**BRIEF DESCRIPTION OF GAME AND GAME OBJECTIVE**

The task for the assignment brief was to create a game to teach school kids some form of science using a computer application game. In this report, I would outline the design and implementation of the game created and further stating my conclusion. The name of the game is Photosynthesis and it aims to teach school kids how plants grow. Below is a list of primary objects clients would be interacting with in the game

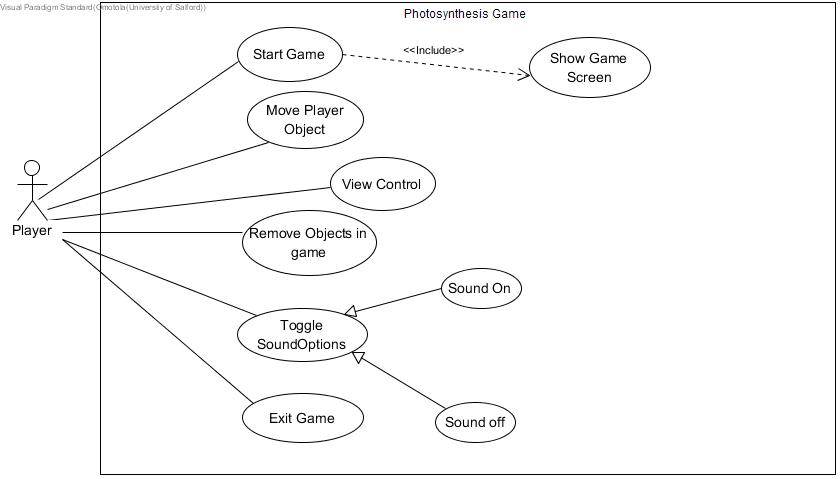
* PlantPod (Player)
* CO2 (Element)
* Water (Element)
* Sun (Element)
* Flame (Enemy)

The task in the game is for the player to gather all the elements so it can grow and avoid the enemy before its health is exhausted. The graphical user interface is simple and easy to follow, it includes sound, buttons, health indicator and images to help interaction with game fun but educational.

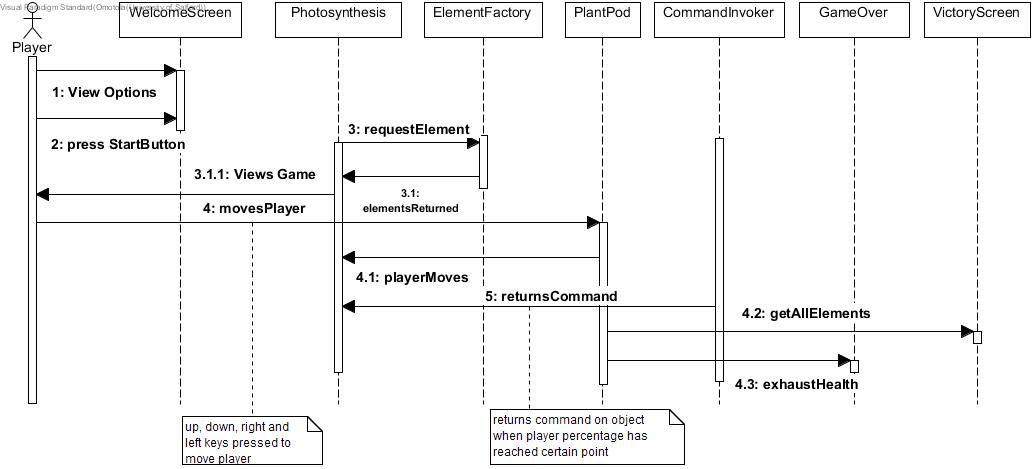
**DESIGN**

In the game, I have implemented five design patterns, the factory pattern, command pattern, delegate pattern, singleton pattern and Null Object pattern.

**USE CASE DIAGRAM**



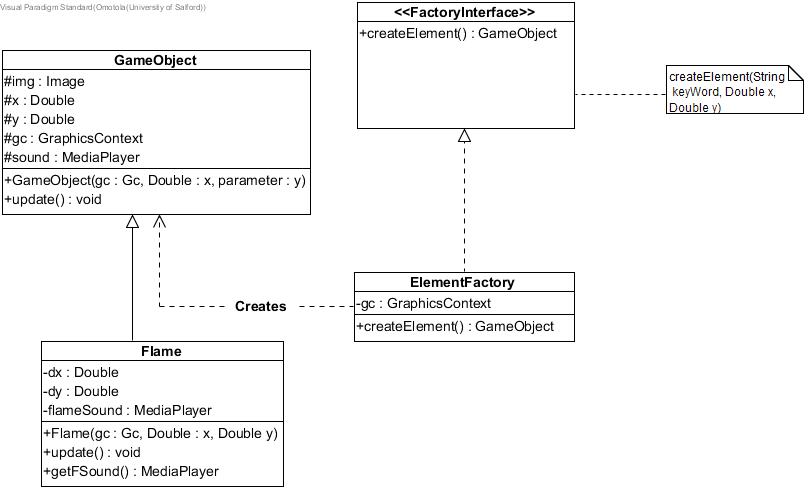
**SEQUENCE DIAGRAM**

****

**FACTORY PATTERN**

The factory pattern is one of the creational patterns used to handle creation mechanism, and in the game, I used this pattern to create instances of several game object. The *FactoryInterface* is implemented by the *ElementFactory* class which gives a concrete realization of the method *createElement()* it instantiates the gameObject at position x and y upon passing a given keyword. The *ElementFactory* instantiates class Sun, Water, Flame, Co2, PlantPod, Growth, and a Null Element classes. They are all subclasses of the superclass *GameObject.* The method call for creating objects is written in the main class *Photosynthesis*.

CLASS DIAGRAM FOR THE FACTORY DESIGN PATTERN



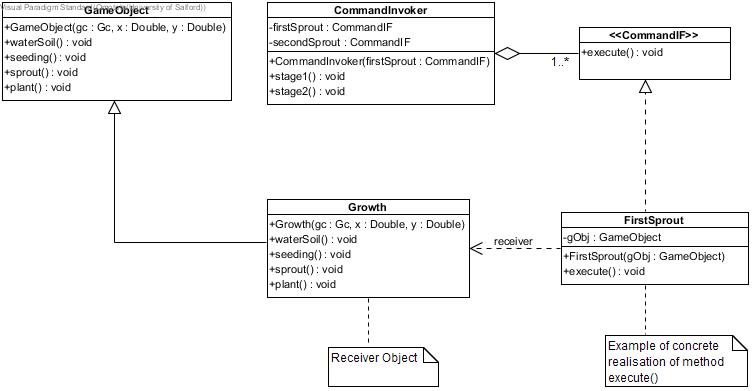
The *ElementFactory* class is instantiated taking a parameter of the GraphicsContext used to draw object on canvas. In *Photosynthesis* class, an arraylist is created to house *element objects* in the game. A for loop is used to create a certain number of each element at random positions on the canvas within a range of 700 – 551. When the correct keyword is passed to the method createElement the associated object is created. *GameObjects* that are not added to the list include the player object and the receiver object.

**COMMAND PATTERN**

The command pattern is classed a behavioural pattern, which handles communication between different objects usually between an invoker and a receiver object. The *CommandInvoker* class decouples invoking the commands from the receiver that performs the task. In the game, I have implemented the *Command pattern* to illustrate the different stages of growth of a plant when player object *PlantPod* takes elements. All commands invoked on the receiver implements the *Command Interface* and provides a concrete implementation of the *execute()* method.

The *CommandInvoker* class servers as an invoker of the command to be acted on the receiver (*Growth class*). This class takes in its constructor the Commands to be executed. I constructed four commands to be executed, therefore the CommandInvoker constructor takes four parameters. Once the invoker class is instantiated the individual commands can be called which executes the command in the actual *Command class* (e.g. *FirstSprout*)*.* Note that the *Commands* alsotakes a parameter of the receiver and it is in the receiver class that the behaviour of the execute method would be carried out.

CLASS DIAGRAM FOR THE COMMAND DESIGN PATTERN



FirstSprout is one of the commands to be executed on the receiver object. In all the command classes, the receiver class is of type GameObject, the specific game Object that the command will be executed on will be passed at runtime to the polymorphic *gObj* variable which in turn calls the method *waterSoil()* which is an overridden method in the receiver class *Growth.*

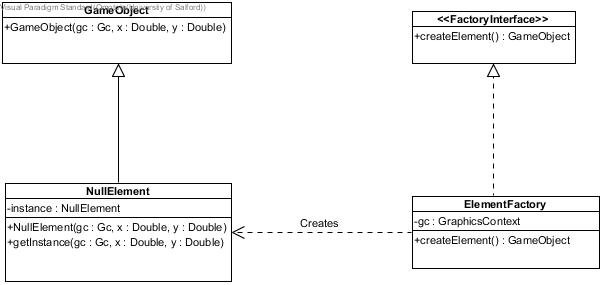
The *Growth* class extends *GameObject* and its method waterSoil() is the method that would be executed, and this case it changes the picture to show the growth of the plant.

The invoker takes command FirstSprout, SecondSprout, ThirdSprout and Plant which takes the receiver object as a parameter. The commands are only called on the receiver when the number of elements collected is up to a certain point. Each element collected is worth 4%

**SINGLETON AND NULL OBJECT PATTERN**

The null object pattern is a behavioural pattern that encapsulates the idea of a do-nothing behaviour. It is often implemented as a singleton which is classed as a creational pattern. Since an instance of a null object will often have the same state it is only reasonable to have just one instance of the object that does nothing. In the game, I have used this pattern to abstract handling null exceptions.

CLASS DIAGRAM



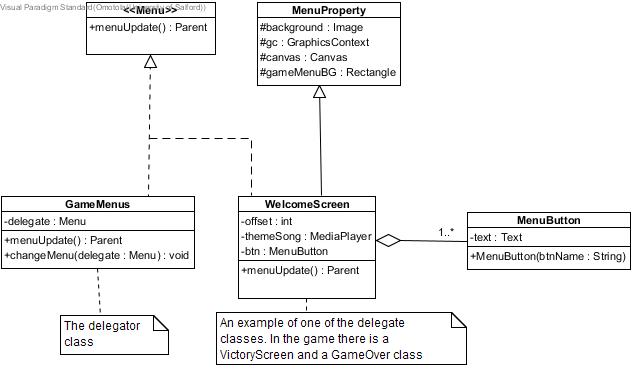
The *NullElement* class extends *GameObject* and only one instance of the class is retrieved from the static global method *getInstance().* The NullElement class is used in the *ElementFactory* class to handle null exception in the event an unfamiliar keyword is passed into the *createElement()* method it returns an instance of the NullElement which does nothing.

**THE DELEGATE PATTERN**

The delegate pattern is used to extend and reuse behaviour, it is often preferable to inheritance when a subclass needs to be a different object at different times. Therefore, in the game I have implemented this pattern to display different menus at different times in the duration of the game.

The Delegator class *GameMenus* delegates all classes that implement the interface *Menu*. The delegate classes in the game include *WelcomeScreen, VictoryScreen* and *GameOver* and provides a concrete realisation of the *menuUpdate()* method in the *Menu* Interface. For abstraction purposes, the delegate objects extend an abstract class *MenuProperty* which contain shared properties between screens. The task of creating the same button across the game was aggregated into a *MenuButton* class and each class that implements *Menu* *has a* relationship with the *MenuButton* class.

**CLASS DIAGRAM FOR DELEGATE PATTERN**

****

**CRITIQUE AND EVALUATION OF CODE**

When designing the structure of the game, I identified each components of the game and what task they were responsible for, and further abstracting the specifics from the general classes by introducing interfaces and abstract classes which promote loose coupling. Writing each class in a cohesive manner so that the unit can function on its own without much dependency. There were a few discrepancies when developing my game which would be criticized below.

**Command Pattern**

This pattern requires that classes be created for every command that is invoked on the receiver, this can be wasteful, because assuming we had 100 commands to be invoked on an object, 100 command classes would need to be created with very minimal changes in each of those classes.

**Use of Global Variables**

This is frowned upon, and defeats the purpose of encapsulation. In the source code, I have made the method *getFSound()* which returns the sound for the *Flame* object globally accessible to other classes. This method was made globally accessible because the flame sound should only be played when the start button is pressed and the start button is in the class *WelcomeScreen*. A reasonable solution to arrive at the desired result was to make the method global, but on the positive side this method cannot be changed by any class other than the *Flame* class.

**CONCLUSION**

In conclusion, having a responsibility driven approach to writing code is effective because it allows the opportunity of code reuse and scaling. The photosynthesis game, can better be improved by adding functionality of allowing users input keywords that create the game objects in the game, this functionality would utilize the NullElement class effectively because if a user enters incorrect elements instead of throwing an exception a nice printed message is shown to user. Also, the sounds played in the game would have been useful in its own class to avoid the use of global variables.

**GAME SCREENS**

