Project

Developing Battleship Game using Advanced Programming Practices

Build 3

Submitted to:

Instructor: Nagi Basha
Department of Computer Science & Software Engineering
Concordia University, Montreal, QC

In Fulfillment
of the Requirements
for SOEN 6441- Summer 2019
Advance Programming Practices

1.0 Objective

To design and develop a Battleship game using agile software development model and MVC architecture with Observer design pattern.

1.1 Team Members

1. Arsalaan Javed - 40085994
2. Prateek Narula - 40091466
3. Sagar Bhatia - 40076907
4. Mridul Pathak - 40078157
5. Mehak Jot Kaur - 40070845

2.0 PROCESS

2.1 Agile Software Development Model

Agile model is a combination of iterative and incremental process model that focuses on small releases aka **Builds** that focuses on only a subset of requirements for software development. When one build completes, we refactor the code for the second build and the cycle continues till all the requirements are completed. This helps in coping up with any changes in the requirements even at later stages of the development process.

2.2 MVC Architecture

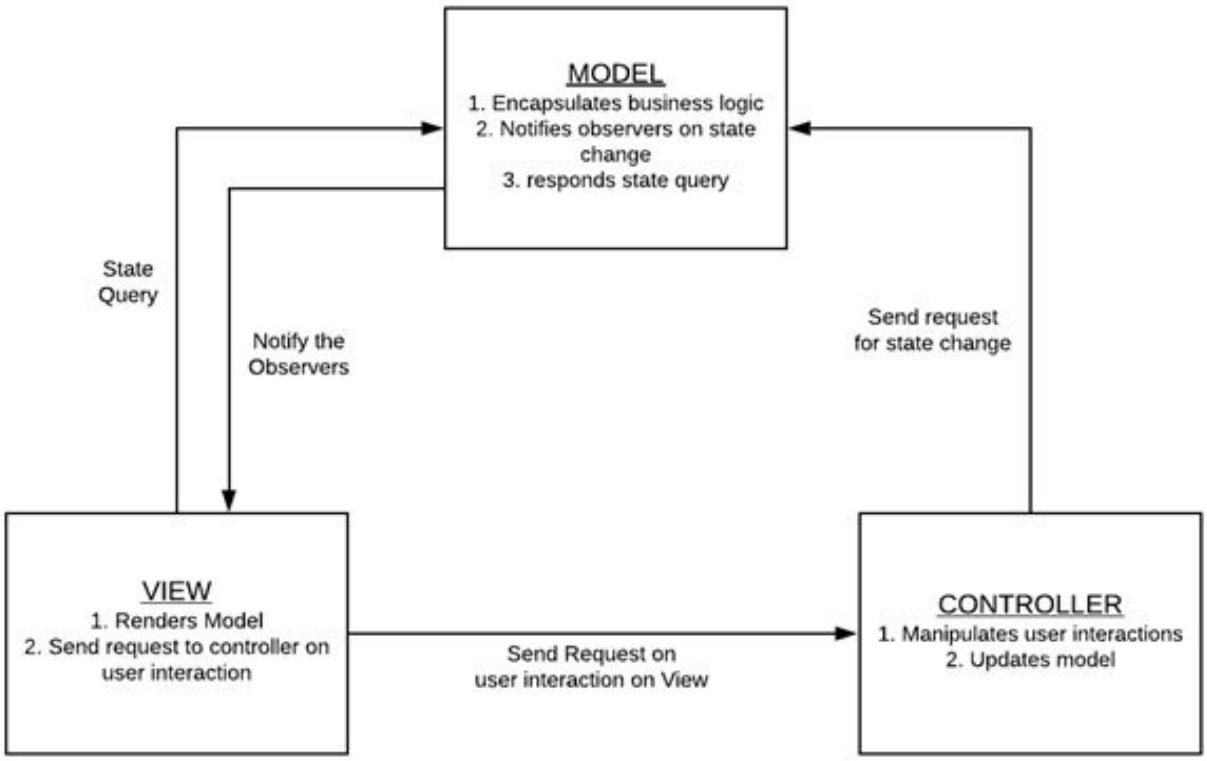
MVC separates the application into three components - Model, View and Controller.

Model: Model represents the shape of the data and business logic. It maintains the data of the application.

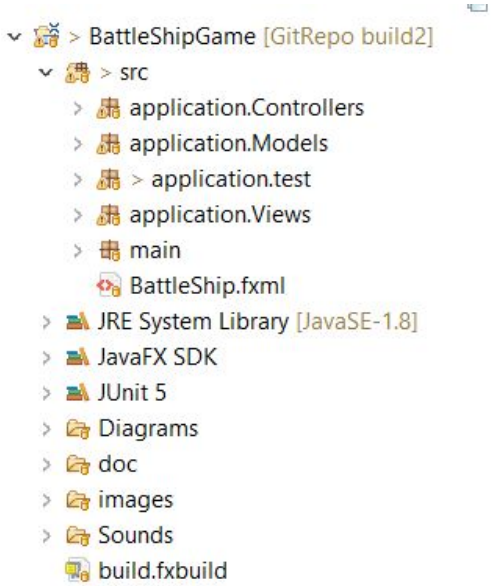
View: View is a user interface. View display data using model to the user and enables them to modify the data via controller.

Controller: Controller handles the user request. Typically, users interact with View, which in-turn raises appropriate request, this request will be handled by a controller and appropriate request is then send to Model for state change.

2.2.1 MVC architecture diagram



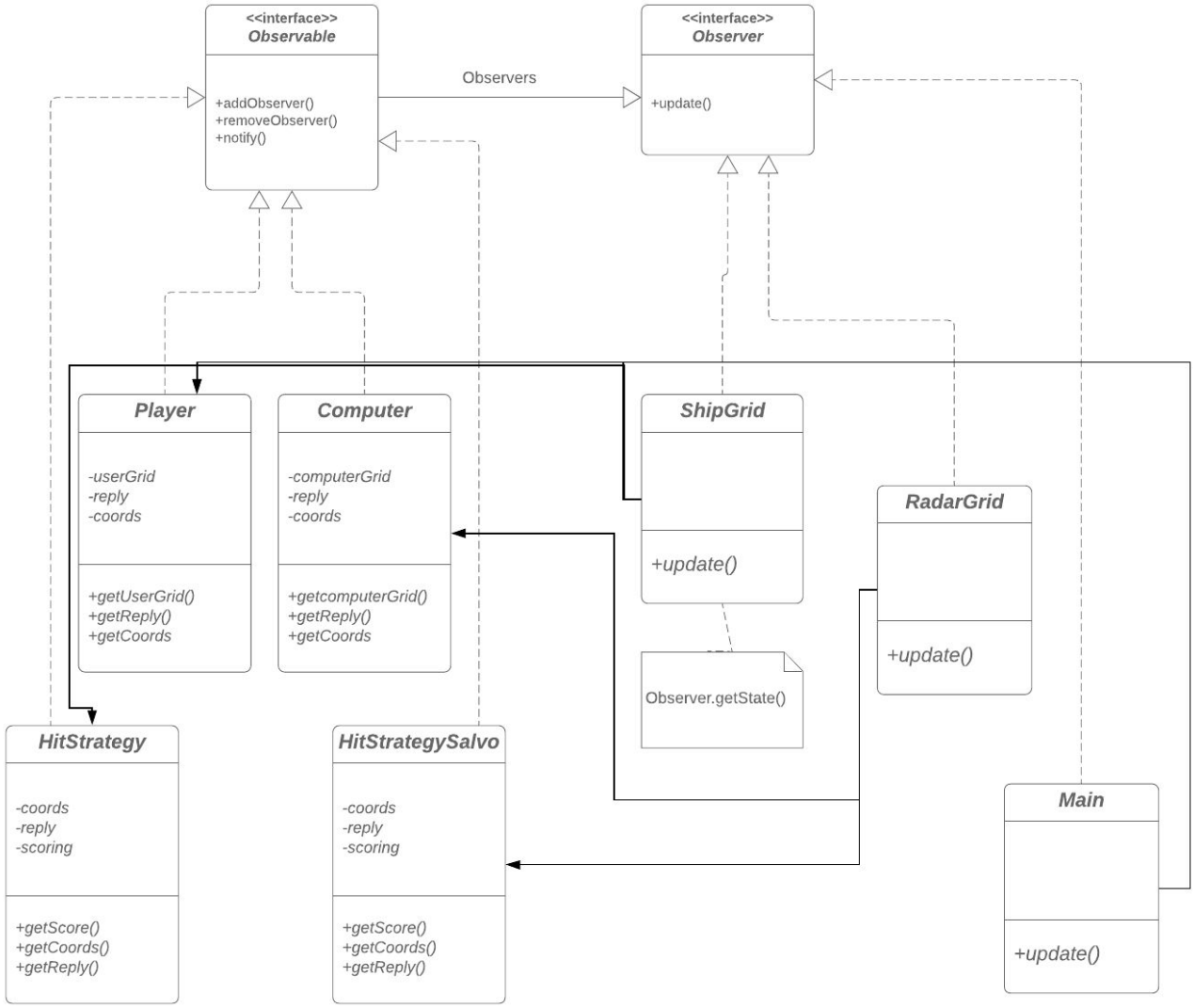
2.2.2 Project Directory Structure



2.3 Observer Design Pattern

The observer pattern is a software design pattern in which an object (observable) maintains a list of its observers, and notifies them automatically of any state changes. Then the observers make a state query and fetches the data.

In the project Observer design pattern is implemented with MVC architecture. Model classes are the subject and implements Observable interface. View classes are the observers and implements Observer interface. The following class diagram illustrating the Observer pattern followed in the project.



2.4 Class Diagram

Model :

HitStrategy.java : This class carries all the three strategy functionality of hitting user grid for computer turn.

HitStrategySalvo.java : This class has the strategy of computer turn in salvation mode.

Computer.java : This class maintains computer data and all of its logic. It serves as an observable for RadarGrid.java class.

Player.java : This class maintains user data and all of its logic. It serves as an observable for ShipGrid.java class.

LoadClass : This class helps in retrieving the saved data of the user for a particular game.

SaveClass : This class helps in saving the data of the user for a particular game.

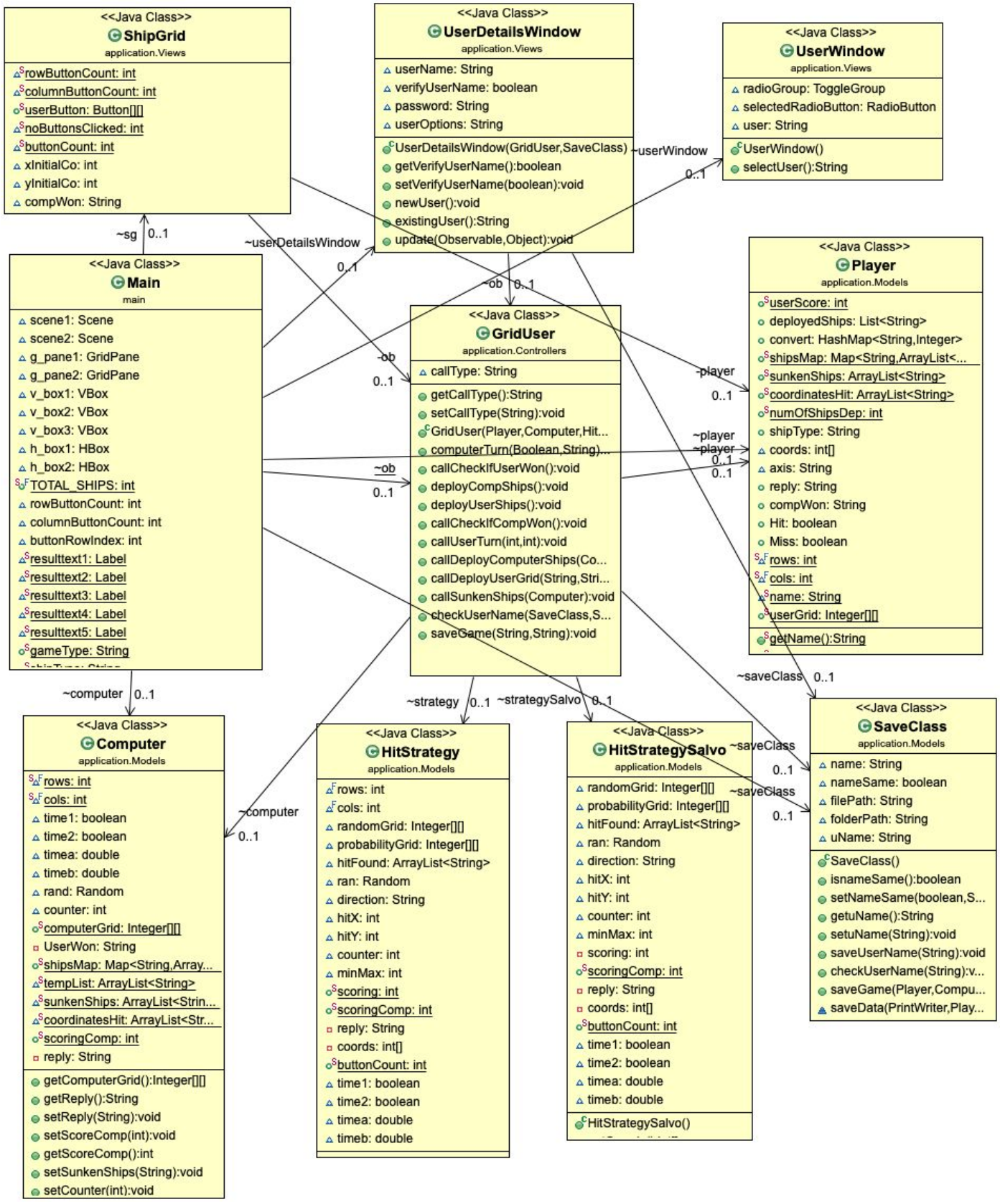
Controller :

GridUser: This class routes the request from views and direct to proper model class.

View :

ShipGrid.java : This is a view class that keeps the view related to user grid and its action listeners.

RadarGrid.java : This is a view class that keeps the view related to computer grid and its action listeners.


<<Java Class>>
ShipGrid
application.Views
rowButtonCount: int
columnButtonCount: int
userButton: Button[][]
noButtonsClicked: int
buttonCount: int
xInitialCo: int
yInitialCo: int
compWon: String
<<Java Class>>
UserDetailsWindow
application.Views
userName: String
verifyUserName: boolean
password: String
userOptions: String
UserDetailsWindow(GridUser,SaveClass)
getVerifyUserName():boolean
setVerifyUserName(boolean):void
newUser():void
existingUser():String
update(Observable,Object):void
<<Java Class>>
UserWindow
application.Views
radioGroup: ToggleGroup
selectedRadioButton: RadioButton
user: String
UserWindow()
selectUser():String
~userWindow
0..1
~sg
0..1
~userDetailsWindow
0..1
<<Java Class>>
Main
main
scene1: Scene
scene2: Scene
g_pane1: GridPane
g_pane2: GridPane
v_box1: VBox
v_box2: VBox
v_box3: VBox
h_box1: HBox
h_box2: HBox
TOTAL_SHIPS: int
rowButtonCount: int
columnButtonCount: int
buttonRowIndex: int
resulttext1: Label
resulttext2: Label
resulttext3: Label
resulttext4: Label
resulttext5: Label
gameType: String
~ob
0..1
<<Java Class>>
GridUser
application.Controllers
callType: String
getCallType():String
setCallType(String):void
GridUser(Player,Computer,Hit...
computerTurn(Boolean,String)...
callCheckIfUserWon():void
deployCompShips():void
deployUserShips():void
callCheckIfCompWon():void
callUserTurn(int,int):void
callDeployComputerShips(Co...
callDeployUserGrid(String,Stri...
callSunkenShips(Computer):void
checkUserName(SaveClass,S...
saveGame(String,String):void
-ob
0..1
-player
0..1
~player
~player
0..1
0..1
<<Java Class>>
Player
application.Models
userScore: int
deployedShips: List<String>
convert: HashMap<String,Integer>
shipsMap: Map<String,ArrayList<...
sunkenShips: ArrayList<String>
coordinatesHit: ArrayList<String>
numOfShipsDep: int
shipType: String
coords: int[]
axis: String
reply: String
compWon: String
Hit: boolean
Miss: boolean
rows: int
cols: int
name: String
userGrid: Integer[][]
getName():String
~computer
0..1
~strategy
0..1
~strategySalvo
0..1
~saveClass
0..1
<<Java Class>>
Computer
application.Models
rows: int
cols: int
time1: boolean
time2: boolean
timea: double
timeb: double
rand: Random
counter: int
computerGrid: Integer[][]
UserWon: String
shipsMap: Map<String,Array...
tempList: ArrayList<String>
sunkenShips: ArrayList<Strin...
coordinatesHit: ArrayList<Str...
scoringComp: int
reply: String
getComputerGrid():Integer[][]
getReply():String
setReply(String):void
setScoreComp(int):void
getScoreComp():int
setSunkenShips(String):void
setCounter(int):void
~computer
0..1
<<Java Class>>
HitStrategy
application.Models
rows: int
cols: int
randomGrid: Integer[][]
probabilityGrid: Integer[][]
hitFound: ArrayList<String>
ran: Random
direction: String
hitX: int
hitY: int
counter: int
minMax: int
scoring: int
scoringComp: int
reply: String
coords: int[]
buttonCount: int
time1: boolean
time2: boolean
timea: double
timeb: double
<<Java Class>>
HitStrategySalvo
application.Models
randomGrid: Integer[][]
probabilityGrid: Integer[][]
hitFound: ArrayList<String>
ran: Random
direction: String
hitX: int
hitY: int
counter: int
minMax: int
scoring: int
scoringComp: int
reply: String
coords: int[]
buttonCount: int
time1: boolean
time2: boolean
timea: double
timeb: double
HitStrategySalvo()
~saveClass
0..1
~saveClass
0..1
<<Java Class>>
SaveClass
application.Models
name: String
nameSame: boolean
filePath: String
folderPath: String
uName: String
SaveClass()
isnameSame():boolean
setNameSame(boolean,S...
getuName():String
setuName(String):void
saveUserName(String):void
checkUserName(String):v...
saveGame(Player,Compu...
saveData(PrintWriter,Play...

3.0 AI Implementation and Logic

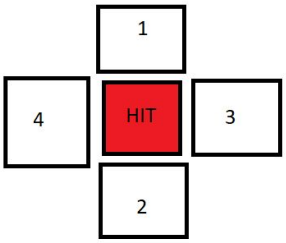
The AI implementation illustrates the strategy followed by the computer to try to hit the user ships. In this project the AI code is in the java class HitStrategy. The logic is further divided into three modes:

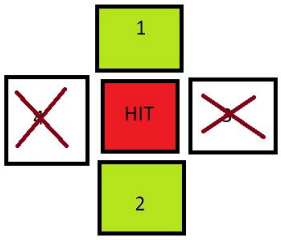
3.1 Easy Mode

In this mode the HitStrategy class will return the x and y coordinates generated randomly over the user ship grid and so the probability of computer to win is less.

3.2 Medium Mode

In this mode the HitStrategy class will initially return a random hit and will keep the state of the last hit saved in the class. As per the result of the last hit result the class will decide to either get a new random hit, if the result of the last hit is a "Miss" or will *return the neighbours* of the last hit in case of a "Hit".





Also, the class will check if it gets coordinates for option 1 and 2 as hit then it will not go for option 3 and 4 (as the class will know that the ship will be placed vertically).

3.3 Hard Mode

In the Battleship game, using the ships that are placed we have 5+4+3+3+2= 17 spaces shaded on a 9 * 11 grid (99 spaces to be filled). Therefore there is a (17/99) = **82.829%** chance of missing our opponents ship on our first call and **17.17%** chance of hitting our opponents ship on our first call.

So we have implemented a more *intelligent AI* logic:

In this we are storing a history of the combinations of ways all the ships can be placed on the grid and maintaining a probability matrix. We are training the probability matrix with **1000** ways of placing ships on the grid. The probability matrix will signifies the probability of a ship to be present at that grid location.

86	112	104	121	154	158	166	158	164	149	101
122	141	129	143	172	160	161	147	163	158	127
174	172	161	180	197	186	195	183	177	162	165
189	171	172	184	196	164	202	194	166	158	178
254	196	176	186	183	166	203	212	189	193	211
283	213	207	216	212	200	210	244	198	197	241
261	210	202	204	181	176	202	234	192	218	232
175	140	155	169	171	160	187	205	152	176	176
74	97	108	120	132	131	120	132	116	115	65
68	87	107	111	117	133	131	111	108	111	66
102	121	142	145	161	159	145	146	137	118	95
132	164	162	174	191	176	175	182	172	164	129
170	208	182	186	213	201	210	231	207	185	151
212	239	199	222	242	229	237	273	246	238	193
231	240	209	241	259	259	255	267	251	245	204
224	210	185	209	199	221	228	226	210	216	185
127	138	143	165	181	194	190	172	169	169	127
61	78	81	105	129	136	132	123	126	92	72

Above is an instance of the probability grid, that tells the probability of the grid location. The red grid location has the least probability i.e. only 61 times out of 1000 times it was used by a ship (0.61% chance of having a ship at a time). Similarly, the green location is most probable to a ship.

In hard mode, instead of choosing a location randomly the HitStrategy class will check for the most probable location to hit and will return the coordinates accordingly. Also, in hard mode we will keep track of the last hit and it will function in the same way as the medium mode when it gets a hit.

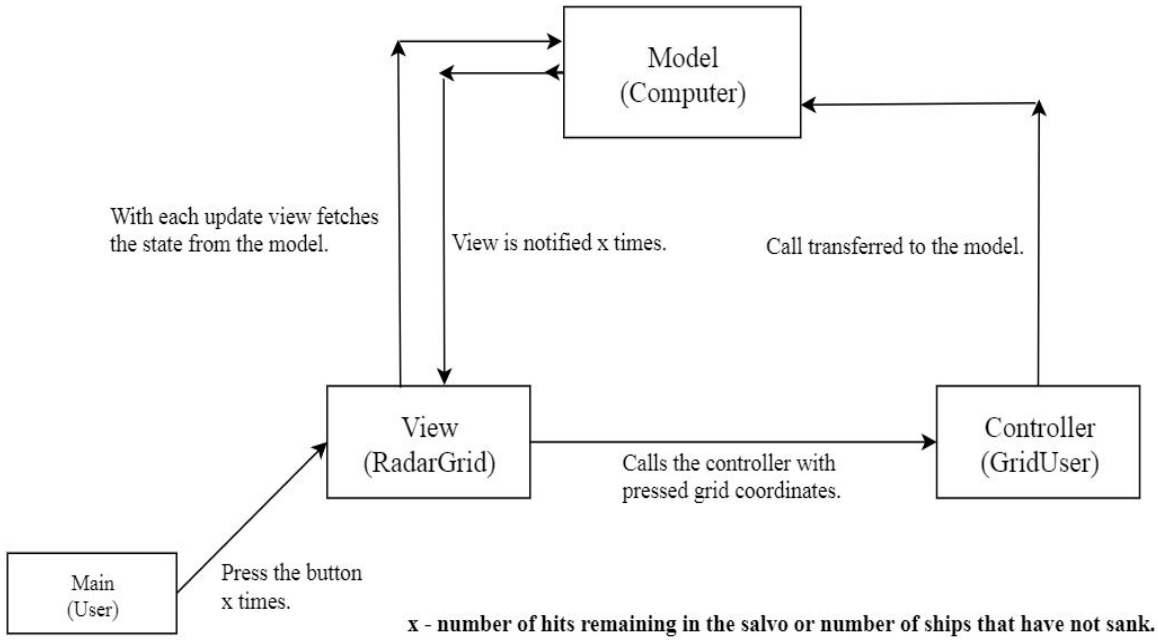
4.0 Salvo Mode

In build-2 , the task is to implement the *Salvo* mode in our battleship game. The rules implemented for the *Salvo* mode are as follows:-

- On the first round of the game, the user(*RadarGrid* view) call out five shots by pressing five different buttons on the *RadarGrid.*
- The calls will then be transferred to the corresponding model(*Computer*) via the controller(*GridUser*).

- Based on the state of the model with respect to where the ships are placed, the model(*Computer*) notifies the view observer(*RadarGrid*) that which buttons were hit(*red*) and which were the ones that were missed(*white*).
- The model also sends the state of ships in case if ship was drowned that is further displayed by the view.
- The above steps are again repeated when the computer plays its turn.
- This goes on till the number of shots in a *Salvo* are reduced to zero. The game is then finished.

Flow diagram to illustrate how the Salvo mode is incorporated in the existing MVC architecture:



In the above flow diagram, the value x, that is the number of hits to be made in the salvo are done are maintained with the *buttonCount* inside the application.

5.0 Scoring

The Scoring for the game is done on the following principles:-

- For each successful HIT a positive increment is done to the score.
- For each failed hit or MISS score is deducted.
- If the Player or Computer score multiple successive hits in under 3 seconds, they get bonus points(Twice more than the Hit points).

The Logic for the scoring is as follows:-

```
if (computerGrid[x][y] == 1) {
                    // change the grid value from 1 to 2 to signify hit

                    computerGrid[x][y] = 2;
                    // the grid changes to 2 signifying hit
                    if (!time1) {
                    //  time1 is set to false by default in declaration
                            timea = java.lang.System.currentTimeMillis();
                    // fetches current time when first hit is registered
                            time1 = !time1;
                    // time 1 is set to true
                    } else if (!time2) {
                    // time2 is set to false as well and for second hit it activates
                            timeb = java.lang.System.currentTimeMillis();
                    // fetches current time when second hit is registered
                            time2 = !time2;
                    // time2 is set to true
                    } else {
                            double t = timeb - timea;
                    // calculating time difference between first and second hit
                            if (t < 3000) {
                    // if time difference less than 3 seconds score Bonus Points
                                    setScoreComp(20);
                            }
                            time1 = false;
                            time2 = false;
                            timea = 0;
                            timeb = 0;
                    // reset all the parameters for next time we need to check consecutive hits
                    }
                    setScoreComp(10);
                    // incrementing computer score for a successful hit

            } else if (computerGrid[x][y] == 0) {
                    // if there is no hit, and grid registers a Miss
                    setScoreComp(-1);
```

6.0 Save and Load game

6.1 Save

The requirement for Build-3 of the project was to allow the user to save and load the state of the game. This was done with the help of the text files. For the save component, data is saved in a text file created by the user's name and all the users are stored in a *Users-List.txt* file. The saves in the user file are sorted on the basis of date and time, and certain states of the game are stored in the file that are useful while loading the game.

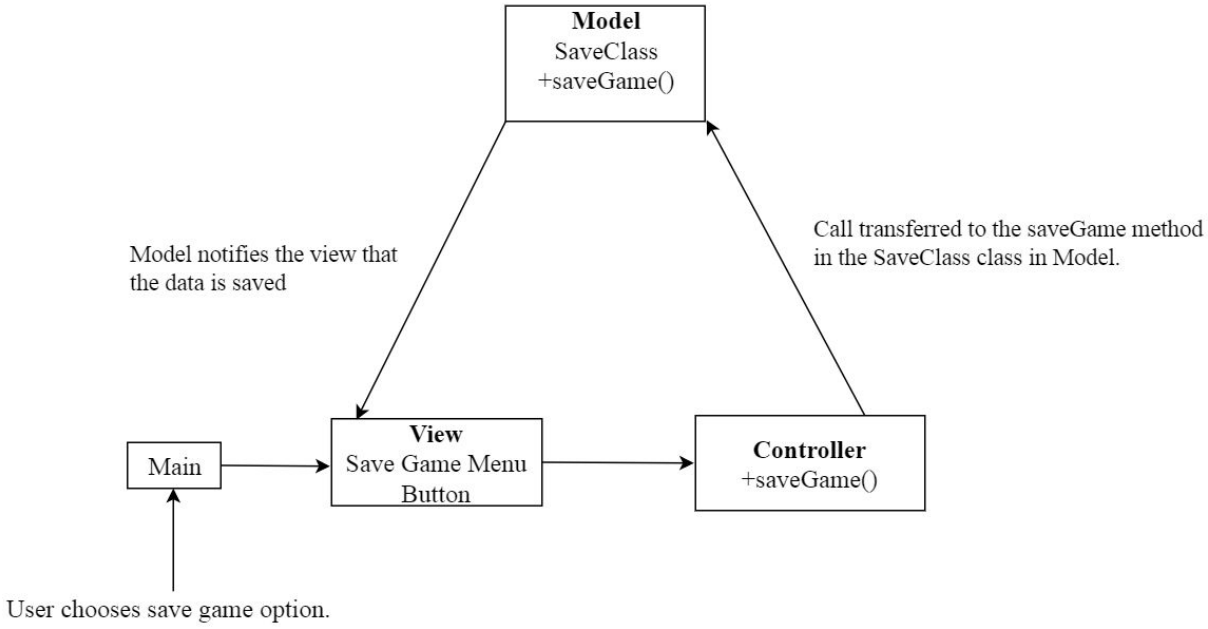


Figure: Representation of the process of saving a game in the MVC architecture.

6.2 Load

Another requirement with the save feature was to load the selected game stated by the user. Loading is done in a two-part process wherein, first the user is asked to select the save to be loaded based on the date and time. If no saves are there, then it is displayed that no saves are available. When the save to be loaded is selected then the game is loaded to the required state

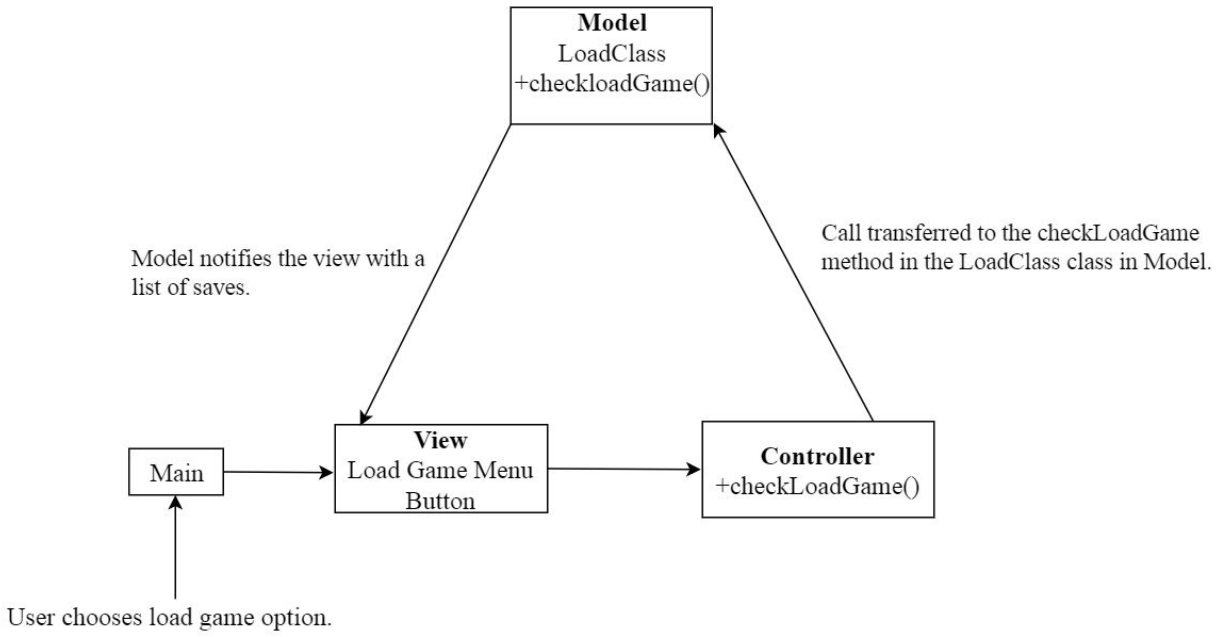


Figure: Representation of the part-one of loading a game in the MVC architecture.

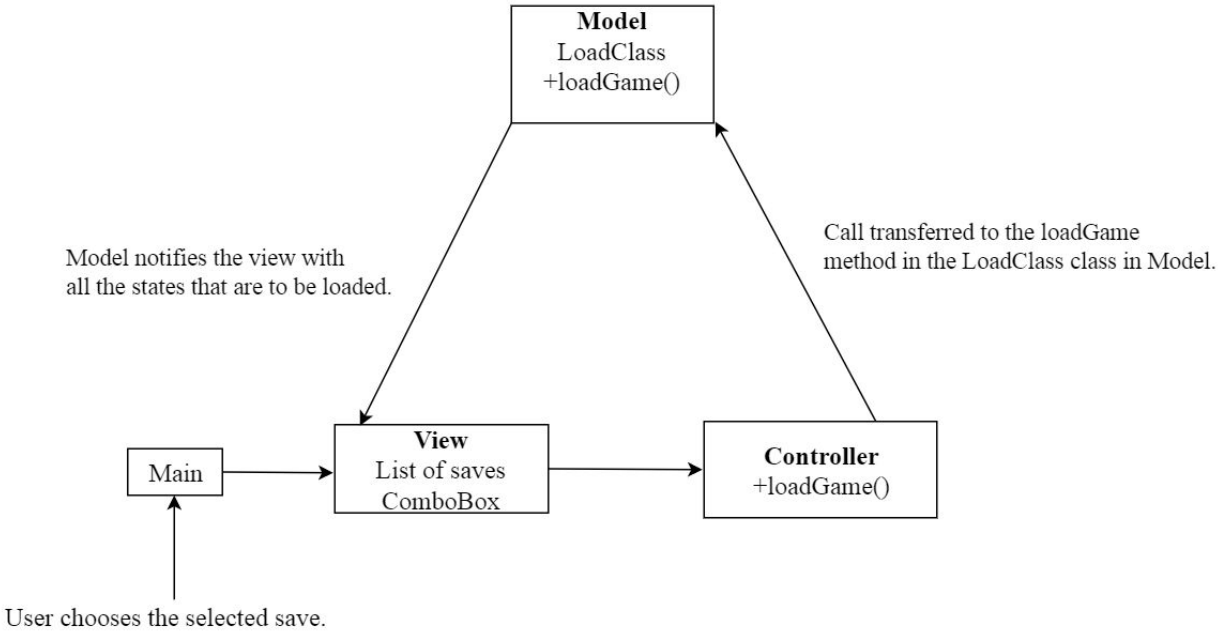


Figure: Representation of the part-two of loading a game in the MVC architecture.

7.0 Versus Player Mode

As per the requirement of Build 3, the game has been enhanced into a two player game battle. In which two people on the same network can play against each other. In our model, both will act as server for themselves. We used **Java socket classes** for sending **UDP** messages to each other server's.

The working of the code is as follows:

- On starting the game one system must select option as Player 1 which will start up the server for player 1 and the server will start listening on port 6795 and waits for the second player to join.
- On starting the game the other person must select option as Player 2 which will start up the server for player 2 and the server will start listening on port 6792 and wait for the first player to join.
- When the second server will start (or whichever will start later), it will send a **notification** saying it's in ready state now and it's name to the other server.
- Once the other got the notification it will send it's **acknowledgement** back and a connection will be established, hence a **handshake** is done between the servers.
- After the connection is established, player 1 and player 2 can start playing the game against each other.

8.0 Exception Handling

8.1 Managed Exception Handling

- PortException : In this when server is launched using the wrong port, it will catch the exception at compile time and will handle it by correcting the port number.
- WrongMethodException : In this when a method is called from views to controller, then at controller it will catch the exception and will display an appropriate message on the console without crashing the application.

8.2 Unmanaged Exception Handling

- NegativeScoreException : In this exception handling, whenever a score tries to go in negative an exception will be thrown and handled by correcting the score to zero. So, the game logic of no score below zero will be maintained in the logic.
- LocationHitException : In this exception, if a player or computer hits the same location again it will handle it by displaying a proper message on the screen.

9.0 Refactored code

The code was refactored and the major changes made were:-

- Earlier the grid for deploying the user ships and computer ships were handled in one class, it handled both the deployment of the user ships and computer ships. We refactored the classes and distributed the functionality into two separate classes,one for deploying user ships (*ShipGrid.java)* and the other for handling the deployment of computer ships (*RadarGrid.java*).
- Earlier we were using a Master Slave architecture , but now we have refactored the code to follow the MVC design implementing Observer Pattern.
- We changed the function and object names to make them more understandable and easily readable, we ensured that the object names, constant names made relevance to the task they were used to perform.
- The methods that were used to set the scores and names of the player and computer were all called separately but that part was merged into two methods so the parts don't get much scattered.
- Grids were now separated into different panes so the calculation of coordinates could be much simpler.
- The single scene is now segregated into different scenes like start up and game play.
- The part of saving and loading during the development was being done in one class but we created different objects so that both the components are segregated.

10.0 Coding standard followed

The coding standards defines how the code is to be written, we used the following conventions while coding :-

Naming:-

- Class Name - UpperCamelCase
- Method Name - LowerCamelCase
- Parameter Name - lowerCamelCase
- Package Name - all Lowercase

Indentation:-

- Proper indent with 1 tab.
- Opening curly braces should be in the same line as the function or class name.

Declarations:-

- All variables to be declared on top of the class before any function definitions.
- Private and Public variables/functions to be grouped together.

Comments:-

- Comments before all code blocks.
- No unwanted commented code should be present.
- Javadoc comments should include all the parameter and return type if any.

11.0 Unit Test Cases

	Test Cases	**Expected Result**	**Actual Result**
	Ship placement		
1.	User Ships cannot be placed ship Diagonally	Fail	Fail
2.	User Ships cannot be on the same location	Fail	Fail
3.	User Ships cannot be placed Right	Pass	Pass
4.	User Ships cannot be placed outside the Grid	Fail	Fail
5.	AI Ship placement correctly	Pass	Pass
6.	Test case to check so that all 5 ships have been deployed	Pass	Pass
7.	Ships cannot be place Adjacent to Each other	Fail	Fail
	AI validation		
8.	AI medium mode HIT case	Pass	Pass
9.	AI hard mode HIT case	Pass	Pass
10.	AI easy mode HIT case	Pass	Pass
	Correct Startup phase		
11.	Computer Grid Initialized correctly or not	Pass	Pass
12.	Player Grid Initialized correctly or not	Pass	Pass

13.	AI Correctly called for salvo mode	Pass	Pass
14.	AI Correctly called for normal mode	Pass	Pass
15.	Test case to check if all ships are deployed	Pass	Pass
16.	Test case to check if array SunkenShips is empty at game start	Pass	Pass
	Calculation Of Winner		
17.	Test Case to verify if the score is increased after a hit for AI	Pass	Pass
18.	Test Case to verify if the score is increased after a hit for Player	Pass	Pass
19.	Test Case to verify if the score is decremented after a miss for AI	Pass	Pass
20.	Test Case to verify if the score is decremented after a miss for Player	Pass	Pass
	Validation of a correct human user play	Fail	Fail
21.	Checking if User won	Pass	Pass
22.	Checking if AI won	Pass	Pass
	Various test for the attack		
23.	Player HIT the computer Grid	Pass	Pass
24.	Player cannot hit on the same location again	Pass	Pass

25.	AI HIT the Player Grid	Pass	Pass
	Various tests for Save Game		
26	Test Case to verify if the file path is correct or not	Pass	Pass
27	Test case to verify if the folder path is correct or not	Pass	Pass
28	Test to verify if the computer's hit coordinates were saved correctly or not	Pass	Pass
29	Test to verify if the User's hit coordinates were saved correctly or not	Pass	Pass
30	Test case to verify if the name is being correctly set	Pass	Pass
	Various Tests for Load Game		
31	Test Case to verify the coordinates and their state are correct	Pass	Pass
32	Test Case to verify whether the correct score is loaded or not	Pass	Pass
33	Test Case to verify whether the correct saves are loaded or not	Pass	Pass
34	Test Case to verify whether the correct userScore is loaded or not	Pass	Pass
35	Test case to verify whether the ship coordinates are colored correctly or not	Pass	Pass
36	Test case to verify whether the correct date is selected for loading the game	Pass	Pass

	Various tests for Vs player networking		
37	Test Case to check whether the other player is winning or not	Pass	Pass
38	Test Case to check whether the correct player number is getting set or not	Pass	Pass
39	Test Case whether the name of the player is getting set correctly in the network or not	Pass	Pass
40	Test whether the coordinate is being hit correctly in the network or not	Pass	Pass