SOFT2201 ASS2 Report

## Discussion on Design from task 1

Having a high level UML diagram initially helped to identify some key classes which I would need to implement based on the pool table requirement, for example a table, a cuestick, balls and pockets. For task 2, there were a lot of different requirements in the design patterns which weren't required earlier, these were difficult to see at first with my already chosen class design.

A lot of the relationships in the task 1 UML diagram were correct and helped me define the concrete products needed. For example, composite aggregation relationships of the table and balls ensured that there was a set list of balls that belonged to only one composite instance of the table at a time. The table was then responsible for creation and deletion of these components utilizing the App and GameWindow to implement the creation of the concrete products.

Task 1 also created more abstraction than necessary in the ball class by separating white and coloured balls but then missed abstraction in others. I improved the dependency inversion from my task 1 classes by creating more interfaces to instantiate concrete classes from, thus isolating dependency from high level complex modules (ie the original pool table that was instantiating everything). **[SEE UML IN APPENDIX]**

## Discussion of design patterns

Factory method pattern: used to instantiate JSON file class readers

There are 4 participants in this design pattern. The **product** is a Reader, and is an interface that has a parse method to read JSON file. **Concrete product** then instantiates this interface, there are two concrete products for the moment, TableReader and BallReader which read the Table part of the JSON file and the Ball part of the JSON file respectively. The **Creator** is the ReaderCreator class which declares the factory method create() and then returns the product Reader. The two **Concrete Creators** override this factory method to return the concrete product, including a TableReaderCreator to create a Table Reader and a BallReaderCreator to create a Ball Reader.

In terms of GRASP principles, this limited coupling as I assigned responsibility of the reading of the JSON files to two separate reading classes and only the information of these would be passed on. SOLID principle is demonstrated in the open close principle as you can extend and create different readers (for example, ie cuestick reader) but the product and creator interfaces are closed for modification, so the readercreator still needs to adhere with the methods provided.

Factory method was useful for my code by allowing the creation of very specific user defined concrete product classes. Since the Balls had all the same attributes, I could specifically read each from the JSON file (eg. Reading Ball mass as a double) but keep this separate from the Table attributes. Also, if any changes happened to the path of the config file, this wouldn't affect the function of the Readers.

The issues I found with this pattern was that both readers needed to be related to the interface Reader and the method parse() needed to have the same return type. I used the nested list List<Arraylist<String>> as the return type to contain all the file values, but this wouldn't be necessary for the Table since it would need only one list of values.

Builder method pattern: used to instantiate Ball classes

There are 4 participants in this pattern. The **Builder** is the interface builder and is an abstract interface for creating parts of the Ball. Since this is for creating different types of balls, the **Concrete Builder** is named BallBuilder and implements the builder interface to assemble a Ball. The **Product** is a Ball class which is under construction by the builder and is returned using the build() method. The **Director** constructs the object by using the builder interface and has an aggregated relationship with the builder.

This method creates high cohesion as we are delegating creation responsibility to the builder class by the director, isolating this responsibility and making the builder classes solely focused on assigning the specific values to the Director class. SOLID principles were used in this pattern as well, for example the dependency inversion principle here is demonstrated by depending on abstractions and not the concrete classes. The low level concern with constructing the balls are separated from the high level logic needed to create these balls.

The method was useful to separate the assembly of the Balls, instead of having repetition to create each individual Ball with different colours and different starting positions, this was handled by the director class. In terms of writing code, this was a drawback since the builder needed to copy all fields from the Ball, requiring lots of setter and getter functions. Also, if I needed to implement any other part of a Ball, for example the balls radius, I would then have to go through and add additional getter and setter in both the Ball class, the builder class, the ball builder class and the director.

Strategy method pattern: used to handle logic of Balls falling into pockets

There are 3 participants in this pattern. The **Strategy** interface is BallFall and declares the method think() to be used in the context of a ball falling into a pocket. The **ConcreteStrategy** are the classes BallFallBlue and BallFallRed, each implementing the think() method for the particular requirement based on the balls colour ie the Blue ball respawning and the red ball disappearing. Finally, the **Context** is the Ball class which maintains reference to BallFall and then uses selection logic to decide on the ConcreteStrategy to implement.

The benefit of using this pattern means that this logic can be reused, since there would be multiple Blue and Red balls on the board at once. The way I have implemented it, the client doesn't need to be aware of the logic to switch between strategies since Ball class is able to handle this purely based on its own Ball colour. This provides flexibility, the SOLID principle of open close is demonstrated since more think() methods could be added for additional coloured balls but the actual think() interface needs to be implemented and is closed for modification. The interface segregation principle is also demonstrated by utilising a single BallFall interface opposed to lots of interfaces with different think() methods. The downside of using this pattern was the creation of numerous objects in the application, in particular the need for an additional class for the red ball, even though it doesn't do anything when it falls into a pocket.

Appendix:

Figure 1: Task 2 UML