MILESTONE 1

PROJECT DOCUMENTATION

Dream Team

Table of Contents

Brief Description of Project

The aim of this project was to create a Battleship game. Each player has a 10x10 grid where their fleet resides. Each fleet is made up of ships of varying lengths, specifically a Battleship of length 4, a Minesweeper of length 2, a Destroyer of length 3, and a Submarine of length 4, which can be either submerged or above water. Ships can be placed either horizontally or vertically. Each player can see their own grid and where the opponent has hit or missed. The other grid will contain what's known so far about the other player's board in terms of hits and misses.

The players take turn inputting coordinates to attack the other player's ships, with weapons called space lasers, sonar pulse, and mines. Mines will attack the player's board at the specified coordinate, and the space laser will attack both a submerged and above water ship. Sonar pulse allows a player to reveal a certain portion of the map. Both the space laser and sonar pulse require a ship to be sunk before that weapon can be utilized.

An attack results in a hit or a miss, and when the captain's quarters are hit twice (in all cases except Minesweeper) or all spaces of a ship are hit, the ship will sink. The game will continue until all of a player's ships have sunk. The captain's quarters are a square on the Submarine, Destroyer, and Battleship, and if it is hit twice, the entire ship will sink. We have also refactored various parts of the project using the methods learned in class. This project also contains the implementation of design patterns, and contains classes that are open to extension and closed to modification as shown in class.

Developmental Process

Iterative Development:
Iterative development allowed us to break down the development and testing of a large project into smaller pieces. It allowed us to design, test, and code smaller subsets of the main project as we worked on the Battleship game feature by feature. This allows for flexibility for changes. For example, when requirements changed between Milestones, iterative development allowed us to be flexible and we were able to reevaluate and design for the altered requirements. Working on smaller parts of the bigger system can also help us find issues with our code before it's too embedded into the structure. We can also test each part of the system individually before we test the entire thing, which cuts down on the amount of time and energy spent to pinpoint errors.

Refactoring:
Refactoring was helpful in that throughout multiple iterations, we were able to clean up existing code in past iterations. This allowed us to first focus on writing code that would pass tests, then revisit that code in later iterations to clean up and rewrite the code to make it more readable or better designed. In this way, refactoring made it easier to deal with changing requirements and such. If there was code that needed to be integrated into the new features and new requirements, we were able to refactor the code such that it fit better with the new requirements without changing the code itself. It was also a great tool to create readable code. We were able to add new features each iteration and then also revise old code to make it look more readable and better for our sanity to use.

Testing:
For this project, we utilized Test Driven Development. This means we started with a specification, created tests for that specification, and wrote the code for that specification. This allowed us to have an idea of what we're expecting as well as the design of the specification before we even started coding. This also helped us clarify what we wanted to test and how it was relevant to the specification and reduced ambiguity or confusion. We learned, through code coverage, how to create tests that would run quickly but validated all of the code we'd written and would reach every statement in the program through at least one test case. Writing these tests helped us be responsive to change, safe from bugs, and helped us write code that was easy to understand. We

used JUnit for TDD, and utilized assertions to write tests that were self contained.

Collaborative Development:
In this project, we utilized the concept of collaborative development through pair programming and team meetings. Through pair programming, we were able to design and discuss code before we implemented it, as well as catch one another's errors. This allowed for increased efficiency, as two people were checking the implementation plan and reading over the code. It also helped us develop our communication and collaboration skills as we worked together to solve problems. We also practiced collaborative development in our team meetings, since we all worked together to write up our user stories and planning, and oftentimes we would discuss the design of certain features as we were thinking of them.

What We Wished We'd Known When We Started the Project:

Initially, we designed the coordinate system as (x,y) coordinates which would make it intuitive for the user to play our Battleship game. However, when we were utilizing these coordinates in the Map class, we ended up translating the x/y coordinates straight into the row/column coordinates when creating our battleship board -- we didn't realize that the X coordinate actually corresponded to the column coordinate (and the Y coordinate corresponded to the row coordinate). The reason we didn't catch this bug and the reason this still passed all of our attack tests was because we accidentally flipped our coordinates once again when retrieving the locations from the Map class itself. This bug showed us that our tests need to be even more granular and more specific -- testing every single aspect of the classes before writing the code for the corresponding section.

Requirements and Specifications

- As a player, I want to be able to submerge a ship even after it's been placed

- As a player, I want to be able to place a Trojan Horse (Blucifier) ship such that another player will lose points if they hit it

- As a developer, I want to add a point tracking system

- As a developer, I want to refactor the Map class (with samples of before and after code refactoring)

- As a developer, I want to identify code smells as targets for refactoring

- As a developer, I want to communicate effectively while pair programming with my partner to ensure that we are working together to code well.

- As a developer I want to make sure that I am properly using Test Driven Development to satisfy the requirements of this project.

- As a developer, I want to use the refactoring strategies we learned about in class to refactor the Map class and some of the methods in the class.

- As a developer, I want to refactor the Submarine class to implement some of the more submarine specific features instead of making that a Ship wide change
- As a developer, I want to make sure that my classes are highly cohesive and loosely coupled

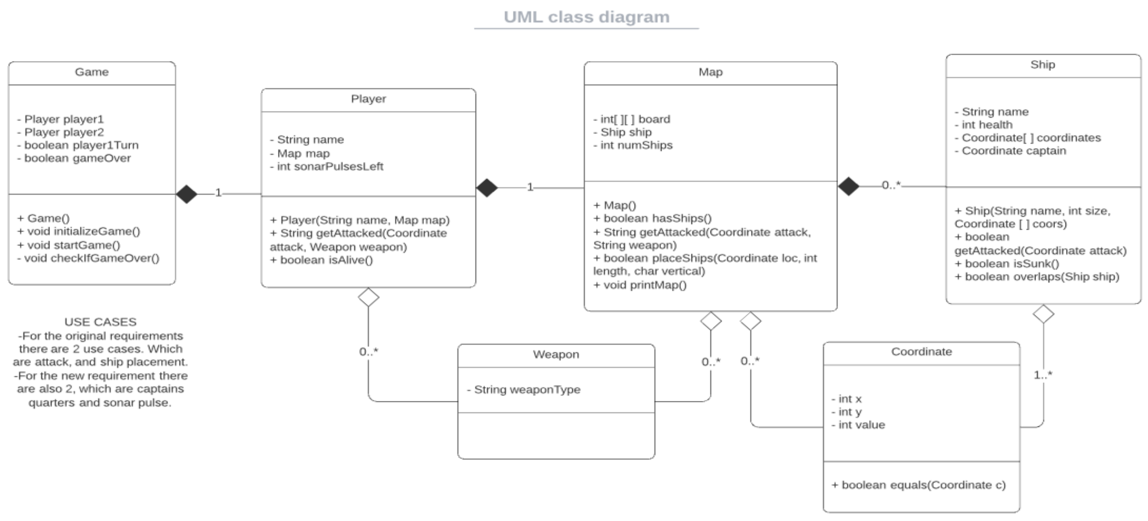
- As a player I should be able to place a submarine.

- As a player, I should be able to fire a space laser that can hit a surface ship and a submarine.

- As a player, I should only be able to use a space laser if I’ve already sunken an enemy ship.
- As a developer, I should implement several design patterns.
- As a player I should be able to use 2 sonar pulses and no more during a game
- As a player I should be able to see the results of the sonar pulses that I use so that I can plan out future attacks clearly.
- As a player I want to be able to choose between weapons when attacking the other player’s map.
- As a developer, I want to ensure that the captain’s quarters are implemented according to the requirements where it takes 2 hits (except for minesweepers) to sink a ship

Architecture and Design

UML Class Diagram from HW#2


UML class diagram
Game
- Player player1
- Player player2
- boolean player1Turn
- boolean gameOver
+ Game()
+ void initializeGame()
+ void startGame()
- void checkIfGameOver()
Player
- String name
- Map map
- int sonarPulsesLeft
+ Player(String name, Map map)
+ String getAttacked(Coordinate attack, Weapon weapon)
+ boolean isAlive()
Map
- int[ ][ ] board
- Ship ship
- int numShips
+ Map()
+ boolean hasShips()
+ String getAttacked(Coordinate attack, String weapon)
+ boolean placeShips(Coordinate loc, int length, char vertical)
+ void printMap()
Ship
- String name
- int health
- Coordinate[ ] coordinates
- Coordinate captain
+ Ship(String name, int size, Coordinate [ ] coors)
+ boolean getAttacked(Coordinate attack)
+ boolean isSunk()
+ boolean overlaps(Ship ship)
Weapon
- String weaponType
Coordinate
- int x
- int y
- int value
+ boolean equals(Coordinate c)
1
1
0..*
0..*
0..*
0..*
1..*
USE CASES
-For the original requirements there are 2 use cases. Which are attack, and ship placement.
-For the new requirement there are also 2, which are captains quarters and sonar pulse.

There is a current UML Class diagram inside the wiki folder on the project github page.

In comparing our first UML Class Diagram to the current UML Class Diagram, our first diagram didn't have the UI class that interacted with the users and also had a lot more functions in the map class making it loosely cohesive. On the new and updated UML class diagram, you can see that our code is tightly cohesive as we have a class just for one specific functionality and the classes are loosely coupled. You can also see that the new diagram has many more classes and methods because of all the features we've added.

Description of Each Class

The Map Class has a board that holds information on the number of the ships, the coordinates of the ships, and the display of the Map. This class collaborates with almost all of the other classes. It also contains the functionality to move ships in a fleet, as well as the ability to undo and redo moves.

The Ship Class holds information about each ship on the map. We have the name and type of the ship, which parts of the ships are attacked, if it is submerged or not, the size of the ship, and a lot more details about the ship. It has all the information you need to know about a specific ship on the map. It also has child classes for each ship.

The Coordinate Class holds information about what is there on each specific coordinate on the Map board. We can check if there is a ship there or not, if that coordinate has been attacked or not, and other information needed about each Coordinate.

The Player Class holds the Name and a specific Map for the Player.

The Game Class keeps track of which player's turn it is.

The UI class displays all the information a player needs to play the game; it also collects and passes all the information needed for the game to keep going.

The Weapon Class holds all the information about the weapon type.

The Driver Class starts the game.

A Quick Rundown on How These Classes Collaborate Together to Get the Game Going

The first class we run is the Driver class that calls the Game class. The Game class will then set up each Player's information, and it does that by calling the UI.Initplayer function which interacts with the user to get the name of the player and specific positions to set up the ships on the board. It sets the name by using the Players Class then sets the ships on the board by collaborating with the setships function in the Map class. The Map class then interacts with the setships function in ship class to set up information about each ship. The Map class also lets you set up each coordinate to a specific state. When setting up it will update the coordinates where the ships are at to coordinate status "SHIP".

Once the Players have set their names and ships on the map, the Game class will then go in a loop to start the turns by letting each player take turns attacking each other. It calls the UI.getattack function in the UI class to get the information needed from the user to set an attack on the others board. Once it gets the coordinates and weapon type from the user, it passes it to the getattacked function in the Map class. The map class then updates the board, the status on each coordinate and also updates the ship information in the ship class.

The loop in the Game class will keep going until one of the players dies. A player is considered dead when the health of all their ships are down to zero. We keep track of the health of the ships in the Ship class and it goes down at the time an attack was successful.

Personal Reflections

Beka-
There were a lot of things that went well in this project, but what I liked the most was the test driven development. The test driven development made this project go smoothly and saved us a lot of time. Also github, with intelliJ was simple to use. It was well organized and merging our codes was easy. Another thing that went well was pair programming, I really felt the saying that "Two heads are better than one". Doing this project with a partner made it so much better. We were able to problem solve way faster than we would by ourselves.
Something that did not go so well, was that there were things that we didn't fully understand, such as Refactoring and Design Patterns. We had a hard time trying to figure out how to do them.
I'm still a little puzzled by the different Design patterns and when exactly to use them.
I learned that getting things done as early as possible makes the process more enjoyable and less stressful. Also communication is very important, discussing changes and commenting on codes is really helpful to make the project go by smoothly.

Noorain-

I learned a lot throughout this project about software development, good programming practices, and collaborating remotely with my team. I learned about the benefits of test driven development and gained more experience with Java. This was my first time pair programming extensively in a project, and I think that I'm going to implement that more in future projects. It helped to be able to talk through code with another person and quickly find errors in logic. I also think we were all on the same page in terms of being motivated to do well, and that led to great collaboration.

Initially, I was intimidated by the scope of this project and struggled with how to proceed given intentionally unclear requirements, but after designing features and pair programming with the team, I was able to get more confidence in being able to design for as well as meet the requirements.

I think next time I would add a specific section in our team Wiki for mapping out the design of specific features and I would regroup quickly with the team after the TA meetings to go over the meeting notes to make sure we were all on the same page. I learned a lot more about the benefits of pair programming, test-driven development, and design patterns through working on this project. I have a better understanding of how to design and implement well structured and extensible code.

Andrew-

I think our group worked really well together and I think the pair programming was especially useful. Coding for tests also helped steer the code writing process. I think group work and similarly long, iterative projects are good. I think the beginning was very rough because no one knew what they were doing or what the goals of the project were. We did not know that this project would be so long nor did we know any good code writing practices so what happened was that we would write so much bad code and then later learn how to write good code that it was a painful process to refactor our existing code. I am still puzzled by how certain design patterns could be fit into this project. I didn't have the chance to put into practice everything I've learned in the class so there are still some points that I am confused on. The main lesson I learned was how to work in a group for long iterative projects and to always write code such that changing it will be simple. The code I had written in the past was always too static and never changeable but now I know how horrible it is to attempt to change that code.

Sai-

The best part of the group development was the pair programming. The pair programming was effective because when one brain hits a roadblock, the partner can jump right in and offer input. I have never done pair programming so I was skeptical at first. I was worried that this would slow down the development and make it so that it would take twice as long to write the same amount of code. The best part that I did not anticipate was that the troubleshooting time was cut down in half or even more. Pair programming made it extremely easy to figure out where the bugs were, very quickly as well. I was also new to Test Driven Development, and that also greatly improved our code quality, speed of development, and allowed us to have a lot of the features implemented one at a time which allowed us to catch bugs very quickly.

In the future, one thing we could do better is estimating the time that each task was going to take us. We often underestimated the effort required to complete a task -- this resulted in a hectic end of the week rush to finish the milestone on time. Another aspect that we could have done better was splitting up the different parts of each new feature that we wanted to

implement. This caused the implementation of the new feature to go in very different directions than what was initially planned -- this was not necessarily a bad thing, but a little more documentation would have been great.
Some parts of the code could still use some refactoring. This is one of my biggest puzzles because this is my first time working with a major refactoring problem and I still need some more practice with it.
My favorite part of the project was the takeaways I got from the value of TDD and pair programming. Following good coding practices makes it so easy to create a finished product with quality code. The practices that we learned in this class helped.