

# **Battleship**

## **Final Report**

**git gud**

Rebekah Blazer, Ryder Gallagher, Jack Nealon

Course: CPSC 224 - Software Development

## I. Introduction and Project Description

The objective of this group project was to develop a java program to run a game of Battleship utilizing teamwork and collaborative programming. Throughout the process this project endured many changes and modifications to the original plan as errors were encountered, restructurings changed our outlook, and accidents happened. Our initial plans included the following:

First, we developed a written plan on how we wanted to approach our program. This included UML charts, sequence diagrams, and other functional requirements identified before starting to code, all of which could be found in our FP2 document. We also developed a comprehensive testing plan to ensure that our program's behavior worked correctly and that all methods produced intended results (this document). This helped with smooth development and minimal errors throughout the project.

The original structure of the game had two players who would play against each other on a 10x10 grid game board. Each player would have 5 ships: one carrier (size 5), one battleship (size 4), one cruiser (size 3), one submarine (size 3), and one destroyer (size 2). The ships would be placed on the game board with a click-and-drag functionality, and each player would then take turns guessing where the opposing player's ships were, the functionality for this was to be determined as we developed our initial game structure.

- If the player's guess hit one of the opposing player's ships, the coordinate square was marked with an 'X' and the grid square colored red.
- If the attack missed, the coordinate square was marked with an 'O' and remained the same blue ocean color.
- A battleship was sunk when there was an 'X' on every coordinate square that the ship occupied.

The game would continue with the players taking turns attacking the other's board until one player had sunk all of the opposing player's ships. At the end of the game, the winner would be displayed on a splash page, and the option to play again is given to the user.

As development progressed, our original plans changed of which the largest and most impactful change was the decision to switch to a single player. Additionally, the click-and-drag ship placement functionality was discarded and replaced with an automatic ship placement feature that randomly placed the ships on the board.

On startup, the current program version produces a window with the player's name and a fully generated 10x10 grid for the player to start taking guesses at with ships already randomly placed on the game board. Because the board's grid is made of buttons, the guessing functionality only requires that the user use the mouse to select their desired coordinate. After the player guesses, the coordinate that the player selected is colored either red or green, red to symbolize a hit ship, and green to symbolize a missed guess. After a coordinate is guessed and revealed, the player can no longer click on that button and must guess at a different coordinate. Currently, there is no end-of-game functionality.

The program utilized an external GUI, which displayed each player their respective game boards and provided an interactive experience for the players. The game was developed using the Java programming language and utilized git and GitHub for collaborative file sharing.

## II. Team Members - Bios and Project Roles

Jack Nealon is a computer science student with a concentration in biology who is interested in software development, data management, and biotechnical applications in the medical industry. His past projects include encryption algorithms, game development and data control algorithms. Jack demonstrates proficiencies in C/C++, Python, NetLogo, and Java. For this project his responsibilities started with the functional design of the game board and of the background structure for the main game. These responsibilities changed and expanded as the project progressed.

Ryder Gallagher is a computer science student minoring in Communication Studies who is interested in data science, software development, and operating systems. He has a background in electricity and digital logic, as he took 4 years of electronics in high school. Ryder's computer science skills include C++, Python (pandas library specifically), and Java. For his project responsibilities he was the one creating the Issues most of the time and assigning people to work, as well as designing a lot of the ship functionality. By the end of the project, we kind of all worked on the same stuff though.

Rebekah Blazer is a computer science student minoring in art and a concentration in art, software security, and software application development. Her past projects include various class assignments, however no current personal projects. She is currently a research assistant for Tomas Guardia at Gonzaga University to develop an online game for Rithmomachia. Rebekah's computing skills include C++ and Java, with limited ability in working with GUIs. For this project, her responsibilities started with working on the Player class, and evolved to working on the main GUI layout alongside Jack and designing and compiling information for the final PowerPoint presentation.

## III. Project Requirements

### Functional Requirements:

Ship Can Be Hit/Sink	
Priority	Medium
Purpose	Advances the game. Once all the ships have been sunk, the game is over.
Inputs/Needs	Coordinate – getX() Coordinate – getY()
Operators/Actors	Coordinate holds the ship Ships have coordinates
Outputs	Button state changes

Players can Guess Coordinates	
Priority	High
Purpose	The main driving ‘actor’ of the game, the player guesses coordinates by clicking buttons on the board. The button will then change state to indicate either a hit or miss (see Buttons can Change State for more information).
Inputs/Needs	Board – makeBoard() Coordinate – addActionListener()
Operators/Actors	Player clicks the button Board holds the coordinate buttons Coordinate button can be clicked
Outputs	The clicked coordinate button changes state and a message is printed in the terminal

### Non-Functional Requirements:

Player can Click Coordinate Buttons	
Priority	High
Purpose	Allows Player to guess and get feedback for coordinates. The player clicks a button on the coordinate grid to guess where a ship is.
Inputs/Needs	Board – makeBoard() Coordinate – addActionListener() Coordinate – getX() Coordinate – getY()
Operators/Actors	Player clicks the button Board holds the coordinate buttons Coordinate Button can be clicked
Outputs	The clicked button changes state Message printed out in terminal

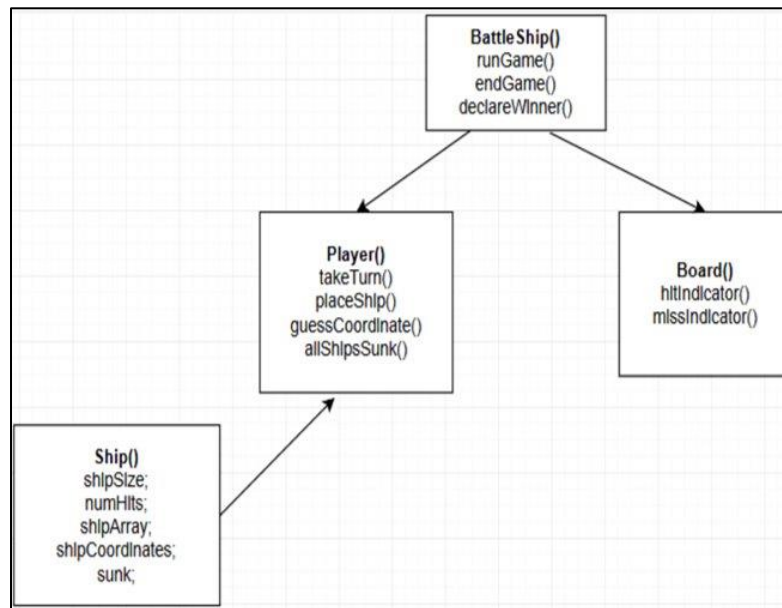
Buttons can Change State	
Priority	Medium
Purpose	Gives feedback to the Player so they know whether their guess is correct or incorrect. A ‘Hit’ will turn the button red and a ‘Miss’ will turn the button green.

Inputs/Needs	Coordinate - addButtonActionListener() Coordinate – Boolean hasShip
Operators/Actors	Player clicks the button The JButton changes state
Outputs	If there is a ship at that coordinate, the button will turn red. If there is no ship at that location the button will turn green. In both cases the button will be disabled

## IV. Solution Approach

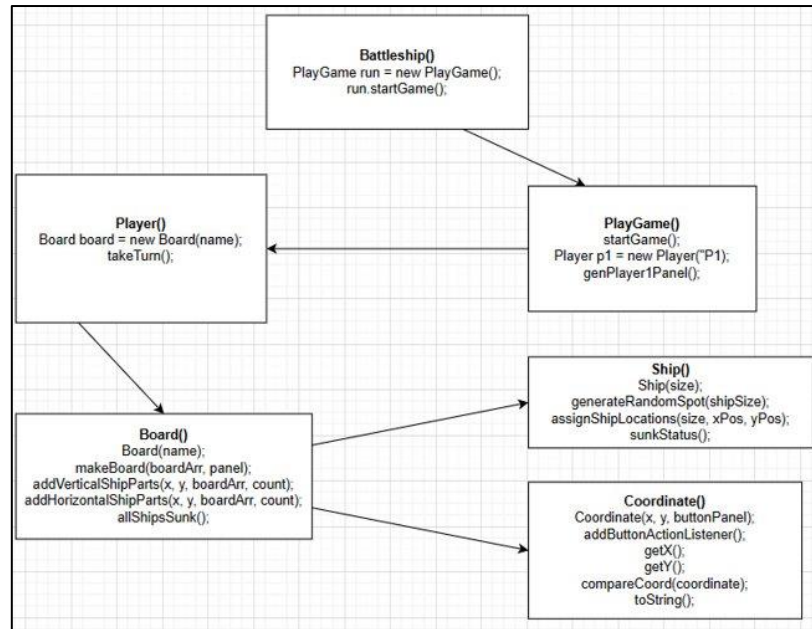
The following UML diagram identified as Figure 1 shows the initial plans for our battleship. There were 4 main classes with basic and rudimentary functions. These were split up among our 3 members, each person taking responsibility for 1 or 2 classes.

**Figure 1**



The following UML diagram identified as Figure 2 shows the final plans for our battleship.

**Figure 2**



As can be seen between Figures 1 and 2, we added a lot of game functionality and added classes as was needed. The coordinate class in specific was necessary to add as it would work better with the GUI than our original plans. The major challenge for us that prompted many changes to our original idea was the transition from terminal-based programming to GUI-based programming. A common example of this type of transition was our focus on user input. Terminal-based programming would just implement a “console input” command while GUI-based programming would depend on where the panel is created, who has access to it, and how to properly read into our variables so that the game could continue. In the end, we did get a working program that steps through the basic requirements for the game of battleship.

As you can see in Figure 1, our original approach was to make just a few classes with a minimal amount of member variables. We were to split these classes among us, and all code them separately. After we had the classes, we were to work together to try to get them to work together. This approach was very flawed because we all had different ideas as to how our classes were going to work together, so we had to eventually go through and change all the classes together to try to get them to work together and have a functioning game. What we ended up with was Figure 2 with a less linear class structure, so that not every class relied entirely on every other one to work.

For more information on how these Classes interact with each other in a sequence diagram, see Appendix A.

## **V. Test Plan**

### **I. Initial Testing Plan**

#### **I.1.1. Unit Testing**

Unit testing will start as an individual assignment. Team members are to do the tests for the class they are assigned to code and should start with testing the default values of the class. These tests should be made and run before any other functions for the class are written and the coder shall not work on the functions until default values work properly. Once default values are running properly, the team member will start working on the main functions for the class. Tests are recommended for each function to minimize the need for code review when pushing to main, however, not required. Once code is pushed to main, team members will test classes' ability to interact with one another. Any errors found will be worked on together as a team. Once all classes are finished and run error-free, the program will be run and tested as a whole. Again, any errors found are to be resolved as a team.

#### **I.1.2. Integration Testing**

Initially, integration testing will be conducted individually as each person develops their assigned class alongside its respective unit test cases. As unit tests are implemented, the respective integration tests will follow to demonstrate that despite passing unit tests, the program behavior will still function properly to ensure a working program. Additionally, once unit tests are completed and as overlap happens, collaborative integration testing will then fall on the two programmers whose classes (methods or variables or behaviors) overlap. As a framework, some integration tests that can be determined now include:

1. Verify that the game is displayed correctly on the user interface.
  - a. That players can interact with the GUI.
2. Test that the game accurately processes player actions and correctly updates the game board.
3. Verify that the game rules are implemented correctly.
  - a. Attempts against the game rules are properly addressed (re-enter, skip, etc)
4. Test that the game ends when one person has sunk all of the opposing ships.
5. Verify that players can take turns attacking and receive feedback on the result of the attack.

#### **I.1.3. System Testing**

System testing will be done with a few hard coded scenarios. This will be implemented after integration testing, so we can assume that the game's individual functionality should work, so these tests will see if the game functions as intended. A few things we could try would be:

1. Hard coded version of the game that has the players performing the intended “normal actions” such as putting in strings as their names, placing the pieces away from each other, and guessing both wrong and right locations for the opponent’s battleships.
2. Hard coded version of the game where the players clearly make some odd decisions like putting in weird values for their names, guessing everywhere except for the ships, placing ships too close together, etc.
3. Hard coded version of the game where the players make the worst decisions possible like trying to make their name 100 characters long, placing ships on top of each other, etc.

#### **I.1.4. Functional testing:**

1. Test if players can take at least 3 turns each, while guessing coordinates (doesn’t need to have any ships it just needs to allow the player to guess and to show where they already guessed).
2. Test to see if the player can guess the same spot over and over or if the proper error message will pop up
3. Test to see if the board starts off blank (with no ships on it)
4. Hard code placing ships, then test to see if the board only shows the ships that you placed and if the board is clear of any hit or miss markers
5. Test to see if a ship will sink if all its coordinates are hit
6. Test to see if all the opponent's ships are sunk, and that the player is declared a winner.

#### **I.1.5. User Acceptance Testing:**

The last step of testing for us will be to select 2 friends to play the game with us and to take note of any bugs we have (if any). If there are any bugs, we will work to rectify those issues before repeating the testing process. If we have extra time, we may take suggestions and make cosmetic fixes and improvements to optimize the UI.

## **II. Actual Testing Plan**

### **II.1.1. Unit Testing**

Our unit testing was very informal with no actual function created for the test. Unit testing was conducted by adding in lines of code to set values, running the program, and erasing that line once the program ran as desired. Some of the unit tests conducted were:

- Creating ships of various lengths and printing out the size to ensure they were correct.
- Set a coordinate button to have a ship and added an action listener that would print “Has ship” if the button had a ship to make sure buttons could register that state and do so without affecting the other buttons.



- Added an action listener to print out results of getX() and getY() for the button clicked to ensure buttons had proper coordinate values as well as ensuring the X and Y values would be returned properly.

Various other unit tests were conducted however, each test was fairly informal. We would have preferred to have more unit tests and had actual functions to demonstrate them, however due to our initial struggle figuring out how to work as a group and various github issues, we were not as organized as hoped.

### **II.1.2. Integration Testing/System Testing**

Due to limited time and disorganized group planning, we combined Integration and System Testing. Once we conducted some basic unit tests, we all pushed the code we had implemented to main. Here we ran into some issues with merging to GitHub and that set us back a touch, but once we figured that out we were able to run tests. We started by simply trying to run the program and see if any errors popped up. There were no visible or glaring errors. At this point in time, we did not have much outside the basic layout for each function and a rudimentary GUI. We also figured out that the most efficient way to work was to have one person code while the other two discussed future coding issues that may arise or were currently an issue. Due to this, we had a lack of need for heavy integration testing as the code was being written by one person and could code between classes with no issue. For System testing, we had a hardcoded scenario we had running as we coded with a single ship on the grid. Then as new code was added we would rerun the program clicking coordinates on the grid and ensuring the button changed to the correct state. Alongside this, we were figuring out a way to add the board to the main GUI interface.

### **II.1.3. Functional testing:**

Functional testing consisted of running through an entire game various times. It is important to note that since we did not have an end-game case, this consisted of clicking all coordinate buttons until they all were disabled. Withing these run-throughs we tested the following:

- Ships can be Hit/Buttons can Change State: We placed a ship at specific coordinates and guessed those coordinates as well as surrounding coordinates. By doing this we ensure that the button will turn red if there is a ship, and that buttons that do not contain a ship will turn green. By doing this, we also ensure the ship was placed horizontally or vertically with the proper length and not horizontally or spread out.
- Player can Guess Coordinates/Player can Click Coordinate Buttons: We added code that made it so that when a button was clicked, it printed a statement to the terminal that included the coordinates of the button clicked and whether it was a Hit or Miss. We then ran through a game clicking coordinates until all buttons were disabled and checking to ensure the proper coordinate was printed into the terminal when the button was clicked. This ensures that the Player could click the JButtons in the coordinate grid, and that the proper coordinate was being registered by the click.

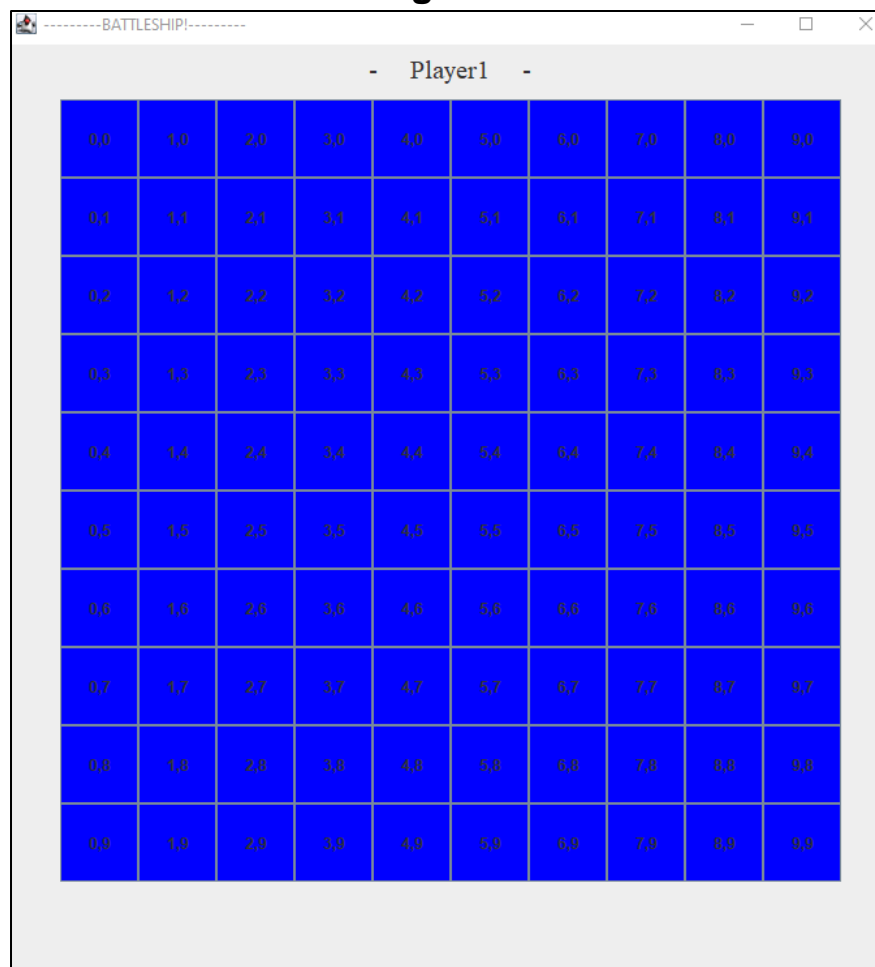
### **II.1.4. User Acceptance Testing:**

User testing was very informal and conducted by us. We were working on our game until a very late hour and could not find a willing and awake participant to test our software. We tried to conduct the test acting as users who had never seen the software before, trying to re-guess already guessed coordinates and attempting to edit text that should not be editable. However, since we did not finish our game completely, we were unable to finish user acceptance testing beyond this. Ideally, we would add more features to complete the game (see VII. Future Work) and then have friends and other users run through a few Battleship games to ensure there are no hidden bugs and fix them if there are.

## VI. Project Implementation Description

On startup, the current program version produces a window with the player's name and a fully generated 10x10 grid for the player to start taking guess at with ships already randomly placed on the game board. Currently the ships are successful in generating within the bounds of the board however there is still a problem with ships overlapping one another. This functionality can be added in future versions. This initial game board can be seen in Figure 3.

**Figure 3**

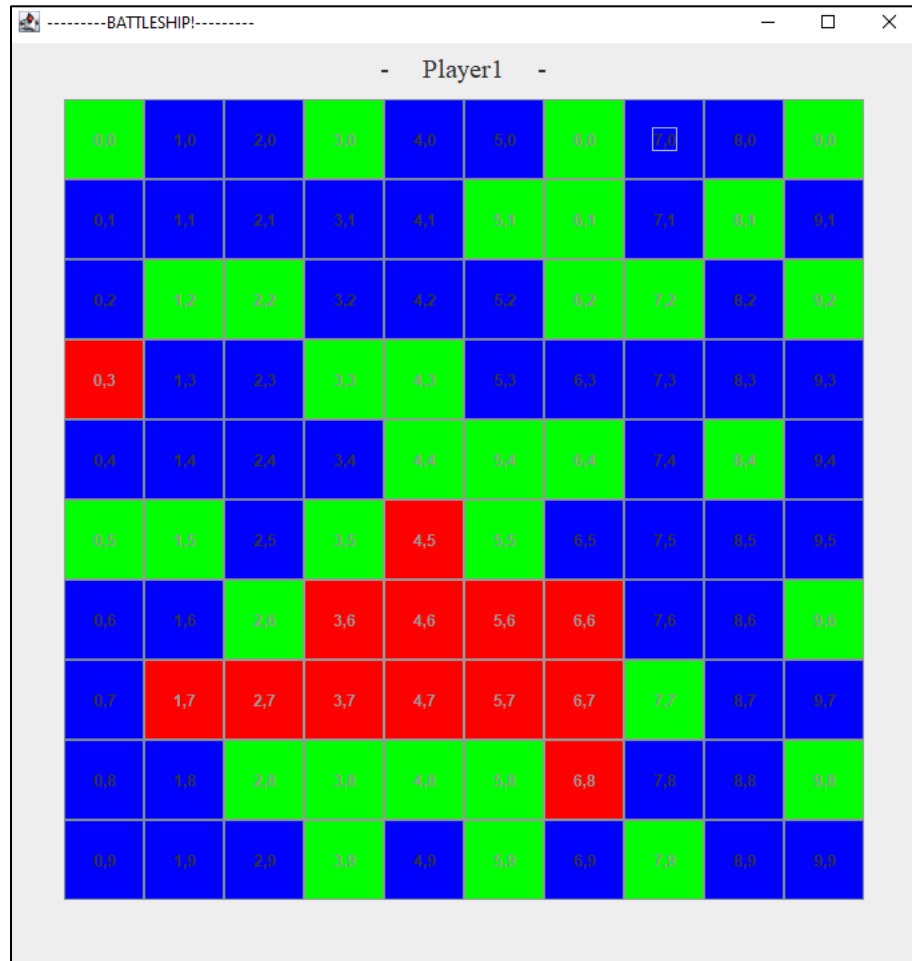


The screenshot shows a window titled "BATTLESHIP!" with a subtitle "Player1". Inside the window is a 10x10 grid of blue squares. Each square contains a coordinate in the format "row,col". The rows are labeled from 0,0 at the top-left to 9,9 at the bottom-right. The columns are labeled from 1,0 to 9,0 in the first row, and so on, following a sequential pattern across the grid.

0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4	9,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5
0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6	9,6
0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7	9,7
0,8	1,8	2,8	3,8	4,8	5,8	6,8	7,8	8,8	9,8
0,9	1,9	2,9	3,9	4,9	5,9	6,9	7,9	8,9	9,9

Because the board's grid is made of buttons, the guessing functionality only requires that the user use the mouse to select their desired coordinate. After the player guesses, the coordinate that the player selected is colored either red or green, red to symbolize a hit ship and green to symbolize a missed guess. This can be seen in Figure 4.

**Figure 4**

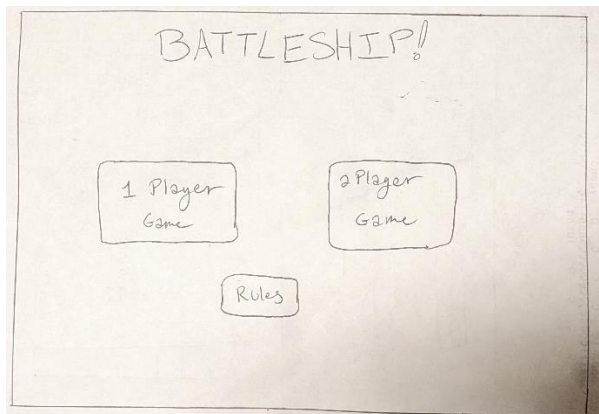


After a coordinate is guessed and revealed, the player can no longer click on that button and must guess at a different coordinate.

Currently, there is no end-of-game functionality, and the program just continues until every coordinate has been guessed, to which the program then sits stagnant. It is also important to note that at the time of submission, we were unable to prevent ships from overlapping.

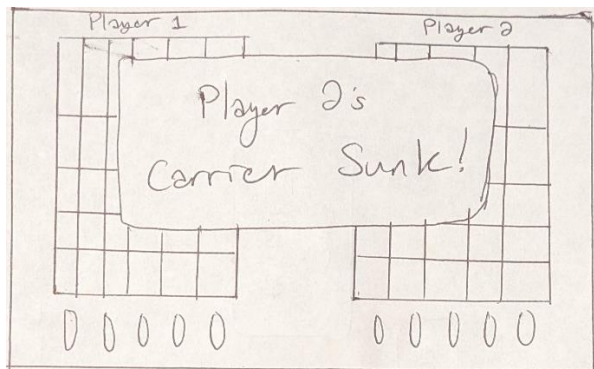
[GU-2023-Spring-CPSC224/final-software-dev-project-git-gud: final-software-dev-project-git-gud created by GitHub Classroom](#)

## VII. Future Work



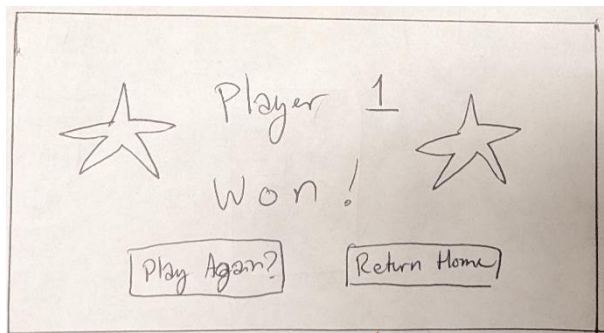
### Implementing a Home/Starting Screen:

Currently, our game starts up immediately to a one-player version with the board. We would like to add a starting screen that allows the user to select a one-player or two-player version. This screen will also have a button that will bring the player to a list of rules for Battleship in case the user is a first-time player.



### Implementing a Ship Sunk Pop-up:

For our current implementation, ships can be 'hit' and technically sink, but there is no indicator for the ship sinking. Adding a pop-up will give the users feedback on the ship that is sunk as well as whose ship was sunk. Ideally, we would also have small ship icons at the bottom of the board that would turn red once the ship has been sunk.



### Winning/Game End Screen:

One of the main things we would have liked to implement had we had time, was a way to end the game. The idea would be each player would have five ships and once their ship count hit zero, the other player would win displaying this screen. The user can then select to play again or return to the home screen.

### Fix Ship Overlapping:

Another of the main things we wanted to implement was to fix the problem of ships overlapping. Currently, when ships are placed randomly ships can overlap. Due to this issue, it is also difficult to keep track of how many ships are left, preventing us from creating an end case.

Overall, had we had more time or figured out how to work together as a group sooner, we would have wanted to implement these items to create a more complete game.

## VIII. Glossary

- **Unit test:** A test that focuses on a small piece of the program (a unit).
- **Integration test:** A test for program behaviors and processes.
- **Functional test:** A test for mandatory methods and behaviors that are essential to make the program run.
- **User acceptance testing:** the final stage of software development where QA testers test the program before launching the final product.
- **Default case:** When the class is implemented, what are the automatic states of the class?
- **High priority function:** Functions that are detrimental to the game running properly and fulfilling functional requirements.
- **Issue:** GitHub issue. Assigned to people and used to keep track of problems and any requirements/tasks that need to be completed.
- **Class:** An object of the program. In our cases, “Player”, “Ship”, “Board”, etc.
- **Function:** A portion of the class that fulfills a certain task.
- **Bug(s):** An error, flaw or fault in the design, development, or operation of the program

## IX. References

Wikipedia contributors, "Battleship (game)," *Wikipedia, The Free Encyclopedia*, [https://en.wikipedia.org/w/index.php?title=Battleship\\_\(game\)&oldid=1151197061](https://en.wikipedia.org/w/index.php?title=Battleship_(game)&oldid=1151197061) (accessed May 11, 2023).

*Kids Toys, Action Figures, Toys Online - Hasbro*,  
[www.hasbro.com/common/instruct/Battleship.PDF](http://www.hasbro.com/common/instruct/Battleship.PDF) (accessed 11 May 2023).

## X. Appendices

### Appendix A:

