**Received code:**

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

The code was designed well for extension. The abstract factory design pattern made it easy to create new versions of products. The interfaces were generally extensible however some getters and setters were not marked virtual which was a problem particularly with the getMass() method in the Ball class as the second stage required it to be calculated differently.

However, when it came to user interaction, the interface of the game class did not allow for mouse events. None of its methods were declared virtual which meant that subclasses could not override its methods. Many concrete implementations of interfaces also left out the keyword virtual from its methods which hindered subclassing.

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

**The internal documentation of the code was done very well. All the doxygen comments were in the header file as expected and in line comments were included where necessary. The documentation concisely explained what each method did, and the inline comments were helpful in figuring out what each code block did.**

**External documentation was not included so in that regard it was poorly done.**

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

**Good:**

* The code was very well done. All allocated memory on the heap was freed as required.
* There was clear separation of responsibility between the classes. The abstract factory created the objects, and the builder built the game from the created objects. The Game class handled the interactions between the table and the balls.
* The code was easy to read, short and concise. Methods were marked with const and override where necessary.

Bad:

* There were methods that returned raw pointers. This created the ownership problem of who was responsible for deleting the pointer. I would’ve used a smart pointer instead.
* Concrete classes did not mark their methods virtual which made them impossible to extend without modifying. I would’ve marked them virtual despite the slower performance.

**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.

Generally, the style of coding was consistent. Member variables used the “m\_” style to denote membership to a class. The class member variables and methods were organised by private, protected and public except in the dialog class where it was public, protected and private.

**Your code:**

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

**Composite Design Pattern: The composite design pattern was applied to the Ball interface. A class StageTwoBall was created to act as the composite and the Ball interface was the component. The leaf class was omitted as leaves could just be composites Balls without any inner balls. The composite ball was responsible for drawing their inner balls, as well as updating the position of the inner balls.**

**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(AbstractPlayableGame) allowed the dialog class to send keyboard and mouse events to the game. The way that this pattern was applied in the code resembled a proxy since many method names were unchanged.

**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 class StageTwoBall extended the Ball class without modifying the interface to achieve the containment of inner balls.
* The composite pattern also single responsibility principle where the StageTwoBall is only class that manages its inner balls. The management included updating the position of the inner balls as well drawing them onto the screen.

Disadvantages

* The composite class StageTwoBall contains methods which are not present in the ball interface. This means the client needs to in one way or another rely on the concrete implementation of StageTwoBall rather than the Ball interface to access its methods.

**Adaptor**

Advantages

* The adaptor allowed code reuse. In the case of this assignment, we reuse the Game class under a new interface which better suits the client needs.
* The new interface is more extensible than the original game class which had no virtual methods.

Disadvantages

* The existing factories and builders had to be significantly refactored to return the new game interface.

For a clearer picture of the UML diagram please see attached documents.

