# Skywars

The game takes place on a two dimensional, square grid. A player can input his name and select the size of the grid he wishes to play on. The game starts with the Master Ship in the top left corner of the grid. A panel below the grid shows the players name and score on the left and a grid of nine buttons on the right which the player can use to move the Master Ship around the grid. Each button press will move the Master ship one square in the relevant direction either vertically, horizontally or diagonally. The centre button will leave the Master Ship in the same location.

Every turn, there is a one in three chance that an enemy ship will appear. These come in three types:

* Battle Stars: these move in a random direction each turn and are twice as likely to appear as either of the other two ships
* Battle Cruisers: these always move towards the Master Ship
* Battle Shooters: these always run away from the Master Ship, unless they are against the upper or left border in which case they move away from the border
* Battle Shooters: these always run away from the Master Ship, unless they are against the upper or left border in which case they move away from the border

If the Master Ship lands on a square with one other enemy, that enemy Ship is destroyed. If the Master Ship lands on a square with two or more enemy ships, the game is over.

Each type of enemy ship is worth a certain number of points, and the players score will increase every time he destroys one.

## Design

The game takes place on a Grid. Grid objects have a size (int) and a gameGrid which is two dimensional array of objects of type Square. each square has two integers (int), it's x and y coordinate on the grid, and a HashSet of ship objects. Each ship as a postion, of type Square, a name, of type String, and a gameGrid, of type Grid. The constructor for Ship requires the input of a grid; forcing ships to to have a particular grid makes for convenience when moving ships around the grid, as the methods for movement of the ships can call on the local gameGrid variable. Ships therefore have a many-to-one relationship with both the grid and a square; the relationship between ships and squares is two-way.

Grid has the method initializeGrid() which populates the gameGrid array with new Squares with corresponding x and y coordinates.

Movement of the ship makes use of the enum Movement. This consists of 8 values, for the eight possible directions, plus a 'non-movement' enum which was for convenience of constructing the GUI; this O enum also aids game play by allowing to stay in the same position for the next turn. A ship's move() method passes in a Movement enum; the method then retrieves the x and y component of the Movement and uses these to calculate the ship's next position. The new coordinates, and the ship, are then passed into the grid's put() method.

Initially, this method caught and ArrayOutOfBounds exception, but it proved more useful for the put() method to call on a private keepInBounds() method to validate the coordinates; if either coordinate is less than zero or greater than or equal to the grid's size, the ship is then placed on the boarder of the grid.

The the game play is carried out by a game object. A game variables include: theGrid (Grid), size (int), player (MasterShip), news (String), random (Random number generator), allEnemys (HashSet of EnemyShips) and gameOver (boolean). The Game constructor passes in a size and a player name (obtained from the GUI); gameOver is set to false, theGrid is initialized and a Master Ship (discussed below) is placed on the grid in the top left corner.

A go() method, which passes in a Movement enum, fires all the necessary actions for a round:

1. Check that gameOver is false.
2. Master Ship notifies it's Observers (Design Pattern, see below) before moving using the passed in enum.
3. moveEnemys() loops through allEnemys and fires their characteristicMove() (see below).
4. checkPlrSquare() switches the number of ships in the player's new position (Square); one ship means that only the player occupies that position, two ships means the player is sharing the Square with one EnemyShip therefore the enemy is destroyed. The only other scenarios would be the player sharing the Square with two or more enemies which means the game is over and so gameOver is set to true.
5. probNewEnemy() uses the random number generator to determine whether or not to produce a new enemy ship.

## Polymorphic Programming, Interfaces and Design Patterns

The class Ship is abstract and has two sub-classes: MasterShip and EnemyShip. EnemyShip is also abstract and has three subclasses: BattleStar, BattleCruiser and BattleShooter. Squares, therefore can have a HashSet of Ships and be able to hold MasterShips and any subtype of EnemyShip. In the same way, the allEnemys HashSet in Game can hold any subtype of EnemyShip. If more sub-classes of EnemyShip are created in future, no changes will need to be made to Game.

EnemyShip implements the Interface CharacteristicMove with method stub characteristicMove(). As EnemyShip is abstract, the enforcement of the method stub is passed to each of the sub-classes, but each sub-class can implement the method differently. This makes the moveEnemys() method in Game much easier to implement. Each characteristicMove() calls move() from the ancestor: BattleStars use a Random number generator to select a random Movement enum to pass in to move(). BattleCruiser and BattleShooter both implement an Observer Interface, based on the Observer Design Pattern: the Observer Interface enforces the update() method which is fired at the end of every round and passes the MasterShip's position to the Observers. The Observer's characteristicMove() then use this, and the vector() method from Movement to calculate their next Movement, and passes this into move(), either towards or away from MasterShip. The MasterShip implements Observable and has a an ArrayList of EnemyShips (which can therefore hold both type of Observer) and has the methods addObserver(), removeObserver() and notifyObservers() enforced.

MasterShip also employs the Singleton Design Pattern to ensure there is only ever one MasterShip.

## The GUI

The GUI is separated from the game logic as much as possible by hold an object of type Game and no other non-swing classes. The only methods used by the GUI which have any effect on the game is Game.go() and the constructor.

The Screen class extends JFrame and is comprised of two nested JSplitPanes. The outer JSplitPane has a vertical split; the top pane contains gridPane which is a JPanel with GridLayout and is made up of ScreenSquares (see below). The bottom pane holds the nested JSplitPane with horizontal split and displays game information on the left and movement buttons on the right. The ScreenSquare class extends JLabel and each instance 'looks at' a different Square in Grid; the getGridPane() method uses a nested for loop to iterate through the Squares in the game's grid and create ScreenSquares looking at the corresponding Grid Squares arranged left to right, top to bottom. The movement buttons (JButtons), in the right hand pane of the nested JSplitPane, are created in a similar way but using one for loop to iterate through all the Movemnt enums and place them in a three by three GridLayout. Each button passes the relevant enum to Game.go() and fires the updateGrid() method which iterates through all the ScreenSquares firing their update() method. ScreenSquare.update() checks the number of ships in the relevant Grid Square and displays and IconImage of either a starry background, an image of the relevant ship (calling the ships imagePath() method) or a 'fleet' of ships if there is more than one ship in the Square.

The infoPane (bottom left) is another JPanel displaying the player's name, news (displays text when an enemy ship is destroyed) and the players score.

Four methods of note in Screen are, initiate(), startGame() and updateInfo():

* initiate() fires when the Screen first runs and displays a startScreen (extends JPanel). This asks the user to input their name and select the grid size they would like to play on.
* The 'start' button on the startScreen fires startGame() passing the player name and the size. This instantiates a newGame and changes the content pane to the JSplitPane with all it's nested and child elements.
* updateInfo() updates the score and displays any news from Game. It also checks the Game's gameOver boolean; if 'true', the top pane is set with an image displaying "Game Over".
  + NB At this point, the movement buttons will not function as the go() method is essentially disabled as it only carries out the relevant actions if gameOver = false.

## Advanced Features

Design Patterns - The Observer and The Singleton.

Variable Grid size – The user selects the grid size from a drop down before playing the game. Values selectable range from 4 to 9, however a nine by nine grid may be impractical on some screens.

Untaught Element – There is no feature which substantially employs elements untaught in the module. However, the GUI does import images using ImageIcons coupled with the getClass() method to create the file path of the image which were not covered in the module.