Software Development 2 (SET11103)

Coursework: Sky Wars

Heidi Koikkalainen

Matriculation Number: 04004626

Due Date: 24th April 2018

Game specification

The coursework assignment was to develop a move-based Sky Wars game with a Graphical User Interface (GUI). The game consists of a Sky (a four-by-four grid), a Master Space Ship and a number of Enemy Ships. The specifications given were:

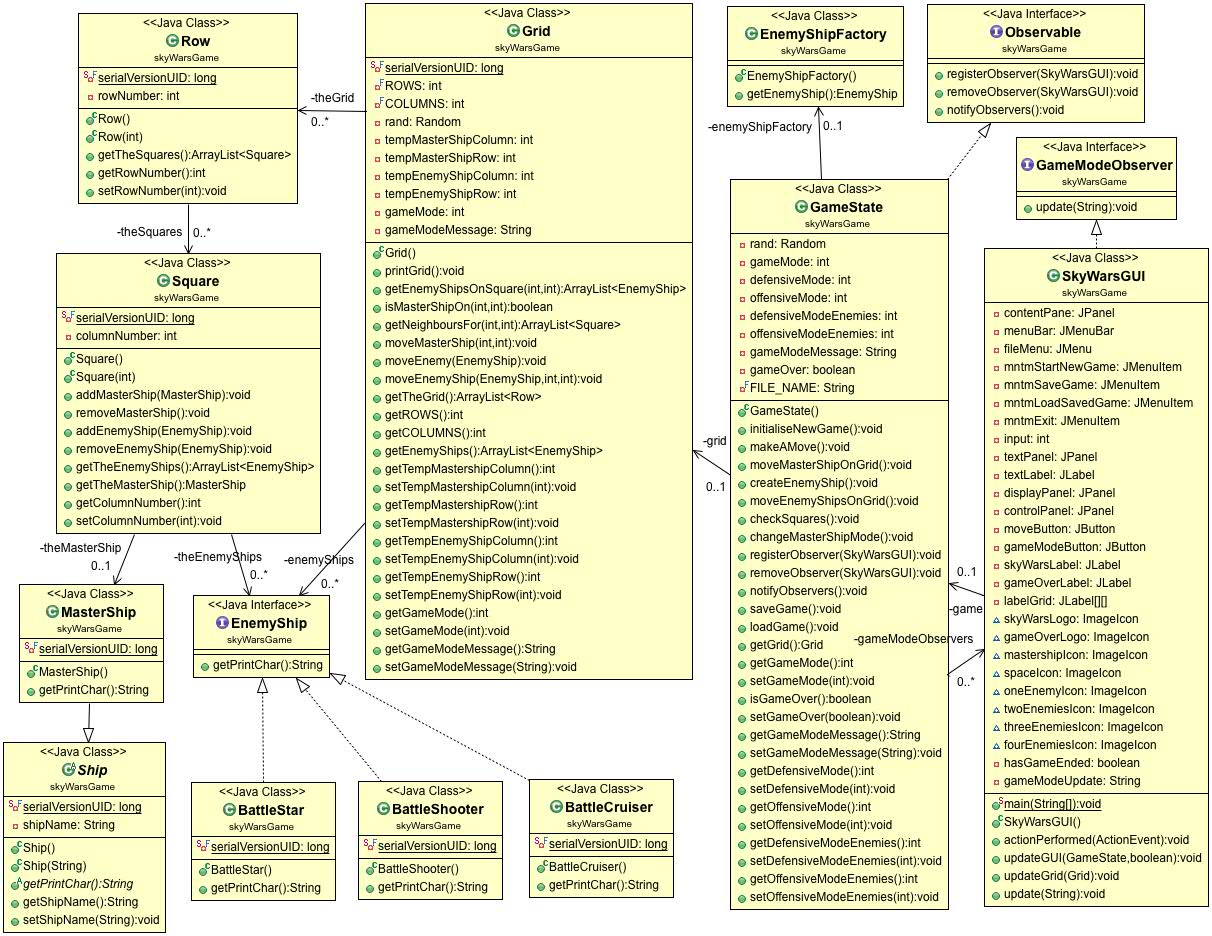
* When the game starts, the Master Ship is placed on a random square in the Sky and it can then move to any of the neighbouring squares (horizontally, vertically or diagonally) which is chosen at random.
* Every time the Master Ship moves, there is a one-in-three chance of an Enemy Ship entering the sky. There are three types of Enemy Ship, the BattleStar, BattleCruiser and BattleShooter, and the type of Enemy Ship entering the sky each time is completely random. New types of Enemy Ship may be available in the future.
* Enemy Ships enter the Sky through the top left corner and can then move randomly to any of the neighbouring squares. The space ships (including the Master Ship) cannot move out of the sky.
* If the Master Ship moves onto a square containing one Enemy Ship, the Enemy Ship gets destroyed and is removed from the Sky. If there are two or more Enemy Ships on the square, the Enemy Ships destroy the Master Ship and the game is over. Any of the space ships cannot be destroyed at the top left corner of the Sky.
* The game should be controlled from a GUI, which should allow the user to start the game and make moves. It should also provide a visual representation of the game and update it as the game status changes.
* The program should include the use of a programming technique that was not taught during the module.

Advanced Features:

* The GUI should allow the user to save and reload a game.
* The Observer pattern should be used at an appropriate place in the program.
* The user should be able to change the mode of the Master Ship during the game. In defensive mode the Master Ship behaves as described above. In offensive mode it takes three Enemy Ships on the same square to destroy the Master Ship.

Game Design

My program implements the four main concepts of object-oriented programming: **Encapsulation**, **Abstraction**, **Inheritance** and **Polymorphism** (Johari, 2017). Encapsulation means binding your data and code together into a single unit (class). It is also a way to hide variables from other classes and, thus, to provide more control over them. This is achieved by declaring the variables of a class private and providing public get and set methods to access and modify the data from a different class. Abstraction means breaking complex systems into simple components, each with their most basic attributes and functions, and is achieved by using Abstract classes and interfaces. Inheritance allows us to use existing classes (including Abstract classes) to create new classes that share some of their attributes and behaviours but have more specific functions. Polymorphism, on the other hand, is the ability of an object to take on multiple forms and behave in a different way in different circumstances. Using these concepts ensures that the program is organised into manageable components that interact with each other, and that the created code is reusable and extensible (Johari, 2017).



The above UML class diagram illustrates my design for the game. The main components of the game are the Sky, Master Ship and Enemy Ships. For the Sky, I created a four-by-four grid by using Square and Row objects. Each Row consists of a collection of Squares, and the Grid, in turn, consists of a collection of Rows. Each Square can add, contain or remove the Master Ship or any number of Enemy Ships. The Grid class can check if a specific Square contains the Master Ship or any Enemy Ships, knows how many Enemy Ships there are in the Sky, and also handles the movements of the ships.

The MasterShip class extends the abstract Ship class and inherits the name attribute and the *getPrintChar()* method.

The GameState class contains the game logic and can initialise a new game, make a move, create a new Enemy Ship (through the EnemyShipFactory class) and check whether there are any Enemy Ships in the same square as the Master Ship. It maintains the game status and determines when the game is over. An instance of the GameState class is created in the constructor of the SkyWarsGUI class and, hence, every time the program is run. The GUI and the application are connected by this association and when the user selects one of the menu options or presses a button from the GUI, a corresponding method in the GameState class is called (apart from ‘Exit’, which closes the program):

* ‘Start New Game’ menu option – *initialiseNewGame()*
* ‘Save Game’ menu option – *saveGame()*
* ‘Load Saved Game’ menu option – *loadGame()*
* ‘Make a Move’ button – *makeAMove()*
* ‘Change Game Mode’ button – *changeGameMode()*

This ensures that there is no game logic inside the GUI.

Polymorphic Programming and Interfaces

Polymorphic programming seemed an appropriate way to create the Enemy Ships as it allows objects of various types and classes to be created through a single, uniform interface, which keeps the code simple and easy to use. I decided to use the Factory Pattern, which uses polymorphism for object creation as well as object use. I first created the EnemyShip interface and then the three types of Enemy Ship classes – BattleStar, BattleCruiser and BattleShooter. All of them implement the EnemyShip interface, which forces them to have a *getPrintChar()* method but allows them to implement it in a different way. During the game the Enemy Ships can be randomly created through the EnemyShipFactory class, and their specific print characters accessed via the same *getPrintChar()* method call (which is known as method overriding, an example of run-time polymorphism).

Graphical User Interface

I used the Window Builder to create the GUI, which opens a ‘Welcome to Sky Wars’ window when the program is run:



The JFrame has four containers:

* A menu bar at the top of the window containing the File menu
* A JPanel which contains a JLabel displaying text
* A JPanel which contains a JLabel or a grid of JLabels with images (providing the visual representation of the game)
* A JPanel at the bottom of the window containing the ‘Make a Move’ and ‘Change Game Mode’ buttons

When the user starts a new game from the File menu, the Sky Wars logo is replaced by a four-by-four grid of JLabels, each containing an image. The *updateGrid(Grid grid)* method loops through the grid finding the location of the Master Ship and adds a JLabel containing the Master Ship image on the grid, filling out the rest of the grid with JLabels containing the space background image. The text label is set to display text: “MasterShip mode: DEFENSIVE” and the ‘Make a Move’ and ‘Change Game Mode’ buttons enabled for use.

The user can then play the game by pressing the buttons as they wish until the game is over or they decide to exit the game. The display panel is updated with the JLabel grid throughout the game, displaying the location of the Master Ship and any Enemy Ships in the Sky. The text label gets updated every time when the ‘Change Game Mode’ button is pressed.



The user can also select to save the game from the File menu, after which they can either choose to continue playing or exit the game. The user can reload a saved game while they are playing or when the program has just started. The text label then displays a message saying “Welcome back!” and the mode of the Master Ship when the game was saved, and the display panel is also restored to the saved state.



When the game is over, the JLabel grid is replaced by the Game Over image, text label set with text “MasterShip destroyed – GAME OVER” and the ‘Make a Move’ and ‘Change Game Mode’ buttons disabled. The user can choose to start a new game, reload a saved one or close the program.

Programming Technique Not Taught During the Module

I used the Factory Pattern to create the Enemy Ships (as described above), which is one of the most commonly used Creational design patterns in Java. By using this pattern, random Enemy Ships can be easily created through the EnemyShipFactory class and the code kept simple and extensible. The Enemy Ship classes can be refactored or new ones added to the game without the code that uses them having to be changed.

Save and Reload Game Status

To add the feature of saving the game status I first made all the classes containing instances to be saved to implement the interface Serializable and generated serialVersionUIDs for each one. I then wrote a *saveGame()* method in the GameState class, which first stores the current game mode and game mode message into the grid and then serializes the current Grid object. The serialization process translates the data into a storable format, which can be used to recreate the object in memory.

The *loadGame()* method first checks if a saved game file exists and if it does, deserializes the data and restores the Grid object to the state it was last saved.

The user can save or reload a game from the GUI whenever they want by selecting the ‘Save Game’ or ‘Load Saved Game’ options from the File menu.

Master Ship Mode

The Master Ship has two modes, defensive and offensive. I added the gameMode variable in the GameState class to store the current Master Ship mode and declared the variables defensiveMode as 0 and offensiveMode as 1. I set the gameMode as defensive when the game starts (in the *initialiseNewGame()* method) and wrote the *changeGameMode()* method to enable the game mode to be changed during the game. It checks the current game mode and, if it is defensive, sets the gameMode variable to offensive, and vice versa.

The gameMode variable is used in the *checkSquares()* method to detect the current game mode and destroy ships accordingly. If the game mode is defensive and there are two or more Enemy Ships in the same square as the Master Ship, the Master Ship gets destroyed and the game is over. In offensive mode it takes three or more Enemy Ships to destroy the Master Ship.

The user can change the Master Ship mode whenever they want during the game by pressing the ‘Change Game Mode’ button on the GUI.

Observer Pattern

I used the Observer pattern to update the text label on the GUI whenever the Master Ship mode is changed. I created an Observable interface (with *registerObserver(SkyWarsGUI gui),* *removeObserver(SkyWarsGUI gui)* and *notifyObservers()* methods), which is implemented by the GameState class, and a GameModeObserver interface (with *update(String gameModeMessage)* method), which is implemented by the SkyWarsGUI.

The GameState class has a variable gameModeMessage, which stores either “MasterShip Mode: DEFENSIVE” or “MasterShip Mode: OFFENSIVE” when the Master Ship mode is changed in the *changeGameMode()* method. The *notifyObservers()* method is then called which in turn calls the *update(String gameModeMessage)* method in the SkyWarsGUI class. The new game mode message is passed in and the text label on the GUI updated accordingly.

Junit Test Suite and JAR File

I have provided a Junit test suite for the main classes and methods in my program, which can be found inside the ‘test’ folder. A JAR file of the program can be found inside the ‘lib’ folder.

References:

Johari, A. (2017). *Object Oriented Programming: Java OOPs concepts with examples*. [online] Available at: <https://www.edureka.co/blog/object-oriented-programming/> [Accessed 17th April 2018]