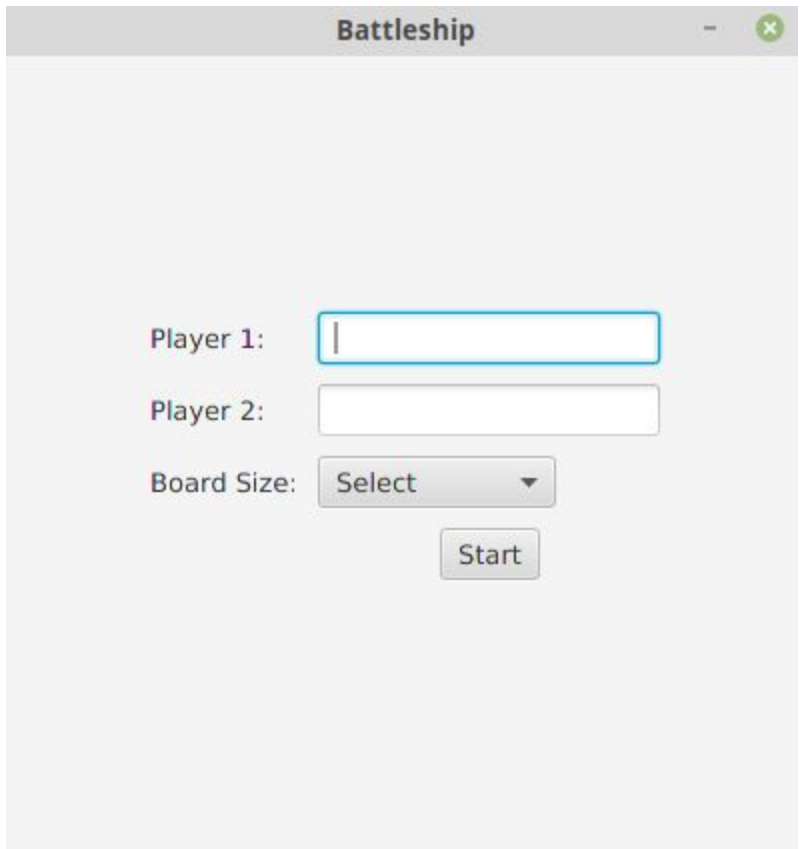


Java Project (Game: Battleship)

Problem Description:

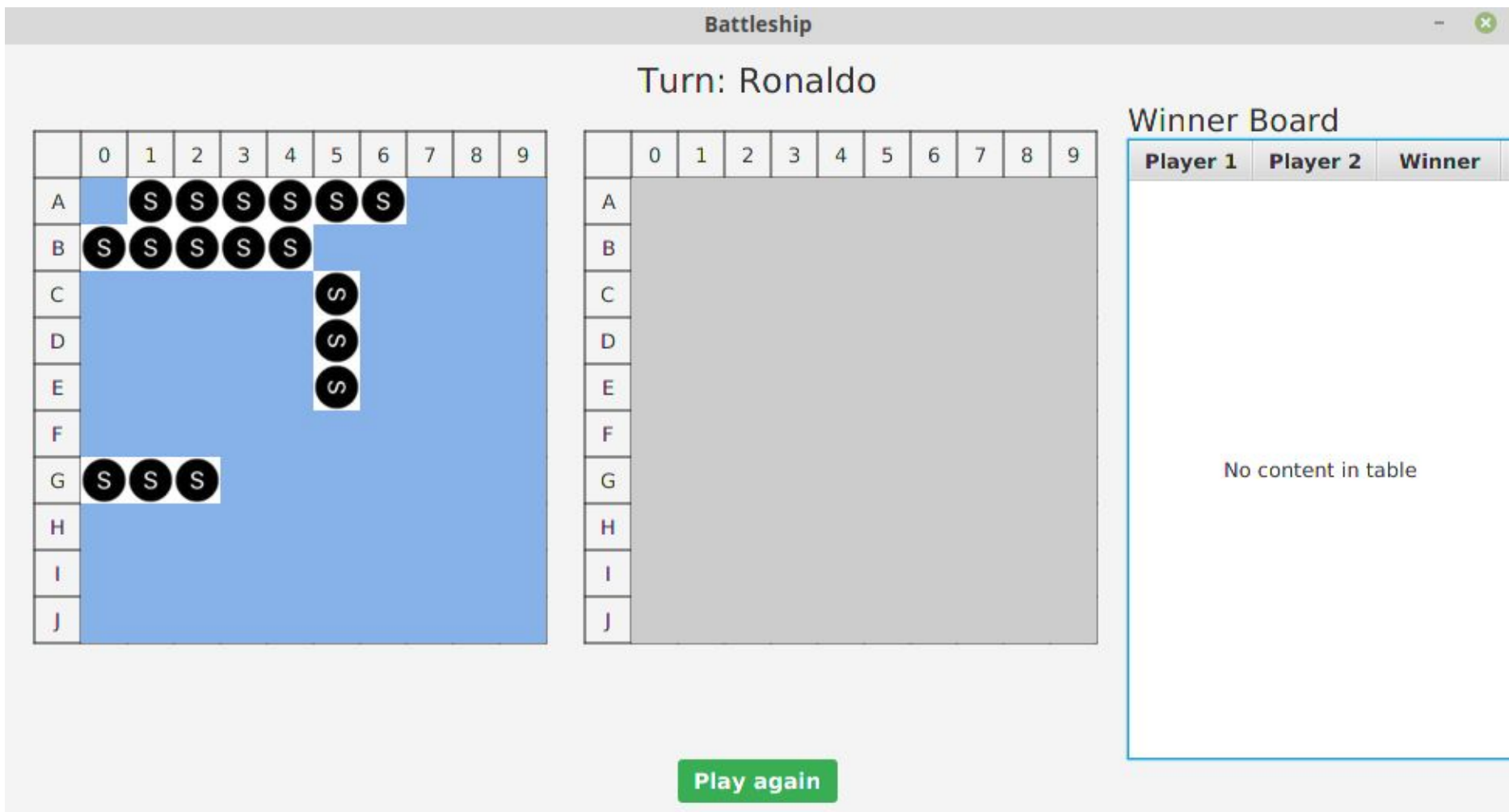
Battleship is a war game played on the ocean by two players. Each player owns his battle area. Each player will get the same number of ships where each ship may have different size placed at some position in non-overlapping fashion. A player can not see the location of his enemy's ships. The player who destroys all the ships of the other player is the winner of the game. Both players will get a chance to launch missiles one by one. Each player is aware of whether its missile hits the enemy's ship or misses.

Introduction:



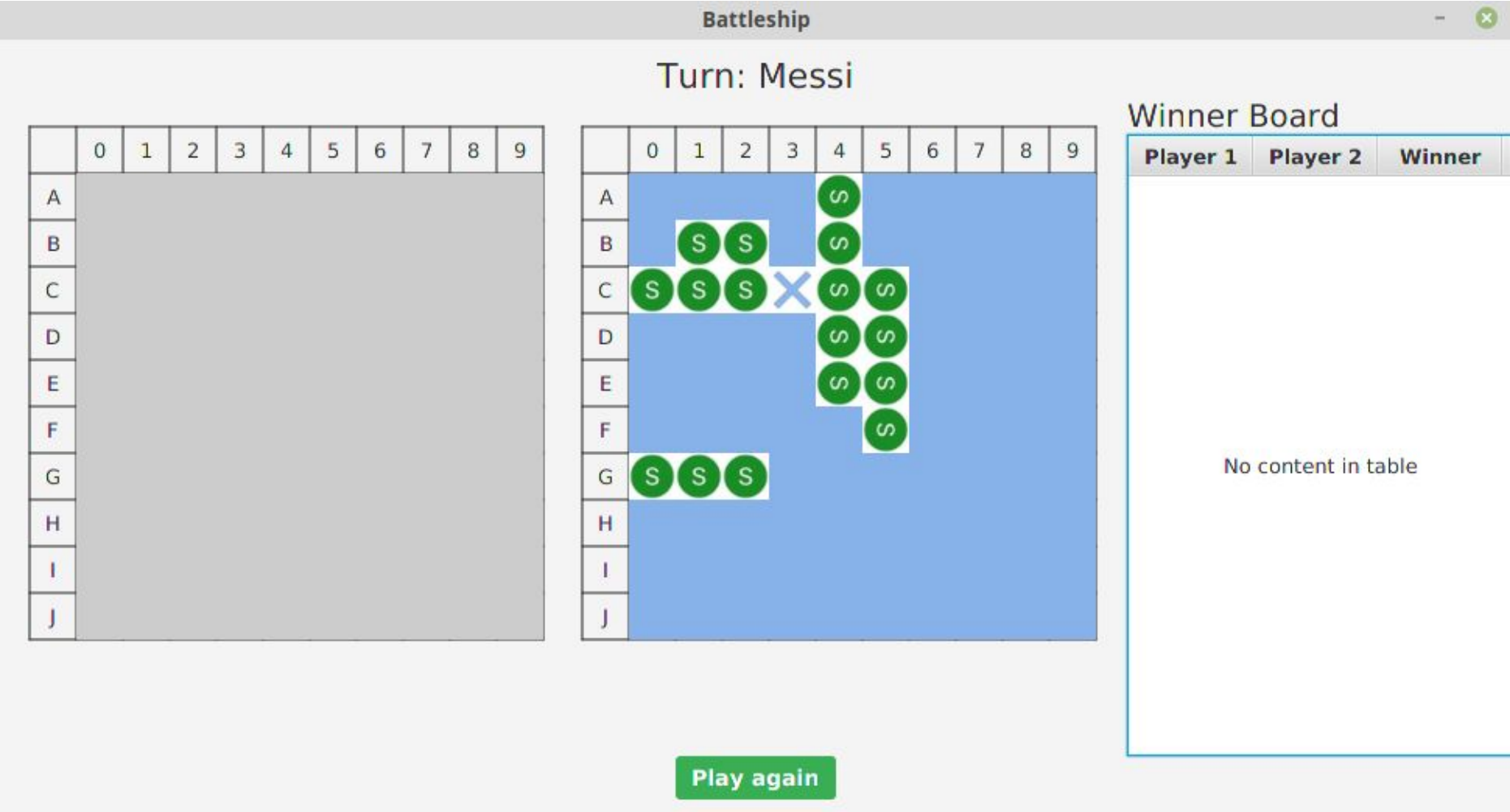
This is the first screen of the game that takes player information and board size.

And the game begins!



This screen has the following components:

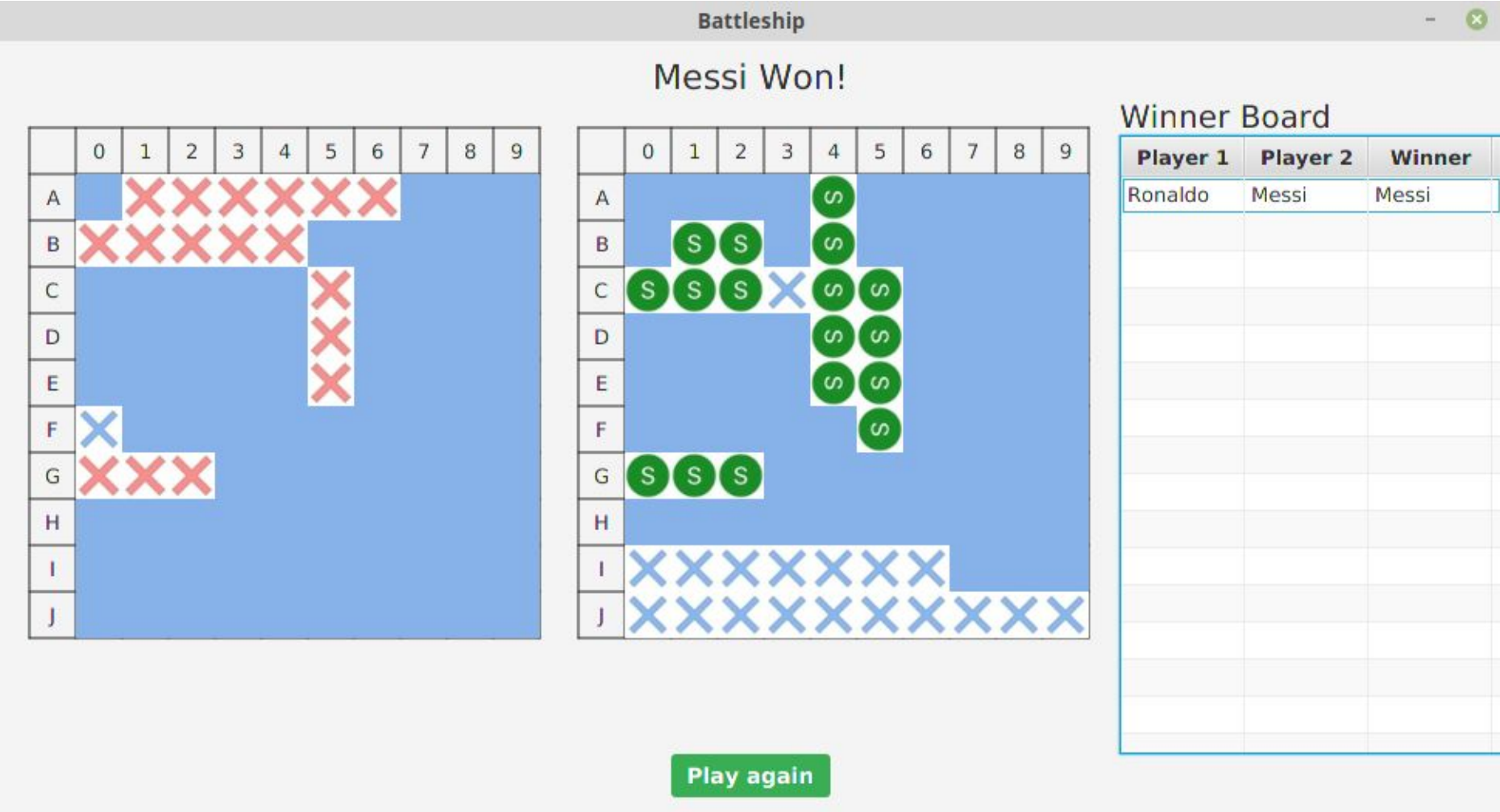
- **Turn Label** - It is an indicator of whose turn it is.
- **Arena** - This is where the game is played. It contains the battle area of both players. Game engine places ships of both players on the arena randomly. Blue colour represents water and S-shaped icon represents the ships (black for player 1 and green for player 2). Grey colour represents the enemy's battle area. Ronaldo (Player 1) can not see the location of Messi (Player 2) ships.
- **Winner Board** - It records the results of previous games.
- **Play again** - If a player wants to start again it can hit this button and the game will restart.



Ronaldo (Player 1) played his turn and missed the target indicated by blue cross icon in the game. Now its Messi's (Player 2) turn. Messi's ships are green in colour.

After a few turns, game ends and Messi is declared as the winner. The entire arena is now visible and Winner Board is updated with the result.

The following screen captures the final move:



- For reference:**
- Blue cross icon represents a miss i.e, player fired the missile and hit the water.
 - Red cross icon represents a hit i.e, player fired the missile and hit the enemy's ship.
 - Black colour S-shaped icons represent Player 1 ships. They are either placed horizontally or vertically.
 - Green colour S-shaped icons represent Player 2 ships. They are either placed horizontally or vertically.

Implementation and Design:

Object-oriented programming allows us to model real-world objects. Each object can have private mutable states and functions that operate on those states. This type of design makes our program more readable, maintainable and easily modifiable.

- The main objects of the battleship game are:
- **Arena** - Arena has information about both players. It knows whose turn it is. A player fires a missile on the arena by providing its identifier and the coordinates. So, the arena also knows if the game is over or not.

Design Decision - I introduced this object because a player does not have any details of the enemy's ships so I needed an object that has information about both players along with whose turn it is.

- **Player** - Represents a player.
- **Ship** - Represents a ship. The ship is of following types - CARRIER of length 5, BATTLESHIP of length 4, DESTROYER of length 3, SUBMARINE of length 3, PATROL_BOAT of length 2. It has either VERTICAL orientation or HORIZONTAL orientation.
- **Board** - Represents a player's board. It uses a 2-D array data structure to store coordinates of ship and water.
- **Game Gui** - Battleship game has two screens - one screen for creating a registration form that captures player information and game configuration (e.g, board size) and another screen for creating the arena GUI. This class is responsible for showing the first screen on stage.
- **Registration Gui** - I have used JavaFX FXML for creating the GUI for the first screen. This class is responsible for creating the arena object by using the player information and game configurations (e.g, board size). It also shows the second screen when the player clicks on the start button.

Significance of Board size - It increases the complexity of the game. A large board size means a large area for placing ships that implies more turns are required to completely destroy all the enemy's ships.

- **Arena Gui** - I have used JavaFX FXML for creating the base GUI for the second screen. This class is responsible for dynamically tiling the arena with the ship icon or the blue colour (for water) using the arena object.
- **Tile Gui** - This class is used by Arena Gui to tile the arena. Since I had to create the tile dynamically I avoided using JavaFX FXML for this.

Design Decision - I began writing GUI first that had high cohesion with the logic of the game. As a result, when I started testing the game I was stuck. Since I could not test the GUI so I was unable to test the logic of the game. I solved this issue by using the Model-View-Controller design pattern.

Model-View-Controller design pattern enhances the power of object-oriented programming. MVC includes the following components - **Model** represents data or state of the program, **View** represents display and **Controller** binds the two together. This design improves cohesion, reduces coupling and improves maintainability, modifiability and simultaneous development. This design enables developers to work separately on model, view and controller. It also eases testing of code. It was very difficult to test code when view and model were tightly coupled.

Differentiating the above-mentioned objects into these three categories:

Model - Arena, Player, Ship, Board

View and Controller - Registration Gui, Arena Gui

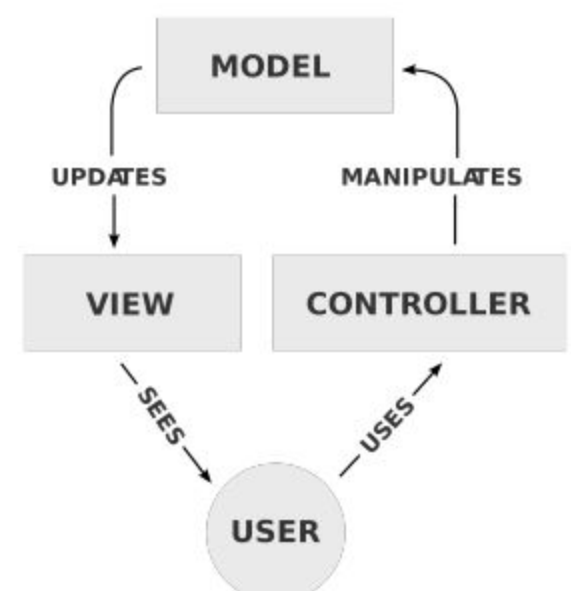
View - Game Gui, Tile Gui

The main logic of the game:

```
Arena.java 83
164 public boolean fireMissile(int enemyX, int enemyY, PlayerId missileFiredBy) {
165     if (!missileFiredBy.equals(turn))
166         throw new IllegalArgumentException("not your turn");
167     boolean hit = false;
168     Player enemy = getOtherPlayer();
169     Board board = enemy.getBoard();
170     TileType tileType = board.getTileType(enemyX, enemyY);
171     if (tileType.equals(TileType.WATER_HIT) || tileType.equals(TileType.SHIP_DESTROYED))
172         throw new IllegalArgumentException("already fired missile on this coordinate");
173     if (tileType.equals(TileType.WATER)) {
174         board.updateTileType(enemyX, enemyY, TileType.WATER_HIT);
175     } else if (tileType.equals(TileType.SHIP)) {
176         hit = true;
177         board.updateTileType(enemyX, enemyY, TileType.SHIP_DESTROYED);
178         Ship ship = enemy.getShipOnBoardPosition(enemyX, enemyY);
179         if (ship != null)
180             ship.hit();
181     }
182     if (!enemy.isDead()) {
183         toggleTurn();
184         if (arenaGui != null)
185             arenaGui.displayArena(false);
186     } else {
187         playerDao.saveResult(player1.getName(), player2.getName(), getPlayer(getTurn()).getName());
188         isGameOver = true;
189         if (arenaGui != null)
190             arenaGui.displayArena(true);
191     }
192     return hit;
193 }
```

Note: Each ship has a health attribute that is initialized with its length (e.g, DESTROYER is of length 3). Every time a ship is hit its health is decremented by 1. A ship is considered destroyed when its health becomes 0. A player is considered dead when all of its ships are destroyed.

MVC Design Implementation - In MVC, Model updates the View but keeps the coupling minimum. I created an interface `IArena` that has a method `displayArena()`. When a player fires a missile on the arena it calls this method. This interface is implemented by the `Arena Gui` class. Using this design, we can easily change the view by creating another implementation for this interface.



The following class diagram represents the **Model** classes:

Their brief description:

Arena class

- Attributes player1 and player2 are final. Reason - I don't want to modify players in the middle of the game. They are fixed when you construct the object and can not be changed later.
- The constructor requires both players to create arena object. Neither of the players can be null.
- Stores whose turn it is.
- Stores state of the game i.e, whether the game is over or not.
- Stores Arena Gui object in interface IArena.
- Has a method getOtherPlayer() which returns the enemy player in the arena.
- Has a method getGridSize() that returns a player's board size + 1. The extra one is for headers that describe the coordinates of the battle area (e.g, A4, C5).
- Has a method toggleTurn() that changes the turn attribute to the other player.
- Has a method fireMissile() - already explained.

Player class

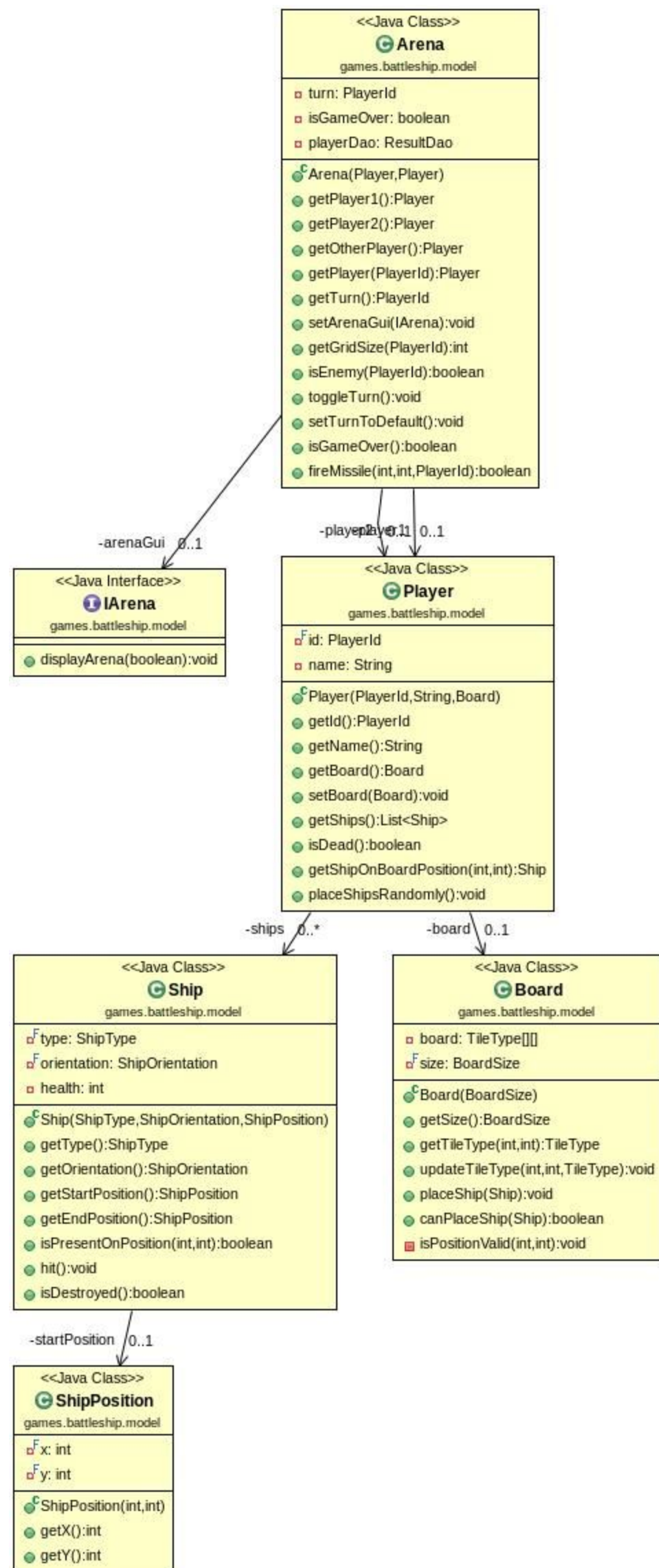
- Attribute playerId is final. Reason - I don't want to modify it in the middle of the game. It is fixed when you construct the object and can not be changed later.
- Attribute name is not final since it can be modified in the middle of the game. (functionality planned for future)
- Attribute board is not final since when a player clicks on Play Again I have to reinitialize the board attribute.
- Ship attribute is not final since either I can place ships through the GUI (functionality planned for future) or the game engine will place them.
- The constructor requires playerId, name and board to create player object. Basic validations like null and empty string checks are done.
- Has a method isDead() - already explained.
- Has a method getShipOnBoardPosition() that returns the ship on the position specified in the parameters. It loops over all the ships and uses the Ship class method isPresentOnPosition() to check if a particular ship is present on that position or not.
- Has a method placeShipsRandomly() that places ships randomly on the board. It uses Board class method canPlaceShip() to determine if the ship can be placed on random coordinates or not and placeShip() method to place the ship in case it succeeds.

Ship class

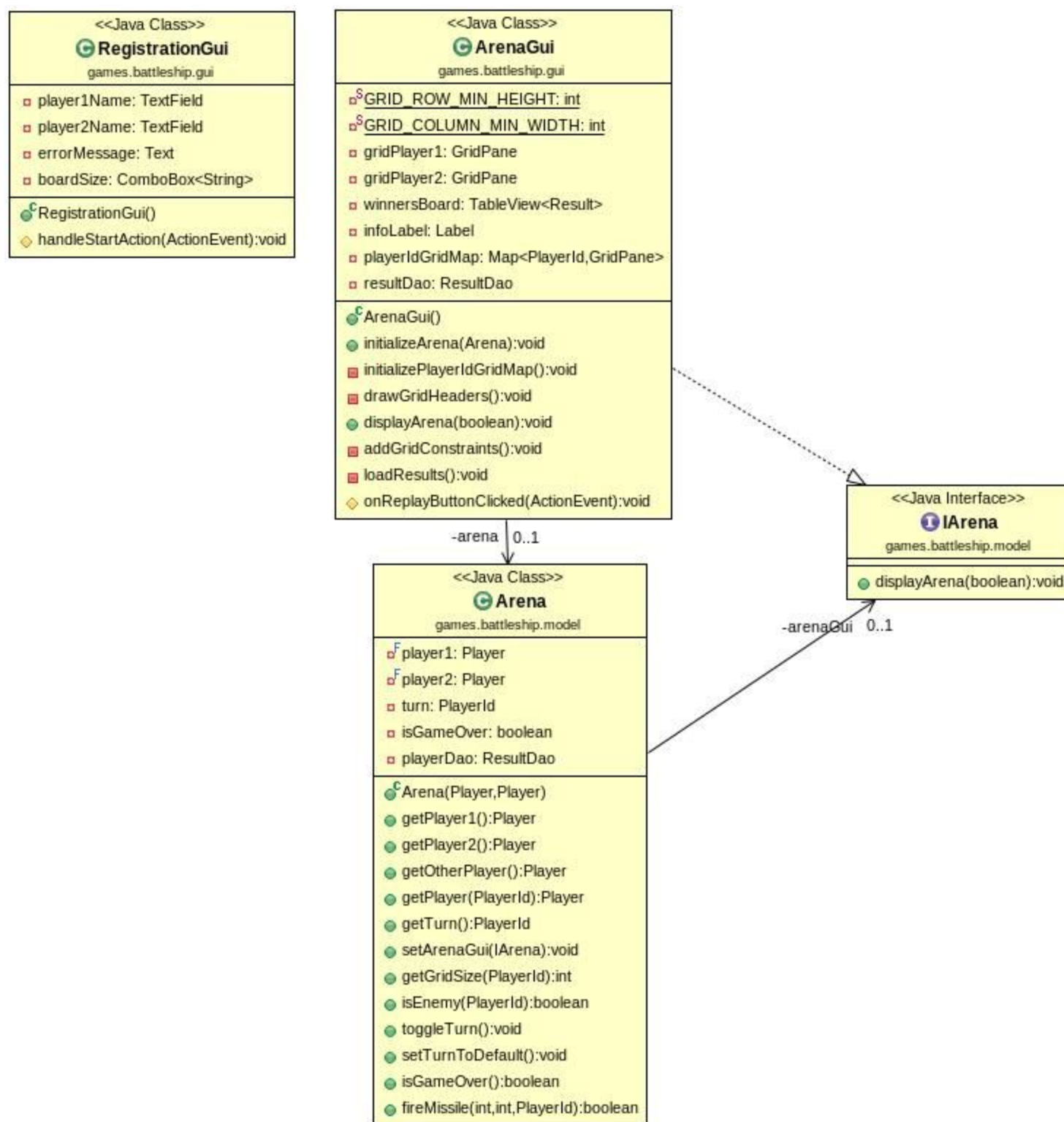
- Attribute type, orientation and position are final. Reason - I don't want to modify them in the middle of the game. They are fixed when you construct the object and can not be changed later.
- The constructor requires type, orientation and start position of the ship. Basic validations like null checks are done.
- Has a method getEndPosition() that returns the end position of the ship on the basis of its start position, length and orientation.
- Has a method isPresentOnPosition() that checks if the ship is present on that position or not.
- Has a method hit() that decreases the health of ship - already explained.
- Has a method isDestroyed() that checks if the ship is destroyed or not - already explained.

Board class

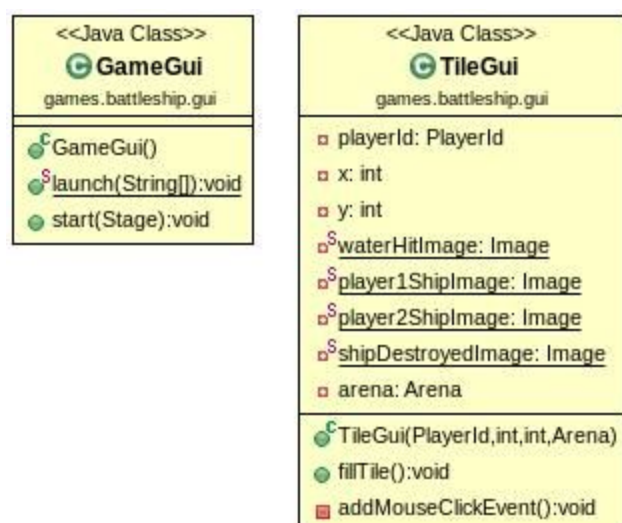
- Attribute size is final. Reason - I don't want to modify them in the middle of the game. They are fixed when you construct the object and can not be changed later.
- It uses a 2-D array data structure to store coordinates of ship and water.
- The constructor requires the size of the board. Basic validations like null checks are done.
- Has a method getTileType() that returns that type tile on given coordinate. Basic validations are done. For e.g, x or y coordinate given should be valid i.e, should not be negative or equal to board size, or greater than board size.
- Has a method placeShip() that places the given ship on board. Basic validations like ship position should be valid are done.
- Has a method canPlaceShip() that checks if the given ship can be placed on the board or not. Logic - Confirm if there is another ship on that location.



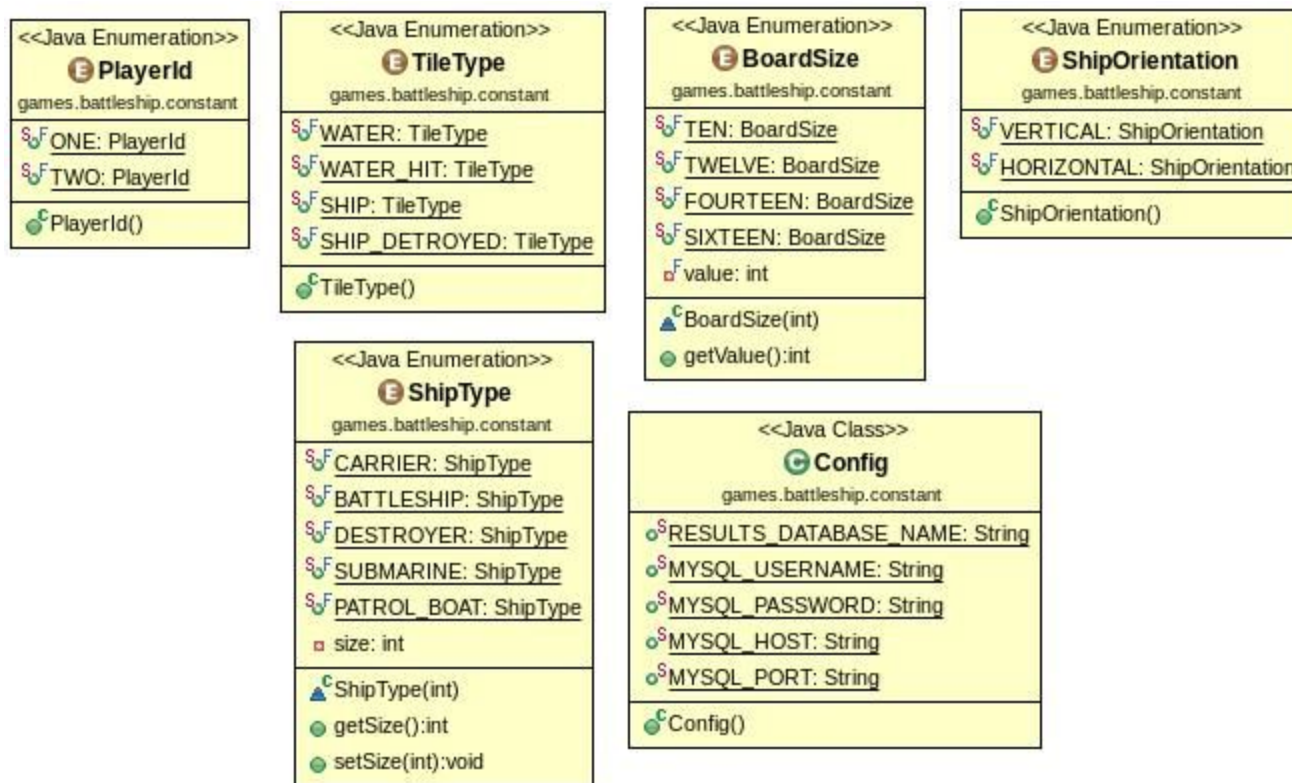
The following class diagram represents the **View** and **Controller** classes:



The following class diagram represents the **View** classes:



The following class diagram represents the constant classes:



The following class diagram represents helper classes:



Unit Testing Strategy:

Testing Arena class:

Method: getPlayer1()

Case 1: Check if the arena object can be created. It should throw an exception if player1 is null.

Case 2: Should be able to successfully fetch player 1 from arena object.

Method: getPlayer2()

Case 1: Check if the arena object can be created. It should throw an exception if player2 is null.

Case 2: Should be able to successfully fetch player 2 from arena object.

Method: getOtherPlayer()

Case 1: By default, the turn is of player 1. Create an arena object and check if this method returns player 2.

Case 2: Toggle the turn using arena object method. Check if this method returns player 1.

Method: getGridSize(PlayerId playerId)

Case 1: Create an arena object using player 1 board size as 10 and player 2 board size as 10. Check if this method returns 11.

Case 2: Create an arena object using player 1 board size as 12 and player 2 board size as 12. Check if this method returns 13.

Case 3: Create an arena object using player 1 board size as 14 and player 2 board size as 14. Check if this method returns 15.

Case 4: Create an arena object using player 1 board size as 16 and player 2 board size as 16. Check if this method returns 17.

Method: isEnemy(PlayerId playerId)

Case 1: By default, the turn is of player 1. Create an arena object and check if this method returns true for player 2.

Case 2: Toggle the turn using arena object method. Check if this method returns true for player 1.

Method: toggleTurn()

Case 1: By default, the turn is of player 1. Create an arena object and toggle turn. Check if getTurn() method returns the id of player 2.

Case 2: Toggle the turn again. Check if getTurn() method returns the id of player 1.

Method: setTurnToDefault()

Case 1: By default, the turn is of player 1. Create an arena object, toggle turn and call this method. Check if getTurn() method returns the id of player 1.

Case 2: Toggle the turn again. Check if getTurn() method returns the id of player 1.

Method: isGameOver()

Case 1: By default, the game is not over. Create an arena object and check if this method returns false.

Case 2: Create player1 and place ship (PATROL_BOAT) on (0, 0). Create player2 and place ship (PATROL_BOAT) on (0, 0). Create an arena object using both players. Player 1 fires missile on the coordinate (0, 0). Player 2 fires missile on the coordinate (1, 1). Player 1 fires missile on the coordinate (0, 1). Now, check this method returns true.

Method: fireMissile(int enemyX, int enemyY, PlayerId missileFiredBy)

Case 1: By default, the turn is of player 1. Create an arena object. Player 2 fires missile on the arena. This method should throw an exception since its player 1 turn.

Case 2: Create an arena object. Player 1 fires a missile on player 2's ship. This method should true.

Case 3: Create an arena object. Player 1 fires a missile on water. This method should false.

Case 4: This method should throw an exception in case a player tries to fire on the same coordinate again.

Testing Player class:

Method: getId()

Case 1: Create a player object with playerId ONE and check if this method returns the same player id.

Case 2: Create a player object with playerId TWO and check if this method returns the same player id.

Case 3: It should throw an exception in case player object is created with NULL id.

Method: getName()

Case 1: Create a player object with some name and check if this method returns the same player name.

Case 2: Create a player object with some name and check if this method returns the same player name.

Case 3: It should throw an exception in case player object is created with NULL name.

Method: getBoard()

Case 1: Create a player object with board of size 10 and check if this method returns the same board size.



















Case 2: It should throw an exception in case player object is created with NULL board.

Method: getShips()

Case 1: By default, a player should have 0 ships. Check if the number of ships is 0.

Case 2: Create a player object and place all ships randomly. Check if the number of ships is 5.

Coverage Report:

Element	Coverage	Covered Instruction	Missed Instruction
▼ BattleShip	 74.8 %	3,387	1,139
▼ src/main/java	 57.2 %	1,186	886
▶ games.battleship.gui	 0.0 %	0	775
▶ games.battleship.dao	 59.8 %	101	68
▼ games.battleship.model	 97.3 %	732	20
▶ Arena.java	 92.5 %	196	16
▶ Player.java	 97.8 %	182	4
▶ Board.java	 100.0 %	175	0
▶ Result.java	 100.0 %	41	0
▶ Ship.java	 100.0 %	114	0
▶ ShipPosition.java	 100.0 %	24	0
▶ games.battleship.constant	 97.1 %	232	7
▼ games.battleship.validator	 90.9 %	70	7
▶ RegistrationValidator.java	 90.9 %	70	7
▶ games.battleship.main	 0.0 %	0	6
▼ games.battleship.util	 94.4 %	51	3
▶ RegistrationUtil.java	 94.4 %	51	3
▶ src/test/java	 89.7 %	2,201	253

References:

- All icons are taken from <https://icons8.com>
- <http://tutorials.jenkov.com/javafx/index.html>
- <https://www.callicoder.com/javafx-registration-form-gui-tutorial/>