**Pattern-Based Software Development**

**Team ID: B1**

**Group Assessment**

**Project Specification:**

In this project we were asked to create a Java application from scratch in order to learn how to correctly apply software design patterns and their practical implications. However, finding a good domain for an application that would be able to utilise at least eight different design patterns proved to be a more challenging task than previously thought. We had to put thought into the practical use cases of the design patterns and the advantages that come with them before deciding the best candidate for our project. At the end, the team decided to develop a variation of the popular arcade game Pacman for desktop.

The game follows the typical Pacman structure. There is a board that resembles a maze that contains simple dots, fruit enhancements, ghosts and of course, the Pacman. The user can select on run time between three different board sizes with unique layout, different number of ghosts and different number of fruit enhancements. As the game progresses and the player collects more points, based on the board size, the difficulty of the game will change. There is a total of three different game levels which change the behaviour of the ghosts. In the first level, the ghosts move completely random, in the second level the ghosts calculate the Euclidean distance between their current location on the grid and Pacman’s and lastly, in the last level (level 3) the ghosts move based on the Manhattan distance metric which is specifically designed to work on 2D grids. Moreover, the fruit enhancements are self-explanatory. There are four different kinds of fruits which offer unique bonuses to the gameplay. One gives extra points, another stops the ghosts from moving for a fixed amount of amount, one other gives the Pacman immortality and now it is able to pass through and “eat” ghosts without dying and the last one is a composite fruit which offers a random combination of two of the previously mentioned functionalities.

In order to complete the game, you have to complete all three levels of difficulty and then a success screen is shown, and the game is terminated. If you manage to lose all three of your available lives before completing the game, then a screen is shown prompting you to return to the game board selection in order to start over again.

Lastly the game can be played by either using two different sets of keys (arrows or wasd) which move the Pacman to the respective direction or by using the mouse where the left and right clicks as well as scrolling move the Pacman to the right direction.

**Pattern Specification:**

For the project purposes we had to use the following six patterns:

1. Model-View-Controller
2. Composite
3. Strategy
4. Factory
5. Singleton
6. Adapter

Also, we have included in our implementation the following two patterns:

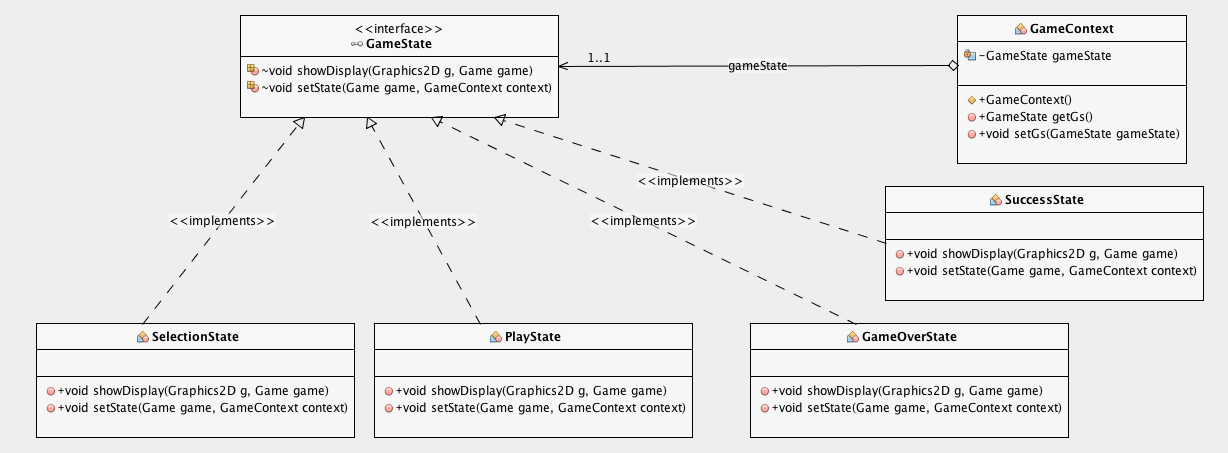
1. Iterator
2. State

As an architectural pattern, Model-View-Controller stands at the core of the implementation which there are three different packages, the model, view, controller which communicate through the controller without any direct interaction between the view and the model. The rest of the software design patterns were used as follows and will be more thoroughly described in the rest of the report:

1. Composite – fruit functionalities and their composition in the composite fruit
2. Strategy – based on the level of the game, the ghosts move by the use of a different movement algorithm
3. Factory – different board sizes with different attributes can only be initialized through the factory to make it easier and correct every time
4. Singleton – singleton makes sure only one factory is created during runtime, and every other need of instantiation comes through the same, initial factory object
5. Adapter – combines the keyboard and mouse interfaces for user input
6. Iterator – iterate through the list of ghosts in order to move them all at the same time, every time
7. State – Different views comprise the different state of the game. The states are self-explanatory: Selection, Play, GameOver, Success

**Pattern Descriptions**

1. **State Pattern**

****

**Motivation**

State pattern is a design pattern that handles different behaviors needing to change dynamically at runtime. Selection state shows the welcome message created with the awt library of Java.Play state has been used after choosing the board of the game. Drawing the components of the game such as : ghosts, walls, fruits etc. takes place in play state. Success state prints the success message after passing all the levels that contain different conditions. Game over state runs only if Pac-Man has no lives remaining.

**Implementation**

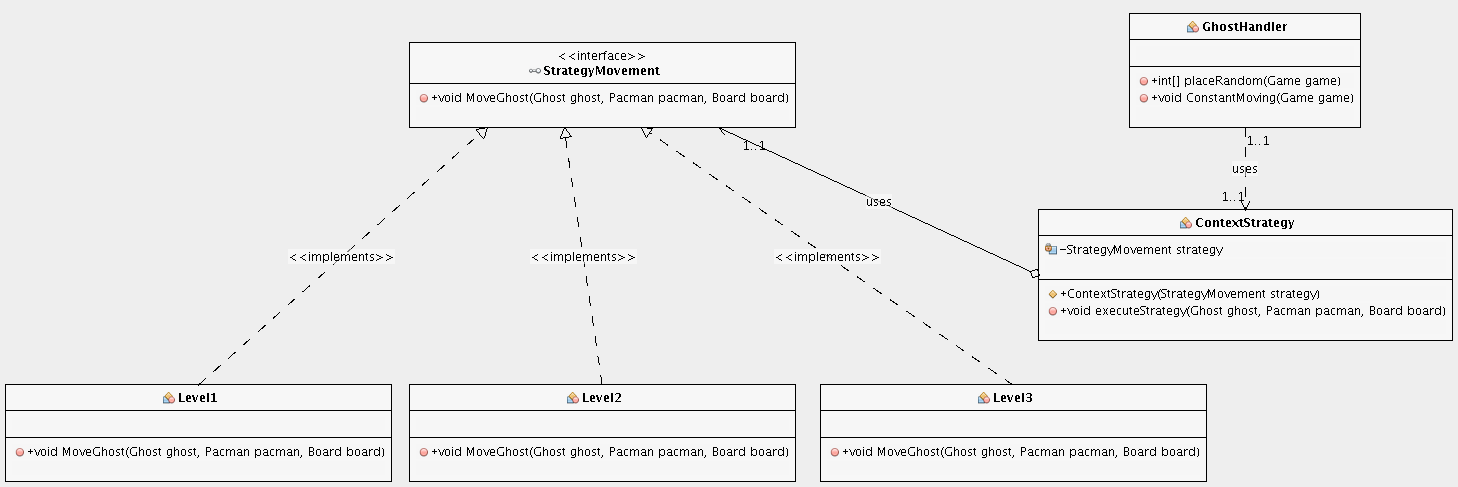
Pacman has 4 different states such as: Selection, Play, Success and Game Over.

All the state concrete classes have the same showDisplay() function that is implemented by the GameState interface. This function plots the necessary graphs in order to show different screens changing during the game. GameContext is created for containing as well as changing the states by using the interface.

**Outcomes**

Consequently, the flexibility of adding new independent states to the game is improved by utilizing the structure of the State pattern.

**2. Strategy Pattern**

****

**Motivation**

The strategy pattern is used when we want to interchange between a family of algorithms on run time. The motivation to use this pattern is efficiently in our project comes from the need to have three different difficulty levels that change the way the ghosts move. In the first level the ghosts move randomly, in the second level they chase pacman using the Euclidean distance metric, and in the third level they chase pacman using the Manhattan distance metric.

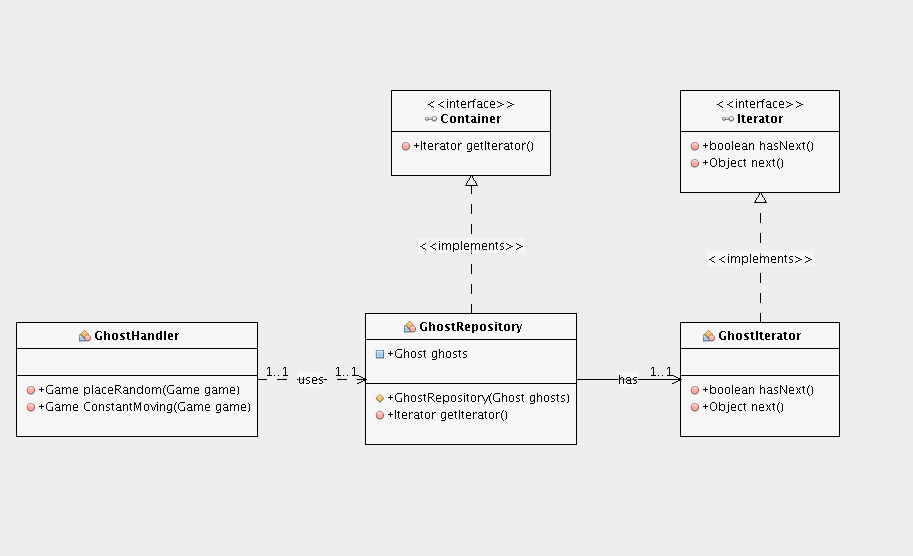
**Implementation**

The pattern consists of the interface “StrategyMovement” which provides the common function “MoveGhost” that all concrete strategy classes should implement. The concrete classes are level1, level2, level3. Each one of them implements a different algorithm for selecting the next available move that each ghost will do. ContextStrategy class maintains a reference to the strategy and depending the context (level) calls the respective movement function through the interface. The client class for this pattern is the “GhostHandler” class which through the “ConstantMoving” function utilises the different movement algorithms.

**Outcomes**

This implementation is easily extensible, which means if we want we can just add more levels with different movement algorithms and that would only be feasible by only adding one extra class that implements the StrategyMovement interface. Also, this implementation increases cohesion among the classes that implement the same interface. Lastly, this provides looser coupling and less complex classes as each algorithm is separated into a different class and the context class has access to all of them through a single point of connection.

**3. Iterator**



**Motivation**

We decided to include the iterator pattern as one of the optional design patterns because we found the need to iterate over all objects of the collection. The collection is the collection of ghosts that exists the game and we want to iterate over all of them, every time, over a fixed period of time in order to constantly change their movement. The best pattern for such a job, is the Iterator pattern.

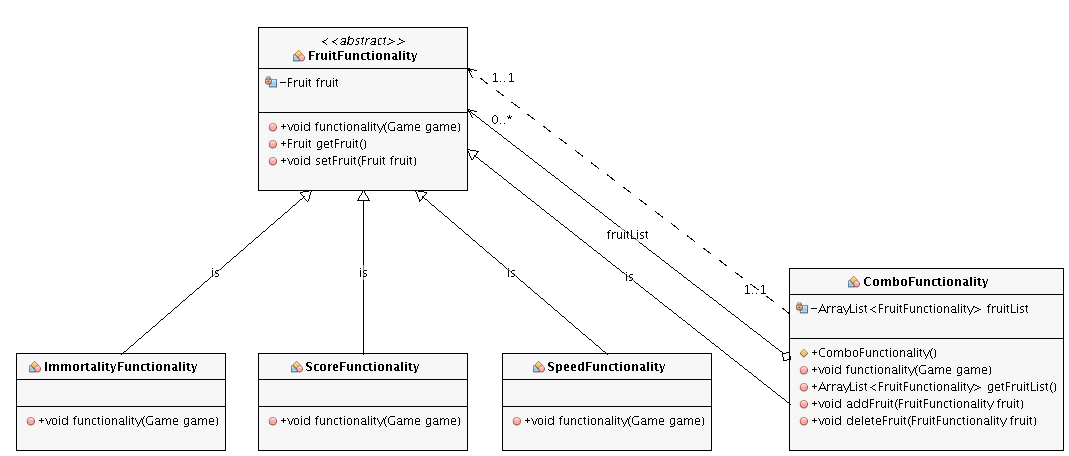
**Implementation**

The “GhostRepository” class is the class that implements the Container interface and is responsible to return and implement the Iterator object. To do this we have to create a private class named “GhostIterator” that implements the Iterator interface which consists of two functions, one that returns the next object and one that checks if there is a next object. The client program is again the GhostHandler class that in combination with the strategy pattern, every time ConstantMoving function is called, all the items of the collection are checked through the iterator pattern.

**Outcomes**

This implementation allows us to make sure we iterate over every object of the collection (ghosts) even if their number changes at runtime or mid-loop. Therefore, we have better algorithmic correctness and better encapsulation as through the use of iterator we make sure that we do not care about the underlying implementation and instead provide a generic interface to use only for iterating over any collection of objects.

1. **Composite Pattern**



**Motivation**

Composite Pattern is used when an object can be composed of some other smaller objects. Using this pattern creates a hierarchy among objects and client code can ignore the difference between single and composed objects.

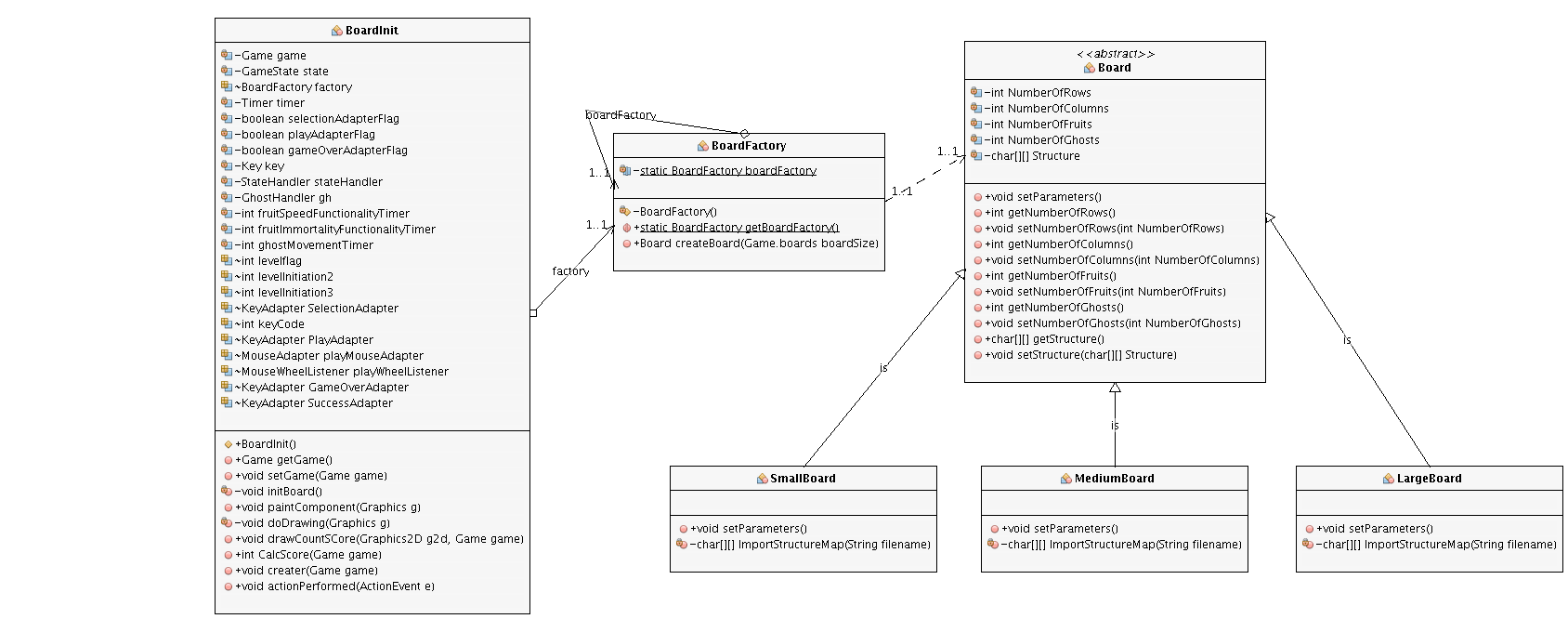
**Implementation**

In the application, there is an abstract class named as “FruitFunctionality” which is responsible for the effects of the fruits. Each type of fruit on the game board gives a different advantage to Pacman. There are 3 different single concrete functionality classes that inherits the abstract “FruitFunctionality” class. They are “ImmortalityFunctionality”, “ScoreFunctionality” and “SpeedFunctionality”. There is also a composite class (“ComboFunctionality”) which has a list of single functionality classes. As all of the functionality classes are inherited from the abstract class, they all have the same method (“functionality”). Whenever a fruit is eaten by Pacman, this method is called. The “functionality” method in the “ComboFunctionality” class calls the “functionality” methods in other classes based of the fruit list its instance has.

**Outcomes**

Thanks to the usage of Composite Pattern, “ComboFunctionality” class is created and being used with the least complexity possible it can cause in the application. Moreover, the game board has a list of “FruitFunctionality class” and it does not differentiate the actual implementation of the class. It calls the “functionality” method to perform the effect of the fruits regardless of the fruit type.

1. **Factory**



**Motivation**

Factory Pattern is used when there is a need for a class that is only going to be responsible for object creation. It separates object creation and usage.

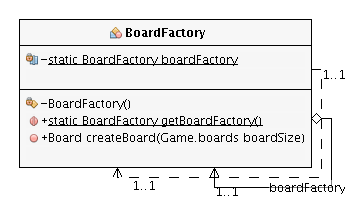
**Implementation**

There are 3 concrete classes which are inherited from abstract “Board” class. They are “SmallBoard”, “MediumBoard” and “LargeBoard”. “createBoard” method in “BoardFactory” class takes a parameter to decide which board class to initiate and return.

**Outcomes**

“BoardInit” class makes use of this factory class and create a board through “createBoard” method. Thanks to this way, initiation of board classes are completely isolated from their usage.

1. **Singleton**



**Motivation**

Singleton Pattern is used when there is only one instance of a class is required for software applications. It restricts the creation of more than one instance of the class and therefore saves memory and avoids unnecessary complexity.

**Implementation**

It is applied on Factory classes quite often. In the application, there is a Factory class named as “BoardFactory”. This class has a static variable of itself. The constructor of the class is private and static “getBoardFactory” method is used to initiate the static variable only once and then each time it is called, the initiated “BoardFactory” instance is returned through the method.

**Outcomes**

“BoardFactory” class is being used to initiate different size of boards based on the parameter passed to “createBoard” method. There is no need to have more than one instance of this class and Singleton Pattern restricts this situation as explained above. “BoardInit” class calls static “getBoardFactory” method to get the instance of the factory class (“BoardFactory”) and when there is a need for creating a board “createBoard” method is called via the only instance of the class.