Battleship

Project Documentation

CSCI 4448: Object Oriented Analysis and Design

Team ApplePear: Kevina Wong, Vienna Wong, Yubin Go, Yvonne Liu

Repository Link Here

Description of Project **4**

Development Process **4**

Development 4

Refactoring 4

Example i. Point Class Refactoring: Encapsulation Field 5

Example ii. Ship Class Refactoring: Extract Interface 5

Example iii. Ship Class Refactoring: Replace Conditional w/ Polymorphism 6

Example iv. GameBoard Class and Ship Class: Move Method 6

Example v. Move Class: Replace Subclass with Fields 7

Example vi. Move Class and Command: Remove middleman 7

Testing 7

Collaborative Development 8

Requirements and Specifications **8**

Requirements 8

Functional Requirements 8

Non-Functional Requirements 8

Specifications 9

Coding Standards 9

Naming Conventions 9

Comments 9

Indentation & Spacing 9

Architecture and Design **10**

UML Diagrams 10

Description of System 11

Game Class 11

GameBoard Class 11

Ship Interface & Subclasses 11

Player Class 11

Personal Reflections **12**

Kevina Wong 12

Vienna Wong 12

Yvonne Liu 12

Yubin Go 12

I. Description of Project

This project is an object-oriented Java implementation of the commonly-known board game, Battleship. The two-player game begins by allowing both players to place both their ships at specific locations. The main objective of the game is guess which coordinates to attack and sink all of your opponent's ships before they sink yours. Each ship may vary in size or shape, and each player has the same numbers and types of ships.

This project is elevated from the traditional board game because we have newly added ships, like the "Tower Ship" and the "L-Ship". We also have newly added weapons, like the "Sonar Pulse" and the "+ Bomb". Furthermore, a GUI component is added to the game to elevate the experience for the player rather than it being a text-based game. Although, the text-base option is fully implemented as well.

II. Development Process

a. Development

We primarily use test-driven development (TDD), meaning that we write unit tests for each case prior to writing code for the function. An initial issue we faced was that our team was unfamiliar with TDD, and had some issues with knowing how to write good unit tests. We also struggled with writing tests prior to writing code.

Along the semester, we have fully implemented the classes and the entirety of the text-based version of the game. For the final portion, we are currently working on implementing a graphical user interface (GUI). Due to a current lack of time, we are striving to complete at least the base Battleship game for the GUI. That is - the Battleship game where the players are able to place a ship, attack a location, sink a ship, and win/lose the game. The text-based implementation has many additional features, such as specialty weapons and moving ships. However, due to the current time constraint, that is not in our criteria but might be in the future as we develop the GUI.

b. Refactoring

As we learned about more refactoring techniques throughout the semester, we frequently refactored our code along the way where it was applicable. We ensured that we were constantly refactoring to uphold high cohesion and loose coupling and that our code reflects our strong understanding of design principles in object-oriented programming. We refactored to make sure that we were avoiding bad code smells such as cloning. The issue that we had with refactoring is that we really didn't learn many refactoring techniques until later in the semester. Thus, early on, we were writing tightly coupled

code, code with a lot of cloning, and overall poorly written object-oriented code. Specific examples of refactoring within our code can be found below. Note that the refactoring that we have done in our code isn't limited to these few major examples.

```java
package edu.colorado.applepear.classes;

//Created a Point class for getting location of ship in Sh
public class Point {
    public int x;
    public int y;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }

    public int getY() { return y; }
}
```

Not refactored code

```java
package edu.colorado.applepear.classes;

/**
 * Point Class used to hold coordinate style objects
 */

public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }

    public int getY() { return y; }
}
```

Refactored code

Example i. Point Class Refactoring: Encapsulation Field
The code here shows that the un-refactored code previously had public attributes. After refactoring, the attributes have been changed to private.

```java
lic class Ship {
public ArrayList<Point> location;
private String shipName;
private GameBoard gb;

//Constructor

public Ship(){
    location = new ArrayList<>();
    this.shipName = shipName;
    this.gb = gb;
}

public void setShipName() {
    Scanner myInput = new Scanner(System.in);

    if (location.size() < 1) {
        shipName = "";
        System.out.println("Enter coordinates First.");
    }
    else if (location.size() == 2){
        shipName = "minesweeper";
    }
    else if (location.size() == 3){
        if ( (location.get(0).y) == ((location.get(1).y)) && (location.get(0).y) ==  (location.get(2).y)){
            shipName = "destroyer";
        }
        else if ( (location.get(0).x) == ((location.get(1).x)) && (location.get(0).x) ==  (location.get(2).x)){
            shipName = "destroyer";
        }
        else if (location.get(0).equals(location.get(1).equals(location.get(2)))){
            shipName = "tower";
        }
        else {
            shipName = "L";
        }
    }
```

Not refactored code

```java
package edu.colorado.applepear.classes;

import java.util.List;

// Kevina - Rearranged ship class and relationships, made ship int
public interface Ship {

    String getShipName();
    int getShipHealth();
    List<Point> getLocation();
    boolean getUnderwater();
    CaptainsQuarters getCaptainsQuarters();

    void setShipName();
    void setShipHealth(int newHealth);
    void setCaptainsQuarters();
    void setLocation(List <Point> points);

    Boolean isShipSunken();
    void updateHealth(Point location);
    List<Point> input(int[][] shipMap);
}
```

Refactored code

Example ii. Ship Class Refactoring: Extract Interface
The code here shows that previously, we used a large amount of if-else statements to code for each type of ship. After refactoring, Ship was extracted as an interface, and each type of ship was concretely implemented through subclasses.

```java
public void setShipName() {
    Scanner myInput = new Scanner(System.in);

    if (location.size() < 1) {
        shipName = "";
        System.out.println("Enter coordinates First.");
    }
    else if (location.size() == 2){
        shipName = "minesweeper";
    }
    else if (location.size() == 3){
        if ( (location.get(0).y) == ((location.get(1).y)) && (location.get(0).y) ==  (location.get(2).y)){
            shipName = "destroyer";
        }
        else if ( (location.get(0).x) == ((location.get(1).x)) && (location.get(0).x) ==  (location.get(2).x)){
            shipName = "destroyer";
        }
        else if (location.get(0).equals(location.get(1).equals(location.get(2)))){
            shipName = "tower";
        }
        else {
            shipName = "l";
        }
    }
    else if (location.size() == 4){
        shipName = "battleship";
    }
    else {
        System.out.println("Hmm... That's a new ship. Go ahead and give it a name!");
        String newShipName = myInput.nextLine();
        shipName = newShipName;
    }
```

Not refactored code

```java
public class Battleship implements Ship {

    /**
     * Class Attributes
     */
    public ArrayList<Point> location;
    private String shipName;
    private int health;
    private CaptainsQuarters ct;

    /**
     * Constructor
     */
    public Battleship(){
        location = new ArrayList<>();
        shipName = "battleship";
        health = 4;
        boolean isSunken = false;
        boolean underwater = false;
    }
```

Refactored code

Example iii. Ship Class Refactoring: Replace Conditional w/ Polymorphism
Previously, we had a large if-else piece of code to assign names for each type of ship. After refactoring, each type of ship is named in it's subclass.

```java
public List<Point> minesweeperInput(){
    Scanner myInput = new Scanner(System.in);
    String input = "";

    System.out.println("Place Minesweeper (2 blocks wide): ");
    System.out.println("Enter \"1\" or \"2\" \n 1. Horizontal \n 2. Vertical ");
    boolean temp = true;
    while (temp) {
        input = myInput.nextLine();
        if (input.equals("1") | input.equals("2")) {
            temp = false;
        } else {
            System.out.println("Input must be \"1\" or \"2\"! ");
        }
    }
    if (input.equals("1")) {
        System.out.println("Enter the X-coordinate of the right-most block of your ship: ");
        String inputValX = myInput.nextLine();
        System.out.println("Enter the Y-coordinate of the right-most block of your ship: ");
        String inputValY = myInput.nextLine();

        Point p1 = new Point(Integer.parseInt(inputValX), Integer.parseInt(inputValY));
        Point p2 = new Point( x: Integer.parseInt(inputValX) + 1, Integer.parseInt(inputValY));

        return Arrays.asList(p1, p2);


    } else if (input.equals("2")) {
        System.out.println("Enter the X-coordinate of the top-most block of your ship: ");
        String inputValX = myInput.nextLine();
        System.out.println("Enter the Y-coordinate of the top-most block of your ship: ");
        String inputValY = myInput.nextLine();

        Point p1 = new Point(Integer.parseInt(inputValX), Integer.parseInt(inputValY));
        Point p2 = new Point(Integer.parseInt(inputValX), y: Integer.parseInt(inputValY) + 1);
```

Not refactored code

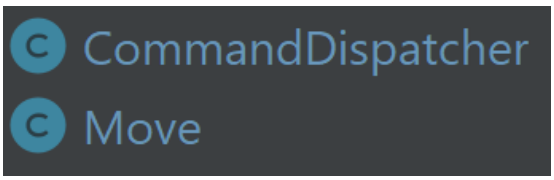
```java
@Override
public List<Point> input(int[][] shipMap) {
    Scanner myInput = new Scanner(System.in);
    String input3 = "";
    System.out.println("Place Battleship (4 blocks wide): ");
    System.out.println("Enter \"1\" or \"2\" \n 1. Horizontal \n 2. Vertical ");

    boolean temp = true;
    while (temp) {
        input3 = myInput.nextLine();
        if (input3.equals("1") | input3.equals("2")) {
            temp = false;
        } else {
            System.out.println("Input must be \"1\" or \"2\"! ");
        }
    }
    if (input3.equals("1")) {
        temp = true;
        while (temp) {
            System.out.println("Enter the X-coordinate of the left-most block of your ship: ");
            String inputValX = myInput.nextLine();
            System.out.println("Enter the Y-coordinate of the left-most block of your ship: ");
            String inputValY = myInput.nextLine();
            boolean criteriaA = (shipMap[Integer.parseInt(inputValY)][Integer.parseInt(inputValX)] == 0);
            boolean criteriaB = (shipMap[Integer.parseInt(inputValY)][Integer.parseInt(inputValX) + 1] == 0);
            boolean criteriaC = (shipMap[Integer.parseInt(inputValY)][Integer.parseInt(inputValX) + 2] == 0);
            boolean criteriaD = (shipMap[Integer.parseInt(inputValY)][Integer.parseInt(inputValX) + 3] == 0);

            if (criteriaA && criteriaB && criteriaC && criteriaD) {
                int x1 = Integer.parseInt(inputValX);
                int y1 = Integer.parseInt(inputValY);
                Point p1 = new Point(x1, y1);
```

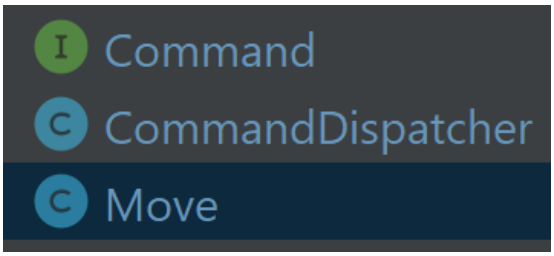
Refactored code

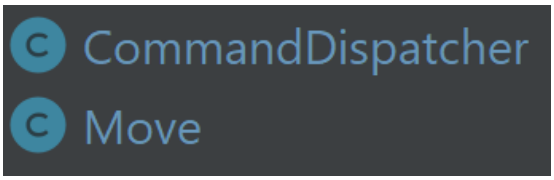
Example iv. GameBoard Class and Ship Class: Move Method
Input methods were previously in the GameBoard Class and moved to the ship class after refactoring.



Not refactored code Refactored code

Example v. Move Class: Replace Subclass with Fields
Previously, we had each direction a player could move as a subclass. After refactoring, we removed all of the subclasses and added each direction as a parameter in the Move constructor





Not refactored code Refactored code

Example vi. Move Class and Command: Remove middleman
The Command interface was unnecessary after we replaced the subclasses in *example v.*

c. Testing

As mentioned in the *Development* section above, we did adhere to the test-driven development method for programming this game. Thus, we always ensure that we have 100% class coverage prior to finalizing any new releases. Furthermore, we always strive for 100% method coverage and line coverage as well. Also mentioned above as well, we initially had some issues knowing and figuring out how to write well-written unit tests. However, as we progressed through the semester and learned more refactoring techniques, we were able to effectively refactor our code to more well-written code.

d. Collaborative Development

Our team frequently uses GitHub and IntelliJ's VCS integration for code collaboration. The team also uses Discord and Zoom as a frequent form of communication. Initially, we had issues with collaboration because half the team resides in the U.S. and the other half resides in Asia. This made it difficult to communicate with each other due to such a drastic time difference. We were able to resolve this issue by frequently working in pairs based on time-zone. Fortunately, we were able to take advantage of a 24 hour work rotation.

III. Requirements and Specifications

a. Requirements

Functional Requirements

- Each player must be able to position five ships - one of each kind.
- Each player must be able to select a location in which they would like to attack, yielding the following possible results: Miss, Hit, Sink, or Surrender.
- Each player must be able to win the game.
- Each player must be to sink ships

Non-Functional Requirements

- The game should support two players.
- The program should be implemented so that they are scalable. Adding extra players or ships should require minimal changes to the code.
- The program should run quickly and accurately. Attacked coordinates and placed ships must be displayed accurately and immediately for every player.

b. Specifications

i. There are currently five types of ships: Battleship, Minesweeper, Destroyer, L-Ship, and Tower.
ii. There are three types of weapons: Regular Missile, Plus Missile, Sonar Pulse

c. Coding Standards

i. Naming Conventions

1. Classes and Interfaces must use whole words and nouns. The first letter of each word must be capitalized. ie.) Class GameBoard
2. Methods must use verbs and camel case. ie.) attackShip ( )
3. Variables should be in camel case. ie.) string shipName

ii. Comments

1. Documentation comments are expected above each method to describe the parameter(s), return type, and how the method is expected to function.
2. Single line comments are expected where there may be unintuitive code.

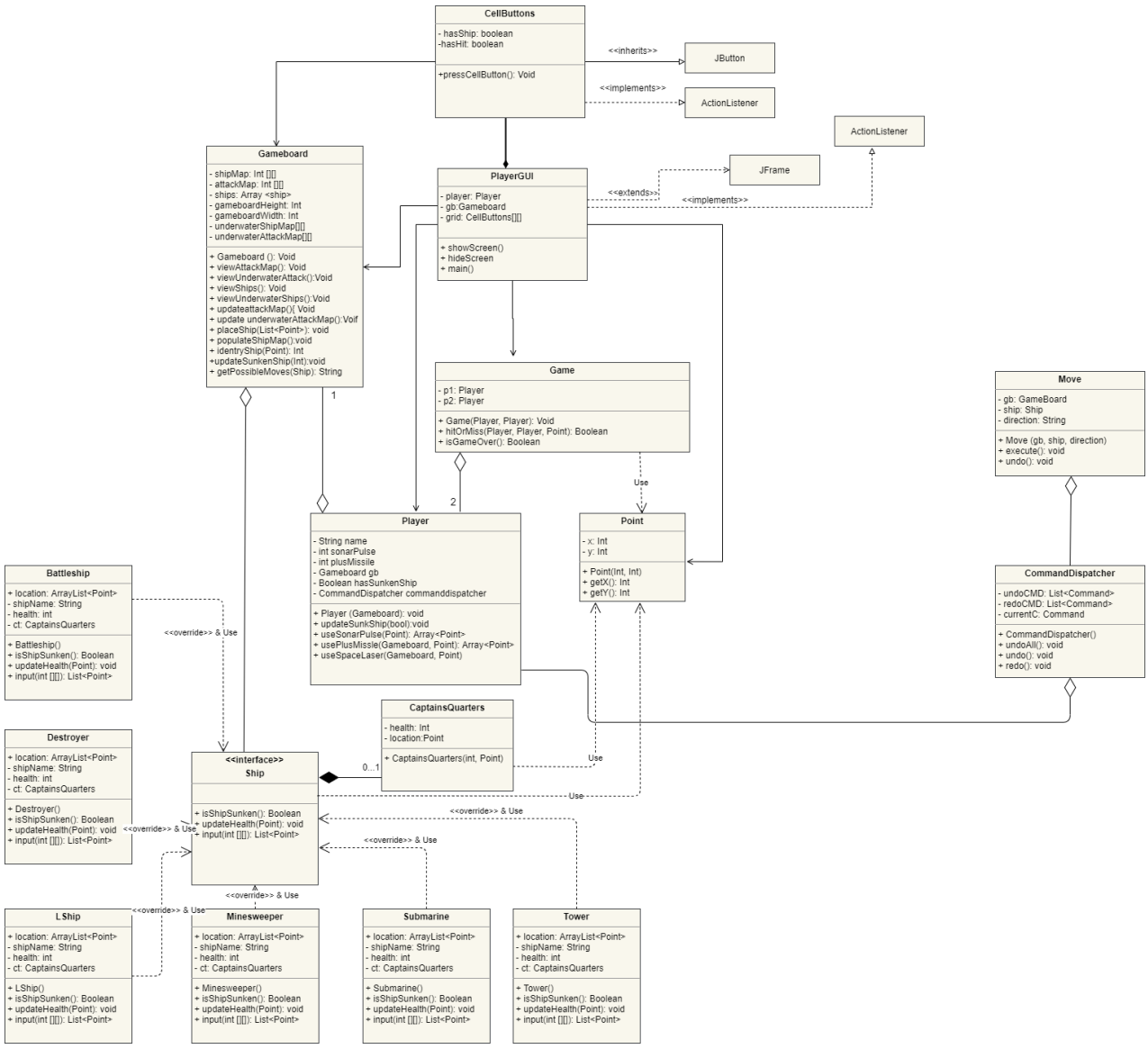
iii. Indentation & Spacing

1. There should be a space after commas.
2. There should be a space before and after arithmetic and boolean operators.
3. Proper indentation should be used throughout the code.

IV. Architecture and Design

a. UML Diagrams

The UML Diagram can also be found here.



b. Description of System

i. Game Class

The Game class holds attributes relevant to the overall game. These attributes are two players. It also holds relevant methods that are to be executed by the game such as checking if the game is over and checking if an attack hit or missed an opponent's ship.

ii. GameBoard Class

The gameboard class holds several 2D arrays acting as game board maps. It holds the maps of where players have placed their own ships and maps of where the player has attacked. Its methods consist of managing and updating any actions that may change one's map. Furthermore, it is also responsible for printing functions that allow a player to view a map.

iii. Ship Interface & Subclasses

The ship interface defines how each of the ships should look and hold methods every ship has. Each subclass holds its own attributes specific to the type of ship it is. The methods consist of methods that manage the ship's attributes, like its health, location, name, and status.

iv. Player Class

The player class holds attributes that are specific to each player. Each player needs to be constructed with a gameBoard. The player class also holds attributes such as a player's name, number of specialty missiles, how many ships a player has sunken, and where the player has attacked. The methods consist of any methods that manage those attributes.

V. Personal Reflections

a. Kevina Wong

There was a lot that I personally learned from this project. I feel that I am stronger in object-oriented programming and understanding design principles and patterns. Something that I thought went well in the project was how well our team worked together despite the dramatically different time zones amongst our team members. Something that didn't go as well as we expected was the GUI implementation. It was a difficult learning experience for all of us, and although we didn't entirely finish the GUI component, I am proud of the text-based game we created and how much of the GUI we were able to complete..

b. Vienna Wong

This project allowed me to apply a lot of design principles that I learned in class this semester. Despite our lack of time implementing the GUI, I feel that I learned a lot about the implementation process. In the future, I'd like to continue developing this game and its GUI. I think one of the strongest parts of the project was our team dynamic. Every member of the team was able to use their strengths to contribute to every milestone. Because of how much practice with design principles I had in this project, I feel that I will be able to apply these concepts in future projects.

c. Yvonne Liu

Overall, I enjoyed this project and learning Java while applying object-oriented skills and concepts from the class to our project. I think that we worked very well as a team, and we were able to hit every milestone meeting most of the expectations of the instructors. I learned a lot of design patterns and applied them, and I also learned how to work on a team project remotely. Something that could be improved is the delegation of tasks and maybe more efficient use of our meeting times; some group members had to do more than others since code is hard to split up.

d. Yubin Go

Overall, I enjoyed the battleship project, and I believe our project went well. It was quite hard to apply design patterns to the code since I never had an experience with it. Also, using TDD style was not difficult in the beginning because we have to think about the whole structure of the project first. Although, learning OOAD elements such as various design patterns, abstraction, encapsulation, and diagrams will be very helpful and essential for my future career.