SOEN 6441- Advanced Programming Practices

Summer 2019

Architectural design for Battleship Game Project

Submitted to: Prof. Nagi Basha

Team 2

Gurleen Sethi	40079565
Sandeep Singh	40043110
Vaibhav Malhotra	40079373
Karandeep Karandeep	40104845
Navjyot Kaur Minhas	40073551

Table of Contents

1. Introduction

1.1 Purpose of the Document

This document states about the architectural design model used to implement the battleship game. The design model implemented is **Model-View-ViewModel (MVVM).** The underlying approaches are the known approaches which are used for the smooth and effective game development.

- **Programming Concepts**: Some of the well-known programming practices are followed to maintain effectiveness of project development. For example, pair programming, where two programmers worked on same workstation to maintain productivity.
- **Simple Design**: Simple workflow of design made it easier to read and avoid faults. The simplicity in design lets us to understand and maintain the code effectively.
- **Continuous Integration**: Github is used as version control for the project where all programmers are made to work with different branches and commit accordingly. Maintenance of frequent changes, rollbacks helped increase the productivity by automatic integration.
- **Coding standards**: Simple and understandable coding standards are followed including naming and file organization conventions.

1.2 Scope of Document

This design document covers the development of basic battleship game using different builds featuring different requirements of the game.The detailed explanation of design are covered in different sections below. The motives of design decisions made are crucial for implementing the entire project and it also help effective build. This document will cover overall project architecture by explaining from the base until final build process.

2. Architecture Overview - MVVM

Model-View-ViewModel (MVVM) provides a way to separate the GUI (View) from business logic (Model) of the application. The ViewModel is responsible for exposing model data to the view in a way that they are easily manageable/presentable.

Model: The layer or the domain layer contains the core logic of the application.

View: The layer that is responsible for the appearance of what the user sees on the screen.

ViewModel: The layer is responsible for exposing data to view layer. View layer binds to the data provided by this layer. This layer tends to map the concepts of data towards the concept of view layer.

2.1 Why MVVM is a better choice than MVC or MVP?

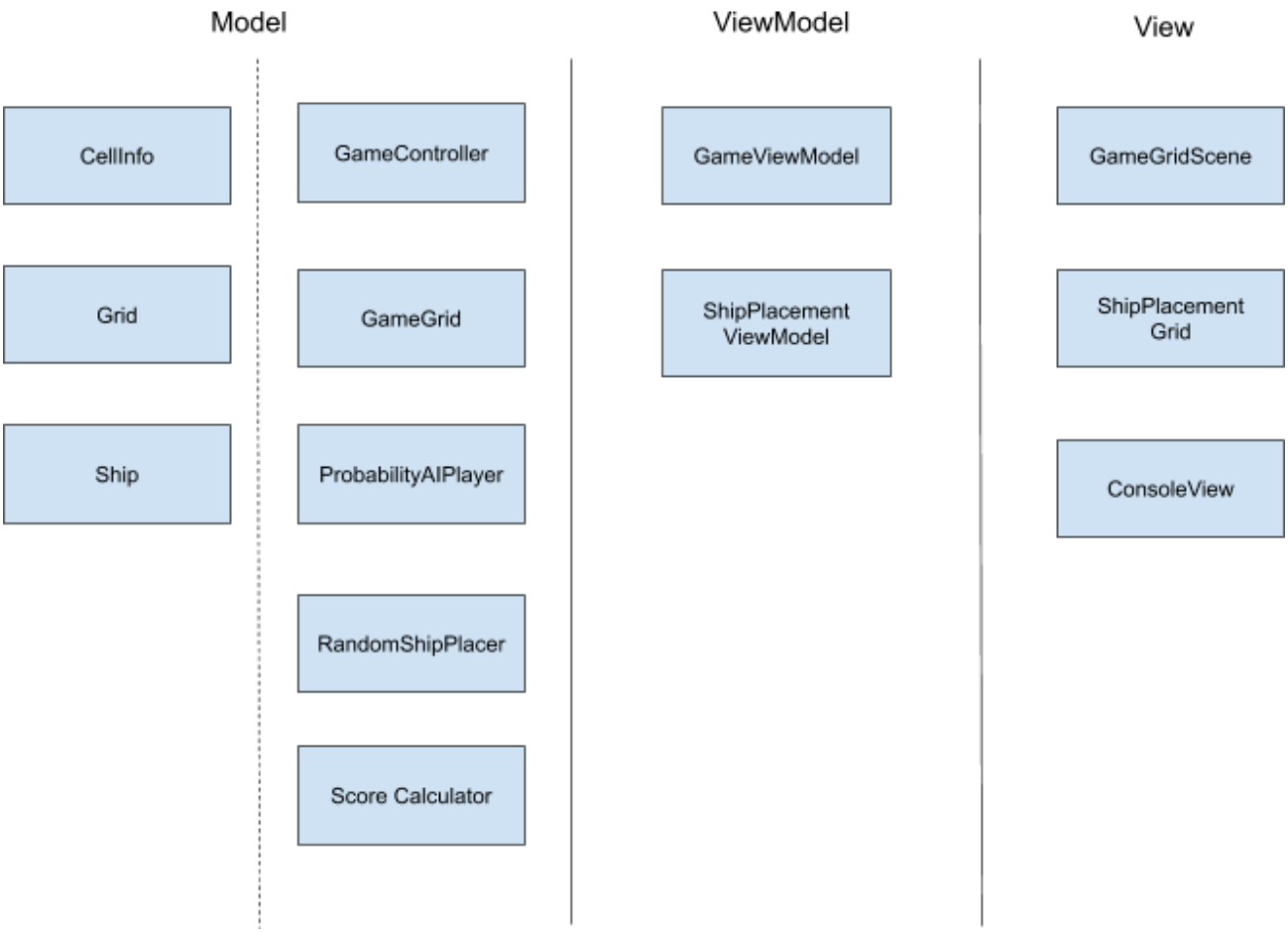
The obvious benefit that MVVM provides over MVC is that the View layer is independent of Model layer. In MVC the View layer is tightly coupled with other layers, which makes testing of the views tedious and sometimes leads to application logic creeping into the view player itself, whereas in MVVM the View is independent of core business logic of the application.

A practical example of proving the independence of View layer is its replacement. View layer in MVVM can be easily be changed without having any effect of the core logic. To prove this, we added the ability to play the game from console as well, thus replacing our GUI.

MVVM leads in a better separation of concerns. Transparent communication between layer is achieved, view layer doesn't know what is being communicated between Model and ViewModel, similarly Model layer has no Idea how View layer is being served by the ViewModel.

2.2 Project breakdown in MVVM

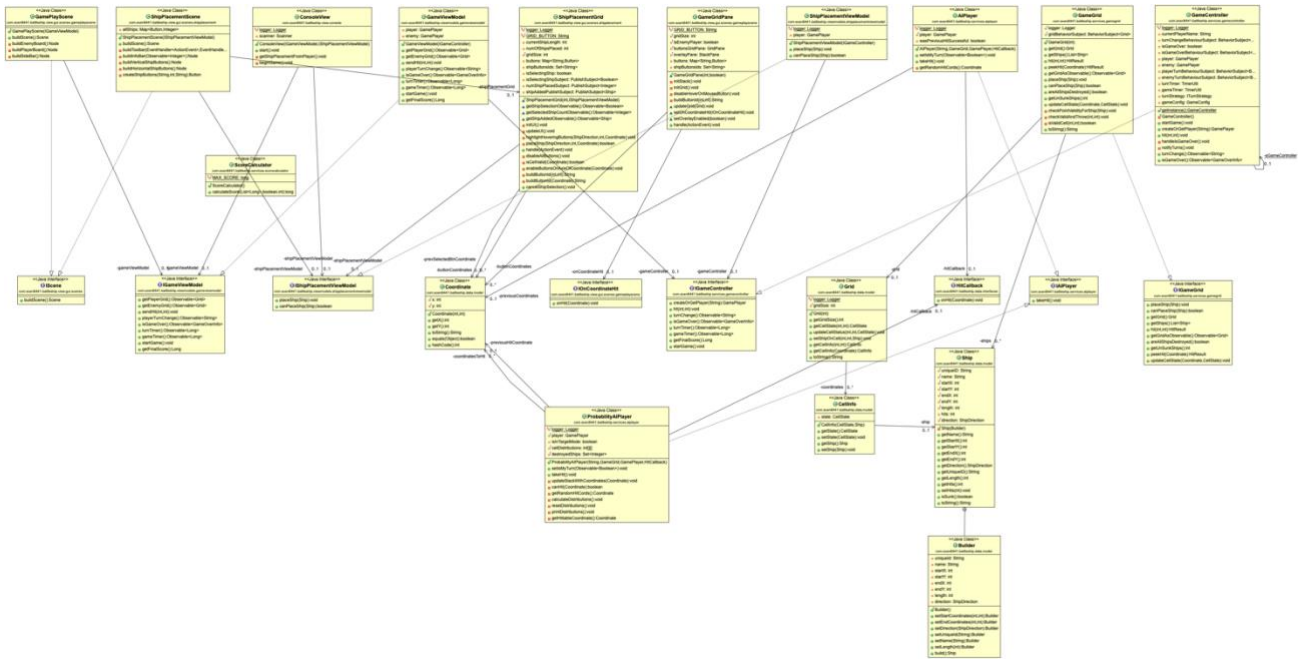
Below is a representation of how some of the classes from the project fit into the various layers of MVVM.



The model layer itself can be broken down into sub layers, one containing the logic and the other containing data model and access.

3. UML Diagram

Click to Zoom



Class Diagram

4. Class Structure

4.1 Common Classes

SceneRoutes	Contains the routes for navigator class to navigate between scenes.

4.2 Data classes

CellInfo	Contains information for a single cell on the grid.
Coordinate	Class to hold x-y coordinates.
GameOverInfo	Class to hold information required when the game is over.
Grid	Class to represent a grid on xy plane.
Ship	Class to represent a ship and all its properties.

4.3 Enums

CellState	Denotes the current state of a cell, whether the cell is empty, has a hit, as a ship that has been hit or destroyed.
Direction	Represents a direction on xy plane. Up, Down, Left, Right.
HitResult	Represents the result of an attempted hit on grid.
ShipDirection	Direction the ship is laid on the grid. Vertical or Horizontal.

4.4 Exceptions

CoordinateOutOfBoundException	The pair of x,y coordinate are out of bounds of a plane.

DirectionCoordinatesMismatchException	Directions deduced from start and end coordinates don't match the
InvalidRouteException	Invalid route passed in SceneNavigator.
InvalidShipPlacementException	Trying to place ship on coordinates that are either invalid or already have a ship on it.
NavigatiorNotInitialisedException	Singleton instance of Navigator is not initialised before use.

4.5 Service Classes

IAIPlayer	Interface to represent an AI player.
AIPlayer	Concrete implementation of IAIPlayer. Class for computer player option, keeps record and predicts the best possible next move.
IGameController	Interface to represent game controller.
GameController	Concrete implementation of IAIPlayer. Driver class for the game. Implements IGameController to perform.
IGameGrid	Interface to represent a game grid.
GameGrid	Concrete implementation of a IGameGrid. Wraps the Grid class and adds logic to it.

4.6 Utility Classes

RandomShipPlacer	Provides utility to place random ships on a GameGrid.
GridUtils	Provides utility to support console version of the grid.

4.7 View Classes

ConsoleView	Display class for Console version of the game.
SceneNavigator	Helper class to navigate between Scenes.
GameGridPane	Class to display and update grid view
GamePlayScene	Class to setup gameplay
IOnCoordinateHit	The interface on CoordiateHit.
InitialUserInputScene	Class to implement Welcome screen XML file.
UserInputSceneController	Controller class to implement Welcome screen functions.
ShipPlacementGrid	Class logic for placement of Players ships.
ShipPlacementScene	Class to display Ship placement screen.
IScene	Interface to build scenes.
ISceneBuilder	Functional Interface to build scenes.
GUIView	Starting class of the application. It loads all the scenes from initial user input to game play scene.
IView	Interface to abstract view layer.

4.8 View Model Classes

IGameviewmodel	Interface for Gameview Model to take data from data.
Gameviewmodel	This class takes data from model and passes over to observer. The view model takes data from this class and displays on GUI.
IPlayerviewmodel	Interface of the player which takes data from model and displays the updated data on user screen.

IShipplacementviewmodel	Interface for the place ship method which validates the data from controller and passes to model.
ShipPlacementViewModel	This class takes data from model and passes the updated data from the view and vice versa

4.9 Main Classes

App	Main start to the application and has the main method, it sets the primary stage for the game and passes to AppwithGui class.
AppWIthGUI	Sets the connection between the controller and view model.
GamePlayer	Class to instantiate the player type from which both human player and AI player will take properties.

5. Functional Requirements

- ✓ Placement of Ships
- ✓ Respecting constraints of placement of ships
- ✓ Switching turns
- ✓ Verification of invalid moves or repeated moves
- ✓ AI Engine
- ✓ Announcement of Winner

5.1 Placement of Ships: The entry point of application is the placement of ships. The user of the application is given a window with options to select the points on grid to place ships. The user gets 5 sizes of ships and can place the ship anywhere on the screen with the conditions that ships cannot overlap each other, cannot go out of the grid and, can only be either vertical or horizontal. The first user has to select the starting edge of the ship and then he will be given choices in each direction to place the end point of the edge of the ship. When the placement of all ships is done, then the user is taken to the next screen which is the game play scene. The ship placement on the computer side is automated but following the same constraints that ships cannot overlap and cannot go out of the grid and can only be either vertical or horizontal. The RandomShipPlacer is the class which places the ships for computer/AI.

5.2 Respecting constraints of placement of ships: The constraints which are being followed in placing ships for both computer player and human player are as follows :

a. Ships cannot overlap each other.
b. Ships cannot be placed out of the grid size.
c. Ships can only be placed either vertically or horizontally.

5.3 Switching turns: The switching turns for players is very basic following the original rules of the game. One player starts the game, and he/she got the chance to hit the enemy ship cell. If it is a hit on the enemy ship, then they get another chance to hit on enemy, and if they miss the hit. i.e., enemy ship is not present on the hit then player 2 gets the turn and follows the same rules as player one. The turns finish when either of the players is out of ships, and the opponent wins. To be noted that, when either of the players is making a move then the opponent’s screen is overlapped by a white screen to enforce fair play in the game.

5.4 Verification of invalid moves or repeated moves: When either of the players hit on the same spot on which they already hit before then that hit is counted as valid move and turn is given to the opponent.

5.5 AI Engine:

Build 1: The first hit which AI makes is chosen randomly. If a hit is successful, it checks the surrounding cells for ship placement and move towards that direction. This continues till the moves are valid and ships are being found. Once there are no ships in the path, it selects another coordinate at random and continues with the same process.
Build 2: Probability distribution algorithm is used to implement the AI Engine. This is a much-improved technique compared to what was being used in Build 1. This algorithm tries to predict the most probable location of a ship based on superposition of all possible locations of an enemy ship.

5.6 Announcement of Winner: The first one to run out of ships is announced to lose the game. The total number of ships are 5 and in order to destroy a whole ship, the player has to hit on every cell of the ship.

6. Refactoring

1. Turn refactoring in GameController: Build 2 required salva variation of battleship. Turn change logic was being handled in GameController.

```
HitResult result = enemy.getGameGrid().hit(x, y);

if (result == HitResult.MISS || result == HitResult.ALREADY_HIT) {
  aiPlayer.takeHit();
}

turnChangeBehaviourSubject.onNext(this.currentPlayerName);
```

We refactored the above piece of code and delegated the logic of handling turn variation into strategy classes. Turn handling is now represented using strategy pattern, namely there are two types: SimpleTurnStrategy and SalvaTurnStrategy. Both these classes implement the ITurnStrategy interface. GameController depends ITurnStrategy and is not dependent on concrete implementations. This makes the design more extensible, allowing us to add any kind of turn strategy without having to change GameController again.

```
HitResult result = this.turnStrategy.hit(playerToHit, new Coordinate(x, y));

GamePlayer playerToSwitchTurnTo = this.turnStrategy.getNextTurn(player, enemy, result);
```

2. Ship placement constraints: Build 2 required to update ship placement constraints where in two ships cannot be placed alongside each other. Ship placement constraints are being handled in a method named checkPointValidityForShip(Ship ship). The checkPointValidityForShip had unit test for the placement, which helped the refactoring faster and we were more confident about it. Also logic for this is now directly being used in UI grid placement as well. First UI placement had its own logic, but now this logic is used to directly to show user where ship can be placed validly.

3. AI Player: Build 2 required an improved AI. We used probability distribution algorithm to predict the most probable location of a ship based on superposition of all possible locations of an enemy ship. We added the algorithm to a new class instead of modifying existing AI. We used a common interface IAIPlayer to merge the functionality of both the classes. This makes the design more flexible, allowing us to add and test new strategies without having to impact the main AIPlayer class.

7. Bibliography

https://www.thesprucecrafts.com/the-basic-rules-of-battleship-411069