Battleship: Ultimate

Project Documentation

Team Object Grind

Jake Bonner and Jennifer Gurtler

Contents

Part I

Introduction

Project Description

This project is a modified version of the classic board game known as Battleship. The original version of the game consists of two players, who each have have a fleet of ships and a 10×10 grid that represents the ocean. At the beginning of the game both players place their ships on their grid, positions unknown to the other player. The players then alternate turns attacking coordinates of the other player's grid. If the coordinate a player attacks has a ship, then it is a hit, otherwise it is a miss. If all spaces that a ship occupies are hit, then that ship sinks. The first player to sink all of their opponent's ships is the winner.

We expanded upon this standard version of Battleship by adding several new features to the game, including: three different game sizes, a second layer to the grid, a new type of ship, more weapons at each player's disposal, and the ability for players to move their entire fleet.

Platforms and Technologies

Throughout this project we utilized five main platforms and technologies. Using these technologies improved our ability to communicate and work together virtually, control and centralize all of our code, safely refactor and test code, and ultimately provided a better experience for the team.

Discord

We used discord for both synchronous and asynchronous communication. It allowed us to speak together online and share our screens for pair programming.

Github

We used Github for version control, hosting all of our source code and diagrams online.

Java and IntelliJ IDEA

All of our source and test code was developed using IntelliJ IDEA and was written in the Java programming language. IntelliJ provided a very advanced yet easy to use interface for writing our code and safely refactoring.

ArgoUML

We used ArgoUML to create all of our UML class diagrams.

Part II

Development Process

Extreme Programming

Throughout this project our team followed many practices of Extreme Programming, an agile software development framework that aims at producing higher quality software. The main aspects of this process that we used include: Stories, Planning Game, Weekly Cycle or Iterative Development, Continuous Integration, Test Driven Development, Pair Programming, Incremental Design, and Refactoring.

Stories

This practice involves having stories that describe what the product should do, in terms meaningful to customers or users. These stories are intended to be short descriptions of what users want to be able to do with the product, and can be used for planning, or serve as reminders of key features throughout development.

All of the requirements throughout this project - both those that were either assigned to us and those that the team chose - we wrote in the form of user stories or use cases. By having these brief descriptions of what the user should be able to do with our software, we were able maintain a high level of what our code needed to accomplish, and ultimately how our system needed to behave.

Weekly Cycle and Planning Game

The idea of Weekly Cycle is to have fixed length iterations of development, which gives teams a way to reflect on progress frequently, plan for shorter periods of time, and better estimate the amount of work remaining for features. At the beginning of each iteration is a practice known as the Planning Game. This involves meeting at the beginning of a cycle to reflect on progress, examine user stories and determine how to approach them, and ultimately plan for the implementation of new features. The goal by the end of an iteration is to have tested and running deliverables in alignment with the selected stories from the beginning.

This project was split into two week iterations, or Milestones. At the beginning of each Milestone we would either receive new requirements from course staff, or create our own user stories. We would then have a Planning Game during which we decided how to approach the new stories, including any needed changes or additions to our existing design. After the planning game we split up work, continually evaluated our progress, and presented the required deliverables at the end of the two week period.

Pair Programming

Pair Programming means that all software is developed by two people sitting at the same machine. One person will typically do the coding while the other will ask questions, make comments, offer advice, and provide feedback. This allows for continuous code reviews and faster responses to problems that may stump one person on their own.

In the context of the COVID-19 pandemic, our team was unable to do pair programming while sitting at the same machine. Instead, we used Discord for communication, and whoever was coding would share their screen with the other team member(s). When we were unable to work synchronously, we each created branches of our repository in Github, which allowed us to work on separate features or sections of code. We would then come together, discuss our changes and give each other feedback, and proceed by merging our changes and resolving any issues.

Continuous Integration and Test Drive Development (TDD)

Continuous Integration is the practice of immediately testing code changes when they are added to a larger code base. The benefit of this is that there are fewer changes incorporated into new versions of the code, and in turn integration issues are much easier to catch and fix.

Instead of following the flow of developing code, writing tests for that code, and then running those tests, the practice of TDD involves writing tests first. The idea is the following: write a failing automated test, run the failing test, develop code that makes the test pass, run the test, and then repeat. Programming in this way reduces the feedback cycle to identify and resolve issues, resulting in less bugs that permeate the code and find their way into production.

Any time we added new features or implementation details during this project, we followed these two Extreme Programming practices. We would start by writing failing tests with JUnit, then develop source code until the tests passed. By always creating tests first and having a consistent test suite, along with making small changes that were always tested against our entire code base, it was easy to locate bugs and solve integration issues quickly.

Incremental Design and Refactoring

The practice of Incremental Design involves doing work up front to understand the "big picture" of the system, and then diving into the details of a particular aspect when a specific feature is needed. This approach reduces the cost of changes, allowing teams to make design decisions when necessary based on the most current requirements and information, rather than planning ahead too much and making changes that may end up not being used.

A key part of Incremental Design is the process of Refactoring, a technique for improving the internal structure of existing source code while preserving its external behavior. This practice encourages teams to first focus on having features and code that work, and once that has been achieved, to make small changes that improve the quality, structure, and performance of the program.

For about the first half of this project our team was only focused on meeting requirements and having a system that worked. After having features locked in place we then analyzed our source code and found many places that could be refactored. By completing many refactorings over the course of the last few Milestones, not only did the quality of our design and code improve, but also the readability and ease of interpretation for the majority of our source code. For specific examples of refactoring see the Wiki for descriptions (Milestone 5, Milestone 6 and Final Submission), and snapshots of before/after refactoring in the directory "/DOCUMENTS/Refactoring".

Part III

Requirements and Specifications

Use Cases

1. A user has a fleet of ships, consisting of a combination of four ship types - Minesweeper, Destroyer, Battleship, and Submarine.
2. A user has a grid that represents the ocean that they can place their fleet in.
3. A user can see their own grid - where they have placed their ships and where their opponent has attacked - and they can also see where they have attacked, and the result of each attack, on their opponent's grid. The locations that a user's ship occupies are black on that their own grid.
4. A user can place a ship on their grid horizontally or vertically, but not diagonally. They can place a ship by entering a coordinate for both ends of the ship, and the ship is placed on all locations at and between the coordinates. Each ship can only be placed on certain layers of the grid, as follows:
    - Minesweeper: The Surface (top layer) only.
    - Destroyer: The Surface (top layer) only.
    - Battleship: The Surface (top layer) only.
    - Submarine: Both the Surface (top layer) and Underwater (bottom layer).
5. A user has access to various weapons throughout a game, which become available to them when certain states are reached in the game.
    - A user will start a game with the Bomb weapon at their disposal.
    - A user will receive a Space Laser after sinking one of their opponent's ships. The Space Laser will replace the Bomb.
    - A user will receive a Sonar Pulse after sinking one of their opponent's ships.
    - A user will receive a Horizontal Air Strike after sinking two of their opponent's ships.
6. On their turn, a user can see a list of their weapons and the index of each weapon. They can choose a weapon to use by entering the index next to that weapon.
7. A user can attack their opponent's grid with a weapon by entering a coordinate.
    - A user can attack their opponent with a Bomb. When a user attacks with this weapon it attacks the single coordinate that they choose. The attack either results in a HIT or MISS. If there is a ship at the coordinate then it is a HIT. Otherwise it is a MISS.
    - A user can attack their opponent with a Sonar Pulse. When a user attacks with this weapon it attacks in a diamond pattern centered on the coordinate that they choose. The attack either results in FREE or OCCUPIED for each location that it spans. If there is a ship at a coordinate then it is OCCUPIED, otherwise it is FREE.
    - A user can attack their opponent with a Space Laser. When a user attacks with this weapon it attacks the single coordinate that they choose. The attack either results in a HIT or MISS. If there is a ship at the coordinate then it is a HIT. Otherwise it is a MISS.
    - A user can attack their opponent with a Air Strike. When a user attacks with this weapon it attacks a straight line - either vertical or horizontal - of coordinates centered on the coordinate that the user chooses. The attack either results in a HIT or MISS for each location that it spans. If there is a ship at the coordinate then it is a HIT. Otherwise it is a MISS.
8. When a user attacks their opponent with a weapon and it is a HIT, that location will have an 'X' at it. If the attack is a MISS, then it will have an 'O' at it. A user can see these changes on their opponent's grid.

9. When a user attacks their opponent with the Sonar Pulse, if there is a ship at a location (OCCUPIED) then that location on the grid will become red. If there is not a ship at a location (FREE), then it will become blue. A user can see these changes on their opponent's grid.

10. A user can only use each of their weapons on specific layers of the grid.

    - A user can attack with a Bomb on the surface layer only.
    - A user can attack with a Space Laser on the surface layer and underwater.
    - A user can attack with a Sonar Pulse on the surface layer only.
    - A user can attack with an Air Strike on the surface layer only.

11. A user can only use each of their weapons a certain number of times, which depends on the game size that the user chooses (see later use case).

12. When a user hits a location of a ship with either the Bomb, Space Laser, Horizontal Air Strike, or Vertical Air Strike, that ship loses one point of health.

13. When a user hits all locations occupied by a ship, that ship sinks.

14. When a user sinks all of their opponent's ship, their opponent surrenders, and the user wins the game.

15. Each ship that a user has on their grid has a Captain's Quarters. When a user destroys a Captain's Quarters, the attack counts as a HIT, and the entire ship sinks, regardless of the status of the other locations that it occupies. However, if a user hits a captain's quarters but does no destroy it, then the attack counts as a MISS. The Captain's Quarters for each ship has a different amount of health, which equals the number of times a user must attack it before they destroy it:

    - Minesweeper: 1
    - Destroyer: 2
    - Battleship: 2
    - Submarine: 1

16. A user can move their fleet of ships on their grid, specifying a direction of North, East, South, or West, where North is towards the top of the board. When a user chooses to move their fleet, if at least one ship cannot move, then the entire fleet cannot move. A ship cannot move if it will leave the grid or run into another ship. When the move command is given, if the fleet can move, then each ship moves one location/position in the given direction.

17. A user can undo the most recent action that they took on a turn. They can undo any number of actions on the current turn, but they cannot undo actions from a previous turn. A user can undo the placement of one of their ships or a movement of their fleet, but cannot undo an attack with a weapon.

18. When a user starts the game, they can choose to play a Small, Medium, or Large Game.

    - When a user selects a Small Game,
      - A user and their opponent have a grid that is 7×7.
      - A user and their opponent start the game with the following fleet of ships: 2 Minesweepers, and 1 Destroyer.
      - Once a user has a weapon, they can only use each weapon the following number of times:
        * Bomb: 49 times

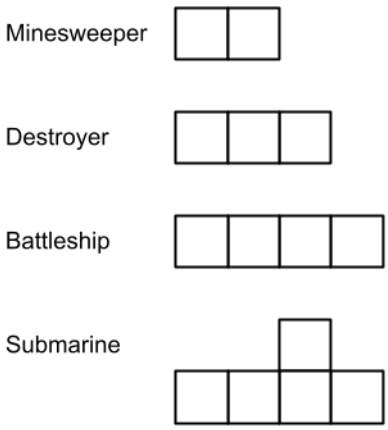
        * Space Laser: 49 times
        * Sonar Pulse: 1 time
        * Horizontal Air Strike: 1 time
- When a user selects a Medium Game
    - A user and their opponent have a grid that is 10×10.
    - A user and their opponent start the game with the following fleet of ships: 1 Minesweeper, 1 Destroyer, 1 Battleship, and 1 Submarine.
    - Once a user has a weapon, they can only use each weapon the following number of times:
        * Bomb: 100 times
        * Space Laser: 100 times
        * Sonar Pulse: 2 time
        * Horizontal Air Strike: 1 time
- When a user selects a Large Game
    - A user and their opponent have a grid that is 13×13.
    - A user and their opponent start the game with the following fleet of ships: 2 Minesweepers, 2 Destroyers, 1 Battleship, and 2 Submarines.
    - Once a user has a weapon, they can only use each weapon the following number of times:
        * Bomb: 169 times
        * Space Laser: 169 times
        * Sonar Pulse: 3 time
        * Horizontal Air Strike: 1 time

Visualizing Game Specifications

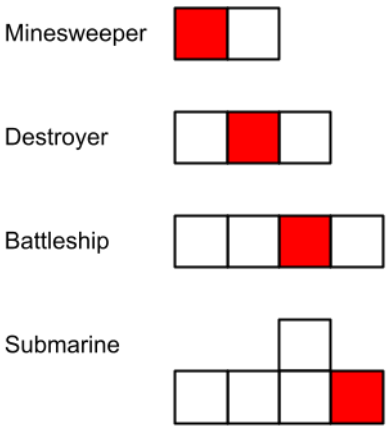
At the beginning of a game, both players receive the same fleet of ships, the same arsenal of weapons, and a grid for placing and moving their ships. The specifics of these components depends on the size of the game that the user selects. Here are the differences between these objects for the different game sizes: the Grids differ in their number of rows and columns, the Fleets have varying combinations of the four ship types, and the weapons within the Arsenals have varying numbers of uses and areas over which they attack. In the following section we give visual representations of all of these differences, and a general overview of each component of the game.

Ships and Captain's Quarters

The four types of ships and the location of their Captain's Quarters are shown below. The number of squares represents the adjacent locations that each ships takes up on a grid, and the captain's quarters are highlighted in red.



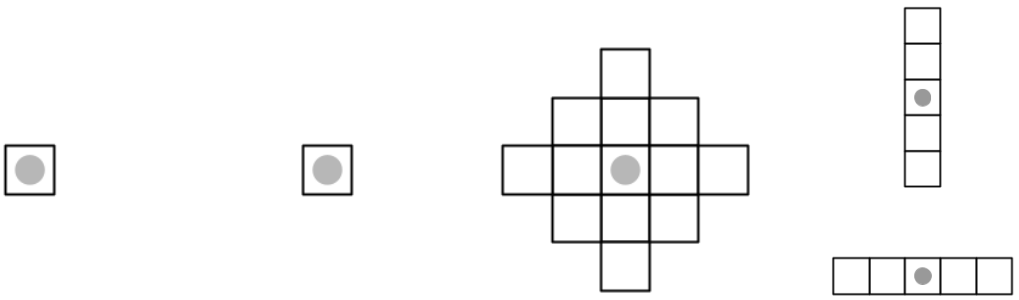
(a) Ship Types



(b) Captain's Quarters

Weapons

As stated in the use cases, there are four weapons that a player has access to throughout a game. The shape of each of the attacks are shown below, where the gray dot is the location that the weapon is used on. Each location that the weapon spans over will be attacked according to the use cases.



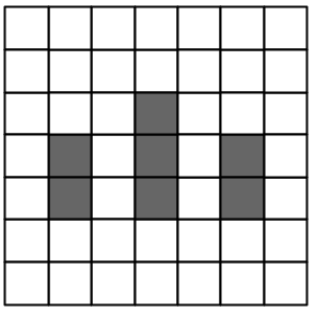
(a) Bomb (b) Space Laser (c) Sonar Pulse (d) Air Strikes

Figure 2: Attack Patterns for all Weapons

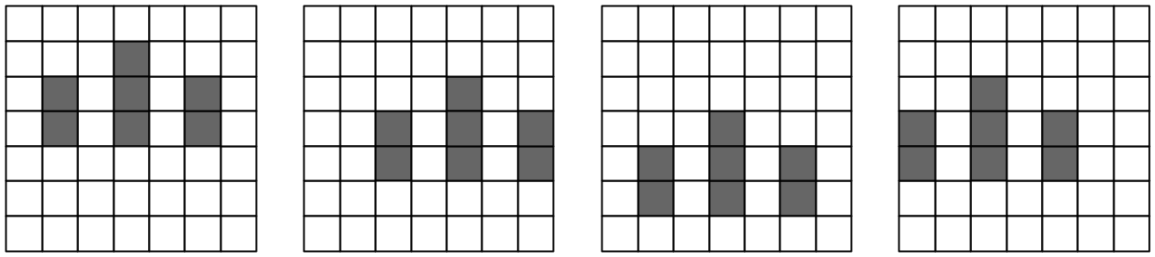
Fleet Movement

Consider the following fleet of ships on a grid, where the ships are the gray squares.

Figure 3: Example Fleet Placement



The figure below shows the result of a successful fleet movement in each of the four possible directions, given the starting positions in Figure 3 above.



(a) North (b) East (c) South (d) West

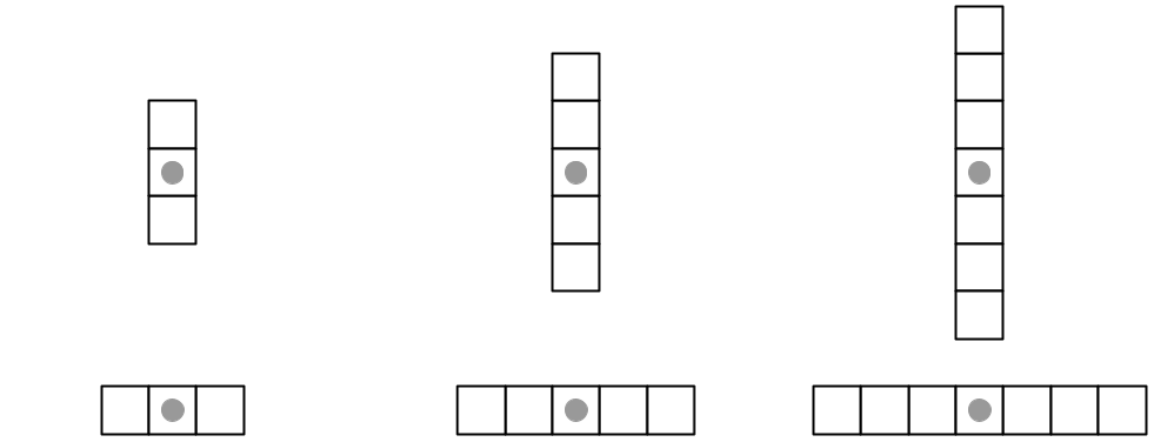
Figure 4: Fleet Movements: N, E, S, W

Arsenal

The Arsenal differs between Game sizes in the number of uses that each of the weapons has. The table below shows these number of uses for the three game sizes.

Game Size	Bomb	Space Laser	Sonar Pulse	Horizontal Air Strike	Vertical Air Strike
Small	49	49	1	1	1
Medium	100	100	2	1	1
Large	169	169	3	1	1

The only weapon that changes in the number of locations that it attacks for each game size is the Air Strike - both Horizontal and Vertical. These differences are shown in the figure on the top of the next page, where the gray dot signifies the location that the attack was used.

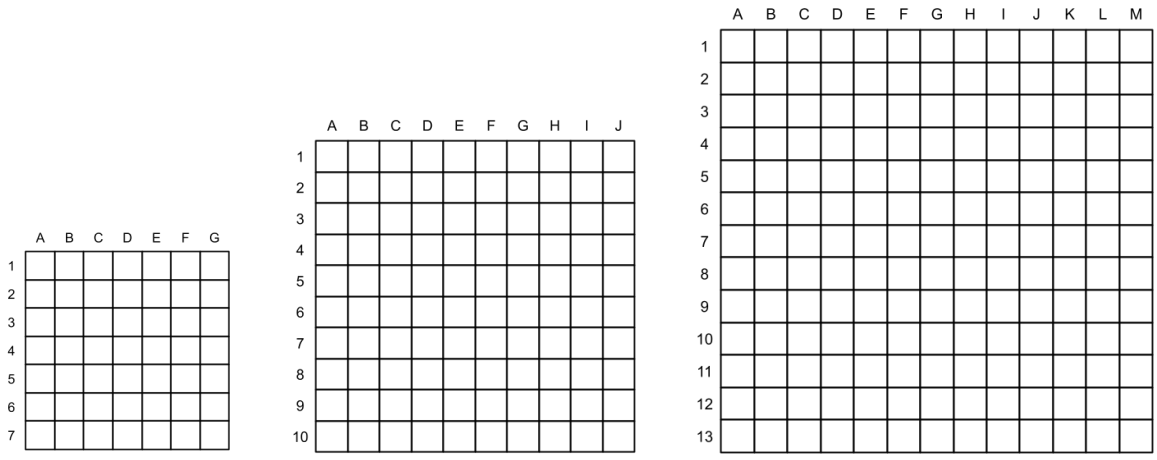


(a) Small Air Strikes, 3 spaces (b) Medium Air Strikes, 5 spaces (c) Large Air Strikes, 7 spaces

Figure 5: Small, Medium, and Large Air Strikes

Grid

The sizes of the Grid for the Game sizes are 7×7, 10×10, and 13×13, for Small, Medium, and Large, respectively. These grids are shown in the figure below.

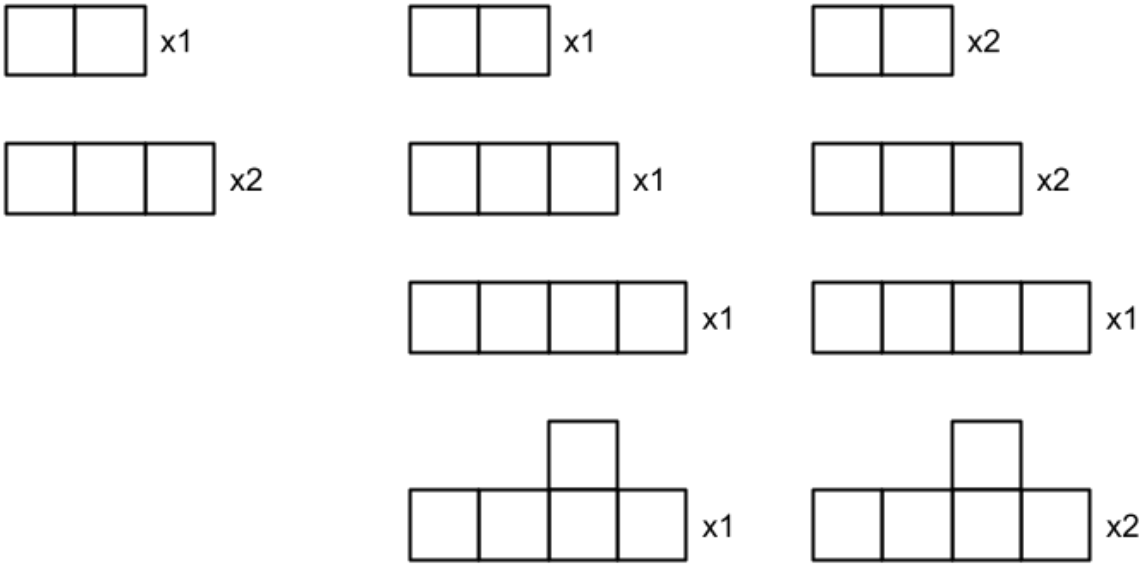


(a) Small Grid, 7×7 (b) Medium Grid, 10×10 (c) Large Grid, 13×13

Figure 6: Small, Medium, and Large Grids

Fleet

As stated before, the fleets for the three Game sizes consist of a combination of the four types of ships - Minesweeper, Destroyer, Battleship, and Submarine. In terms of the number of ships, the Small, Medium, and Large fleets have three ships, four ships, and seven ships, respectively. The three different fleets are depicted below, where the number to the right of each ship represents the number of that type in the fleet.



(a) Small Fleet, 3 Ships (b) Medium Fleet, 4 Ships (c) Large Fleet, 7 Ships

Figure 7: Small, Medium, and Large Fleets

Part IV

Architecture and Design

Architecture

Overview

Our application is divided into a Business Object Layer and a Graphical User Interface (GUI), both of which were coded in Java through the IntelliJ IDEA. The Business Object Layer is the core set of classes that we designed throughout the project, representing all components of the game, and the state of the game while it is being played. The GUI allows users to interact with the business object layer and ultimately have the ability to play the game. The structure of the GUI is designed for two users to play against one another on the same machine. This interface gives each player a way to place their fleet on their grid, and select and use a weapon on their opponent's grid. Throughout the game, on a given player's turn they can see, side by side, their own grid - where they have placed their ships and where their opponent has attacked - as well their opponents grid, which shows the results of their attacks throughout the game. The interface switches between the player's until one of them sinks all of their opponent's ships, and thus wins the game.

Tests

As stated before, we practiced Test Driven Development throughout this project. In order to write our tests, run them, keep them well organized in suites, and ultimately determine our code coverage, we used JUnit5. JUnit is a unit testing framework for the Java programming language, and it provided an incredible amount of value for the team during this project.

Design

Class Diagram Overview

1 Enumerations

2 Game

2.1 Grid

2.2 Arsenal

2.2.1 Weapon

2.2.1.1 Weapon Attack Pattern

2.3 Command

2.4 Fleet

2.4.1 Ship

2.4.1.1 Captain's Quarters

2.4.1.2 Placement Behavior

2.5 Game Parts Factory

2.6 GUI

A couple notes about the class diagrams in this section:

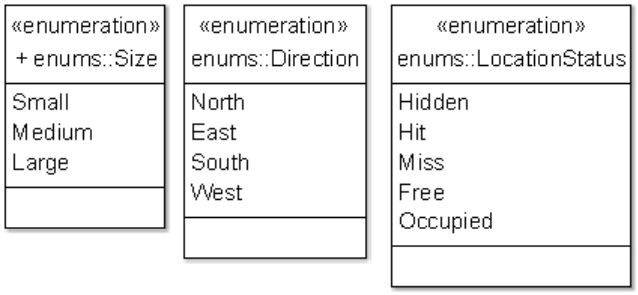
- Any class that realizes/implements an interface will not have the interface's functions listed in its class.
- Any class that generalizes/inherits from a super class will not have any of the attributes or methods from the super class listed in its class, even if the subclass overrides any methods from the super class.
- The only methods and attributes that are listed in a subclass, or class that implements an interface, are methods that are unique to that subclass. In other words the only methods and attributes listed in the diagrams are those that were not inherited from a super class, or defined in an interface.

The reason for creating our diagrams according to these remarks above is to save space within this document and improve readability. For more implementation details on any of the classes, attributes, and methods shown in the class diagrams throughout this section, see the source code.

1 Enumerations

There are three enumeration classes that we used throughout our project. The first of these is Size, which is used for creating and distinguishing between the Small, Medium, and Large game. The second enumeration, Direction, represents the four cardinal directions (N, E, S, W), and is used for moving a player's fleet on their grid. The last enumeration is LocationStatus, which represents all of the statuses that an individual Location of a grid can take on throughout a game. These enumeration classes are shown below.

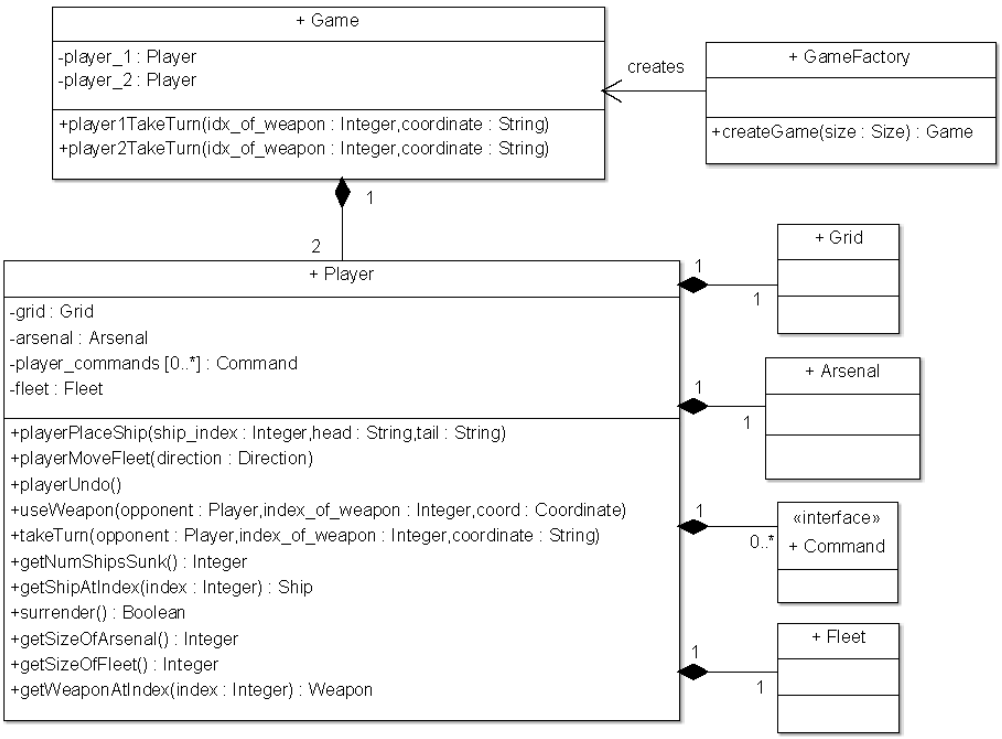
Figure 8: Enumeration Class Diagram



2 Game

At the highest level of our business layer design is the Game class, depicted in the diagram below.

Figure 9: Game and Player Class Diagram

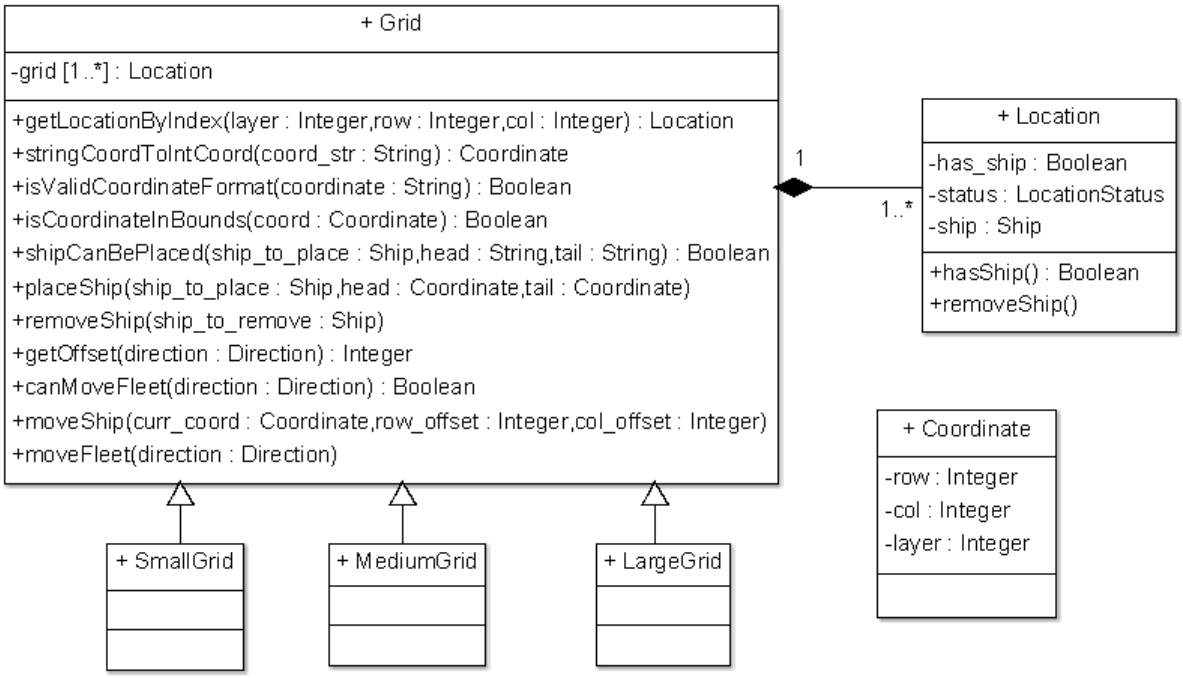


The GameFactory is the only class that knows how to create instances of the Game class, which it does by taking in a Size and creating the corresponding sized game. The Game class itself contains two instances of the Player class, which represent the two players of the game. Each Player has a Grid, an Arsenal, a Fleet, and a stack of Commands. Each of these classes are described in detail throughout this section.

2.1 Grid

As shown in Figure 9, the Player class has an attribute that is an instance of the Grid class. The diagram for this class is shown below.

Figure 10: Grid, Location, and Coordinate Class Diagram

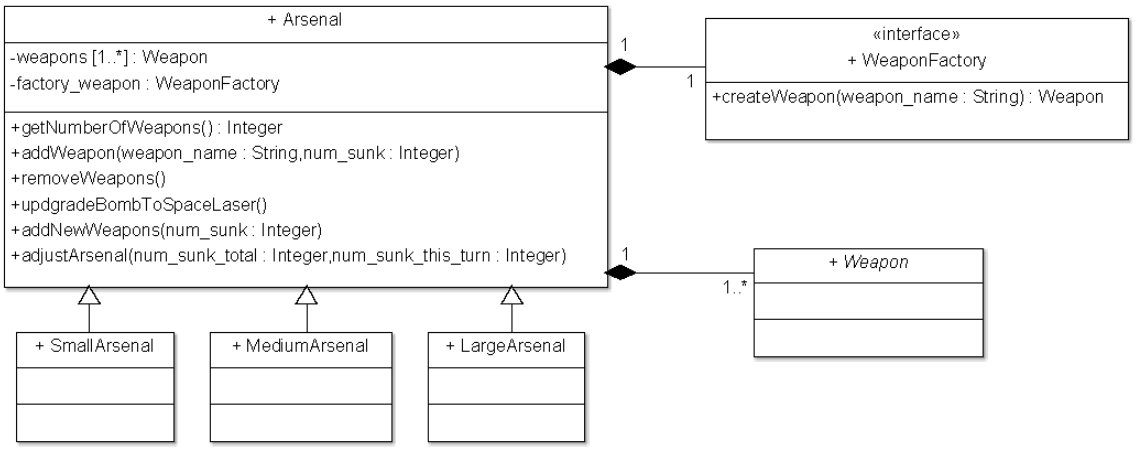


The Grid class contains a 3-Dimensional array of instances of the Location class, and has methods for placing and removing ships, moving a fleet, and ensuring that a coordinate entered by the user is valid for that grid. It also has one subclass for each of the possible game sizes. The Location class represents an individual location of a player's grid, and it keeps track of its own status and whether or not there is a ship at it. There is also a Coordinate class which represents a 3D coordinate - layer, row, and column.

2.2 Arsenal

As shown in Figure 9, the Player class has an attribute that is an instance of the Arsenal class. We show the structure of this class now.

Figure 11: Arsenal Class Diagram

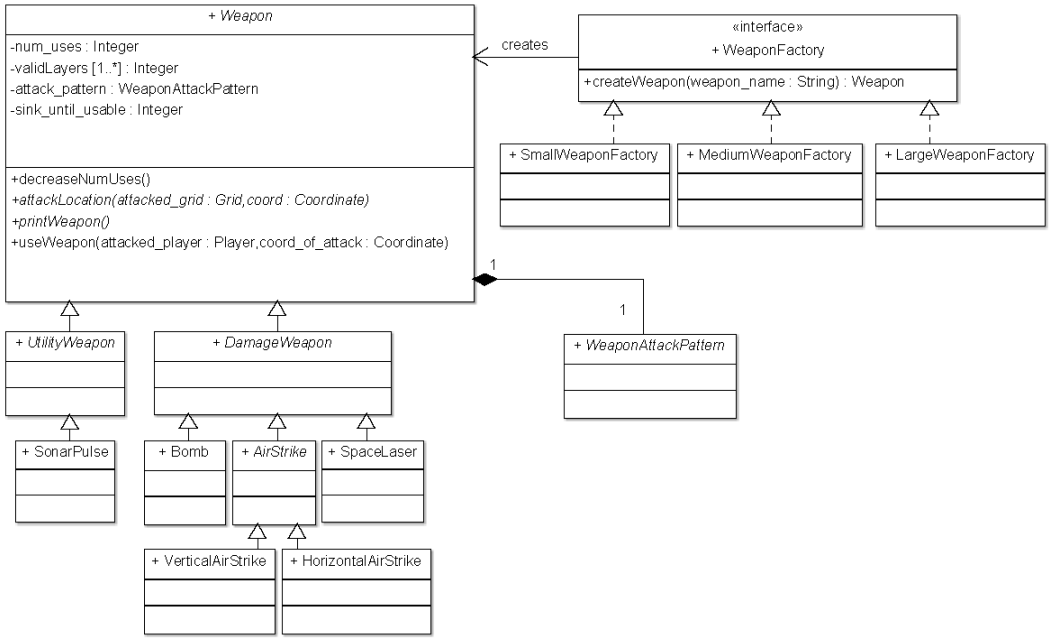


The Arsenal class contains an array of Weapon objects, methods for adding and removing weapons, and has one subclass for each of the three possible game sizes. It also has an attribute that is a WeaponFactory, where each Arsenal subclass has the corresponding subclass of WeaponFactory that creates weapons of the needed size. The WeaponFactory and Weapon classes are described by the next diagram.

2.2.1 Weapon

As shown in Figure 11, the Arsenal class has an attribute that is an array of Weapon objects, and an attribute that is a WeaponFactory. These two classes, and all of their subclasses, are shown below.

Figure 12: Weapon Class Diagram

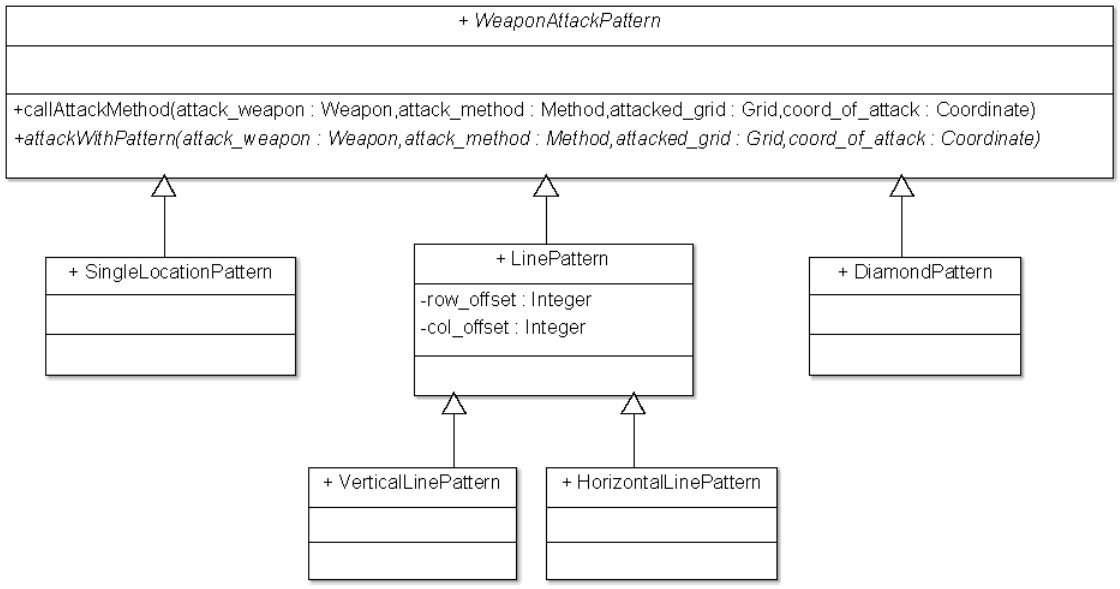


The WeaponFactory interface is the only class that knows how to create Weapon objects, and its subclasses create the weapons used for a particular game size. In terms of the Weapon class, it has attributes representing which layers of the grid it can be used on, how many uses it has remaining, how many ships a player needs to sink before having access to it, and also a WeaponAttackPattern attribute that it follows when it attacks a player's grid. The Weapon has two direct subclasses - UtilityWeapon and DamageWeapon. The UtilityWeapon class represents the weapons that result in FREE or OCCUPIED after an attack, while the DamageWeapon class is those weapons that result in HIT or MISS. All of the subclasses of DamageWeapon and UtilityWeapon simply represent the concrete weapons that a Player can use throughout the game.

2.2.1.1 Weapon Attack Pattern

As depicted in Figure 12, the Weapon class has an attribute that is of type WeaponAttackPattern.

Figure 13: Weapon Attack Pattern Class Diagram

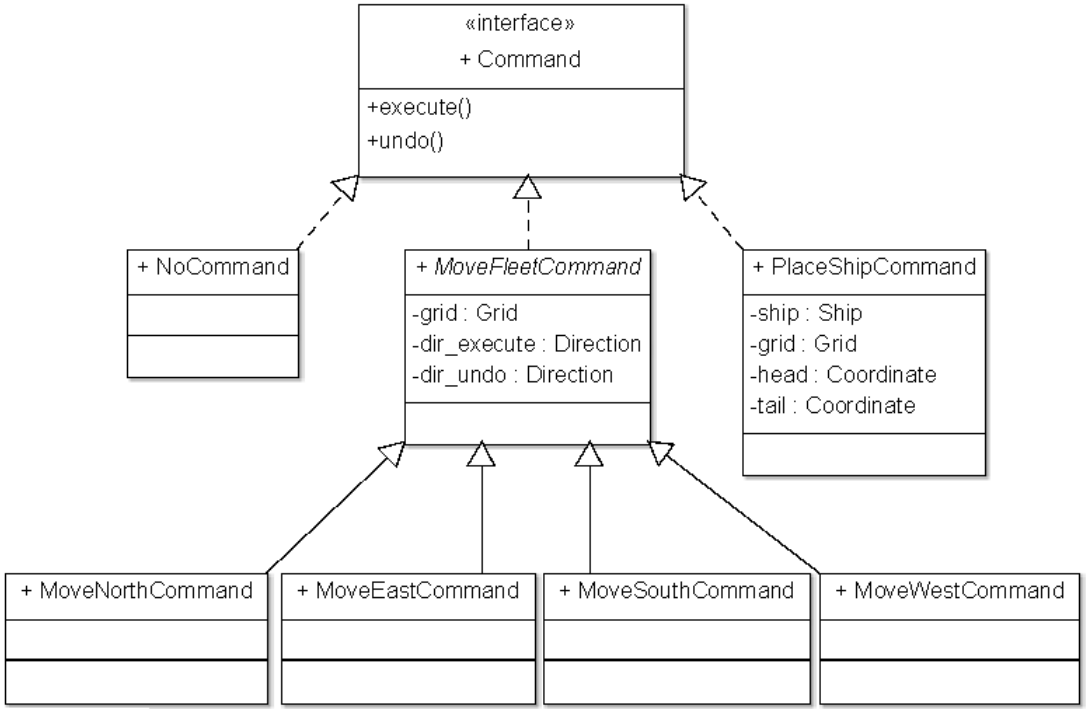


The purpose of the WeaponAttackPattern and its subclasses is to define patterns that each of the Weapon objects follow/use when they are used on a player's grid. The methods defined within the interface take an attack method from a Weapon and apply it to a grid in the pattern that it defines. The SingleLocationPattern simply attacks the single coordinate that is entered, and it is used by the Bomb and SpaceLaser Weapons. The LinePattern and its concrete subclasses attack in either a horizontal or vertical line on the given grid, and the two attributes of the class determine how many locations the attack spans. This pattern is used by the HorizontalAirStrike and VerticalAirStrike Weapons. Lastly, the DiamondPattern attacks the grid in a diamond shape centered on the given coordinate, and it is used by the SonarPulse.

2.3 Command

As shown in Figure 9, the Player class has an attribute of type Command. This interface and its subclasses are shown in the following diagram.

Figure 14: Command Class Diagram

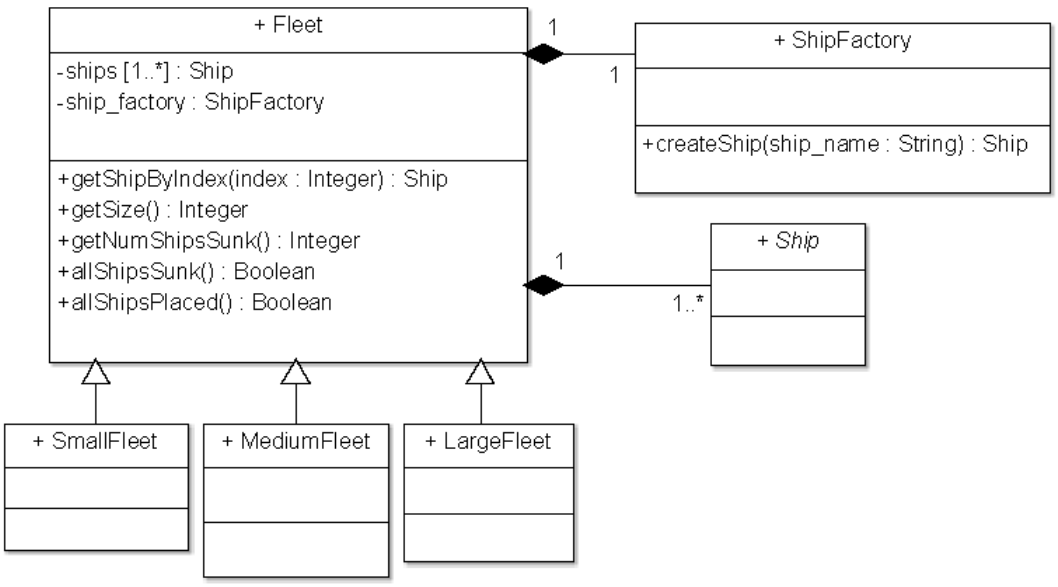


The Command interface represents a command/action that a Player can execute throughout a game, and it defines methods for executing and undoing said command. The NoCommand is simply an object that takes place of null for this class, and it does nothing. The MoveFleetCommand has attributes that represent the direction for executing and undoing fleet movement, and the grid whose fleet will be moving from the command. Each of the MoveFleetCommand subclasses represent a command for moving a fleet in each of the four cardinal directions. Lastly, the PlaceShipCommand represents the placement of a ship on a grid, and its attributes are the ship that is being placed, the coordinates of where the ship will be placed, and the grid that the ship will be placed on.

2.4 Fleet

As shown in Figure 9, the Player class has an attribute of type Fleet. The structure of this class is shown below.

Figure 15: Fleet Class Diagram

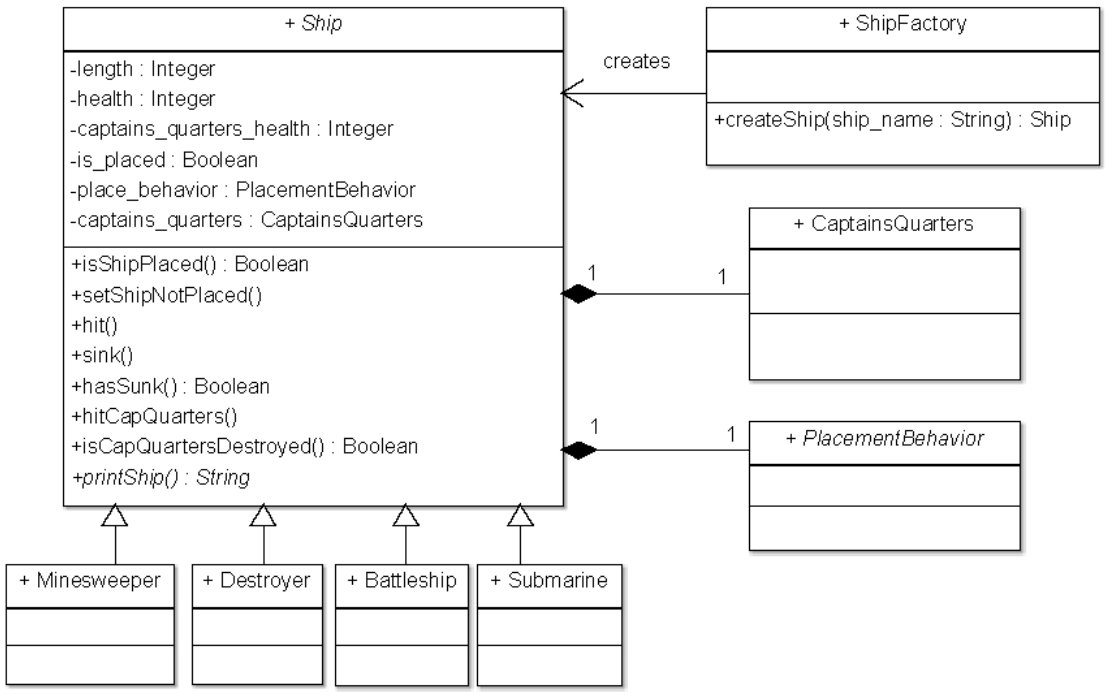


The ShipFactory is the only class that knows how to create Ship objects, and the Fleet class has an attribute of this type so that it can create the ships that are a part of it. The Fleet class also has an attribute that is an array of Ship objects, which represents all of the ships that are in the fleet. It has one subclass for each of the three possible game sizes.

2.4.1 Ship

As shown in Figure 15, the Fleet class has an array of Ship objects, the class of which we now describe.

Figure 16: Ship Class Diagram

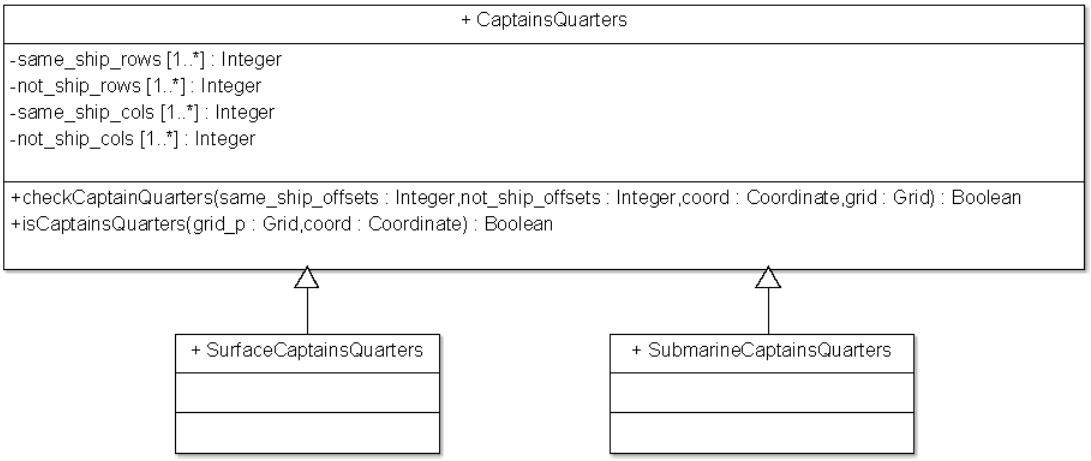


As stated before, the ShipFactory is the only class that knows how to create Ship objects. The Ship class itself contains attributes representing its length, health, and whether or not it has been placed. It also has attributes of type CaptainsQuarters and PlacementBehavior, which are two classes that define the captain's quarters and placement behavior for different ships, respectively. The four subclasses of Ship simply represent the four possible ship types that a player can have in their fleet.

2.4.1.1 Captain's Quarters

As seen in Figure 16, the Ship class has an attribute of type CaptainsQuarters. The diagram for this class is the following:

Figure 17: Captain's Quarters Class Diagram

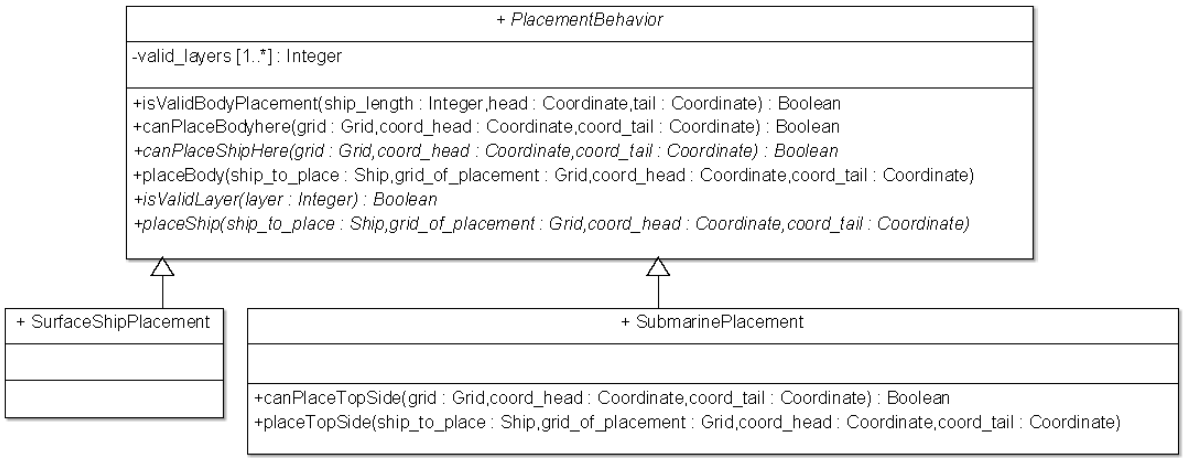


The CaptainsQuarters class defines the behavior for determining if a particular location of a ship on a grid is the captain's quarters. It does this through attributes and methods that check both the needed horizontal and vertical conditions for a captain's quarters. The subclasses SurfaceCaptainsQuarters and SubmarineCaptainsQuarters represent the captain's quarters for the surface Ships - Minesweeper, Destroyer, and Battleship - and the Submarine, respectively.

2.4.1.2 Placement Behavior

As seen in Figure 16, the Ship class has an attribute of type PlacementBehavior. The diagram for this class is the following:

Figure 18: Placement Behavior Class Diagram

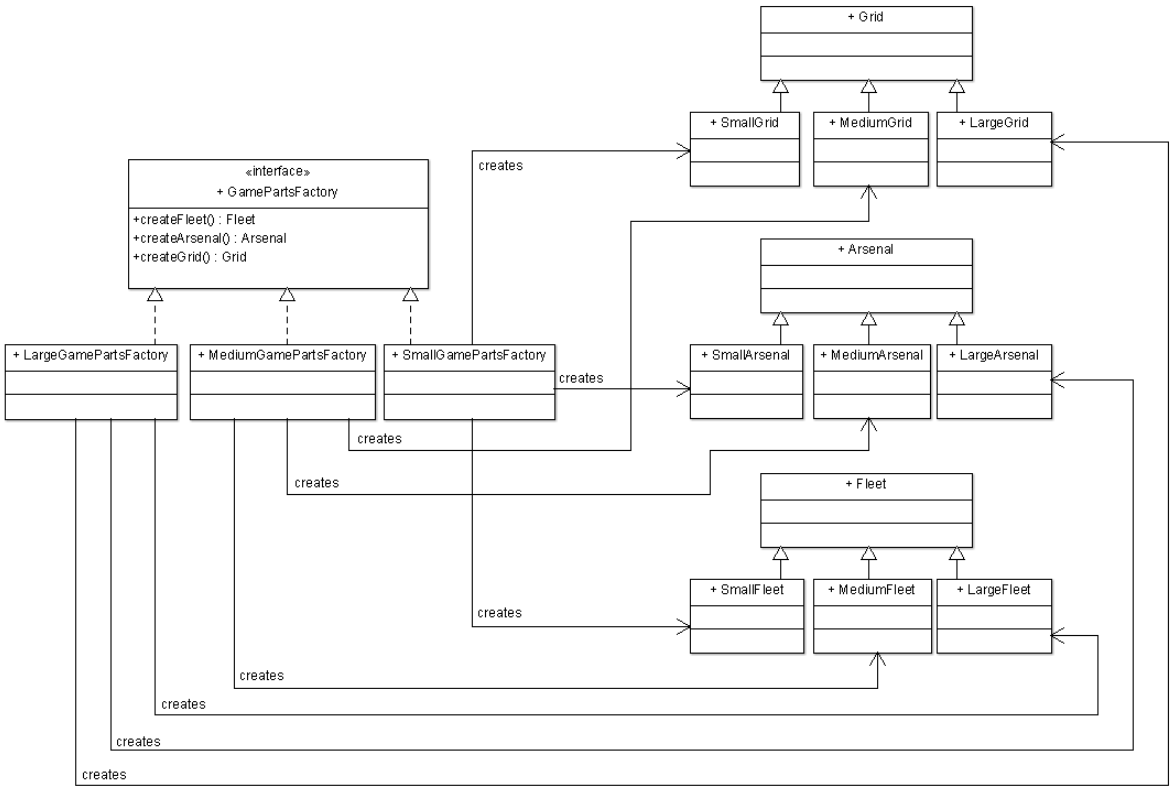


The PlacementBehavior class defines the behavior for how a particular type of ship is placed on a grid, including which layers it can be placed on, and determining if an attempted placement at a given location is valid. The subclasses SurfaceShipPlacement and SubmarinePlacement represent the placement behavior for the surface Ships - Minesweeper, Destroyer, and Battleship - and the Submarine, respectively.

2.5 Game Parts Factory

As mentioned earlier in the user stories, and depicted in the class diagrams for Grid (Figure 10), Fleet (Figure 15), and Arsenal (Figure 11), our design has the ability to create a Game of three different sizes, where the size determines the nature of the components of the Game. In order to create a Grid, Fleet, and Arsenal for each of the three game sizes, we made use of the Abstract Factory design pattern. There is an interface that represents the abstract factory, defining functions for creating a Grid, Fleet, and Arsenal. Then there are three concrete factories - one for each Game size - which implement the functions from the abstract factory, creating the Grid, Fleet, and Arsenal corresponding to each size. The class diagram for this design is shown below.

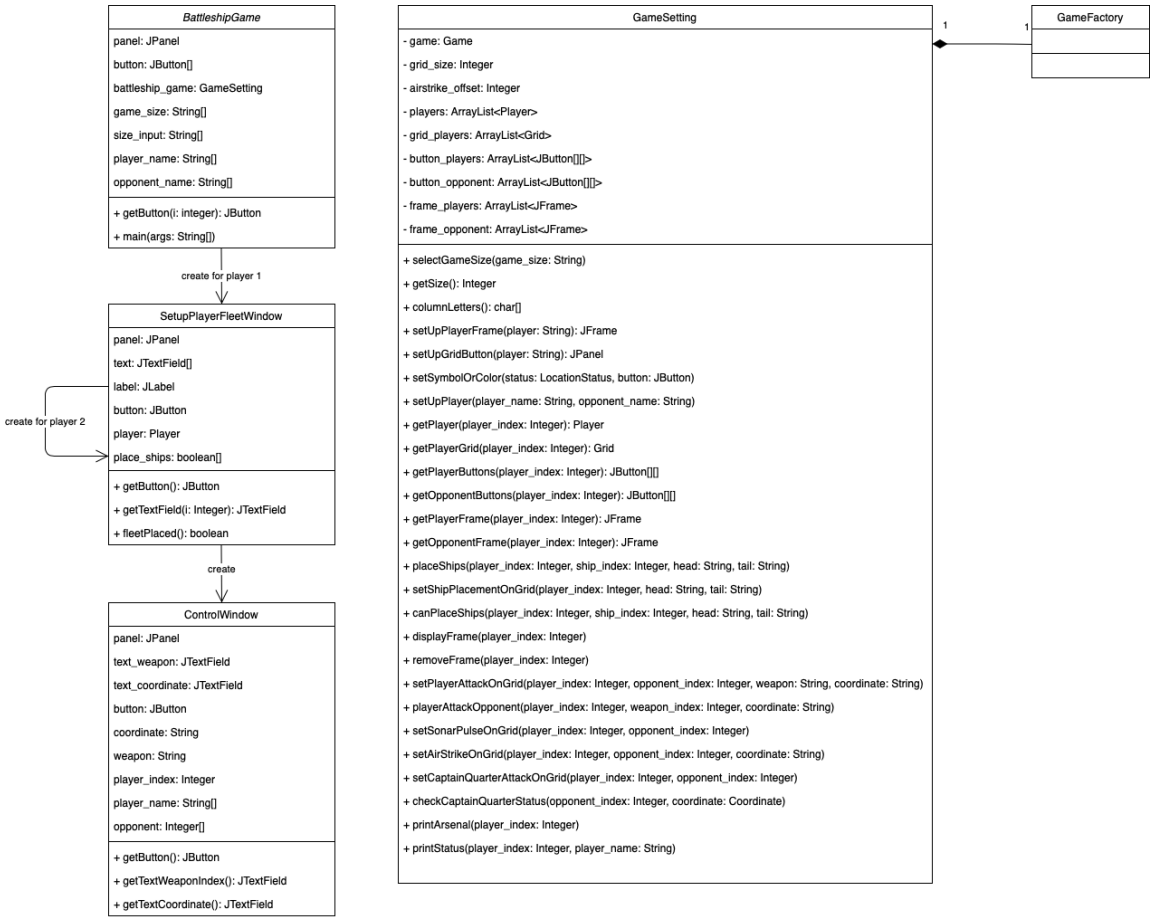
Figure 19: Game Parts Factory Class Diagram



2.6 GUI

The design of GUI can select the game size, place the ships by entering head and tail coordinates, and play the game until one of the two players surrender. There are three classes that present the windows during the game. The BattleshipGame class defines the game size selection window. After the user presses the button of one of the three different game size, the class will create the SetPlayerFleetWindow objects. The SetPlayerFleetWindow class is placing fleet window. When player 1 presses submit button, the current object will create new SetPlayerFleetWindow object but is for player 2. After player 2 presses the submit button, the SetPlayerFleetWindow object will create ControlWindow object where the user can enter weapon index and attack coordinate and the player's own grid frame and opponent grid frame will pop out. The GameSetting class is storing the game status throughout the game which includes both players' status and attack actions. The class diagram for this design is shown below.

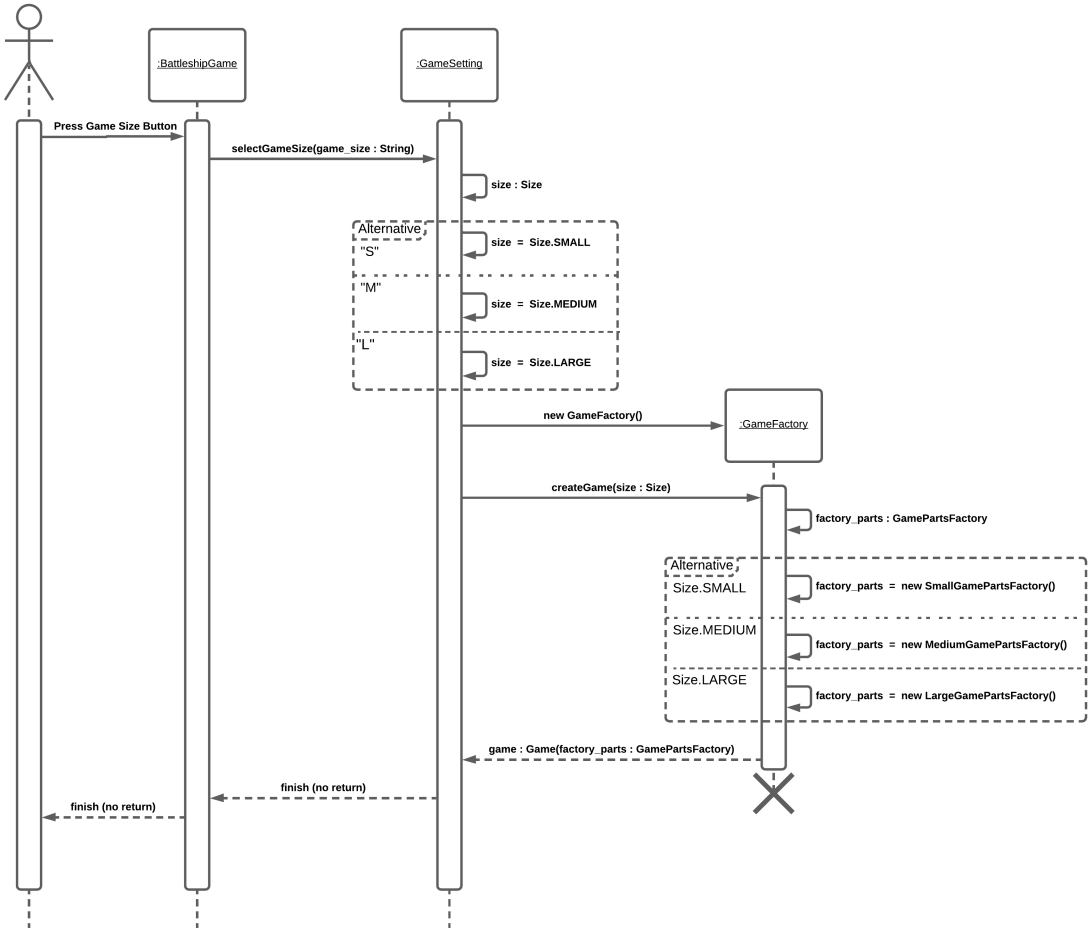
Figure 20: GUI Class Diagram



Sequence Diagrams

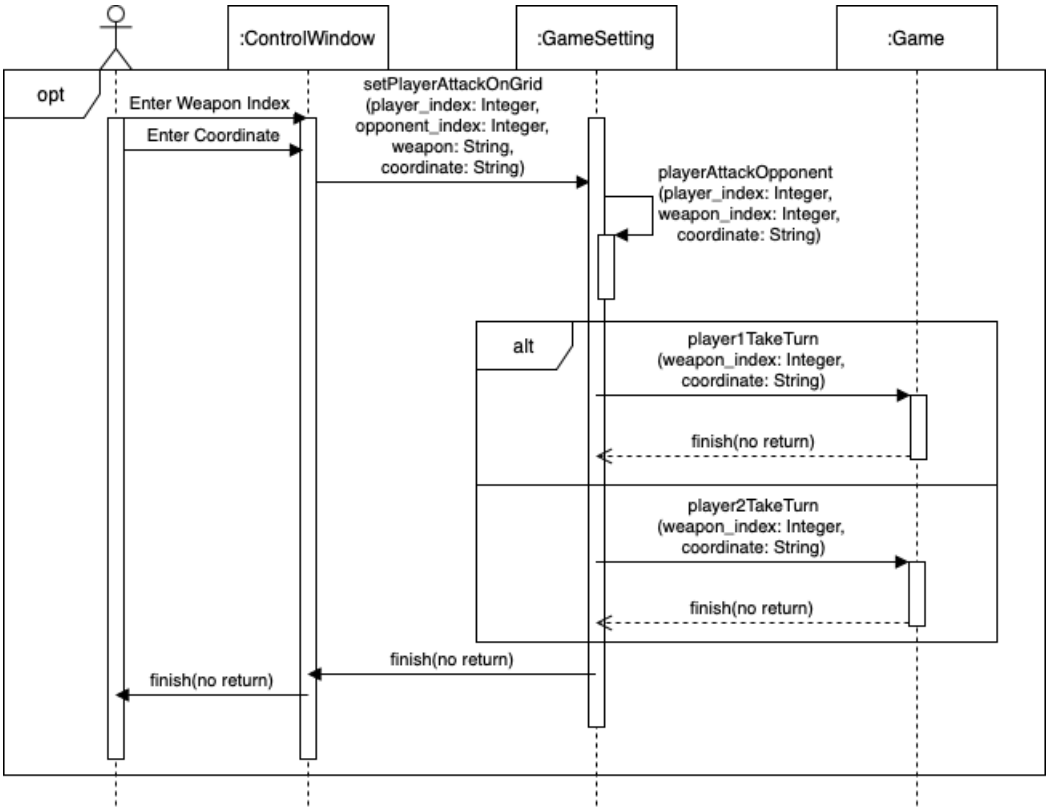
Choosing Game Size

Figure 21: Use Case 18 Sequence Diagram



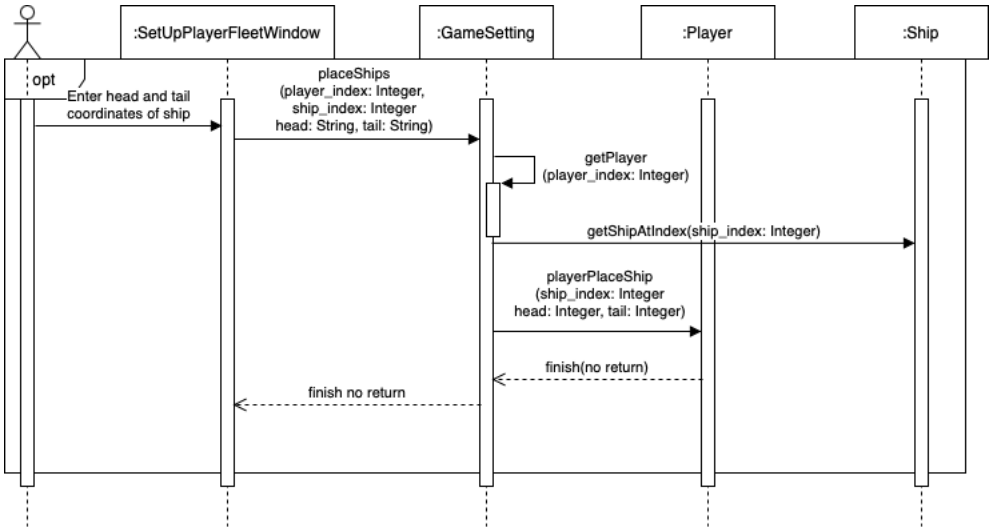
Player Attacks Their Opponent with a Weapon

Figure 22: Use Case 7 Sequence Diagram



Place Ship

Figure 23: Use Case 4 Sequence Diagram



Part V

Personal Reflection

Jake

This project was a great opportunity to expand my knowledge and experience with eXtreme Programming, design patterns, and object oriented design principles. Having no previous experience with coding in Java, and little experience with object oriented concepts and design, the amount that I learned while doing this project was immense. At the beginning of the project, as a group we knew very little about object oriented design and programming, but as we continued throughout the semester it was awesome to watch our skills develop and see the project unfold into something that uses multiple design patterns, and follows many design principles. Overall I am very thankful for this project, as both an opportunity to grow my design and programming skills, as well as connect with other students and work as a team.

Jennifer

This project gives a opportunity to learn about eXtreme Programming, design patterns, and object oriented design principles. At the beginning of the project, I was kind of lost since I didn't have much knowledge about object oriented for design. Throughout the semester, I gain more knowledge on the object oriented concepts and design patterns. In this project, we get to add multiple design patterns which makes the code organize and efficient. The refactoring knowledge helps the team to fix the bad code. I am happy to take this course to improve my knowledge on code design. Also, I am happy to work with other people as a team.