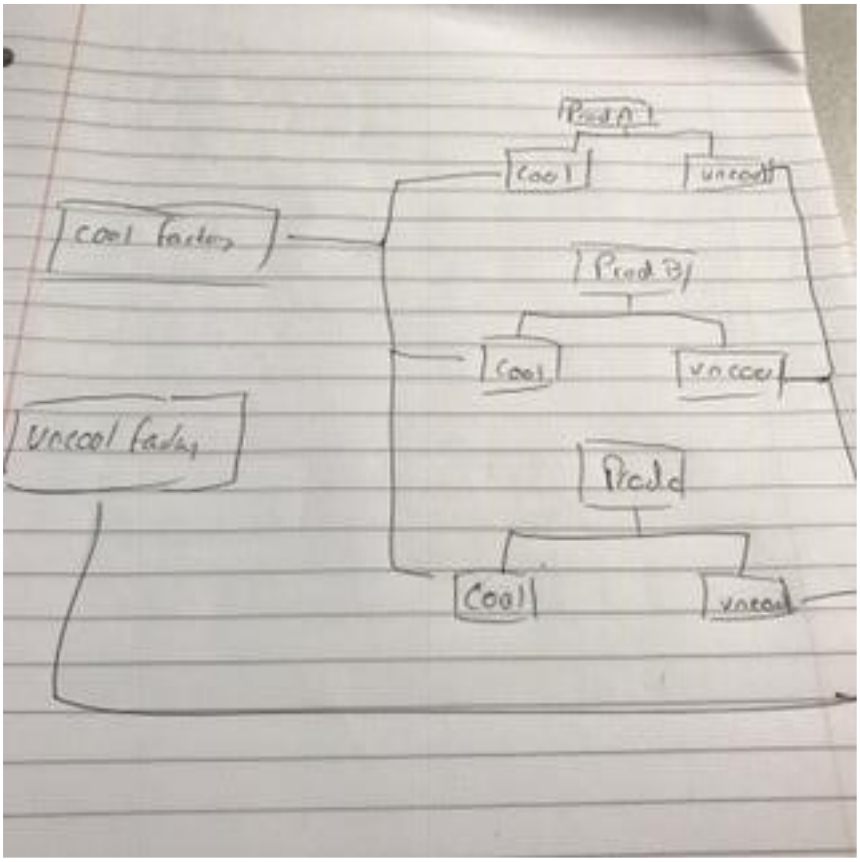
```
puts "Creating the factory"
@@instance=ProductFactory.new


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.

Day 5: Final Lab Report

Problem definition:

Take the "Kangaroo" exercise submitted as pre-work and extend it to build a simulation game where players can pick a "animal" whose move strategy is randomly chosen, and play against other users to see who can finish first. This is similar to the fairground games where mechanical horses race against each other and one of them wins. To do this we should also support a new type of animal that will move along the diagonal only (NE, SE, SW, NW).

The user will get a randomly selected animal that will move in one of these manners and the game will and each animal will print the number of moves it made, as well as the total number of moves it attempted (off grid attempts).

At the end each player will get a report out from their character to define what happened…

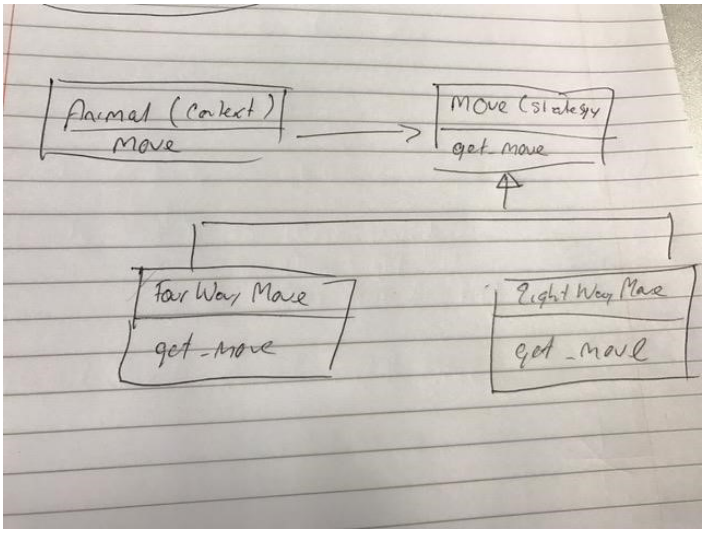
Refactor and extend the pre-work submitted to achieve this;

The patterns that will be applied are;

1. Strategy (there will be three different move patterns)
2. Template Method (There is much shared code between animals and move patterns)
3. Singleton (there should be only one grid)
4. Factory pattern (The game engine will use a factory pattern to create the players in the simulation)
5. STRETCH: Add a new feature that picks a random cell per animal and if that animal lands in that cell gets a special prize. The observer pattern is a great candidate to consider when implementing this.
6. Other patterns may be necessitated in cleaning up and refactoring the solution to support these new requirements)

Applying Strategy and Template Method to the Die and Move Management

First step was to apply the Strategy Pattern and Template Method Pattern to the dice rolling. The strategy is to roll either a 4 or 8 sided die as shown here;



The previous solution also had some shared code to make sure that the move was a valid move (i.e. not off of the grid). The Template Method moves this out of each of the move classes and into the superclass that all moves will rely inherit from.

Template method was used to factor all the move operations, such as repeating until the die throw is within the grid and not outside the bounds of the grid…

The additional strategy (diagonal moving) was added pretty easily.

Adding the new animal the 'Diagaroo'

This was simply a subclass of Animal. Kangaroo and Zigzagaroo will be refactored later to clean up the code and extract templates so the concrete behaviour of the Diagaroo will be added then too.

Creating the Factory

The factory was added to pick a random number between 0 and 2 and the appropriate class was created also;

1. Kangaroo

2. Zigzagaroo
3. Diagaroo

This is an area also where a strategy could be applied if we wanted more behaviour modelled in the creation process but in this case it is not in the requirements so adding it would not be appropriate. It is tempting though that once you start seeing patterns and their value that one could start putting them in places that are needed or not. For this problem space the requirements do not need it at this point.

Adding the new requirement to pick a cell and award a prize to the first animal who lands in that cell

The design here was to add a new class SpecialPrizeChecker which would pick a random cell for that user to get the prize and then would get notified when the player moves and checks to see if it is the special cell. The LocalObservable module that was produced as part of the lab on day 2 was pulled into the project to enable the pattern and the observer was attached when the AnimalFactory creates each player. The observer itself choses the random cell and reports that the animal has landed there, meaning that the behaviour is decoupled from the animal's behaviour. In testing this a 10x10 grid was used in almost all of the test runs, and using this size of grid the player almost always landed on the special cell. It would be interesting to check as the grid size increased did the percentage of players who got a special prize fluctuate by any meaningful pattern.

In this solution the result was a 1..1 observer / player relationship. However a new requirement was needed to have one "grand prize" cell where only one player can get the prize, the first one to land in that random cell. To do this another observer object was created and added to each animal allowing this functionality with minimal changes to the code. This allow provided a 1..n relationship between the observer and all of the players.

Final extension

Whist it is tempting to keep iterating on this problem and adding more and more features, one more check can be done to validate the cohesiveness, and extensibility of the solution. With all the additions and refactoring, the task to add a new type of player should demonstrate the properties listed above. A new player type will be added (Fastaroo) that will only move to the right or up. This player should move quickly to the final cell. To do this the following changes should be needed;

1. Add a new move strategy (right and up moves only)
2. Create a new animal that uses that strategy (Fastaroo)
3. Allow the factory to create this new animal, randomly as with the others.

This proved to be the case with one exception, the die class needed to be modified to also roll a 2 direction mode also. The addition of this player strategy was done in a matter of minutes and all tests ran smoothly.

Conclusion and Summary

Over the course of this final lab exercise the patterns used were logical for the problem space. Initially ***Strategy*** and ***Template method*** were used to re-factor the existing code that was submitted as pre-work to a more extensible framework. The ***Factory Method*** pattern was used to simplify the creation of various players to fit in with the new parameters of the simulations. These three patterns combined really well to support the addition of new types of players or strategies (Diagaroo and Fastaroo). The later was added in a matter of minutes. ***Singleton*** ensured that there was only one instance of the grid. And finally the ***observer pattern*** was used to implement prize cells

per player and also per grid. The code at the end of the exercise was much more decoupled than it started out. The cost of this was a lot more classes and files. For a very big solution (something like Microsoft Word) the sheer number of files, classes and references can become cumbersome to work with resulting in large build systems and scripts.