**Received code:**

**How well designed was the code for extensions, what particular elements aided or hindered extensibility? (10%)**

The codebase was generally easy to make extensions to. The documentation and comments assisted with understanding the code and creating a UML diagram out of it. One of the downsides of the code was that some classes shared header files, for example the Ball and StageOneBall class where in the header file Ball.h. Though this didn’t hinder extension per se, it sometimes made it harder to read code as a I was expecting a different header file for the concrete class.

**How well documented was the code with respect to both external documentation and comments? (10%)**

**The code was very well documented. All the doxygen comments were in the header file as expected and in line comments in code were included where necessary. There**

**Was the coding well done? What would you have done differently? What was good/bad about the implementation? (10%)**

**Good:**

* If this was the 90s, the coding style was very well done. It was generally easy to see

Bad:

* There were methods that returned raw pointers. This creates the ownership problem of who is responsible for deleting the pointer. I would’ve used a smart pointer instead.

**Comment on the style of the code. Were names, layout, code clichés consistent? (10%)**

The separation of the definition from the declaration of classes was not fully followed in the code. In most cases, the constructor and destructor were all defined in the .h files instead of the .cpp file. The layout was consistent throughout the code, save the dialog file. The member variables were declared at the top of the class and methods at the bottom.

Generally, though, the style of code was consistent. Member variables used the “m\_” style to denote membership to a class.

**Your code:**

**Explain the application of the design patterns for your code. (20%)**

**Composite Design Pattern: The composite design pattern was applied to the balls in the game. A new class, StageTwoBall, was used as the composite and the existing Ball interface was treated as the component. There was no leaf class participant as it would add unnecessary complexity to the code.**

**The StageTwoBall was used to contain the inner balls. When drawing child balls on top of itself, the render() method first drew the parent ball, and then called render() on it’s children. To get the effective mass the parent ball, the getMass() method returned the mass of itself and recursively called getMass() on its children.**

Adaptor Pattern: The adapter was used to give a new interface to the existing game class. The existing class did not accept any actions for mouse and keyboard events. The new interface allowed the dialog class to send keyboard and mouse events to the game.

The default action of these mouse events is to not do anything. This maximises compatibility with the stage 1 game class.

**Explain advantage and disadvantages of the design patterns used with respect to your code. (20%)**

**Composite**

Advantages

* The application of the composite design pattern in the code followed the open/closed principle in SOLID.
* The composite pattern also single responsibility principle where the StageTwoBall is only class that manages its inner balls.

Disadvantages

* The StageTwoPlayable game class for some aspects relied on the concrete implementation of Ball (StageTwoBall) to access a specific method. This violated the dependency inversion principle where clients should depend on the abstract class.

**Adaptor**

Advantages

* The adaptor in this code follows the single responsibility principle where it is responsible for translating requests that the client calls to the target interface to the adaptee class.
* The adaptor allows code reuse. In the case of this assignment, we reuse the Game class.

Disadvantages