Battleship Game: Project Documentation

CSCI 4448/5448

Divya Bhat, Tanvi Gopalabhatla, Sahand Setareh, Tahmina Ahmad

Table of Contents

Project Description

Our project is a new, updated version of the traditional Battleship game. The game consists of the traditional features of Battleship, including creating a fleet of ships that include a Battleship, a Destroyer, and a Minesweeper to be placed on the board. The ships are of varying lengths as per Battleship rules, and are ultimately used to defend yourself against your opponents. With the addition of a submarine and a space shuttle, all player's ships can be placed on 3 different maps: ocean, underwater, and in space. All of the ships together are your fleet, and you can choose to move your fleet north, south, east, or west by one cell at any time. You can undo your move fleet or redo your moves as long you're not moving it in the direction that there is a ship on the edge. Our game brings new twists and turns to the original, but the end goal is the same - launch attacks against your opponent in order to sink their ships, and whoever's ships sink first loses!

Development Process

Iterative Development

Our team used iterative development throughout the entire process of building our Battleship game. Iterative development consists of breaking a huge software project into smaller chunks and working on the smaller portions. This was facilitated for us by the instructors of the course, who divided the project into six separate milestones. We divided the project further by splitting the milestones into workable portions. For example, during our preliminary meetings for each milestone we would write up the requirements and our desired implementations of them. This involved making lists, planning, and rough designing. Then, we would split up our plans by the specific features and by the class that the code would fall under and prioritize them in order of what needs to be developed first; then, we would work on each portion at a time in that order. Since we practiced test driven development, we wrote the test for the feature first, and then coded the classes and methods. As we progressed through each milestone, we would make sure that the previously implemented features and code were finished and continue working with the addition of new code.

Refactoring (Screenshots are in Appendix A)

Throughout our project, we went through many phases of refactoring. The biggest one occurred in Milestone 3 when we started to abstract out classes in order to improve extensibility. Since then, we created 6 abstract classes: Ship, Weapon, Boost, Disaster, Map, and Animal. Each of these abstract classes is able to create different subclasses that extend the superclass, and implement the same functions as in the main superclass as well. Doing some major refactoring in this phase made the process moving forward much more efficient. Apart from that, we have refactored many small methods in order to minimize code smells and adhere to coding principles. Here are some of our main refactoring highlights:

- (Code Smell) *Inappropriate Intimacy*: Completely reorganized and re-coded most classes in our project in order to prepare for additional features, a multi layer grid (underwater and above ground), new weapons, and new functionalities.
    - The ship coordinates are no longer baked into ship
    - Ships are abstracted → no longer embedded in ship class
    - Animals → Narwhal and Jaws
    - Maps → OceanMap, UnderwaterMap, and SpaceMap
    - Disasters → Hurricane, Ghost Zone, and Asteroid field
- (Code Smell) *Inappropriate Intimacy*: Code now abides by a design principle of being loosely coupled where our components have little or no knowledge of the definitions of other separate components.
    - Example: Ships are not aware of their locations, simple their size and which direction coordinates need to be added to make up their length
- Created test classes for each of our classes instead of all in one test file
    - Each functionality that is incorporated into the program has its own corresponding test class, so that the tests aren't placed in a single class for better organization
- (Code Smell) *Duplicated Code* and *Oddball Solutions*: Deleted attackUnderSpaceShuttle() function
    - This function was implemented in order to hit all the boats that lie beneath a space shuttle that had just been sunk. To do this, it utilized similar to identical functionality of the deployWeapon() function in bomb, with minor differences in print statements
        - We realized that a huge chunk of code was duplicated and instead we could use existing code implementations!
        - In order to refactor this code, we simply added a couple if-conditions and extracted out print statements into a separate function, and called the Bomb deployWeapon() function in the SpaceLaser. This allowed us to completely remove the attackUnderSpaceShuttle function and eliminate duplicate functions
    - (Code Smell) *Combinatorial Explosion*: Many areas of our code did the same thing with small changes in print statements
        - For instance, our Jaws, asteroids, bomb, and space laser attack methods relied on using similar deployWeapon() functionality, but required small print statement changes
        - In order to refactor this, we created a new function specifically for outputting messages based on a series of print statements

- Simplified checkAvailability() function

  - This function (and many similar functions) used a simple if/else condition to check whether the player had maxxed out their use count for a boost or weapon
  - This was able to be refactored by modifying the condition into a one line return statement
- Changed to lowercase functionality in all functions that takes in user input
  - For any area where there is a string user input, we have refactored the validation to convert all inputs automatically to a lowercase character, to allow for more user freedom during the game
- (Code Smell) *Bloater, Long Method*: MoveFleet() function in Player class
  - This function started out as a very long method, with two different sections of for loops; the first section being a validation for whether the ships can be moved in that direction, and the second section being the actual loop that moves the ships
  - To refactor this, we extracted out the first section into its own function called validateMoveFleet(), which could then be called within the moveFleet() function, as well as any other method that requires validation of moving ships
- (Code Smell) *Comments* and *Dead Code*: Many methods have comments that are superfluous and classes have code that is not being used
  - Method comments need to show the purpose of the function rather with less emphasis on what the code is explicitly doing
  - We made sure to eliminate any comments that are obscure and are not required to understand the code directly below it, keeping minor in-line comments
  - Additionally, we reviewed each and every file and deleted dead code (such as parameters, local variables, getter functions, etc.) that was not being used in the rest of the code.
- (Code Smell) *Inconsistent Naming:* Making sure we followed coding conventions between methods and variable names
  - Initially, we outlined coding conventions that we wanted to practice throughout the project, such as using camelCase for method names, and underscore for variable names
  - Occasionally throughout the project, we started to disregard the coding conventions as we got deeper into coding, so we made sure to refactor and rename all the variables and method names to their respective coding conventions

Testing

We utilized *Test Driven Development* by converting the designated game requirements into tests first, and then developing the software while testing it against the test cases. Many of our tests revolved around checking if the class objects were created properly, and whether they performed intended functionality in every possible scenario. For example, in the weapons tests such as Bomb and Sonar Pulse, we created different tests by placing ships on various parts of the map, performing hits/misses on them, and checking if intended success messages were returned. A major statement that was really beneficial was the assertEquals statement, in which we were able to see a true or false statement determining the success of a function call. Similarly, we did this sequence of steps for testing all of the other classes and possible scenarios. In the end, it became very helpful to run these tests prior to implementing our code so that calling the functions in the main Game class becomes a much smoother process.

Collaborative Development

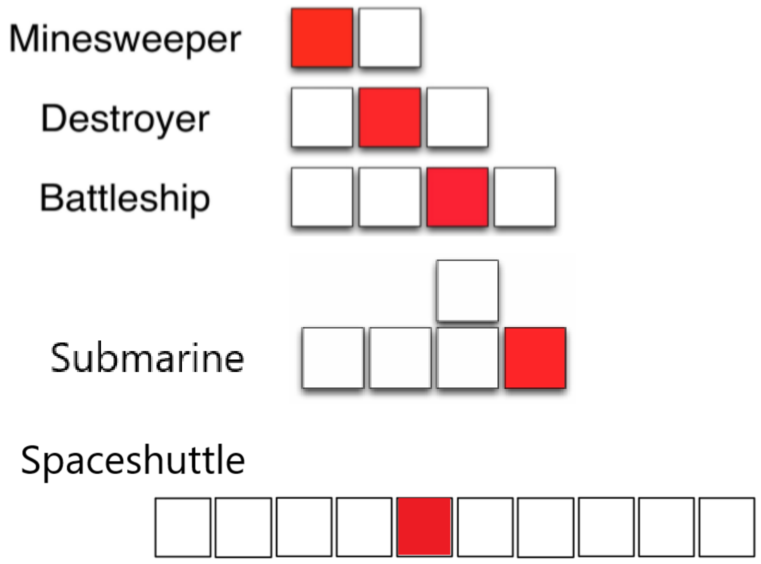
Our team used a collaborative development process while building our game. This means that the majority of the project, if not the entire project, was done whilst working in a group setting. Due to COVID-19, we implemented this process through Zoom meetings. For all our meetings, we had all team members attend for the entire duration of the meeting as much as possible. For the planning and designing phases, we would all discuss our features, action items, design, prioritization, and more. For coding, we often used a system where one person shared their screen while coding, and the other three team members would talk through it. We would switch off coding and screen sharing throughout the meetings. We also split into groups of two and went into breakout rooms, where we used *pair programming*. Both methods were effective for us to finish as much code as possible in an efficient manner.

Requirements and Specifications

Basic Requirements

- Ships: can only be placed horizontally or vertically
    - Minesweeper: 2 cells long
    - Destroyer: 3 cells long
    - Battleship: 4 cells long
    - Submarine: 4 cells long, 2nd to last cell is 2 cells wide (5 total cells)
    - Space shuttle: 10 cells long
- Captain’s Quarters:
    - If the captain’s quarters are “hit”, the entire ship sinks, regardless of the status of its other elements.

- In addition, the captains quarters for battleships and destroyers (but not for minesweepers) are armored, meaning that it takes two attacks on the same square in order to "hit" it
- Locations:



Weapons & Boosts:
The weapons in the game allow the player to play offensively against their opponent. As a player progresses, they receive upgrades or additional weapons in their arsenal. Each player starts with one weapon: the bomb. The bomb can be used to attack any one of the opponent's ocean map ships. When a player successfully sinks their first ship, they receive an additional weapon called the sonar pulse and their bomb upgrades to a space laser. The sonar pulse may only be used a maximum of 2 times and allows a player to view a small section of their opponents battleship grid, allowing them to see if there are ships there. On the other hand, the space laser attacks on all maps, including the space map, ocean map, and underwater map.

The player also has their own set of boosts. At the start of the game, each player is given a Lifesaver boost, which can be used to revive sunken ships on any of their maps. A player must have a sunken ship in order to use the Lifesaver, and can only use this boost up to 2 times.

Animals:
Our game also includes special animals. Before every turn, the player may encounter a shark or a narwhal under one of their ships. A shark will bite off a part of your ship (similar to an attack on that cell), while a narwhal will give you an extra use for your sonar pulse.

Disasters:
There are 3 different types of disasters that can occur on each map - an asteroid field, a ghost zone, or a hurricane. These disasters are randomized before every turn, with a 10% chance of occuring during a player's turn.

1) An asteroid field disaster fires 10 randomly placed asteroids at the current player's space grid and has the chance of hitting a space shuttle in the area. only occurs in space and fires 10 random asteroids and may hit a space shuttle in the area.
2) The ghost zone disaster spans a set of coordinates, of a randomized size, appearing on both the current player's ocean map and their underwater map. Within the ghost zone, the current player faces mysterious ghosts that take over the offensive grid. These ghosts scramble the hit/miss data on the player's offensive grid randomly, thus, when a player encounters a ghost zone, they have the chance of seeing incorrect information about where they hit or missed when they attacked their opponent.
3) Finally the hurricane disaster tosses and turns ships caught on its borders. It only occurs on the ocean map and moves the ocean maps ship in specific directions based on their location on the hurricane border.

Maps:
There are 3 different maps that our game uses. This includes an Ocean Map, a Space Map, and Underwater Map. Certain types of ships can be placed only on certain maps. For example, submarines can be placed on the Underwater map and Ocean map, minesweepers, destroyers, and battleships can only be placed on the Ocean map, and Space Shuttles can only be placed on the Space map. The disaster features can also occur on specific maps. For example, the Ghostzone disaster can occur on the Ocean or Underwater map, the Hurricane can occur only on the Ocean map, and the Asteroid Field can only occur on the Space map.

User Stories

- Player Perspective: As a user of this game, I will have the capabilities of launching attacks that hit, miss, sink my opponent's ships, so that if I sink all my opponent's ships, I would win.
- Player Perspective: As a user of this game, I will have the capabilities of launching attacks that hit, miss, sink my opponent's ships, so that if I sink all my opponent's ships, I would win.
- Player Perspective: As a user of this game, I will have the capabilities of creating a fleet of ships that include a Battleship, a Destroyer, and a Minesweeper, Submarine, and Spaceshuttle and have the ability to place them on my board which will act as my defense against my opponent throughout the game.

- Player Perspective: As a user of this game, I should be able to hit the captain's quarters on an opponent's ship either once or twice and sink the entire ship. (If the ship is a minesweeper, space shuttle, or submarine it will require 1 hit on the captain's quarters, and if the ship is a battleship or destroyer it will require 2 hits on the captain's quarters)
- Player Perspective: As a user of this game, if I have sunk at least 1 ship, I should be able to use a weapon called the sonar pulse twice within the game that allows me to see the status of the cells on a portion of my opponent's battleship grid.
- Player Perspective: As a user of this game, I will have the capabilities of adding a submarine at the surface or below the surface, so that if I sink my opponents submarines, I would increase my chances of winning and if my submarines are hit by my opponent, my chances of winning decrease.
- Player Perspective: As a user of this game, once I sink at least one opponent ship, I will have the capabilities of using a new weapon called the space laser, and have the ability to penetrate through all of my opponents maps. The space laser will = be able to hit a space shuttle, a surface ship, and a sub that is placed at the same coordinate on different maps at the same time, which will act as an attack against my opponent throughout the game.
- Player Perspective: As a user of this game, I should be able to move my fleet. When the command to move is given, depending on whether I choose N, S, E, or W, each ship will move one position in the given direction. I should be allowed to undo and redo moves, but I will not be allowed to move a ship if it is at the edge of my board. This will allow me to have more strategy and options when I'm playing the game, which may increase my chances of winning.
- Player Perspective: As a user of this game, I will have the capabilities of adding a Space Shuttle to the Space Map, so that if I sink my opponent's Space Shuttle, I would increase my chances of winning and if my Space Shuttle is hit by my opponent, my chances of winning decrease. If a Space Shuttle sinks, it counts as a hit against any possible ships on every coordinate that lies underneath the space shuttle on the corresponding OceanMap and Underwater Map.
- Player Perspective: As a user of this game, I will have the capabilities of reviving two sunken ships by using my Lifesaver boost.
- Player Perspective: As a user of this game, my game maps have the capability to randomly place a narwhal or shark under one of the ships, which will either grant me more uses for my sonar pulse, or bite off part of my ships, respectively
- Player Perspective: As a user of this game, at the beginning of every turn, there will be a random disaster that may occur on each of my maps. I will have the ability to see the locations of these disasters, and their effect on my current fleet.
- Player Perspective: As a user of this game, there will be a menu by which the user can play the game and interact with it. This menu feature allows the player to randomly place a fleet, or manually place ships on the grid, as well as show grid

status, use different weapons at will, show score, use boosts, move the fleet, and surrender the game.
- Player Perspective: As a user of this game, the user should be able to generate a fleet of ships and place the fleet randomly on the grid, or manually place each of their ships.
- Creator Perspective: In order to create this game through Java, as the developers, we would implement the appropriate classes and methods allowing us to simulate a modified Battleship game. This game would be developed according to the requirements and features given to us by the instructors as well as according to new features we devise ourselves.

Architecture and Design

Description

The main driver of the game program is the Game class, initially called through the program's Main() method. Game will create two Players, each with their user-inputted names. Maps are created for each player when the players themselves are created, each Map having constituent Offensive and Defensive Grids which keep track of different aspects of the game (Offensive Grid keeps track of a Player's aggressions towards the other player and Defensive Grid stores the locations and statuses of one's own ships). Two Animals, the Narwhal and Jaws are created and placed at random locations on each Map, either giving a player-perk or damaging a player's respective ship when that ship or a ship is placed on the same Coordinate.

Game() then prompts the user to place their ships randomly or manually. If a player chooses to place their fleet of ships at random, Game calls the Fleet() constructor which automatically places a group of ships at valid locations at random. If they choose to place each ship manually, they are given prompts for coordinates and ship directions to which their ships would be placed.

Ships inherit from a Ship superclass, but implement different interfaces (such as Submersible, Orbitable, or ArmoredShip, which restricts their ability to be placed on certain Maps and their unique interactions with certain functions (for example, Armored Ships have a unique defensive behavior). After the process of prompting the user to place their fleet is complete, it is repeated for the second player. Then, the first player's turn begins.

The beginning of each turn brings the chance of a Disaster to a player's SpaceMap, OceanMap, or UnderwaterMap. Game() determines if a Disaster is going to occur that turn, and if so, it appropriately creates a Disaster object and calls applyDisaster() which affects that player's corresponding Map. The player is then notified of the consequences.

Afterwards, the main part of the player turn will commence. Game() displays a variety of options to the user such as viewing their Maps and corresponding Grids, displaying the score and the health of their ships, using a weapon, using a Boost, moving their fleet, undoing a previous fleet maneuver, and surrendering. Every option chosen by the player prompts Game() to call corresponding getters and setters to display additional menus (such as which weapon to use) or information (such as ship statuses and locations on a certain defensive Grid). Many classes employ the abstraction that is a Coordinate Object.

Each Coordinate contains constituent X and Y values, corresponding to a location on a Map. Objects that have multi-turn presence on a Map, such as Ships, generate Coordinates which are kept track of on their associated Maps. Player Actions to move fleet, as well as related undo and redo operations, also access Coordinate attributes. Disasters act uniquely on certain Coordinates themselves. Using this universal representation of location by many classes allows for seamless interactions and relationships between the various aspects of the overall Battleship game, all managed and sequentially accessed by the Game() class.

Design Patterns

Strategy Pattern

"Defines a family of algorithms, encapsulates each one, and makes them interchangeable" (Head First Design Patterns, 2nd Edition)

In our battleship game, we implemented the strategy pattern through our ship, weapon, and boost features. We have an abstract class named Ship that has multiple subclasses including Minesweeper, Destroyer, Battleship, Submarine, and Spaceshuttle that all inherit from Ship. These subclasses offer various different implementations of a ship, encapsulating functionalities within their classes. They can then be placed on the grid interchangeably from within the game class. Similarly, we designed an abstract weapon class and boost class. Both have respective subclasses: bomb, space laser, and sonar pulse inherit weapon and lifesaver inherits boost. From the game class, the user can choose to use one of their weapons or boosts, and the useWeapon() or useBoost() methods are called. These methods take in a weapon or boost respectively, allowing them to conduct various implementations and actions based on what is passed in.

Decorator Pattern

"Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality" (Head First Design Patterns, 2nd Edition)

We also implemented the decorator pattern by adding additional functionalities and requirements to our ship classes. Since our ships include submarines that can exist underwater and space shuttles that can exist in space, two interfaces were created: SubmersibleShip and OribitableShip. These interfaces ``decorate" the ships with certain characteristics that allow them to be differentiated from other ships. This also allows for extensibility because any responsibilities of a certain type of ship that is different from a regular ocean ship can be added to these interfaces. Similarly, we also created an ArmoredShip interface that describes a ship that has an armoured captain's quarters. This cleaned up a lot of our code because it allowed us to keep track of ships that needed to be hit twice to sink vs. ships that would sink immediately if their captain's quarters were attacked.

Command Pattern

"Encapsulates a request as an object, thereby letting you parametrize other objects with different requests, queue or log requests, and support undoable operations." (Head First Design Patterns, 2nd Edition)

The command pattern is used in our game through our move fleet, undo, and redo features. Everytime a player decides to perform these actions, their movement is captured within an action object. This action class encapsulates movement coordinates and the methods that need to be called in order to generate the opposite action, undo the action, or redo the action. The action objects are then added to a stack data structure within the player class which allows the program to keep track of the actions that have been done by the player. This also allows the program to pop off the last committed action in order to undo it. Additionally, the player holds another stack data structure variable that keeps track of the actions that were undo-ed. This allows for the player to be able to redo the move as well.

Composite Pattern

"Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." (Head First Design Patterns, 2nd Edition)

In order to incorporate the composite pattern, we created a class called Fleet that composes together individual ships and places them on the player's maps. It allows the player to provide multiple ships and then places them randomly through a placeFleet() method. The players, who can be considered the clients of the game, treat both the fleet and the individual ships uniformly by being able to place both an individual ship or a whole fleet on their battlefields. Ultimately, the part-to-whole relationship is created from the ship to fleet relationship, where both the ship and the fleet implement an interface called component.

Design Principles

Our project follows many of the basic object-oriented design principles, including a few of the S.O.L.I.D principles. (Quotes taken from Thorben Janssen , "SOLID Design Principles Explained")

- Encapsulation - "Objects should manage their own behavior and state"
    - Many of the objects that are created in our game keep track of their own status, unless doing so would result in high coupling. For instance, a ship object is only aware of its size, coordinates that make up its length, and name, rather than its locations on the actual map
- Single Responsibility - "*A class should have one, and only one, reason to change.*"
    - Each of our classes have a single responsibility or main function that it performs. For example, our Bomb class performs the function of attacking a cell of a ship, our Sonar Pulse class is in charge of displaying a portion of the offensive grid, etc. We ensured not to give too many purposes for each class to keep them loosely coupled
- Open-Closed Principle - *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"*
    - We have 6 abstract classes that are extensible, allowing us to create different types of objects per class. As mentioned previously, the Ship class is able to create new ships, the Map class is able to create new Maps, the Weapons class is able to create new weapons, etc.
    - These abstract classes are closed for modification because they contain private variables and abstract functions that must be implemented in every subclass. Therefore, they are open to extension, and closed for modification
- Interface Segregation - *"Clients should not be forced to depend upon interfaces that they do not use."*
    - We have a total of 4 interfaces: ArmoredShip, SubmersibleShip, OrbitableShip, and Component Interface
    - Each interface is specifically designed to attribute to different types of ship. For example, ships that are armored such as Battleship and Destroyer implement the ArmoredShip interface. Ships that can be placed underwater such as Submarine implement the SubmersibleShip interface. Ships that can be placed in space, such as Spaceshuttle, implement the OrbitableShip interface. Finally, in order to create a part to whole relationship that abides by the composite pattern, the Ship and Fleet implemented the Component interface.
    - Evidently, the clients only implement the interfaces that they utilize and not those that they do not utilize

UML Diagrams

View the last pages of documentation to see our 3 different UML Diagram iterations, respectively (see Appendix B).

Throughout the course of this project, we developed UML diagrams in order to visualize how the architecture of our system looked from the top-down. In homework 2, we developed a diagram for our initial set of features, prior to doing some major refactoring and abstracting out classes. Following that, we developed a second diagram to visualize our new system which was more loosely coupled and extensible. Finally, our last UML diagram that was developed includes our additional custom features, as well as specific cardinality/relationships between the classes.

Personal Reflections

Tahmina Ahmad

Overall, I found the project to be very difficult filled with both good and bad experiences. For example, it was really nice to get reacquainted with Java, as I first learned Java in high school but have not used it since. The Battleship game, especially for the first few milestones, was interesting because it allowed me to work with my team and use data structures that I haven't used since I took Data Structures in 2019. The features that we were implementing allowed for me to understand how different classes and features work together. I also applied what I learned in other classes, like Software Development, to this project. An aspect of this project that was really difficult was the different features that kept building on top of each other. At times, it felt like different features contradicted each other or had to be built on top of each other, when it could be easier by eliminating a few features. Or, we perhaps could have had more direction in the initial milestones that would better lead us to building new functionality on top of old functionality. I also feel like while the grading interviews were a good idea to provide students with some direction, it felt like our work was being overlooked because only one or two features would be looked at, and nothing would be looked at in depth. I think we did a really good job with some parts of our project, but they weren't really actually looked at or acknowledged before we moved on to the next milestone. I am really thankful for my amazing teammates, for being so flexible, so kind, and so motivating during this long and difficult project. Without them or with a different team, I know that the project would have been a much more difficult experience. Being with my friends allowed me to collaborate with them, learn more about them, get closer to them, and spend time with them. It was really cool to learn and develop the game with them. We often spent upwards of 20+ hours together, and although we did pair programming, we also worked together in our group of four, which really built a sense of togetherness for us. Overall, the project was a good experience and I developed some skills, regained some old skills, and had a positive experience.

Divya Bhat

Throughout the course of the project, I was able to see growth in my skills as a professional in the field of computer science as well as a programmer and designer. One of my personal skills that I have improved on is my adaptability. Throughout this project, we were given various additional feature requirements at every milestone that felt completely randomized and extraneous to our most recent code. At many instances, the new requirements felt as if they contradicted everything we had already done. I had to be extremely adaptable and be ready to face whatever was thrown our way and keep a positive and optimistic mindset. Now, I feel more confident and prepared to face obstacles in coding projects in the future. Additionally, I also learned more about working on a team. It was my fortune to have an excellent group to work with. Even after spending endless hours planning and looking at code, my team members stayed motivated through the end. Each member was willing to put in the time and the effort it took to build a coding project of this size. Looking back, one aspect of our process that was essential to our success was the amount of paired programming we did. We built features of the code by sharing our screen over zoom and actively talking through our code. This allowed us to bounce ideas off of each other and hold each other accountable for proper design in our code and completing tasks on time.

I also believe I have become a much better programmer, especially in regards to object-oriented design. The project pushed me to learn more about making code that was extensible and that abided by multiple design principles. Through numerous milestones and multiple refactorings of the code, I've learnt that code is much easier to work with when it follows proper design. Additionally it was a fun challenge to see how we could morph our code in order to accommodate for object-oriented design. For example, we received a comment from our TA that we could implement the command pattern for our move-fleet functionality. Even though we didn't see the need for this pattern in our code, we took this as an opportunity to learn and grow, causing us to sit together and reformat our code to fit the desired expectations.

While I reflect on my growth, I would also like to reflect on my struggles. One aspect of the process I didn't expect was the minimal feedback from instructors. Often it felt like we weren't aware of the expectations of the project and sat through many hours of coding only to have points deducted at the end of a milestone. While my focus was more on our growth and not each point we received, I believe we could have ultimately learned more if we were given better feedback from instructors on how we were progressing on our project. Especially in the moments where we had to completely delete and redo our code in order to accommodate for better design, I look back and wish we had more in-depth discussions with our TAs about our implementation. Instead we were often left to our own devices to figure out how to be better designers and now I know, in the future, I will take more initiative to do so. Overall, however, I look back at my time working with my group and designing our version of the battleship game and see my own progress. I am thankful for the opportunity to build a game of such intricate design with a group of strong designers and programmers.

Tanvi Gopalabhatla

Throughout working on this project, I have encountered many situations - whether it be positive or negative - which have greatly influenced my knowledge and experience of OOP. I will start off by saying that I was very fortunate to have amazing team members that each brought their own skills to the table. Everyone's respective cooperation and willingness to work together day and night really made the process more efficient and enjoyable. Something that became really difficult for me (our team as well) was the foundation of the project. A couple of the early milestones had us lay out custom features we wanted to implement, but also required us to deliver working code. It was only after these couple code deliverables that we started learning more about OOP design patterns/principles and UML/sequence diagrams in class, so refactoring our code to account for those took an excessive amount of time in one week. However, after this major refactoring phase was done, we were able to write better code. Overall, although I struggled with many aspects of this project, I'm really excited that I got to work with a great group and have a final working Battleship game that I can put on my resume!

Sahand Setareh

This project has been incredibly useful for me as an aspiring software engineer. From planning and design to implementation and test-driven development, I have greatly benefited from the multitude of concepts explored through lecture and tangibly applied in a hands-on, team oriented project. Industry practices such as UML/Sequence diagrams, object-oriented analysis and code reviews, refactoring and identifying code smells have all proven to be exceptionally helpful in this regard. I feel confident that the concepts I have learned can aid me in my career for years to come. Not only has this industry-style of development been enlightening and purposeful, it opened my eyes to the benefits of good code design tailored towards testing and purposeful design patterns. I am also incredibly grateful for my teammates. Not only have they been great to work with and learn from a technical perspective, but they have really made a difficult remote semester so much more enjoyable through their uplifting, motivating personalities. I have learned so much about different workstyles and the importance of collaboration, all of which will serve me well in any project.

Citations

Freeman, Eric, and Elisabeth Robson. Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software. 2nd ed., O'Reilly Media, 2020.

Atwood, Jeff. "Code Smells." *Coding Horror*, Coding Horror, 7 Aug. 2019, blog.codinghorror.com/code-smells/.

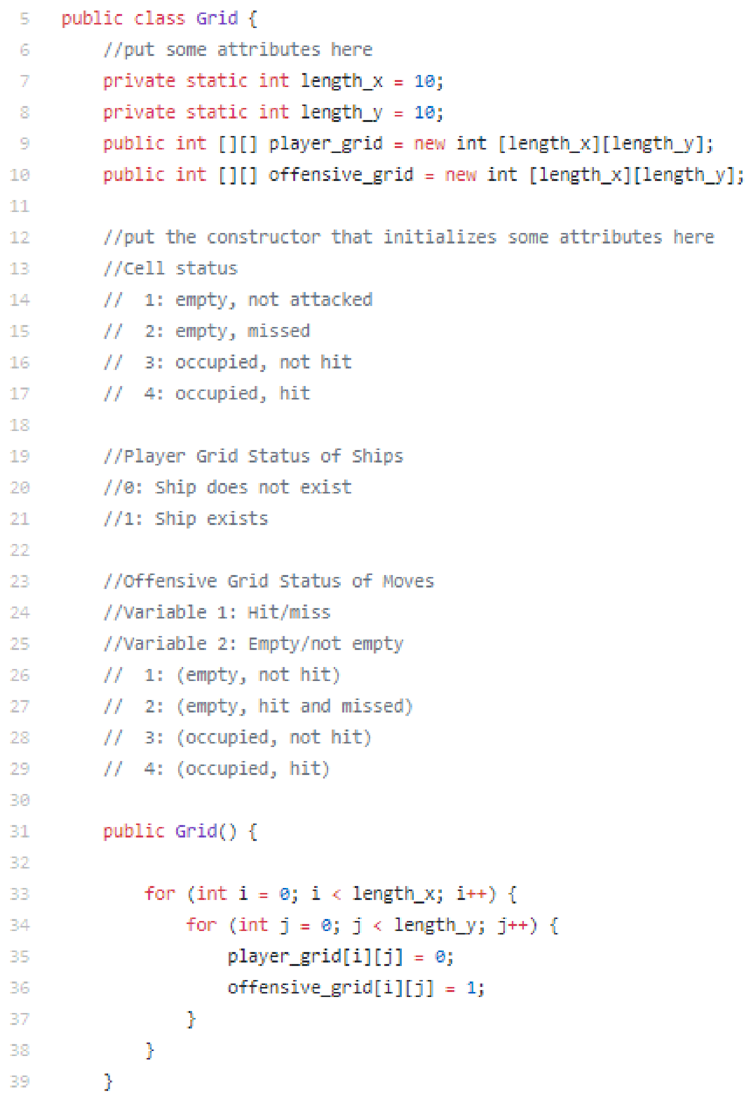
How to Write Doc Comments for the Javadoc Tool, www.oracle.com/technical-resources/articles/java/javadoc-tool.html.

Thorben, Janssen. "SOLID Design Principles Explained: Interface Segregation with Code Examples." *Stackify*, 28 Apr. 2020, stackify.com/interface-segregation-principle/.

APPENDIX A

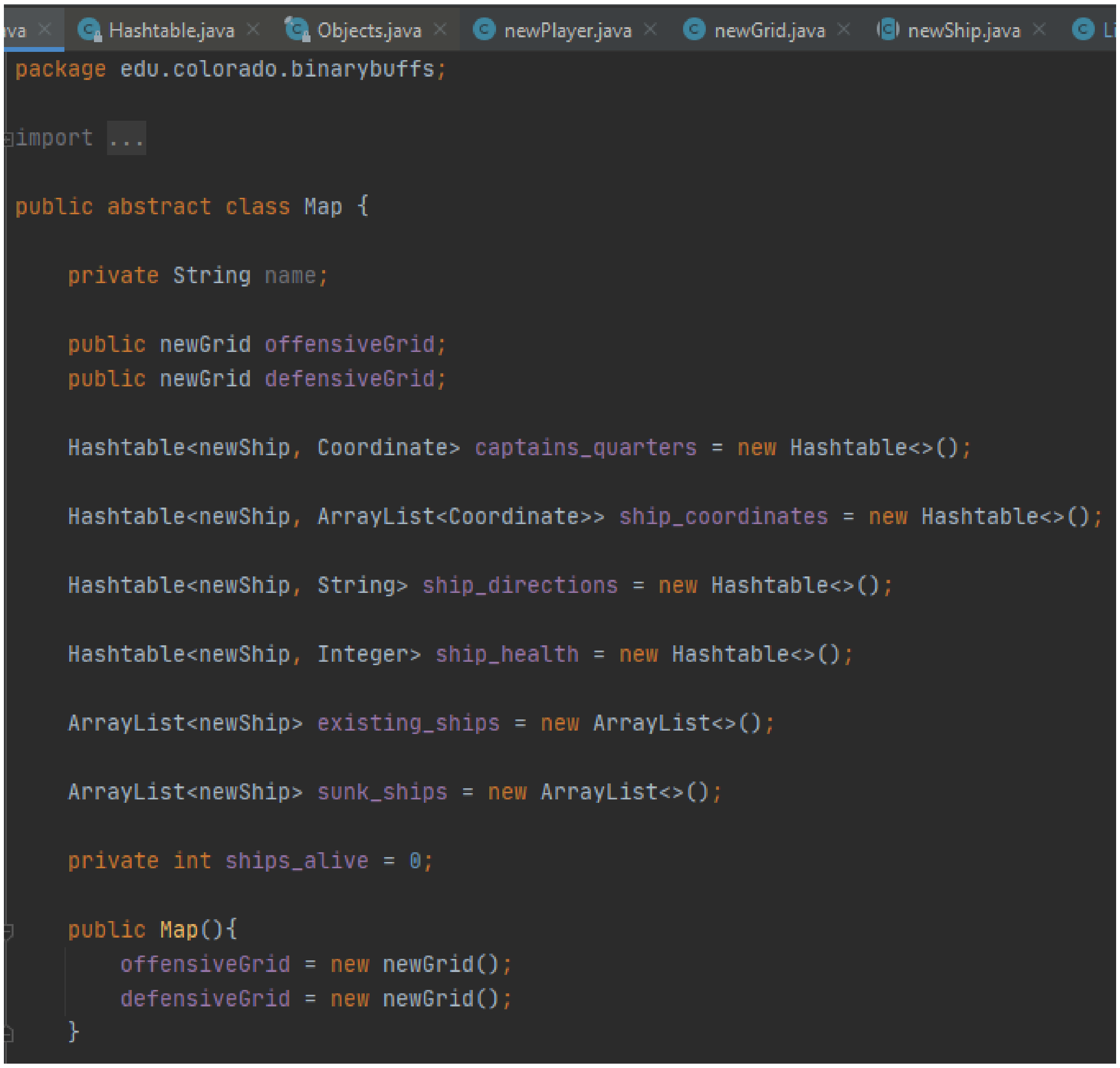
Screenshots

1a) Previous method for implementing player/opponent grids:



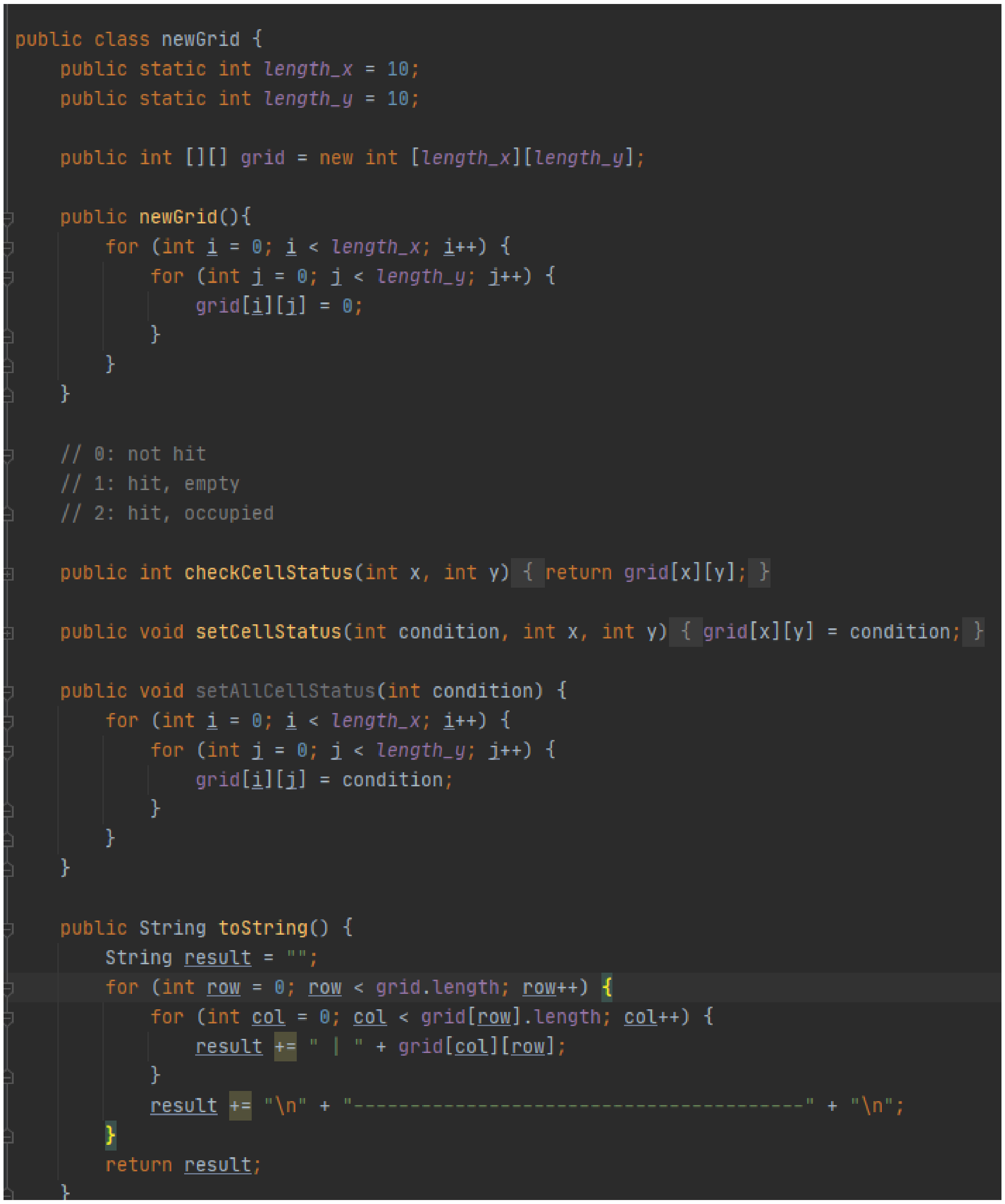
```java
public class Grid {
    //put some attributes here
    private static int length_x = 10;
    private static int length_y = 10;
    public int [][] player_grid = new int [length_x][length_y];
    public int [][] offensive_grid = new int [length_x][length_y];

    //put the constructor that initializes some attributes here
    //Cell status
    //  1: empty, not attacked
    //  2: empty, missed
    //  3: occupied, not hit
    //  4: occupied, hit

    //Player Grid Status of Ships
    //0: Ship does not exist
    //1: Ship exists

    //Offensive Grid Status of Moves
    //Variable 1: Hit/miss
    //Variable 2: Empty/not empty
    //  1: (empty, not hit)
    //  2: (empty, hit and missed)
    //  3: (occupied, not hit)
    //  4: (occupied, hit)

    public Grid() {

        for (int i = 0; i < length_x; i++) {
            for (int j = 0; j < length_y; j++) {
                player_grid[i][j] = 0;
                offensive_grid[i][j] = 1;
            }
        }
    }
```

1b) New method for implementing player/opponent grids and various maps:



```java
package edu.colorado.binarybuffs;

import ...

public abstract class Map {

    private String name;

    public newGrid offensiveGrid;
    public newGrid defensiveGrid;

    Hashtable<newShip, Coordinate> captains_quarters = new Hashtable<>();

    Hashtable<newShip, ArrayList<Coordinate>> ship_coordinates = new Hashtable<>();

    Hashtable<newShip, String> ship_directions = new Hashtable<>();

    Hashtable<newShip, Integer> ship_health = new Hashtable<>();

    ArrayList<newShip> existing_ships = new ArrayList<>();

    ArrayList<newShip> sunk_ships = new ArrayList<>();

    private int ships_alive = 0;

    public Map(){
        offensiveGrid = new newGrid();
        defensiveGrid = new newGrid();
    }
```

1c) Grid class creates a defensive and offensive grid for player

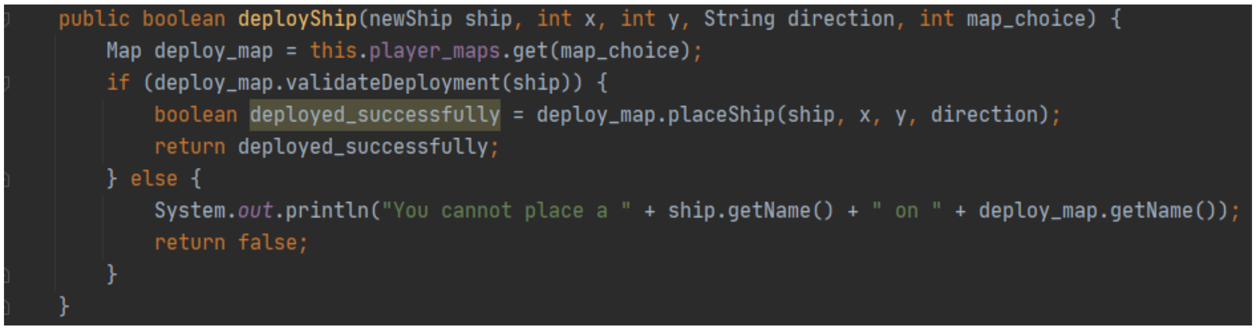


```java
public class newGrid {
    public static int length_x = 10;
    public static int length_y = 10;

    public int [][] grid = new int [length_x][length_y];

    public newGrid(){
        for (int i = 0; i < length_x; i++) {
            for (int j = 0; j < length_y; j++) {
                grid[i][j] = 0;
            }
        }
    }

    // 0: not hit
    // 1: hit, empty
    // 2: hit, occupied

    public int checkCellStatus(int x, int y) { return grid[x][y]; }

    public void setCellStatus(int condition, int x, int y) { grid[x][y] = condition; }

    public void setAllCellStatus(int condition) {
        for (int i = 0; i < length_x; i++) {
            for (int j = 0; j < length_y; j++) {
                grid[i][j] = condition;
            }
        }
    }

    public String toString() {
        String result = "";
        for (int row = 0; row < grid.length; row++) {
            for (int col = 0; col < grid[row].length; col++) {
                result += " | " + grid[col][row];
            }
            result += "\n" + "-----------------------------------------" + "\n";
        }
        return result;
    }
```

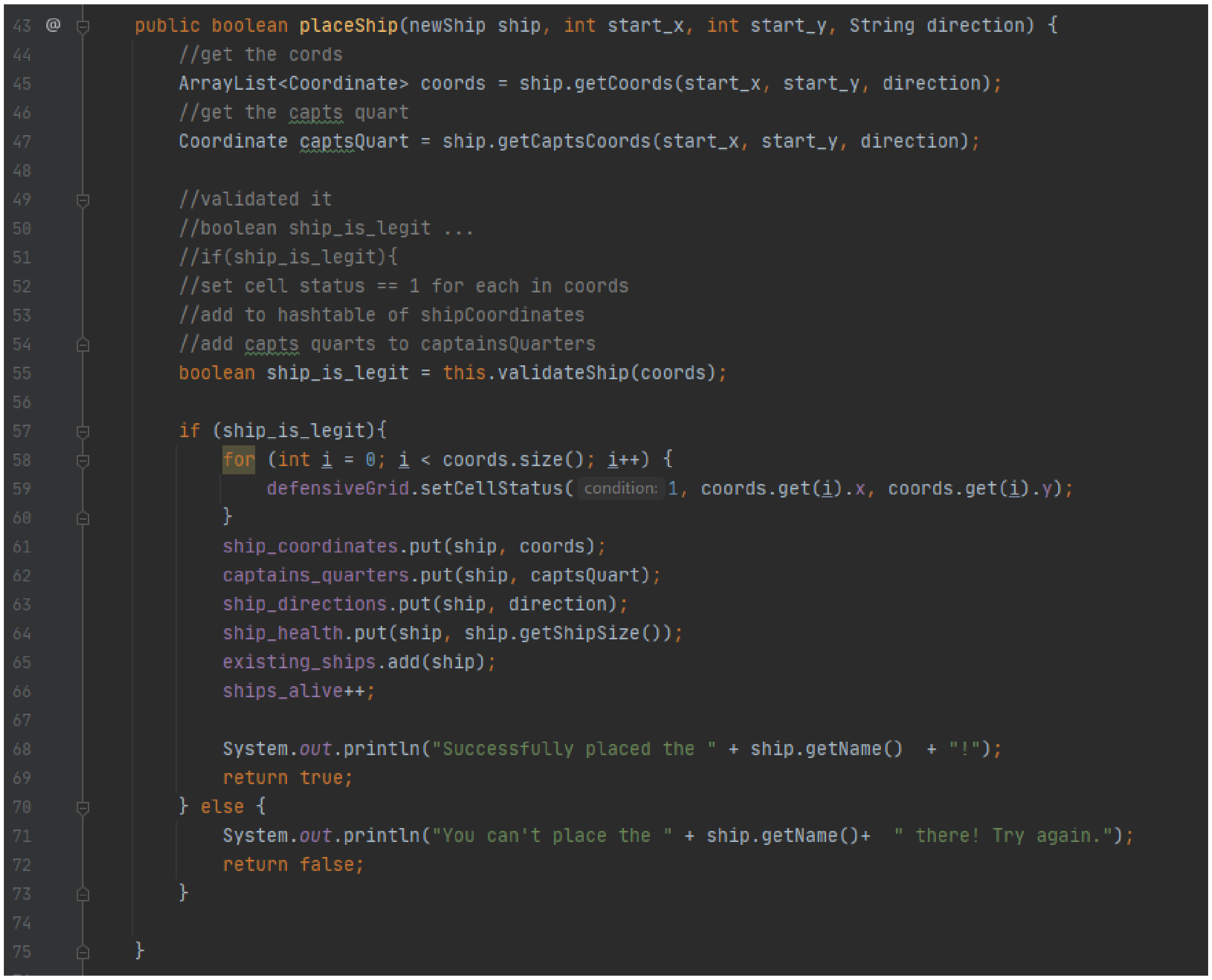
2a) Previous methodology for placing ships: including creating a fleet and then placing it on a grid.

```
 public ArrayList<Ship> createFleet() {

     ArrayList<Ship> fleet = new ArrayList<Ship>();

     Ship Minesweeper = new Ship("Minesweeper", 2);
     Ship Destroyer = new Ship("Destroyer", 3);
     Ship Battleship = new Ship("Battleship", 4);
     fleet.add(Minesweeper);
     fleet.add(Destroyer);
     fleet.add(Battleship);

     this.ship_fleet = fleet;
     this.num_boats = fleet.size();
     return fleet;
 }

public void placeShip(Ship ship, int start_x, int start_y, String direction) {
    int length = ship.getShipLength(ship);
    int end_x = 0;
    int end_y = 0;

    if ((direction.toLowerCase() == "north") || (direction.toLowerCase() == "n")) {
         end_x = start_x;
         end_y = start_y - length;
    }
    else if ((direction.toLowerCase() == "south") || (direction.toLowerCase() == "s")) {
         end_x = start_x;
         end_y = start_y + length;
    }
    else if ((direction.toLowerCase() == "east") || (direction.toLowerCase() == "e")) {
         end_x = start_x + length;
         end_y = start_y;
    }
    else if ((direction.toLowerCase() == "west") || (direction.toLowerCase() == "w")) {
         end_x = start_x - length;
         end_y = start_y;
    }


    if (validateShip(ship.getShipLength(ship), start_x,  start_y,  end_x, end_y)) {
        ship.setShipCoordinates(start_x, start_y, end_x, end_y);

        int num_cells = ship.getShipCoordinates(ship).size();
        ArrayList<Coordinate> ship_cells = ship.getShipCoordinates(ship);
        for (int i = 0; i < num_cells; i++) {
            setCellStatus(1, ship_cells.get(i).x, ship_cells.get(i).y);
        }
    }
}
```

2b) New methodology for deploying ships: takes in a ship object and places it on map.

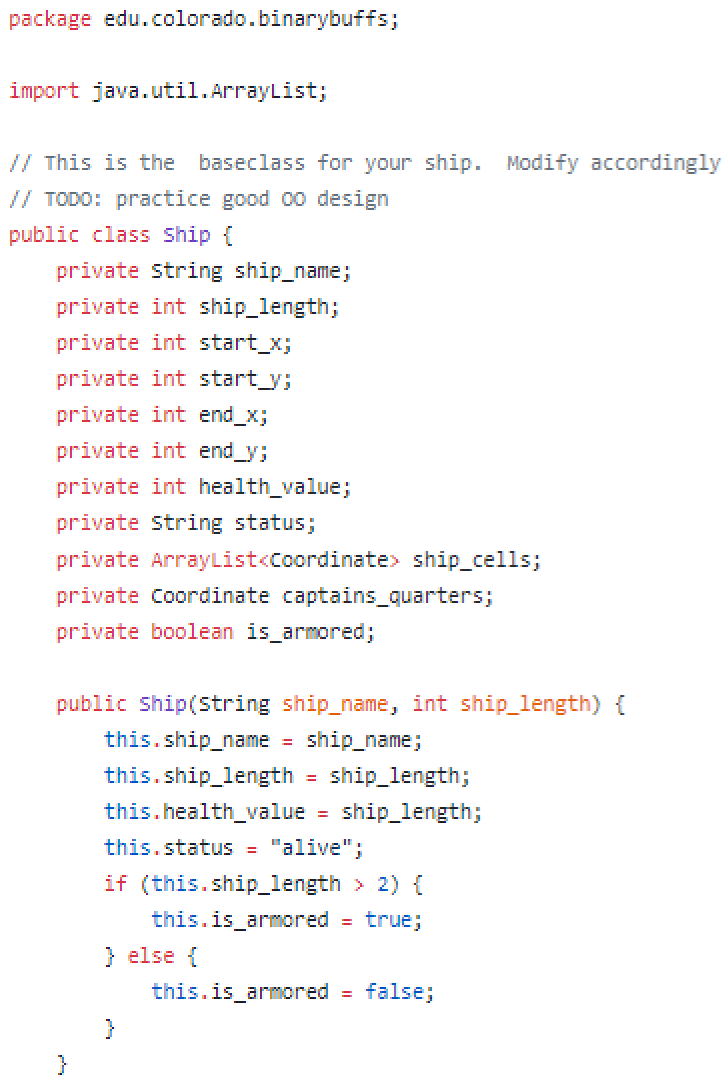


```java
public boolean deployShip(newShip ship, int x, int y, String direction, int map_choice) {
    Map deploy_map = this.player_maps.get(map_choice);
    if (deploy_map.validateDeployment(ship)) {
        boolean deployed_successfully = deploy_map.placeShip(ship, x, y, direction);
        return deployed_successfully;
    } else {
        System.out.println("You cannot place a " + ship.getName() + " on " + deploy_map.getName());
        return false;
    }
}
```



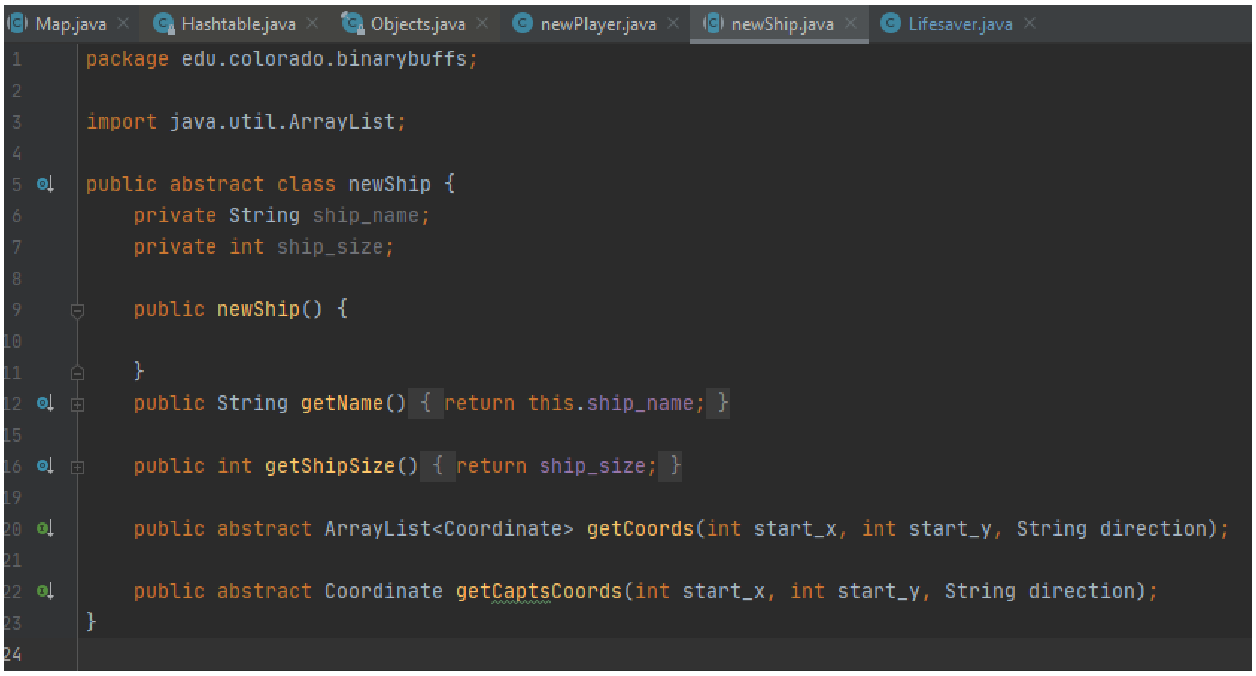
```java
public boolean placeShip(newShip ship, int start_x, int start_y, String direction) {
    //get the cords
    ArrayList<Coordinate> coords = ship.getCoords(start_x, start_y, direction);
    //get the capts quart
    Coordinate captsQuart = ship.getCaptsCoords(start_x, start_y, direction);

    //validated it
    //boolean ship_is_legit ...
    //if(ship_is_legit){
    //set cell status == 1 for each in coords
    //add to hashtable of shipCoordinates
    //add capts quarts to captainsQuarters
    boolean ship_is_legit = this.validateShip(coords);

    if (ship_is_legit){
        for (int i = 0; i < coords.size(); i++) {
            defensiveGrid.setCellStatus( condition: 1, coords.get(i).x, coords.get(i).y);
        }
        ship_coordinates.put(ship, coords);
        captains_quarters.put(ship, captsQuart);
        ship_directions.put(ship, direction);
        ship_health.put(ship, ship.getShipSize());
        existing_ships.add(ship);
        ships_alive++;

        System.out.println("Successfully placed the " + ship.getName()  + "!");
        return true;
    } else {
        System.out.println("You can't place the " + ship.getName()+  " there! Try again.");
        return false;
    }

}
```

3a) Previous method for implementing various ships for player fleet:



```
package edu.colorado.binarybuffs;

import java.util.ArrayList;

// This is the  baseclass for your ship.  Modify accordingly
// TODO: practice good OO design
public class Ship {
    private String ship_name;
    private int ship_length;
    private int start_x;
    private int start_y;
    private int end_x;
    private int end_y;
    private int health_value;
    private String status;
    private ArrayList<Coordinate> ship_cells;
    private Coordinate captains_quarters;
    private boolean is_armored;

    public Ship(String ship_name, int ship_length) {
        this.ship_name = ship_name;
        this.ship_length = ship_length;
        this.health_value = ship_length;
        this.status = "alive";
        if (this.ship_length > 2) {
            this.is_armored = true;
        } else {
            this.is_armored = false;
        }
    }
```

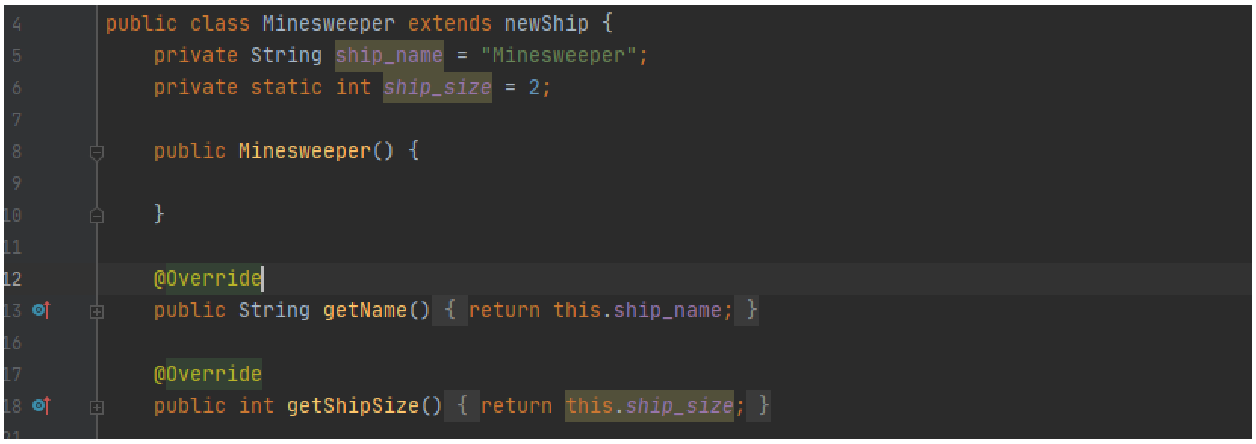
3b) New method for implementing ships - individual ship classes are extended from abstract Ship class:



Map.java × Hashtable.java × Objects.java × newPlayer.java × newShip.java × Lifesaver.java ×

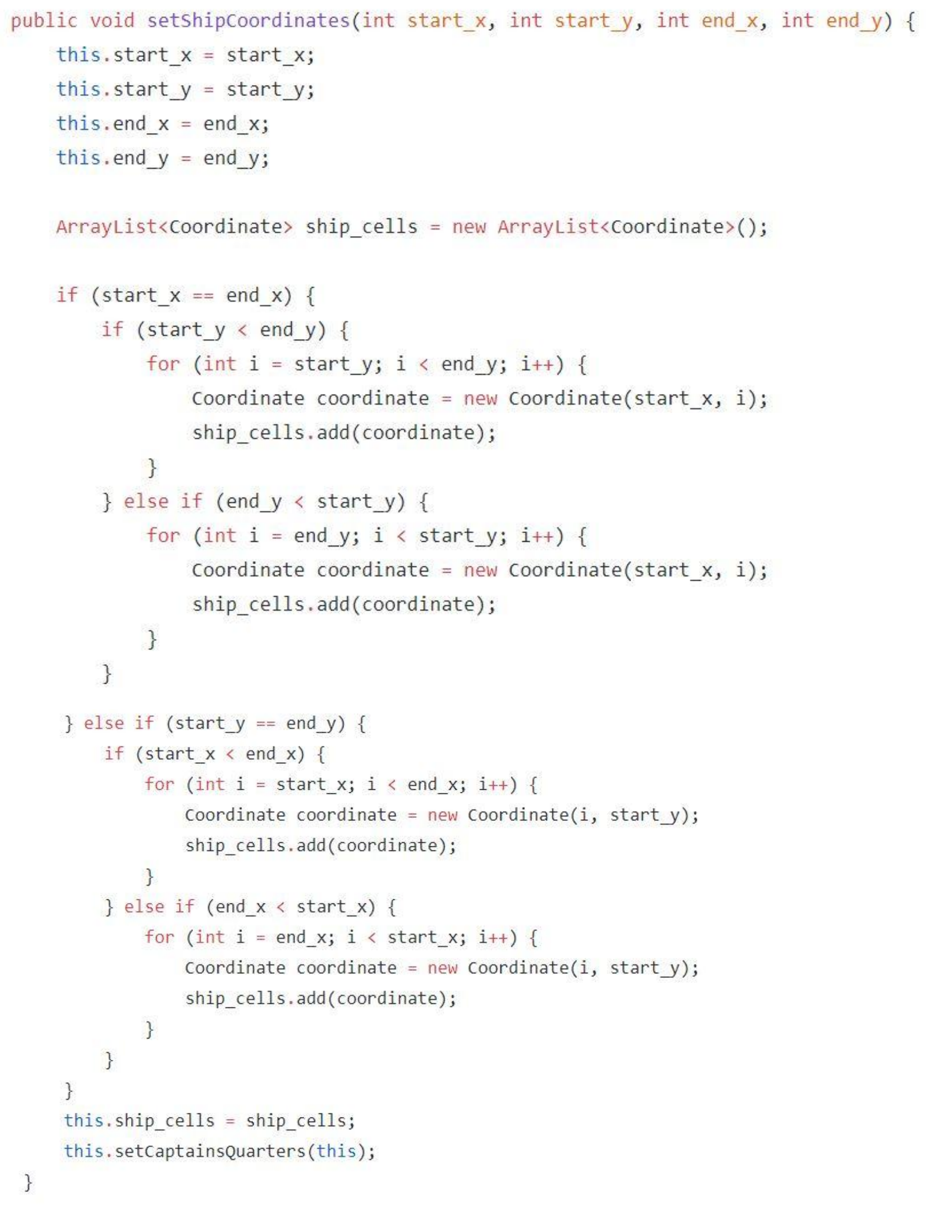
```java
package edu.colorado.binarybuffs;

import java.util.ArrayList;

public abstract class newShip {
    private String ship_name;
    private int ship_size;

    public newShip() {

    }
    public String getName() { return this.ship_name; }

    public int getShipSize() { return ship_size; }

    public abstract ArrayList<Coordinate> getCoords(int start_x, int start_y, String direction);

    public abstract Coordinate getCaptsCoords(int start_x, int start_y, String direction);
}
```

3c) Minesweeper is an individual ship class extended from Ship class



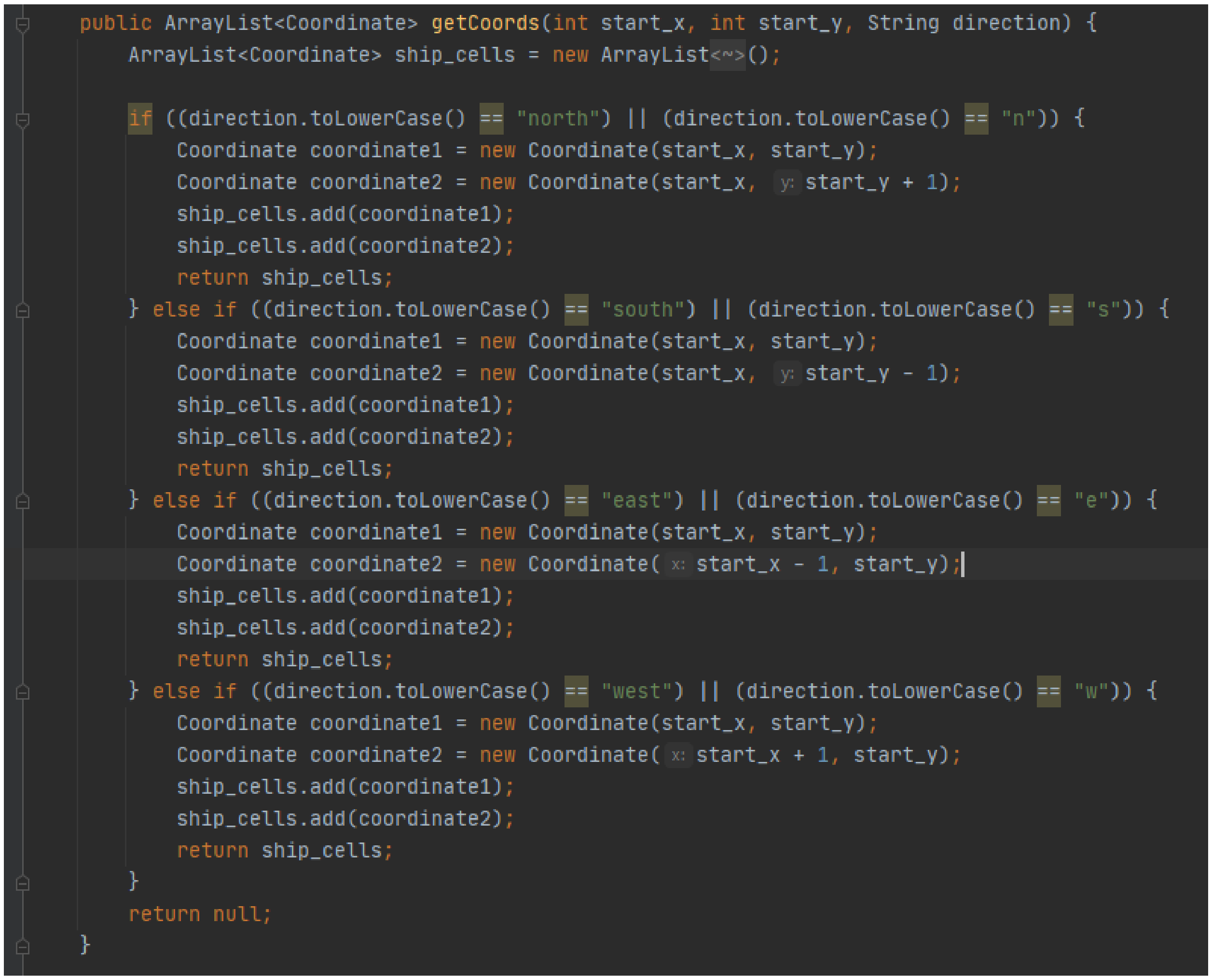
```java
public class Minesweeper extends newShip {
    private String ship_name = "Minesweeper";
    private static int ship_size = 2;

    public Minesweeper() {

    }

    @Override
    public String getName() { return this.ship_name; }

    @Override
    public int getShipSize() { return this.ship_size; }
```

4a) Previous method for setting and getting the ship coordinates:



```
public void setShipCoordinates(int start_x, int start_y, int end_x, int end_y) {
    this.start_x = start_x;
    this.start_y = start_y;
    this.end_x = end_x;
    this.end_y = end_y;

    ArrayList<Coordinate> ship_cells = new ArrayList<Coordinate>();

    if (start_x == end_x) {
        if (start_y < end_y) {
            for (int i = start_y; i < end_y; i++) {
                Coordinate coordinate = new Coordinate(start_x, i);
                ship_cells.add(coordinate);
            }
        } else if (end_y < start_y) {
            for (int i = end_y; i < start_y; i++) {
                Coordinate coordinate = new Coordinate(start_x, i);
                ship_cells.add(coordinate);
            }
        }
    } else if (start_y == end_y) {
        if (start_x < end_x) {
            for (int i = start_x; i < end_x; i++) {
                Coordinate coordinate = new Coordinate(i, start_y);
                ship_cells.add(coordinate);
            }
        } else if (end_x < start_x) {
            for (int i = end_x; i < start_x; i++) {
                Coordinate coordinate = new Coordinate(i, start_y);
                ship_cells.add(coordinate);
            }
        }
    }
    this.ship_cells = ship_cells;
    this.setCaptainsQuarters(this);
}
```

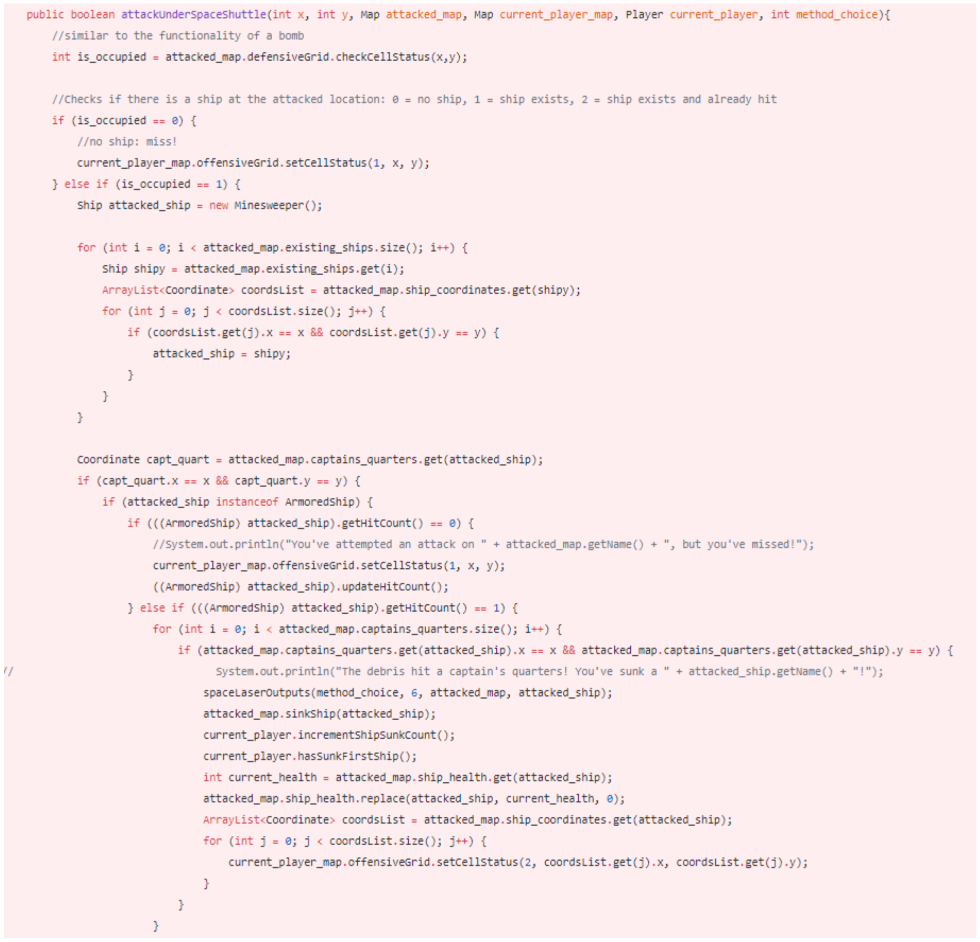
4b) New method for setting and getting ship coordinates: example from Minesweeper (simply provides the offset for each ship)



```
public ArrayList<Coordinate> getCoords(int start_x, int start_y, String direction) {
    ArrayList<Coordinate> ship_cells = new ArrayList<~>();

    if ((direction.toLowerCase() == "north") || (direction.toLowerCase() == "n")) {
        Coordinate coordinate1 = new Coordinate(start_x, start_y);
        Coordinate coordinate2 = new Coordinate(start_x,  y: start_y + 1);
        ship_cells.add(coordinate1);
        ship_cells.add(coordinate2);
        return ship_cells;
    } else if ((direction.toLowerCase() == "south") || (direction.toLowerCase() == "s")) {
        Coordinate coordinate1 = new Coordinate(start_x, start_y);
        Coordinate coordinate2 = new Coordinate(start_x,  y: start_y - 1);
        ship_cells.add(coordinate1);
        ship_cells.add(coordinate2);
        return ship_cells;
    } else if ((direction.toLowerCase() == "east") || (direction.toLowerCase() == "e")) {
        Coordinate coordinate1 = new Coordinate(start_x, start_y);
        Coordinate coordinate2 = new Coordinate( x: start_x - 1, start_y);
        ship_cells.add(coordinate1);
        ship_cells.add(coordinate2);
        return ship_cells;
    } else if ((direction.toLowerCase() == "west") || (direction.toLowerCase() == "w")) {
        Coordinate coordinate1 = new Coordinate(start_x, start_y);
        Coordinate coordinate2 = new Coordinate( x: start_x + 1, start_y);
        ship_cells.add(coordinate1);
        ship_cells.add(coordinate2);
        return ship_cells;
    }
    return null;
}
```

5a) Previous method for attacking under spaceShuttle

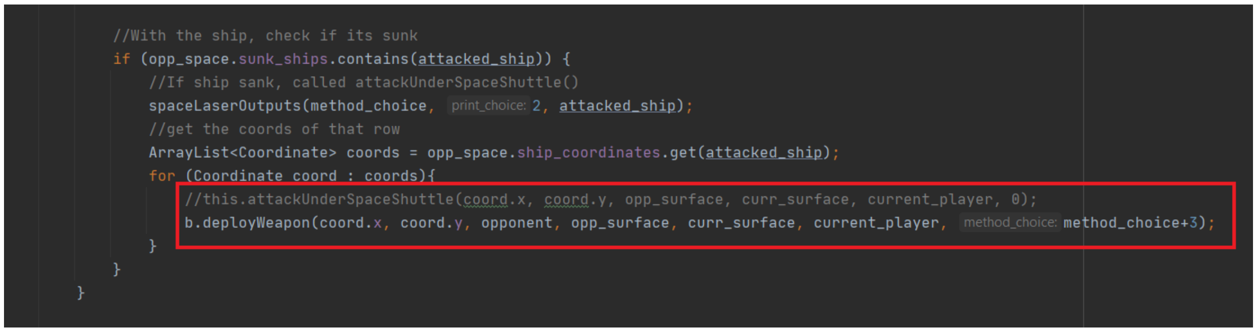
- Used same functionality as bomb deployWeapon() function, with slight print statements modification



```java
public boolean attackUnderSpaceShuttle(int x, int y, Map attacked_map, Map current_player_map, Player current_player, int method_choice){
    //similar to the functionality of a bomb
    int is_occupied = attacked_map.defensiveGrid.checkCellStatus(x,y);

    //Checks if there is a ship at the attacked location: 0 = no ship, 1 = ship exists, 2 = ship exists and already hit
    if (is_occupied == 0) {
        //no ship: miss!
        current_player_map.offensiveGrid.setCellStatus(1, x, y);
    } else if (is_occupied == 1) {
        Ship attacked_ship = new Minesweeper();

        for (int i = 0; i < attacked_map.existing_ships.size(); i++) {
            Ship shipy = attacked_map.existing_ships.get(i);
            ArrayList<Coordinate> coordsList = attacked_map.ship_coordinates.get(shipy);
            for (int j = 0; j < coordsList.size(); j++) {
                if (coordsList.get(j).x == x && coordsList.get(j).y == y) {
                    attacked_ship = shipy;
                }
            }
        }

        Coordinate capt_quart = attacked_map.captains_quarters.get(attacked_ship);
        if (capt_quart.x == x && capt_quart.y == y) {
            if (attacked_ship instanceof ArmoredShip) {
                if (((ArmoredShip) attacked_ship).getHitCount() == 0) {
                    //System.out.println("You've attempted an attack on " + attacked_map.getName() + ", but you've missed!");
                    current_player_map.offensiveGrid.setCellStatus(1, x, y);
                    ((ArmoredShip) attacked_ship).updateHitCount();
                } else if (((ArmoredShip) attacked_ship).getHitCount() == 1) {
                    for (int i = 0; i < attacked_map.captains_quarters.size(); i++) {
                        if (attacked_map.captains_quarters.get(attacked_ship).x == x && attacked_map.captains_quarters.get(attacked_ship).y == y) {
//                              System.out.println("The debris hit a captain's quarters! You've sunk a " + attacked_ship.getName() + "!");
                            spaceLaserOutputs(method_choice, 6, attacked_map, attacked_ship);
                            attacked_map.sinkShip(attacked_ship);
                            current_player.incrementShipSunkCount();
                            current_player.hasSunkFirstShip();
                            int current_health = attacked_map.ship_health.get(attacked_ship);
                            attacked_map.ship_health.replace(attacked_ship, current_health, 0);
                            ArrayList<Coordinate> coordsList = attacked_map.ship_coordinates.get(attacked_ship);
                            for (int j = 0; j < coordsList.size(); j++) {
                                current_player_map.offensiveGrid.setCellStatus(2, coordsList.get(j).x, coordsList.get(j).y);
                            }
                        }
                    }
```

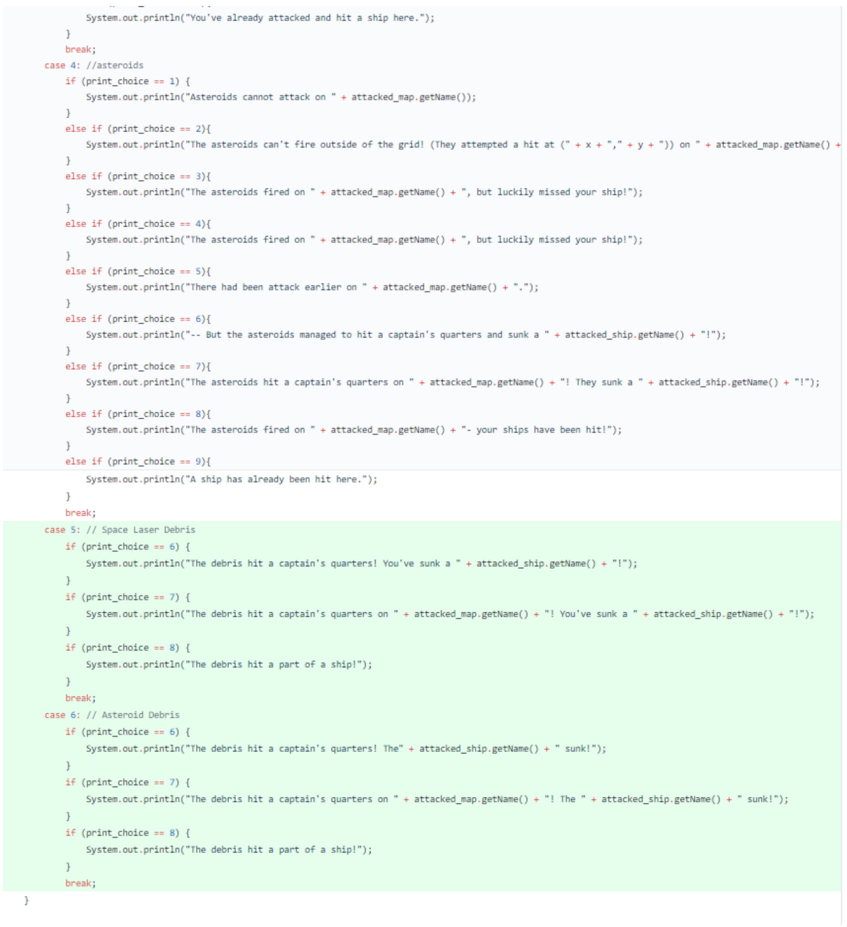
5b) New refactored code for attacking under space shuttle

- Certain print statements got moved to Bomb class, so that we could depend only on deployWeapon() function

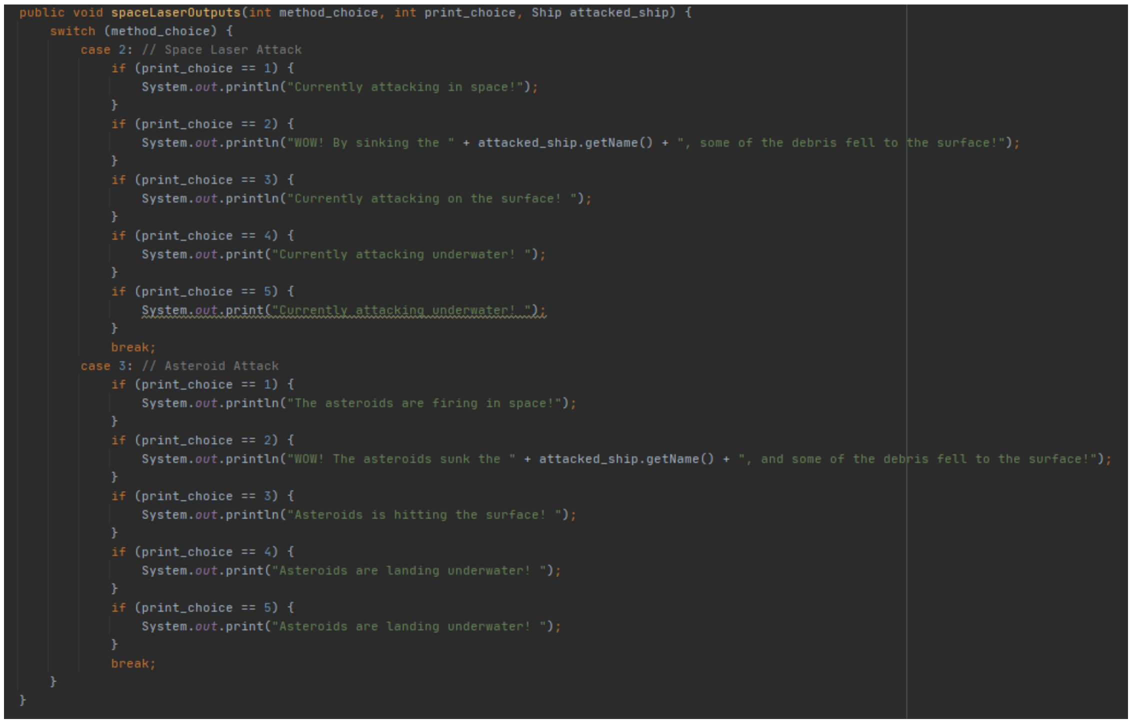


```java
    //With the ship, check if its sunk
    if (opp_space.sunk_ships.contains(attacked_ship)) {
        //If ship sank, called attackUnderSpaceShuttle()
        spaceLaserOutputs(method_choice, print_choice: 2, attacked_ship);
        //get the coords of that row
        ArrayList<Coordinate> coords = opp_space.ship_coordinates.get(attacked_ship);
        for (Coordinate coord : coords){
            //this.attackUnderSpaceShuttle(coord.x, coord.y, opp_surface, curr_surface, current_player, 0);
            b.deployWeapon(coord.x, coord.y, opponent, opp_surface, curr_surface, current_player, method_choice: method_choice+3);
        }
    }
}
```

5c) Print statements were redundant in spaceLaserOutputs() function, moved them to Bomb

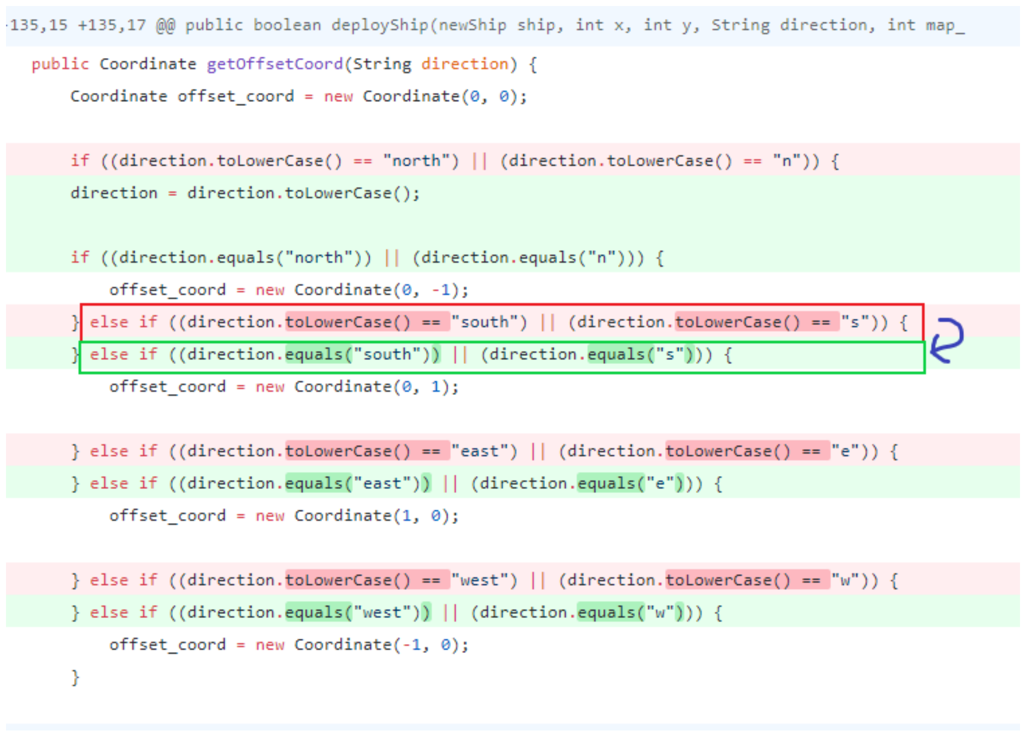


```java
                System.out.println("You've already attacked and hit a ship here.");
            }
            break;
        case 4: //asteroids
            if (print_choice == 1) {
                System.out.println("Asteroids cannot attack on " + attacked_map.getName());
            }
            else if (print_choice == 2){
                System.out.println("The asteroids can't fire outside of the grid! (They attempted a hit at (" + x + "," + y + ")) on " + attacked_map.getName() +
            }
            else if (print_choice == 3){
                System.out.println("The asteroids fired on " + attacked_map.getName() + ", but luckily missed your ship!");
            }
            else if (print_choice == 4){
                System.out.println("The asteroids fired on " + attacked_map.getName() + ", but luckily missed your ship!");
            }
            else if (print_choice == 5){
                System.out.println("There had been attack earlier on " + attacked_map.getName() + ".");
            }
            else if (print_choice == 6){
                System.out.println("-- But the asteroids managed to hit a captain's quarters and sunk a " + attacked_ship.getName() + "!");
            }
            else if (print_choice == 7){
                System.out.println("The asteroids hit a captain's quarters on " + attacked_map.getName() + "! They sunk a " + attacked_ship.getName() + "!");
            }
            else if (print_choice == 8){
                System.out.println("The asteroids fired on " + attacked_map.getName() + "- your ships have been hit!");
            }
            else if (print_choice == 9){
                System.out.println("A ship has already been hit here.");
            }
            break;
        case 5: // Space Laser Debris
            if (print_choice == 6) {
                System.out.println("The debris hit a captain's quarters! You've sunk a " + attacked_ship.getName() + "!");
            }
            if (print_choice == 7) {
                System.out.println("The debris hit a captain's quarters on " + attacked_map.getName() + "! You've sunk a " + attacked_ship.getName() + "!");
            }
            if (print_choice == 8) {
                System.out.println("The debris hit a part of a ship!");
            }
            break;
        case 6: // Asteroid Debris
            if (print_choice == 6) {
                System.out.println("The debris hit a captain's quarters! The" + attacked_ship.getName() + " sunk!");
            }
            if (print_choice == 7) {
                System.out.println("The debris hit a captain's quarters on " + attacked_map.getName() + "! The " + attacked_ship.getName() + " sunk!");
            }
            if (print_choice == 8) {
                System.out.println("The debris hit a part of a ship!");
            }
            break;
    }
```



```java
public void spaceLaserOutputs(int method_choice, int print_choice, Ship attacked_ship) {
    switch (method_choice) {
        case 2: // Space Laser Attack
            if (print_choice == 1) {
                System.out.println("Currently attacking in space!");
            }
            if (print_choice == 2) {
                System.out.println("WOW! By sinking the " + attacked_ship.getName() + ", some of the debris fell to the surface!");
            }
            if (print_choice == 3) {
                System.out.println("Currently attacking on the surface! ");
            }
            if (print_choice == 4) {
                System.out.print("Currently attacking underwater! ");
            }
            if (print_choice == 5) {
                System.out.print("Currently attacking underwater! ");
            }
            break;
        case 3: // Asteroid Attack
            if (print_choice == 1) {
                System.out.println("The asteroids are firing in space!");
            }
            if (print_choice == 2) {
                System.out.println("WOW! The asteroids sunk the " + attacked_ship.getName() + ", and some of the debris fell to the surface!");
            }
            if (print_choice == 3) {
                System.out.println("Asteroids is hitting the surface! ");
            }
            if (print_choice == 4) {
                System.out.print("Asteroids are landing underwater! ");
            }
            if (print_choice == 5) {
                System.out.print("Asteroids are landing underwater! ");
            }
            break;
    }
}
```

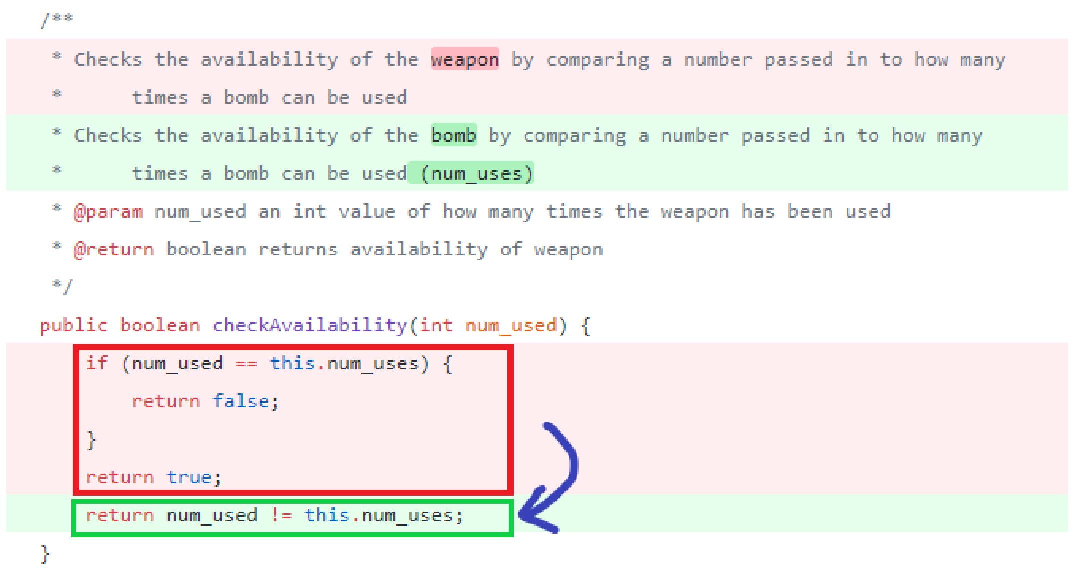
6) Old to new refactored method for comparing string user input ("==" to ".equals())



```
135,15 +135,17 @@ public boolean deployShip(newShip ship, int x, int y, String direction, int map_
public Coordinate getOffsetCoord(String direction) {
    Coordinate offset_coord = new Coordinate(0, 0);

    if ((direction.toLowerCase() == "north") || (direction.toLowerCase() == "n")) {
    direction = direction.toLowerCase();

    if ((direction.equals("north")) || (direction.equals("n"))) {
        offset_coord = new Coordinate(0, -1);
    } else if ((direction.toLowerCase() == "south") || (direction.toLowerCase() == "s")) {
    } else if ((direction.equals("south")) || (direction.equals("s"))) {
        offset_coord = new Coordinate(0, 1);

    } else if ((direction.toLowerCase() == "east") || (direction.toLowerCase() == "e")) {
    } else if ((direction.equals("east")) || (direction.equals("e"))) {
        offset_coord = new Coordinate(1, 0);

    } else if ((direction.toLowerCase() == "west") || (direction.toLowerCase() == "w")) {
    } else if ((direction.equals("west")) || (direction.equals("w"))) {
        offset_coord = new Coordinate(-1, 0);
    }
```

7) Old to new refactored method for checkingAvailability of weapon

- Shortened if/else statement into a one line return statement



```
/**
 * Checks the availability of the weapon by comparing a number passed in to how many
 *      times a bomb can be used
 * Checks the availability of the bomb by comparing a number passed in to how many
 *      times a bomb can be used (num_uses)
 * @param num_used an int value of how many times the weapon has been used
 * @return boolean returns availability of weapon
 */
public boolean checkAvailability(int num_used) {
    if (num_used == this.num_uses) {
        return false;
    }
    return true;
    return num_used != this.num_uses;
}
```

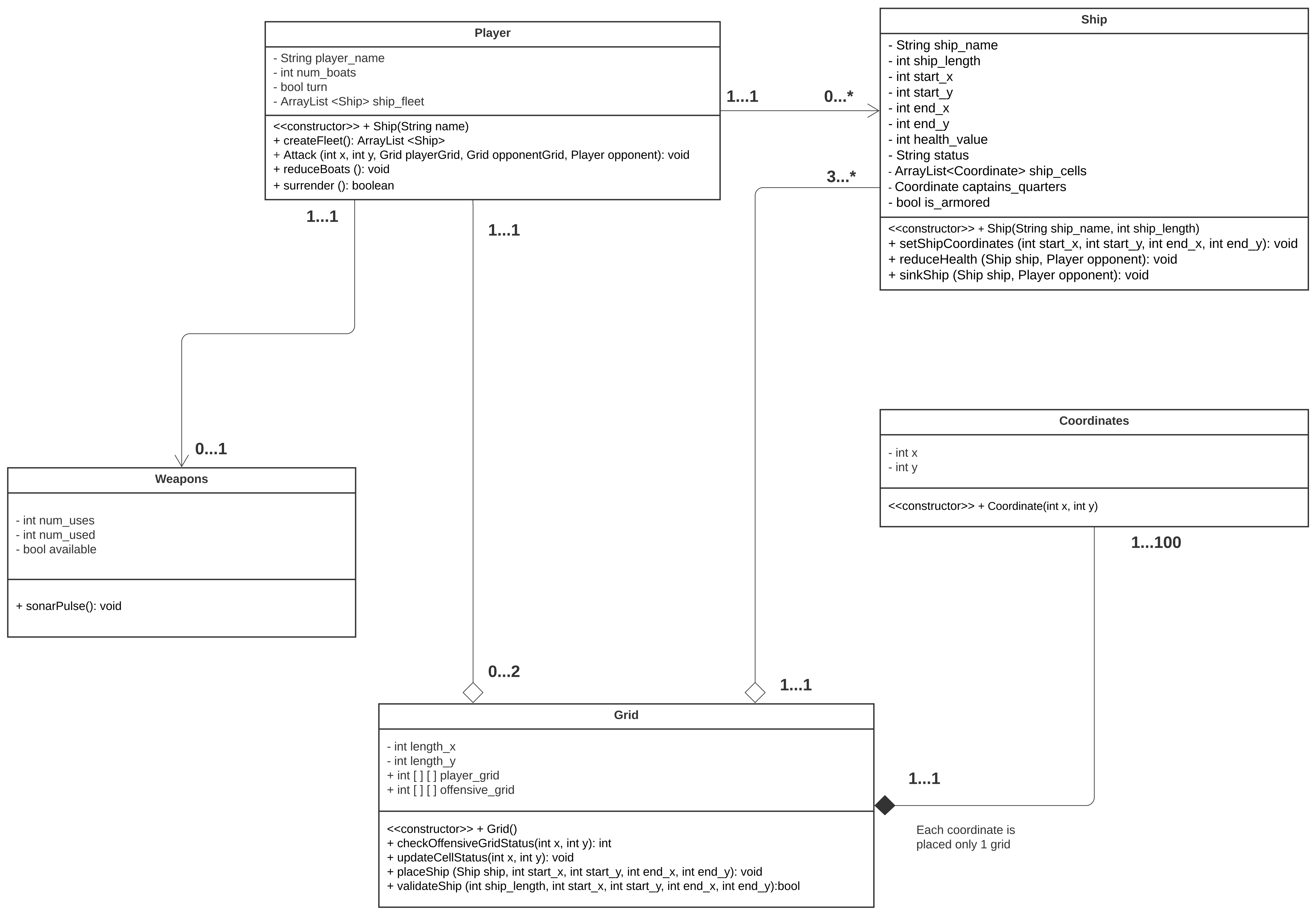
APPENDIX B

UML Diagrams (Attached below)

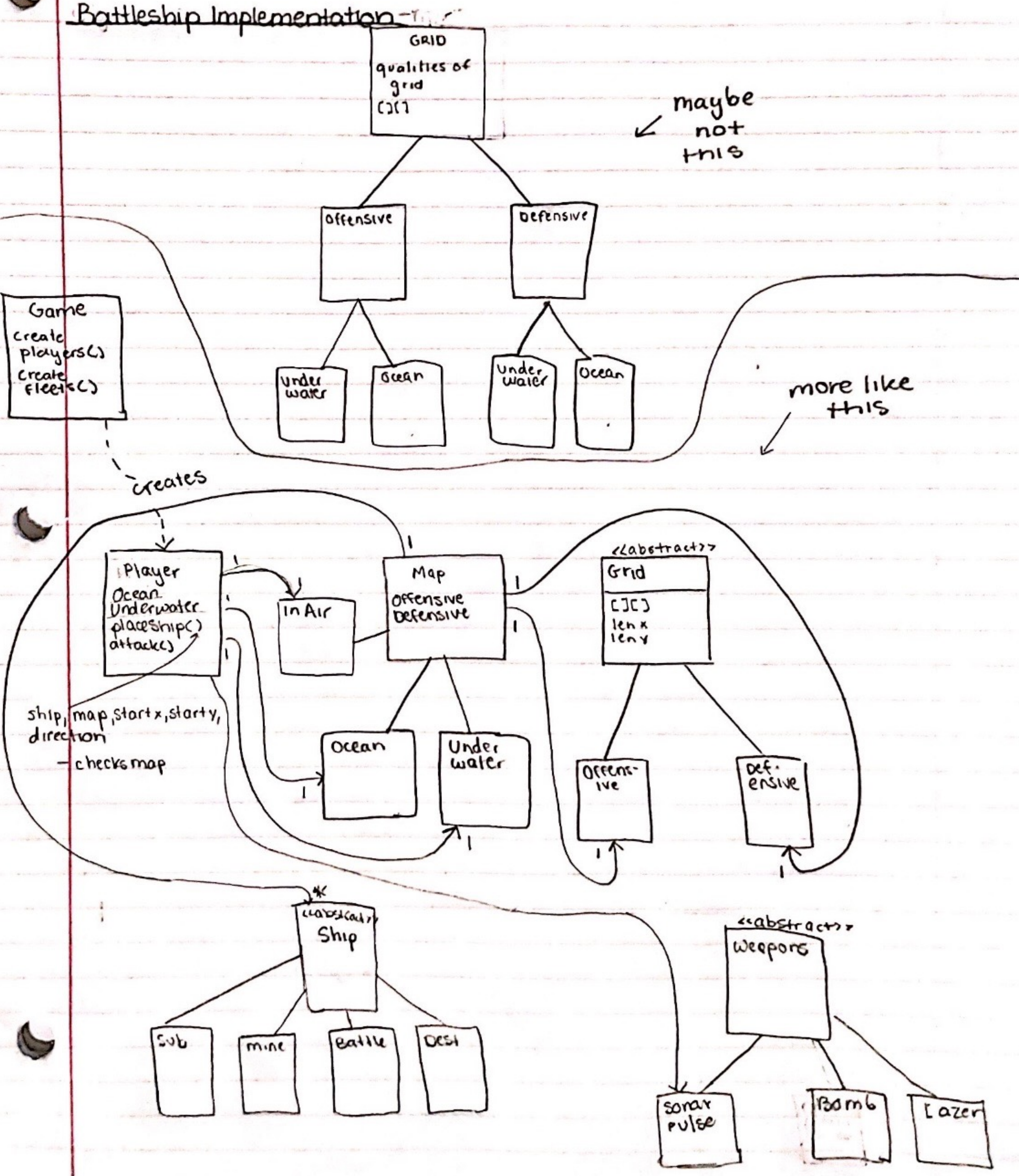
CSCI 4448 HW2

taah6009 | April 24, 2021

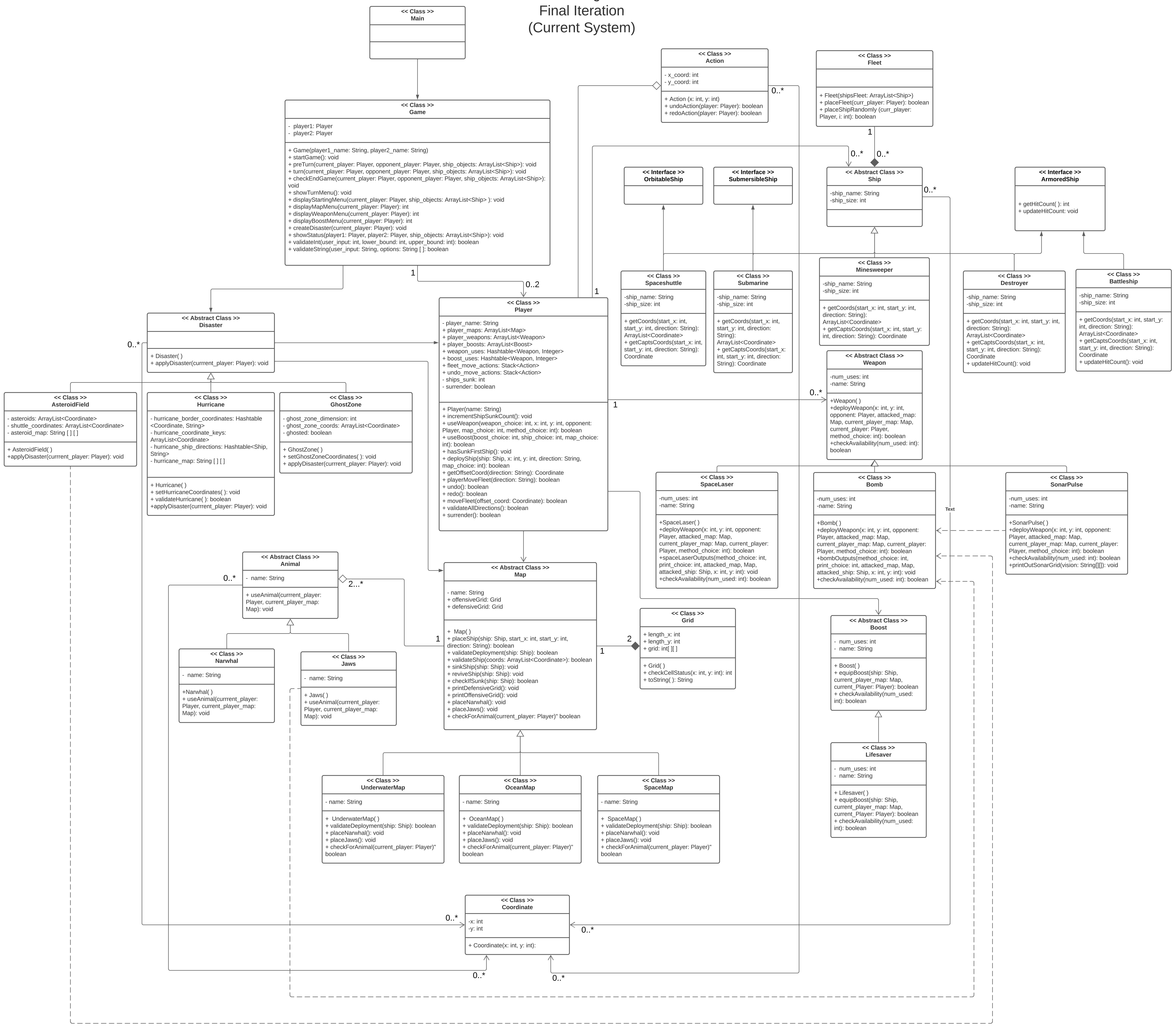
UML Diagram
Iteration 1


Player
- String player_name
- int num_boats
- bool turn
- ArrayList <Ship> ship_fleet
<<constructor>> + Ship(String name)
+ createFleet(): ArrayList <Ship>
+ Attack (int x, int y, Grid playerGrid, Grid opponentGrid, Player opponent): void
+ reduceBoats (): void
+ surrender (): boolean
Ship
- String ship_name
- int ship_length
- int start_x
- int start_y
- int end_x
- int end_y
- int health_value
- String status
- ArrayList<Coordinate> ship_cells
- Coordinate captains_quarters
- bool is_armored
<<constructor>> + Ship(String ship_name, int ship_length)
+ setShipCoordinates (int start_x, int start_y, int end_x, int end_y): void
+ reduceHealth (Ship ship, Player opponent): void
+ sinkShip (Ship ship, Player opponent): void
1...1
0...*
3...*
1...1
1...1
0...1
Weapons
- int num_uses
- int num_used
- bool available
+ sonarPulse(): void
Coordinates
- int x
- int y
<<constructor>> + Coordinate(int x, int y)
1...100
0...2
1...1
Grid
- int length_x
- int length_y
+ int [ ] [ ] player_grid
+ int [ ] [ ] offensive_grid
<<constructor>> + Grid()
+ checkOffensiveGridStatus(int x, int y): int
+ updateCellStatus(int x, int y): void
+ placeShip (Ship ship, int start_x, int start_y, int end_x, int end_y): void
+ validateShip (int ship_length, int start_x, int start_y, int end_x, int end_y):bool
1...1
Each coordinate is placed only 1 grid

UML Diagram
Iteration 2



UML Diagram
Final Iteration
(Current System)



<< Class >> Main
<< Class >> Game
- player1: Player
- player2: Player
+ Game(player1_name: String, player2_name: String)
+ startGame(): void
+ preTurn(current_player: Player, opponent_player: Player, ship_objects: ArrayList<Ship>): void
+ turn(current_player: Player, opponent_player: Player, ship_objects: ArrayList<Ship>): void
+ checkEndGame(current_player: Player, opponent_player: Player, ship_objects: ArrayList<Ship>): void
+ showTurnMenu(): void
+ displayStartingMenu(current_player: Player, ship_objects: ArrayList<Ship>): void
+ displayMapMenu(current_player: Player): int
+ displayWeaponMenu(current_player: Player): int
+ displayBoostMenu(current_player: Player): int
+ createDisaster(current_player: Player): void
+ showStatus(player1: Player, player2: Player, ship_objects: ArrayList<Ship>): void
+ validateInt(user_input: int, lower_bound: int, upper_bound: int): boolean
+ validateString(user_input: String, options: String [ ]: boolean
<< Class >> Action
- x_coord: int
- y_coord: int
+ Action (x: int, y: int)
+ undoAction(player: Player): boolean
+ redoAction(player: Player): boolean
<< Class >> Fleet
+ Fleet(shipsFleet: ArrayList<Ship>)
+ placeFleet(curr_player: Player): boolean
+ placeShipRandomly (curr_player: Player, i: int): boolean
<< Interface >> OrbitableShip
<< Interface >> SubmersibleShip
<< Abstract Class >> Ship
-ship_name: String
-ship_size: int
<< Interface >> ArmoredShip
+ getHitCount( ): int
+ updateHitCount: void
<< Class >> Spaceshuttle
-ship_name: String
-ship_size: int
+ getCoords(start_x: int, start_y: int, direction: String): ArrayList<Coordinate>
+ getCaptsCoords(start_x: int, start_y: int, direction: String): Coordinate
<< Class >> Submarine
-ship_name: String
-ship_size: int
+ getCoords(start_x: int, start_y: int, direction: String): ArrayList<Coordinate>
+ getCaptsCoords(start_x: int, start_y: int, direction: String): Coordinate
<< Class >> Minesweeper
-ship_name: String
-ship_size: int
+ getCoords(start_x: int, start_y: int, direction: String): ArrayList<Coordinate>
+ getCaptsCoords(start_x: int, start_y: int, direction: String): Coordinate
<< Class >> Destroyer
-ship_name: String
-ship_size: int
+ getCoords(start_x: int, start_y: int, direction: String): ArrayList<Coordinate>
+ getCaptsCoords(start_x: int, start_y: int, direction: String): Coordinate
+ updateHitCount(): void
<< Class >> Battleship
-ship_name: String
-ship_size: int
+ getCoords(start_x: int, start_y: int, direction: String): ArrayList<Coordinate>
+ getCaptsCoords(start_x: int, start_y: int, direction: String): Coordinate
+ updateHitCount(): void
<< Abstract Class >> Disaster
+ Disaster( )
+ applyDisaster(currrent_player: Player): void
<< Class >> Player
- player_name: String
+ player_maps: ArrayList<Map>
+ player_weapons: ArrayList<Weapon>
+ player_boosts: ArrayList<Boost>
+ weapon_uses: Hashtable<Weapon, Integer>
+ boost_uses: Hashtable<Weapon, Integer>
+ fleet_move_actions: Stack<Action>
+ undo_move_actions: Stack<Action>
- ships_sunk: int
- surrender: boolean
+ Player(name: String)
+ incrementShipSunkCount(): void
+ useWeapon(weapon_choice: int, x: int, y: int, opponent: Player, map_choice: int, method_choice: int): boolean
+ useBoost(boost_choice: int, ship_choice: int, map_choice: int): boolean
+ hasSunkFirstShip(): void
+ deployShip(ship: Ship, x: int, y: int, direction: String, map_choice: int): boolean
+ getOffsetCoord(direction: String): Coordinate
+ playerMoveFleet(direction: String): boolean
+ undo(): boolean
+ redo(): boolean
+ moveFleet(offset_coord: Coordinate): boolean
+ validateAllDirections(): boolean
+ surrender(): boolean
<< Abstract Class >> Weapon
-num_uses: int
-name: String
+Weapon( )
+deployWeapon(x: int, y: int, opponent: Player, attacked_map: Map, current_player_map: Map, current_player: Player, method_choice: int): boolean
+checkAvailability(num_used: int): boolean
<< Class >> AsteroidField
- asteroids: ArrayList<Coordinate>
- shuttle_coordinates: ArrayList<Coordinate>
- asteroid_map: String [ ] [ ]
+ AsteroidField( )
+applyDisaster(currrent_player: Player): void
<< Class >> Hurricane
- hurricane_border_coordinates: Hashtable <Coordinate, String>
- hurricane_coordinate_keys: ArrayList<Coordinate>
- hurricane_ship_directions: Hashtable<Ship, String>
- hurricane_map: String [ ] [ ]
+ Hurricane( )
+ setHurricaneCoordinates( ): void
+ validateHurricane( ): boolean
+applyDisaster(currrent_player: Player): void
<< Class >> GhostZone
- ghost_zone_dimension: int
- ghost_zone_coords: ArrayList<Coordinate>
- ghosted: boolean
+ GhostZone( )
+ setGhostZoneCoordinates( ): void
+ applyDisaster(currrent_player: Player): void
<< Class >> SpaceLaser
-num_uses: int
-name: String
+SpaceLaser( )
+deployWeapon(x: int, y: int, opponent: Player, attacked_map: Map, current_player_map: Map, current_player: Player, method_choice: int): boolean
+spaceLaserOutputs(method_choice: int, print_choice: int, attacked_map, Map, attacked_ship: Ship, x: int, y: int): void
+checkAvailability(num_used: int): boolean
<< Class >> Bomb
-num_uses: int
-name: String
+Bomb( )
+deployWeapon(x: int, y: int, opponent: Player, attacked_map: Map, current_player_map: Map, current_player: Player, method_choice: int): boolean
+bombOutputs(method_choice: int, print_choice: int, attacked_map: Map, attacked_ship: Ship, x: int, y: int): void
+checkAvailability(num_used: int): boolean
<< Class >> SonarPulse
-num_uses: int
-name: String
+SonarPulse( )
+deployWeapon(x: int, y: int, opponent: Player, attacked_map: Map, current_player_map: Map, current_player: Player, method_choice: int): boolean
+checkAvailability(num_used: int): boolean
+printOutSonarGrid(vision: String[][]): void
Text
<< Abstract Class >> Animal
- name: String
+ useAnimal(currrent_player: Player, current_player_map: Map): void
<< Abstract Class >> Map
- name: String
+ offensiveGrid: Grid
+ defensiveGrid: Grid
+ Map( )
+ placeShip(ship: Ship, start_x: int, start_y: int, direction: String): boolean
+ validateDeployment(ship: Ship): boolean
+ validateShip(coords: ArrayList<Coordinate>): boolean
+ sinkShip(ship: Ship): void
+ reviveShip(ship: Ship): boolean
+ checkIfSunk(ship: Ship): boolean
+ printDefensiveGrid(): void
+ printOffensiveGrid(): void
+ placeNarwhal(): void
+ placeJaws(): void
+ checkForAnimal(current_player: Player)" boolean
<< Class >> Grid
+ length_x: int
+ length_y: int
+ grid: int[ ][ ]
+ Grid( )
+ checkCellStatus(x: int, y: int): int
+ toString( ): String
<< Abstract Class >> Boost
- num_uses: int
- name: String
+ Boost( )
+ equipBoost(ship: Ship, current_player_map: Map, current_Player: Player): boolean
+ checkAvailability(num_used: int): boolean
<< Class >> Narwhal
- name: String
+Narwhal( )
+ useAnimal(currrent_player: Player, current_player_map: Map): void
<< Class >> Jaws
- name: String
+ Jaws( )
+ useAnimal(currrent_player: Player, current_player_map: Map): void
<< Class >> Lifesaver
- num_uses: int
- name: String
+ Lifesaver( )
+ equipBoost(ship: Ship, current_player_map: Map, current_Player: Player): boolean
+ checkAvailability(num_used: int): boolean
<< Class >> UnderwaterMap
- name: String
+ UnderwaterMap( )
+ validateDeployment(ship: Ship): boolean
+ placeNarwhal(): void
+ placeJaws(): void
+ checkForAnimal(current_player: Player)" boolean
<< Class >> OceanMap
- name: String
+ OceanMap( )
+ validateDeployment(ship: Ship): boolean
+ placeNarwhal(): void
+ placeJaws(): void
+ checkForAnimal(current_player: Player)" boolean
<< Class >> SpaceMap
- name: String
+ SpaceMap( )
+ validateDeployment(ship: Ship): boolean
+ placeNarwhal(): void
+ placeJaws(): void
+ checkForAnimal(current_player: Player)" boolean
<< Class >> Coordinate
-x: int
-y: int
+ Coordinate(x: int, y: int):
0..*
1
0..2
0..*
0..*
1
0..*
0..*
1
0..*
0..*
2...*
1
2
1
0..*
0..*
0..*
0..*