Battleship

Project Documentation

Team 20 - CSCI 4448

Vanessa Reyes

Joseph Petrafeso-Crosby

Priscila Trevino

Salvatore Pacifico

Table of Contents

Introduction to The Battleship Project

This project was developed in Spring 2021, for CSCI 4448:Object-Oriented Analysis and Design by Team 20. It is based completely in Java developed in the IDE IntelliJ and using Extreme Programming (XP). This game is based and played in the terminal of a given IDE. Our development process was heavily based on our learning of refactoring, design patterns, and object-oriented design.

Battleship is a strategy-based guessing game played by two users. Our implementation is set for a human user to play against an AI. The AI used in the game was developed using functions within the project code that allows the AI to have a knowledge of hits. The AI will base all of its shots on its knowledge, and if they are no stored knowledge of any hits, the AI will randomly guess its shot. This Battleship game has the familiar features of the classic board game, but also has new and updated features to allow the user a fresh experience in the game of Battleship (discussed more in the Requirements and Specifications section).

Our project development had a heavy focus on the open-closed principle, where our code should be open for extension but closed for modification. Moreover, although our game is set up for a human user against an AI, the structure of our code has the ability to have a human versus human gameplay without the need of having to make significant modifications outside of the Game Manager (discussed more in the Architecture and Design section). As well, if we wanted to add more features, such as new types of weapons or ships, we can handle these requirements as our code is set up to extend rather than modify.

Furthermore, we also made use of multiple design patterns, including the Observer Pattern, the Command Pattern, the Strategy Pattern, and the Factory Pattern. Each pattern allowed our code to be more extendable rather than modifiable, highly cohesive, and loosely coupled. These patterns are discussed more in the Architecture and Design section of this documentation.

Goal

Create a Battleship game that will allow a player to play against an AI computer within a UI that is based in the terminal. Each player will have a turn each round where they will be able to shoot against their opponent and see the progress that they make in the game. As well, create different features that will make the game more advanced and fun for the player. Overall, our project allows for players to play a modified version of Battleship that has different features and game power-ups that will create a more immersive experience and unique gameplay.

Motivation

Our team through this project would be a great way to begin our foundation in Object Orientated Programming so that we can understand concepts such as Inheritance, Delegation, Design Patterns, Refactoring, and much more. As well, the members of our team have not had previous experience in programming in Java so this project was also a great opportunity for us to learn how to work with Java and how OOP works within it. Moreover, this project was also a great way for our team to gain more experience working with others on a larger code base as well as collaborating in a virtual environment that is very common within the industry. Overall, our team gained many skills that we will be able to apply to future projects and we feel that this project has motivated us to continue learning more about OOP and become better programmers.

Development Process

Before we began any coding, our team began our process of development by working on UML diagrams such as class diagrams as well as sequence diagrams. This process allowed our team to develop a foundation of what classes our program required and how these classes would communicate with each other; furthermore, our sequence diagrams gave us further insights into processes and steps that our project would follow to complete and delegate tasks. When our team began the process of coding, we needed to choose a software development process that would suit our needs and requirements best. Thus, our team followed a version of the Extreme Programming (XP) software development process. XP is made up of three main parts: test-driven development (TDD), pair programming, and refactoring. Our team followed the component of test-driven development (TDD) by writing test cases first before developing and implementing any code. The benefit of TDD is that it allows our team to create proper specifications on how our code should be written and implemented, developing a better program design and higher quality code, code flexibility, as well as reducing the time required for project development. Moreover, with XP we as a team pair programmed throughout our development process. We made some modifications to our process of pair programming as we only had options of remote collaboration as a team. Therefore, we used pair programming by splitting our team into different pairs each week, where each team meets remotely in Zoom and focuses on similar components in the project. This allowed for each pair to think together on code design to make more efficient code, fewer mistakes, and find ways to collaborate by sharing ideas and being effective in knowledge sharing. Lastly, with XP, our team followed the component of refactoring our code. When our team felt that we had a suitable foundation of our code and everyone had an understanding of the system, we as a team began working together to find ways to refactor our code. The refactoring process allowed our team to improve the design of our code, created more understandable code, find and solve bugs that allowed for a faster program, and allowed each member to change the way that they think as a developer when implementing future code.

Throughout our coding implementation, our team worked with an iterative development process. Every two weeks, we as a team had different requirements that we implemented into our project design such as new features and design patterns. We found that with an iterative development process we as a team had to continue changing our code design and system to fulfill our new requirements. A change in requirements created issues in each iterative process as we had to rethink our system and come together as a team to come up with a solution. However, having an iterative process allowed our team to learn how to create a system that featured design patterns that would allow our code to be dynamic and handle changes in requirements. We as a team learned how to implement different design patterns and see how effective these patterns are in future iterations. Eventually, we were able to create a design that was able to handle new requirements that were developed and allowed our team to easily implement these new features as well as spend time implementing other features that we thought would create a more efficient and fun game.

Requirements and Specifications

Requirements

Throughout the development process of this project, we have had many iterations and requirement changes. Below, we have listed all the requirements that have been implemented in our game.

Player Requirements

In the Player class, the requirements that are met deal with a player's game piece placement, the player's shot attempts, and Computer AI. The player class delegates operations to take user input or create random coordinates at the beginning of the game to communicate with the Board class to place all the player's game pieces. As well, Game Management communicates with the Player class to tell when a player's shot position input is needed or when there need to be random shot coordinates to be created. Lastly, the Player class has an AI implementation to allow for the user's opponent to have a sense of the game and create a more difficult game for the user.

Game Management Requirements

In Game Management, there are a lot of requirements that are implemented. One requirement is that we created a UI that meets the requirements of showing each player two grids, where one is their fleet and their opponent's fleet for each different type of board. As well, this UI provides a menu to meet the requirements of a clear, user-friendly UI. Another requirement that is implemented in Game Management is that each player has a turn at their opponent's game pieces each round. We also make sure that each player is given the required set of game pieces at the beginning of the round: Minesweeper, Battleship, Destroyer, Submarine, and Bomber; furthermore, we allow players to choose one additional game piece to their set to meet our other requirements/feature. As well, Game Management also accomplishes the feature of knowing when to change the player's shot type to other shot types and setting the limits on how many of these shots are allowed after certain requirements are met (ex: space laser). Lastly, Game Management also handles the requirements of keeping track of when the game is supposed to end as well as when certain features (good/bad luck) are supposed to begin.

Board Requirements

Our Board class delegates and handles many requirements for our game. One requirement that is handled in this class is making sure that each player has a 10 by 10 grid that represents their board that holds/displays their game pieces. As well, the Board class handles collecting data on where game pieces are placed and what their status is in the game such as coordinates and placement direction. That means if an opponent hits a game piece it will show and update its data to know that the piece is sunk or has lower health. This implementation in the class also follows the requirements of handling the Captain's Quarter on each ship and communicating with the Ship class to update the status of its health. Thus, we can tell users when a Captain's Quarters has been hit and if that caused the ship to sink based on the ship's health. Moreover, the board also deals with show requirements, meaning that when a player uses a sonar this class handles showing the requirements for that shot type. As well, this class is also able to communicate with the Player class to know shot types and how that is supposed to affect which boards, this meets requirements such as the space laser.

Game Piece Requirements

Some requirements that are met in the Game Piece class are handling the many different types of game pieces and keeping all their unique requirements defined for each one. For example, we can handle knowing if a piece can be placed underwater, on the surface, or in the air. As well, we can keep track of a game piece's health information which helps meet requirements such as sinking, Captain's Quarters features, and progress in the game.

Game Probabilities Requirements

In this class, we can handle the requirements and features of dealing with calling commands to determine if certain features are supposed to happen in the game. This class allows us to delegate tasks such as good luck and bad luck in the game.

User Stories/Use Cases

- As a player, I want to be asked where I can place my ships in order to begin the game.
- As a player, I want to know where my hits and misses are on the board to know my progress in the game.
- As a player, I want to be able to play against a computer that has AI so that I can have a smart opponent when playing the game.
- As a player, I want to be asked where I want my first and following shots to go so that I can continue the game and play against the AI.
- As a player, I want to be able to see whether I hit/missed a shot so that I know my progress in destroying my opponent's game pieces.
- As a player, I want to play a full game of battleship and know when I have either won or lost the game.
- As a player, I want to know if my ship placement or shot decision is invalid so I do not overlap my ships.
- As a player, I want to know where my opponent's shot was and if it hit/missed my boats so I know the status of my game pieces and my opponent's progress.
- As a player, I want each of my ships to have a Captain's Quarter so that my opponent can destroy my ship if it is hit.
- As a player, I want to know when I hit the captain's quarter on my opponent's ships so I know how the hit affects my opponent's game piece and my progress.
- As a player, I want the entire ship to be destroyed when the captain's quarter is hit twice when I hit an opponent's Minesweeper.
- As a player, I want to see that I have hit an opponent's Captain's Quarter and that part of the ship is weak when my opponent's board is displayed.
- As a player, I want to be able to use the sonar radar so that I can pick a place to see if my opponent's game pieces are in that location.
- As a player, I want to be able to place both ships on the surface and underwater so that I can use all the boards in the game.
- As a player, I want a new ship called a submarine that can be placed either underwater or surface.

- As a player, I want a new weapon called Space Laser after I have destroyed one of my opponent’s game pieces.
- As a player, I want my space laser to go through the regular board and underwater board.
- As a player, I want to see if I hit anything underwater with the space laser so I know my shot results and progress in the game.
- As a player, I want to be able to see both the surface and underwater board so I know my progress in the game and not shoot in the same place.
- As a player, I want to have a Bomber that can be placed in the air.
- As a player, I want to have the option of adding an extra ship to my fleet by using user input to pick my choice.
- As a player, I want to play against an AI that takes its shots based on previous knowledge so I can have a smart opponent to play against.
- As a player, I want to have a new weapon based on my extra ship choice to be able to customize my gameplay.
- As a player, I should have a chance at getting bad luck on a turn in order to lose a turn.
- As a player, I should have a chance at getting good luck on a turn in order to get an extra shot at my opponent’s game pieces.
- As a programmer, I want to split up Game Management, so that all responsibility is not on one function.
- As a programmer, I want to be able to present a clean UI to players so that they understand the status of the game.

Architecture and Design

At the fundamental level, our application is a fully playable terminal game. Our code is equipped with object-oriented programming fundamentals in order to create a Battleship game that can delegate tasks and handle different types of instances of objects to communicate together. Within our code, we created a selected amount of major components that define the game and are split within their own directories. Throughout the code, we have implemented several design patterns that have allowed us to create readable code, make our system be loosely coupled, and adaptable to future iterations. As well, our code has test coverage and has been through many iterations of refactoring to improve our system and design. Furthermore, as a team, we developed a consistent coding style/requirements that have allowed our code to look consistent throughout. When the code has completed its build and executed, within the terminal we have created a readable UI that allows the user to play the Battleship game with consistent feedback from their decisions and the progress of the game and system.

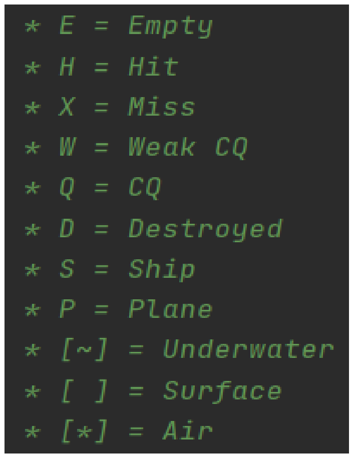
Directories

We organized our directions into our five major components: Board, Game, GamePiece, GameProbabilities, and Player. Within the Board directory, we have our main board class, a subdirectory that holds our interfaces, as well as another subdirectory that holds our behaviors that implements these interfaces. The Game directory consists of six classes that work together to create our game (GameManagement with the help of our factories) and then run the game to be playable (Main). The GamePiece directory contains our abstract class GamePiece and our current game pieces that extend from the abstract class. It also has a subdirectory that holds one interface which implements the observer pattern. The GameProbabilities directory holds four classes and a subdirectory that has one interface and a subdirectory of commands that implement the command interface. Lastly, the Player directory contains our abstract class Player, our current two players (user, computer), and a subdirectory that holds our interfaces as well as another subdirectory that holds our behaviors that implement these interfaces. This organization helped us break down what would have been a very large UML into five subsections that describe the major components of our game.

User Interface

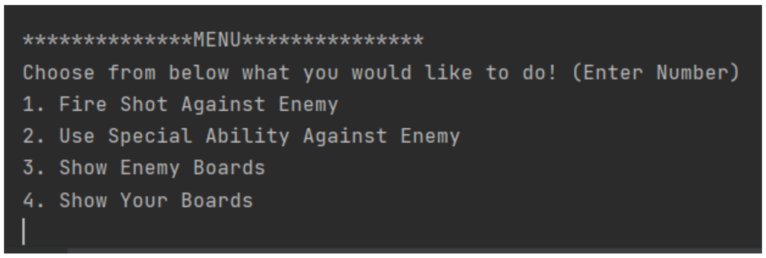
Our user interface displays our game via text-based output. The user can interact with the game by typing text-based commands. When the game first starts, our program gives the user three game piece options to choose from to add to their fleet. After the user enters which game piece they want, they are then prompted to give coordinates on where they would like to place their game pieces and if they want to place their game pieces vertically or horizontally. Once all their game pieces have been placed, the user's turn begins, and we display a menu for them. This menu (Turn Menu Display) allows the user to have control over how they want to play their game and it gives the game a more organized structure. As well, our game asks the user where they want to shoot, what abilities they want to use, and which board they want to see. We also labeled each board that gets displayed so that the user can distinguish between their boards and their opponent's boards. Throughout our program, we ensure that all input given by the user is valid so that our game runs smoothly and no errors are encountered.

Legend for board display



```
* E = Empty
* H = Hit
* X = Miss
* W = Weak CQ
* Q = CQ
* D = Destroyed
* S = Ship
* P = Plane
* [~] = Underwater
* [ ] = Surface
* [*] = Air
```

Turn Menu Display



```
**************MENU***************
Choose from below what you would like to do! (Enter Number)
1. Fire Shot Against Enemy
2. Use Special Ability Against Enemy
3. Show Enemy Boards
4. Show Your Boards
|
```

Design Patterns

We made use of multiple patterns throughout the development of this project. We focused on four main patterns that helped our code achieve the standards of extendable rather than modifiable, highly cohesive, and loosely coupled.

Factory Pattern

The Factory Pattern was used in our project for both the creation of boards and each player's game pieces. Inside of GameManagement, we initialized each factory to prepare for the creation of both the boards and the game pieces. The Factory Pattern allows us to have a more flexible type of object creation so that our program does not need to know what needs to be created until runtime. For example, our FleetFactory is used once we know which pieces a player chooses for the duration of the game. This way, we don't need to have a long list of new creations and if-statements where we need to know the object before runtime, but rather at runtime, our GameManagement class can decide exactly what needs to be instantiated in this instance of the game.

For the Board Factory, we implemented a factory that takes in z-values and creates an array of boards that the players will have throughout the game. This would allow us to expand our game to allow multiple depths to the boards, underwater and air, without having to modify our existing code significantly. With this, in the future, if a player wanted no air game pieces but wanted multiple underwater game pieces at different depths, the factory would allow the GameManagement class to decide what boards needed to be instantiated.

Strategy Pattern

The Strategy Pattern was used in both the implementation of the Board and the Player classes in this project. For Player, we had two instances of the Strategy Pattern, the ShotBehavior, and the PlacementBehavior. The ShotBehavior allows the behavior of a player's shot to dynamically be changed at runtime, so the Player can use multiple different kinds of weapons throughout the game. Similarly, the PlacementBehavior is implemented to have a distinction between placement that requires input and placement that is random. At runtime, this behavior is assigned to each player for the initialization of the game.

For the Board, we had two instances of the Strategy Pattern, the CreateCooridnatesBehavior, and the ShowBehavior. The CreateCoordinateBehavior allows the board to dynamically change the way it calculates the proper coordinates for a ship at runtime. The ShowBehavior allows the board to dynamically change the way it shows its elements at runtime, depending on what is necessary information needed at the time.

By using the Strategy Pattern, this allows our code to defer the decision of which of the algorithms, which are encapsulated separately, to use until runtime, allowing our code to be more extendible, flexible, and reusable.

Observer Pattern

The Observer Pattern is used between our Board class and our GamePiece class in this project. In this case, the Board is the subject and the GamePiece objects are the observers. In Battleship, the GamePiece does not need to know where it is at. The only information that the GamePiece should know is what kind of piece it is, and whether it is still operating or has been destroyed. Giving the GamePiece the information of its whereabouts can lead to tight coupling and unnecessary information sharing between classes.

The Observer Pattern is the solution to this problem. Now, the board is aware of what coordinates are occupied, which makes sense as the board is composed of those same coordinates. Each piece only knows its type, health, and if it is still operating. The board has a list of pieces, all of whom are “subscribed” to the board, who wait for updates on their status. When the board detects that a coordinate with a ship on it has been hit, it sends an update to the piece that has been hit, telling it that it has now lost health or that it is now destroyed. Once a ship is destroyed, it no longer receives updates from the board.

With this, the GamePiece class does not need to know its coordinates to be able to update its information on operations and health. The GamePiece waits for updates from the Board, and no unnecessary information is shared.

Command Pattern

The Command Pattern is used within our GameProbabilities class and allows communication to the Player class. We are using this pattern to handle calls to different classes to delegate the task of knowing when to call our probability calculators methods so that the features of bad luck and good luck are to be enabled.

In the GameProbabilites class, we have a class that creates a ProbabilitiesController that handles creating a new controller that has defined methods that are provided to the controller to call and execute when called upon. When this controller is created, we provide this object to the Player to call commands and execute them during a call to a Player's turn. The controller is set to handle two different commands from two different classes: the BadLuckCalc and the GoodLuckCalc classes. The controller takes the defined methods from each class that calculates a player's bad/good luck for the current round and defines them to different slots in the controller. When a slot is called, this then executes the method that is defined to that slot. After a slot is called, the controller provides feedback on whether the command was successful or not to tell the player if they need to perform other actions.

Coding Style

At the beginning of this project, our team had agreed upon a coding standard to follow throughout development. The standards were:

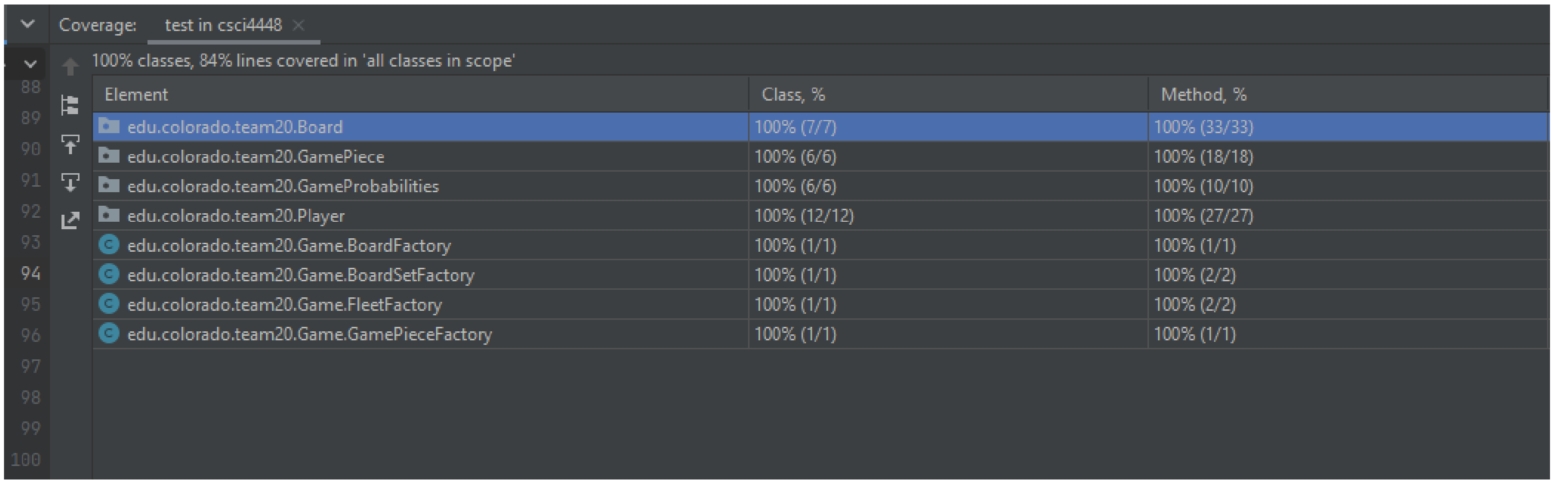
- Proper Indentation
- Comments explaining functionality
- Commits to proper branches for review
- Detailed git commit messages
- Consistent function/variable naming and format

We did code checkups during every meeting and work session to make sure our code was up to the agreed-upon standards. Along with that, we used the automatic styling that IntelliJ provides. This includes automatic indentation where necessary, the Analyze tool that checks for dead code and commented out code that is no longer being used, and automatic alignment and wrapping for conditional statements. Each git commit had a comment that described the general tasks performed during that stage of development.

Testing

Throughout this project, we have followed Test Driven Development, meaning before we can fully develop classes, functions, and other implementations, we must convert them to test cases that test against all possible scenarios written in the code. Throughout our process, we have been able to stay mostly consistent with 100% class and method coverage with each new milestone. When creating new classes, we first took the approach of writing a test that was intended to fail. Then, we write code that would pass the test. We repeated this process until we had a class/function that is fully covered by tests and successfully passing them consistently. For the project, we used the JUnit extension available in IntelliJ that allows us to view the test coverage of all of our code. One of the struggles we faced was getting line coverage up to 100%, as we used multiple conditions based on random events, so not every line of code was executed every time, but we made sure to cover every method to make sure the general function was working. Below are screenshots of the test coverage data that show that we have 100% test coverage in the classes and methods upon finalization of this project. To run your test coverage in IntelliJ with our code, refer to the Appendix for the instructions and necessary setup to achieve correct results and data.

Test Coverage Data



100% classes, 84% lines covered in 'all classes in scope'

Element	Class, %	Method, %
edu.colorado.team20.Board	100% (7/7)	100% (33/33)
edu.colorado.team20.GamePiece	100% (6/6)	100% (18/18)
edu.colorado.team20.GameProbabilities	100% (6/6)	100% (10/10)
edu.colorado.team20.Player	100% (12/12)	100% (27/27)
edu.colorado.team20.Game.BoardFactory	100% (1/1)	100% (1/1)
edu.colorado.team20.Game.BoardSetFactory	100% (1/1)	100% (2/2)
edu.colorado.team20.Game.FleetFactory	100% (1/1)	100% (2/2)
edu.colorado.team20.Game.GamePieceFactory	100% (1/1)	100% (1/1)

Examples of Refactoring

Throughout our development of this project, we were able to learn and apply many different refactorings that allowed our code to be highly cohesive, easier to read, and rid of all dead, duplicated, and unnecessary code and comments.

To view a complete list of all of the major refactorings done throughout our development as well as screenshots to document the refactorings, please refer to the project GitHub, in the RefactoringExamples folder:
https://github.com/sapa3451/4448Team20/tree/main/RefactoringExamples

Lazy Classes

A lazy class refers to one that isn't doing enough to pay for itself. In our case, we had an abstract Board class that has 2 subclasses, UnderwaterBoard and AirBoard. This refactoring was a significant one, allowing us to get rid of these two subclasses that had little to no responsibilities. The responsibilities they did have, were duplicate algorithms that could be refactored into one. They were only dealing with the changes that were needed for the different depths they had. We got rid of the extended classes and changed the Board class to be a single class. Instead, Board now has a z-value that represents the depth and height of the board, allowing for a simplification and easier instantiation of the boards without the need for subclasses that have no major responsibilities and duplicate code.

Game Management

A significant refactoring we were able to perform was in the GameManagement class. Within the class, we had a single function handling all of the responsibility, BeginGame(). This function did everything from placing ships, initializing variables, taking turns, performing special shots, etc. We found that this was way too much responsibility on a single function and made the flow of our game very hard to read and understand. We split up the function into multiple functions, allowing for better readability and flow. We now have a constructor that initializes everything the game needs, an initializeGame function that places the ships, a Sonar and SpecialShot function to handle those abilities, and EndGame function that checks to make sure the game is not over, and a much smaller BeginGame function, that handles the shots and turns.

Duplicated Code/Middle Man

Dealing with our interfaces, we found that we had redundant interfaces dealing with our boards that had almost identical behavior causing hundreds of lines of duplicated code. Our show behaviors were initially made to differentiate between different elevations of the boards; air, surface, and underwater. We found that once we had refactored the Board class to become a single class with a z value, these show behaviors became redundant and unnecessary. We found a way to refactor our 6 show behavior classes down to 2 show behavior classes, one for the player and one for the computer, using the board's z value to now implement the needed differences. Also, we found that once we had a single Board class, we could once again reduce a behavior interface, our MarkBehavior, to now just be a function within the Board class itself. This was a middle man problem, as the Board was delegating responsibility to an interface that only had a single behavior. The interface would delegate the responsibility to the single behavior that never changes. We removed the MarkBehavior interface as a whole and reduced it to a function within Board itself, cutting out the middleman.

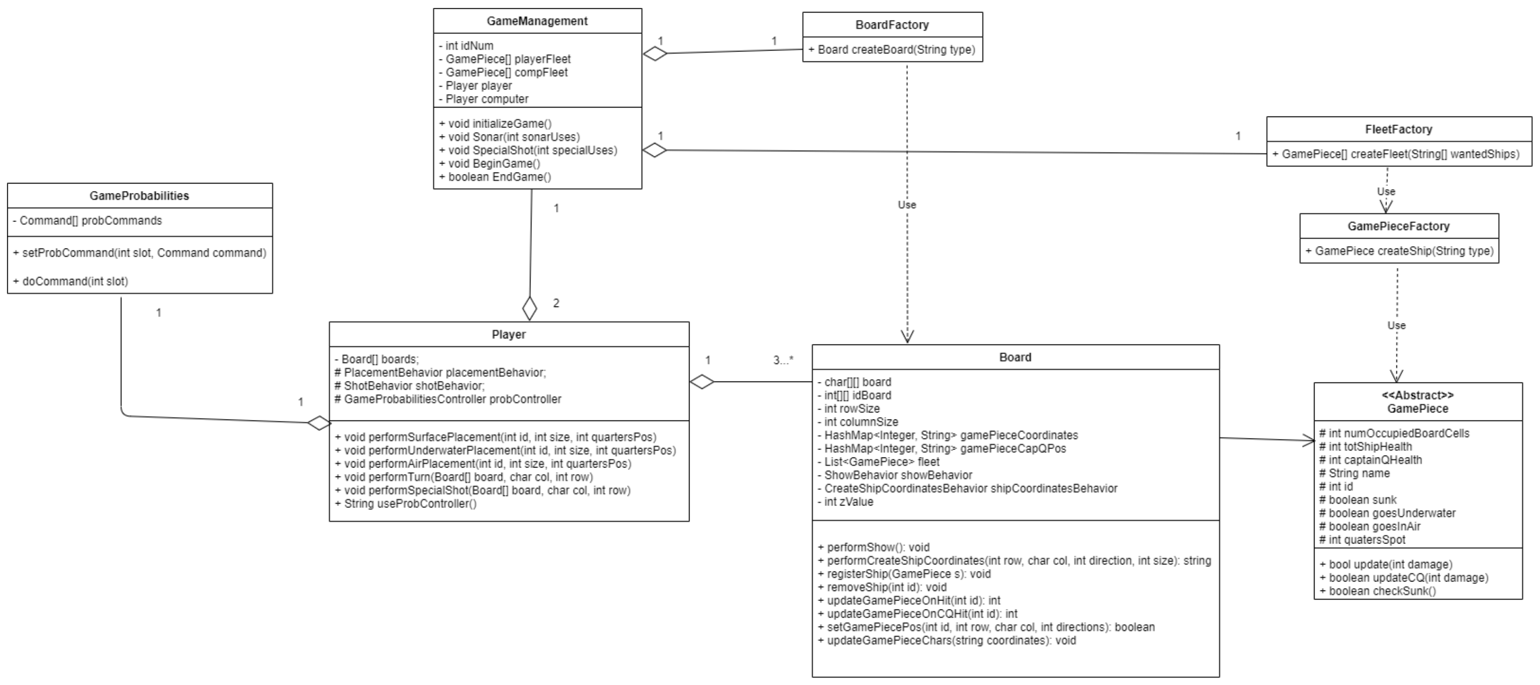
UML Diagrams

We have included screenshots of each UML diagram. If the images are hard to read, please refer to the link: https://drive.google.com/file/d/1E47LrGTu2Zf2LexCN_m9DP4uhBylnzPq/view?usp=sharing

Class Diagrams

Below, we have split up the class diagrams to make each piece of our program easier to understand.

Main Classes



GameManagement
- int idNum
- GamePiece[] playerFleet
- GamePiece[] compFleet
- Player player
- Player computer
+ void initializeGame()
+ void Sonar(int sonarUses)
+ void SpecialShot(int specialUses)
+ void BeginGame()
+ boolean EndGame()
BoardFactory
+ Board createBoard(String type)
FleetFactory
+ GamePiece[] createFleet(String[] wantedShips)
GamePieceFactory
+ GamePiece createShip(String type)
GameProbabilities
- Command[] probCommands
+ setProbCommand(int slot, Command command)
+ doCommand(int slot)
Player
- Board[] boards;
# PlacementBehavior placementBehavior;
# ShotBehavior shotBehavior;
# GameProbabilitiesController probController
+ void performSurfacePlacement(int id, int size, int quartersPos)
+ void performUnderwaterPlacement(int id, int size, int quartersPos)
+ void performAirPlacement(int id, int size, int quartersPos)
+ void performTurn(Board[] board, char col, int row)
+ void performSpecialShot(Board[] board, char col, int row)
+ String useProbController()
Board
- char[][] board
- int[][] idBoard
- int rowSize
- int columnSize
- HashMap<Integer, String> gamePieceCoordinates
- HashMap<Integer, String> gamePieceCapQPos
- List<GamePiece> fleet
- ShowBehavior showBehavior
- CreateShipCoordinatesBehavior shipCoordinatesBehavior
- int zValue
+ performShow(): void
+ performCreateShipCoordinates(int row, char col, int direction, int size): string
+ registerShip(GamePiece s): void
+ removeShip(int id): void
+ updateGamePieceOnHit(int id): int
+ updateGamePieceOnCQHit(int id): int
+ setGamePiecePos(int id, int row, char col, int directions): boolean
+ updateGamePieceChars(string coordinates): void
<<Abstract>>
GamePiece
# int numOccupiedBoardCells
# int totShipHealth
# int captainQHealth
# String name
# int id
# boolean sunk
# boolean goesUnderwater
# boolean goesInAir
# int quatersSpot
+ bool update(int damage)
+ boolean updateCQ(int damage)
+ boolean checkSunk()
Use
Use
Use
1
1
1
1
1
2
1
1
1
3...*

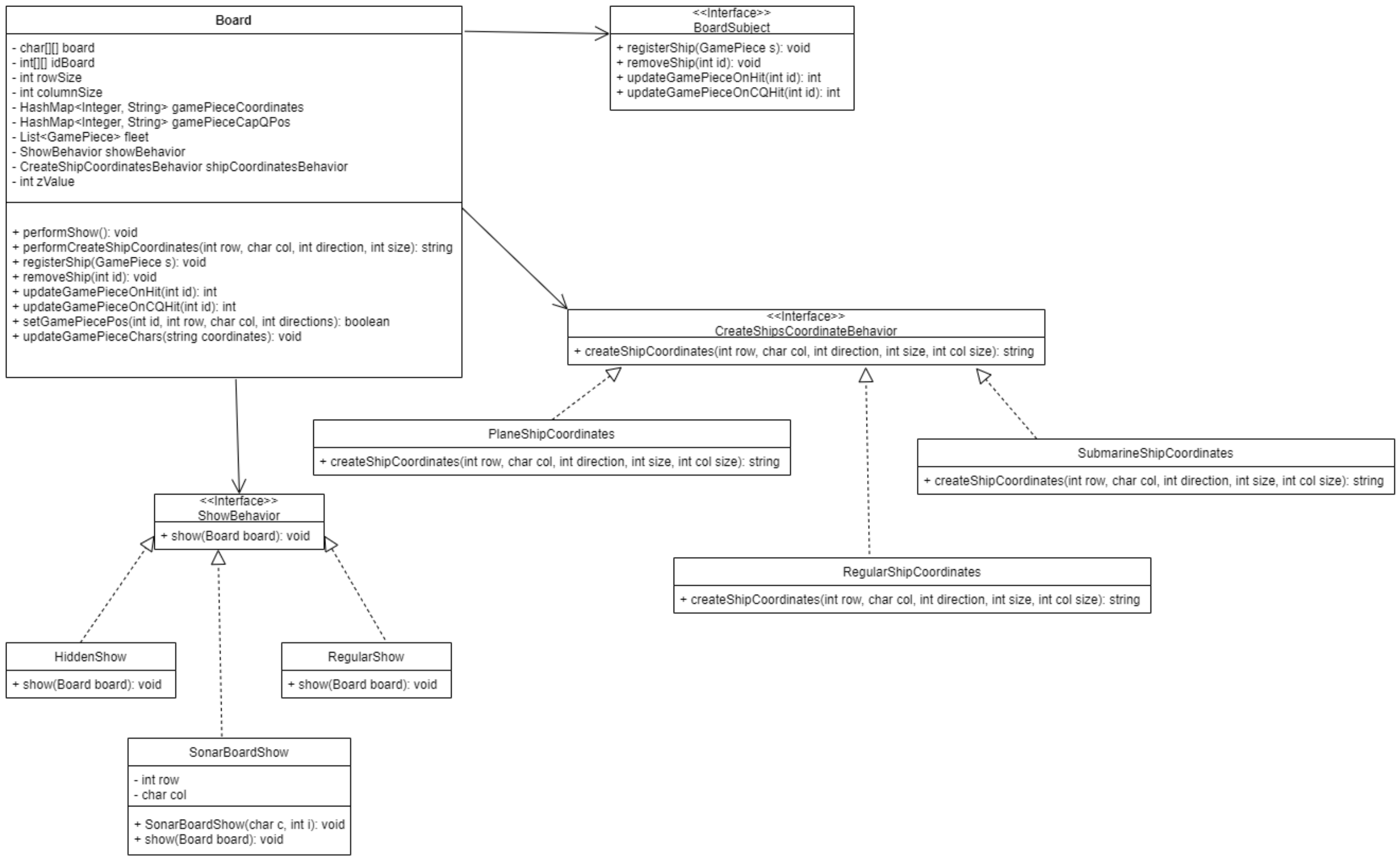
Comments:

These are the main classes involved in the process of the game and the initialization.

GameManagement has 2 players and 1 of each Factory, which creates the boards and game pieces.

Each Player has 3 boards and 1 controller for the GameProbailities.

Board Class


Board
- char[][] board
- int[][] idBoard
- int rowSize
- int columnSize
- HashMap<Integer, String> gamePieceCoordinates
- HashMap<Integer, String> gamePieceCapQPos
- List<GamePiece> fleet
- ShowBehavior showBehavior
- CreateShipCoordinatesBehavior shipCoordinatesBehavior
- int zValue
+ performShow(): void
+ performCreateShipCoordinates(int row, char col, int direction, int size): string
+ registerShip(GamePiece s): void
+ removeShip(int id): void
+ updateGamePieceOnHit(int id): int
+ updateGamePieceOnCQHit(int id): int
+ setGamePiecePos(int id, int row, char col, int directions): boolean
+ updateGamePieceChars(string coordinates): void
<<Interface>>
BoardSubject
+ registerShip(GamePiece s): void
+ removeShip(int id): void
+ updateGamePieceOnHit(int id): int
+ updateGamePieceOnCQHit(int id): int
<<Interface>>
CreateShipsCoordinateBehavior
+ createShipCoordinates(int row, char col, int direction, int size, int col size): string
PlaneShipCoordinates
+ createShipCoordinates(int row, char col, int direction, int size, int col size): string
SubmarineShipCoordinates
+ createShipCoordinates(int row, char col, int direction, int size, int col size): string
RegularShipCoordinates
+ createShipCoordinates(int row, char col, int direction, int size, int col size): string
<<Interface>>
ShowBehavior
+ show(Board board): void
HiddenShow
+ show(Board board): void
RegularShow
+ show(Board board): void
SonarBoardShow
- int row
- char col
+ SonarBoardShow(char c, int i): void
+ show(Board board): void

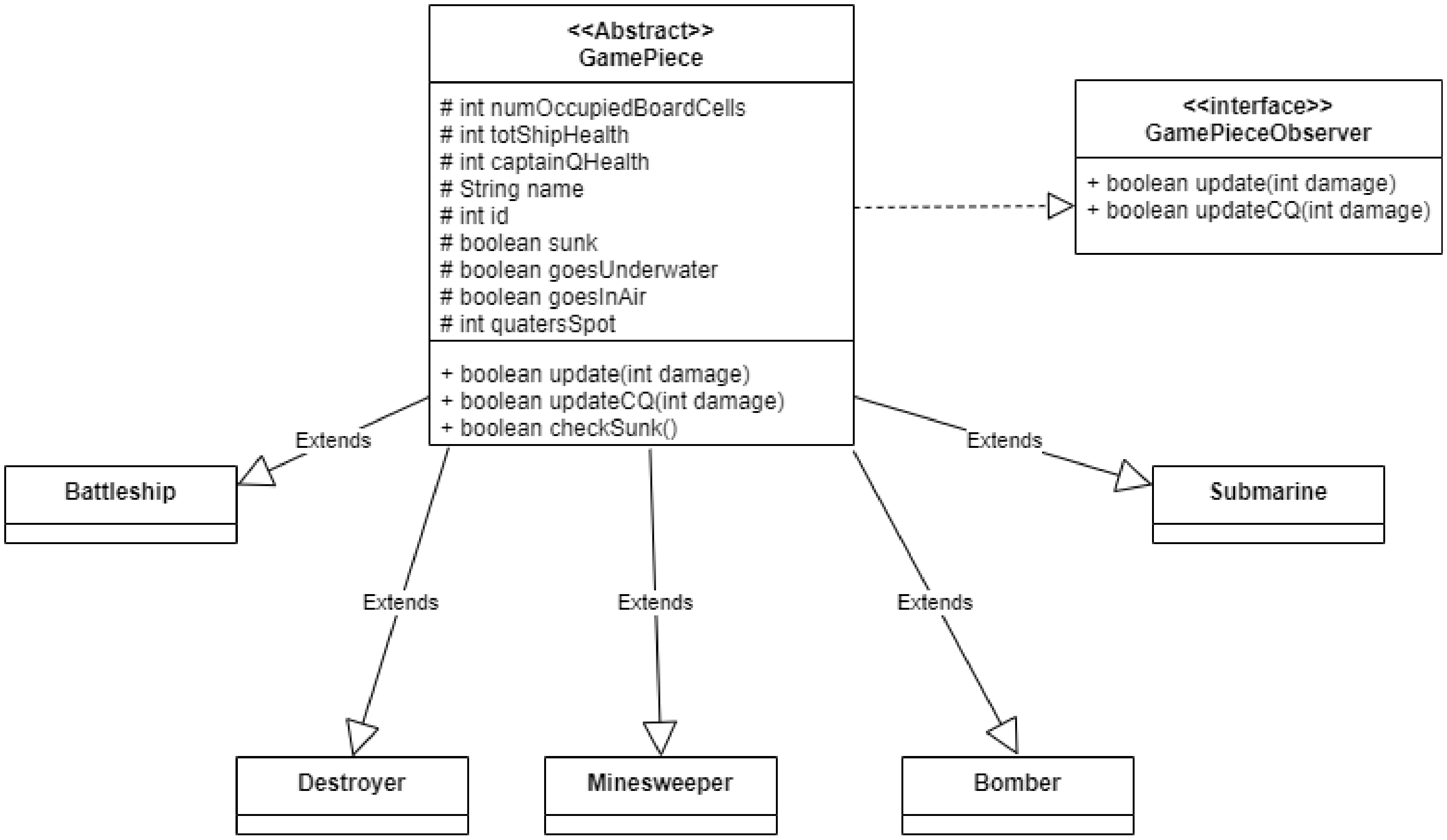
Comments:

This is the class diagram involving the main responsibilities and connections of the Board.

Board implements 1 interface, BoardSubject.

Board has 2 interface behavior objects, CreateCoordinateBehavior and Show Behavior, which handle the behavior of displaying the Board to the user and create coordinates for specific ships.

GamePiece Class

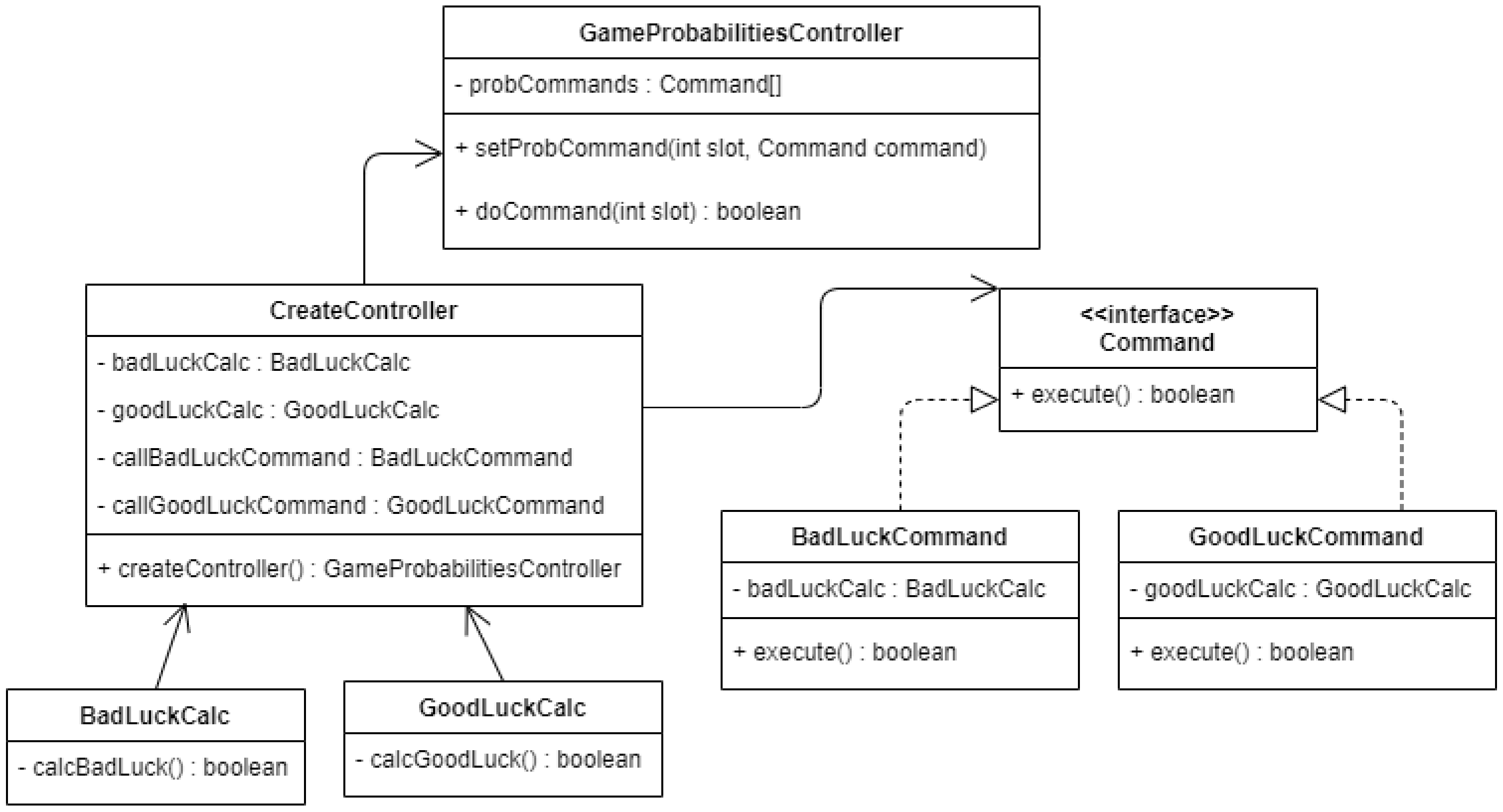


Comments:

This class diagram shows the main extensions and interface usage of GamePiece.

GamePiece currently has 6 subclasses: Battleship, Destroyer, Minesweeper, Bomber, and Submarine. This is open to extension to add a new piece easily, just by creating another extension.

GameProbabilites Class

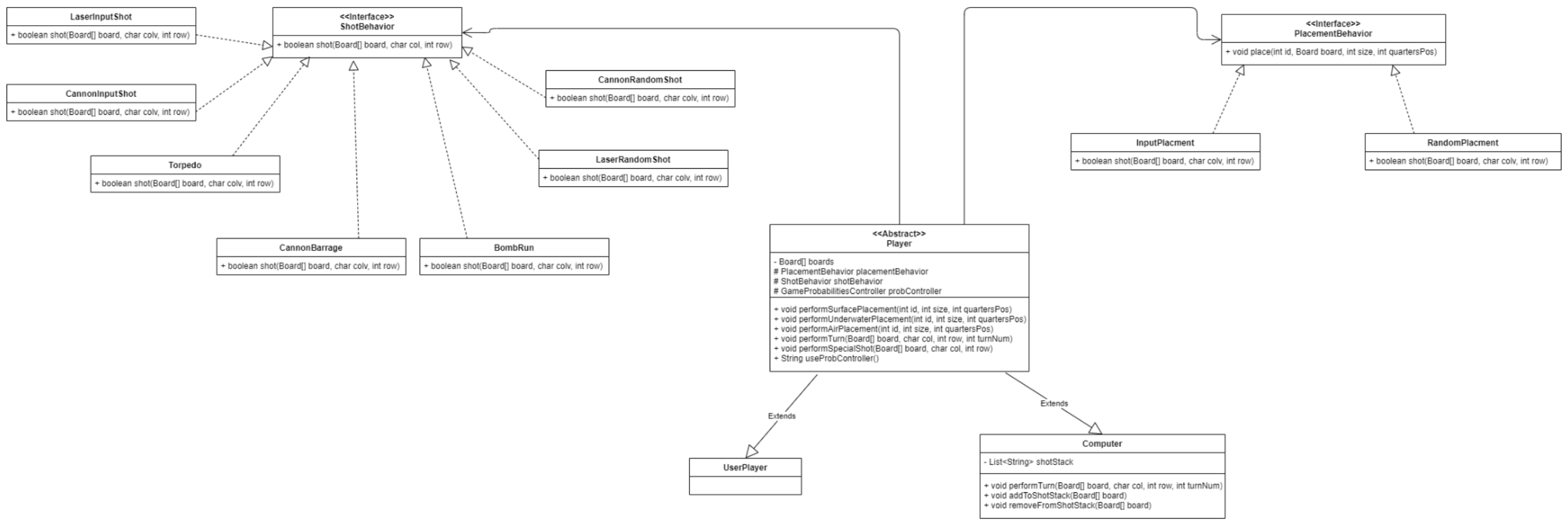

GameProbabilitiesController
- probCommands : Command[]
+ setProbCommand(int slot, Command command)
+ doCommand(int slot) : boolean
CreateController
- badLuckCalc : BadLuckCalc
- goodLuckCalc : GoodLuckCalc
- callBadLuckCommand : BadLuckCommand
- callGoodLuckCommand : GoodLuckCommand
+ createController() : GameProbabilitiesController
<<interface>> Command
+ execute() : boolean
BadLuckCommand
- badLuckCalc : BadLuckCalc
+ execute() : boolean
GoodLuckCommand
- goodLuckCalc : GoodLuckCalc
+ execute() : boolean
BadLuckCalc
- calcBadLuck() : boolean
GoodLuckCalc
- calcGoodLuck() : boolean

Comments:

This class diagram shows how the Command Pattern is used to delegate calls of different methods.

We have created a controller that holds commands to calling two different methods from different object types (in our case, two different probability calculators). We allow the UserPlayer to obtain an object of a controller that holds these commands that will get called each of the user's turns. This controller will execute the methods to calculate good or bad luck and provide feedback on which commands were successful for the Player to determine how their turn is to be handled (miss or receive another turn).

Player Class


LaserInputShot
+ boolean shot(Board[] board, char colv, int row)
<<Interface>>
ShotBehavior
+ boolean shot(Board[] board, char col, int row)
CannonInputShot
+ boolean shot(Board[] board, char colv, int row)
CannonRandomShot
+ boolean shot(Board[] board, char colv, int row)
Torpedo
+ boolean shot(Board[] board, char colv, int row)
LaserRandomShot
+ boolean shot(Board[] board, char colv, int row)
CannonBarrage
+ boolean shot(Board[] board, char colv, int row)
BombRun
+ boolean shot(Board[] board, char colv, int row)
<<Interface>>
PlacementBehavior
+ void place(int id, Board board, int size, int quartersPos)
InputPlacment
+ boolean shot(Board[] board, char colv, int row)
RandomPlacment
+ boolean shot(Board[] board, char colv, int row)
<<Abstract>>
Player
- Board[] boards
# PlacementBehavior placementBehavior
# ShotBehavior shotBehavior
# GameProbabilitiesController probController
+ void performSurfacePlacement(int id, int size, int quartersPos)
+ void performUnderwaterPlacement(int id, int size, int quartersPos)
+ void performAirPlacement(int id, int size, int quartersPos)
+ void performTurn(Board[] board, char col, int row, int turnNum)
+ void performSpecialShot(Board[] board, char col, int row)
+ String useProbController()
Extends
Extends
UserPlayer
Computer
- List<String> shotStack
+ void performTurn(Board[] board, char col, int row, int turnNum)
+ void addToShotStack(Board[] board)
+ void removeFromShotStack(Board[] board)

Comments:

This is the class diagram for the Player and its extensions and interfaces.

Player has 2 subclasses: ComputerPlayer and UserPlayer (UserPlayer being a human player).

Player has 2 interface behavior objects, PlacementBehavior and ShotBehavior.

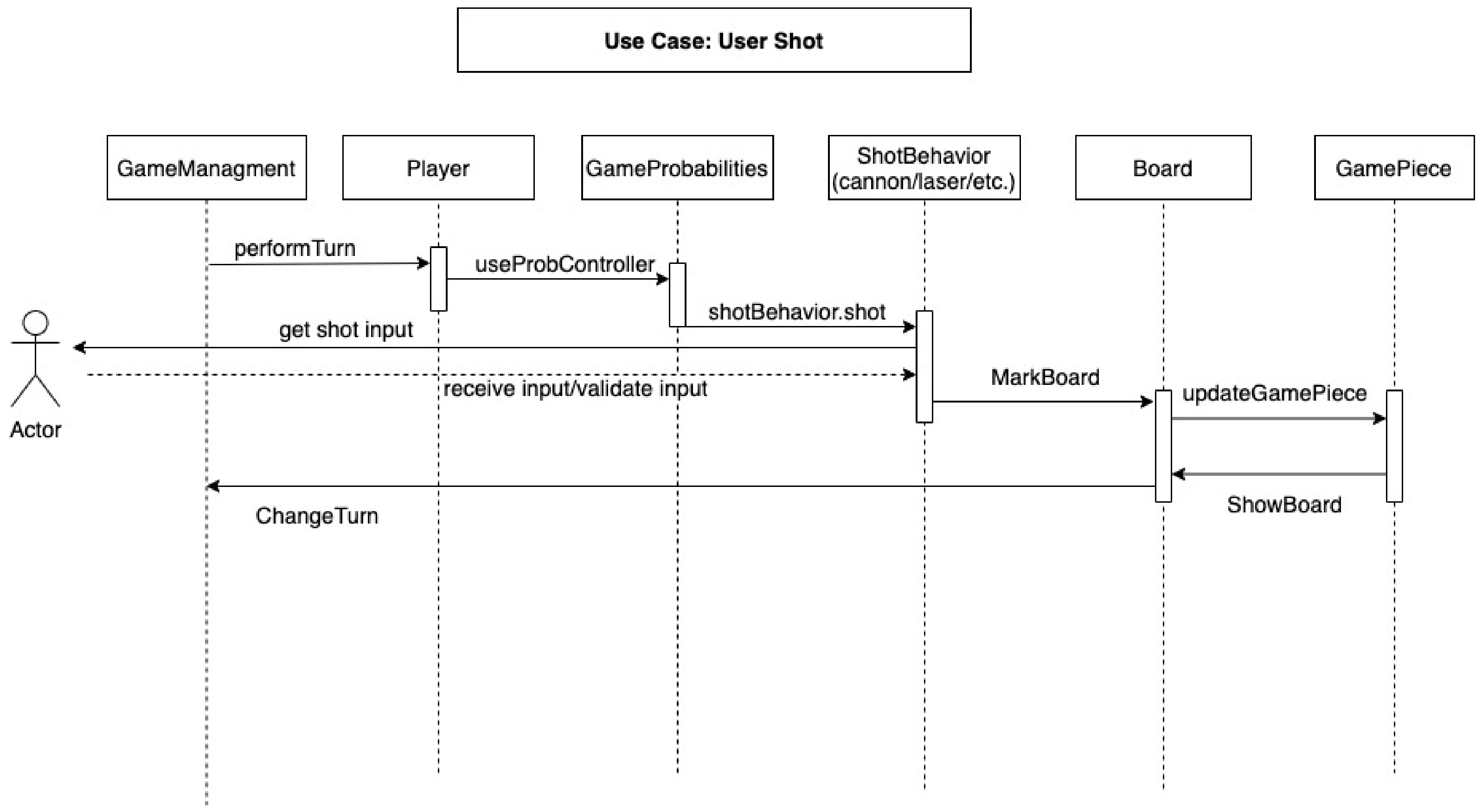
ShotBehavior handles input shots for humans and random shots for the computer and any special abilities.

PlacementBehavior handles input for human placement and random placement for computers.

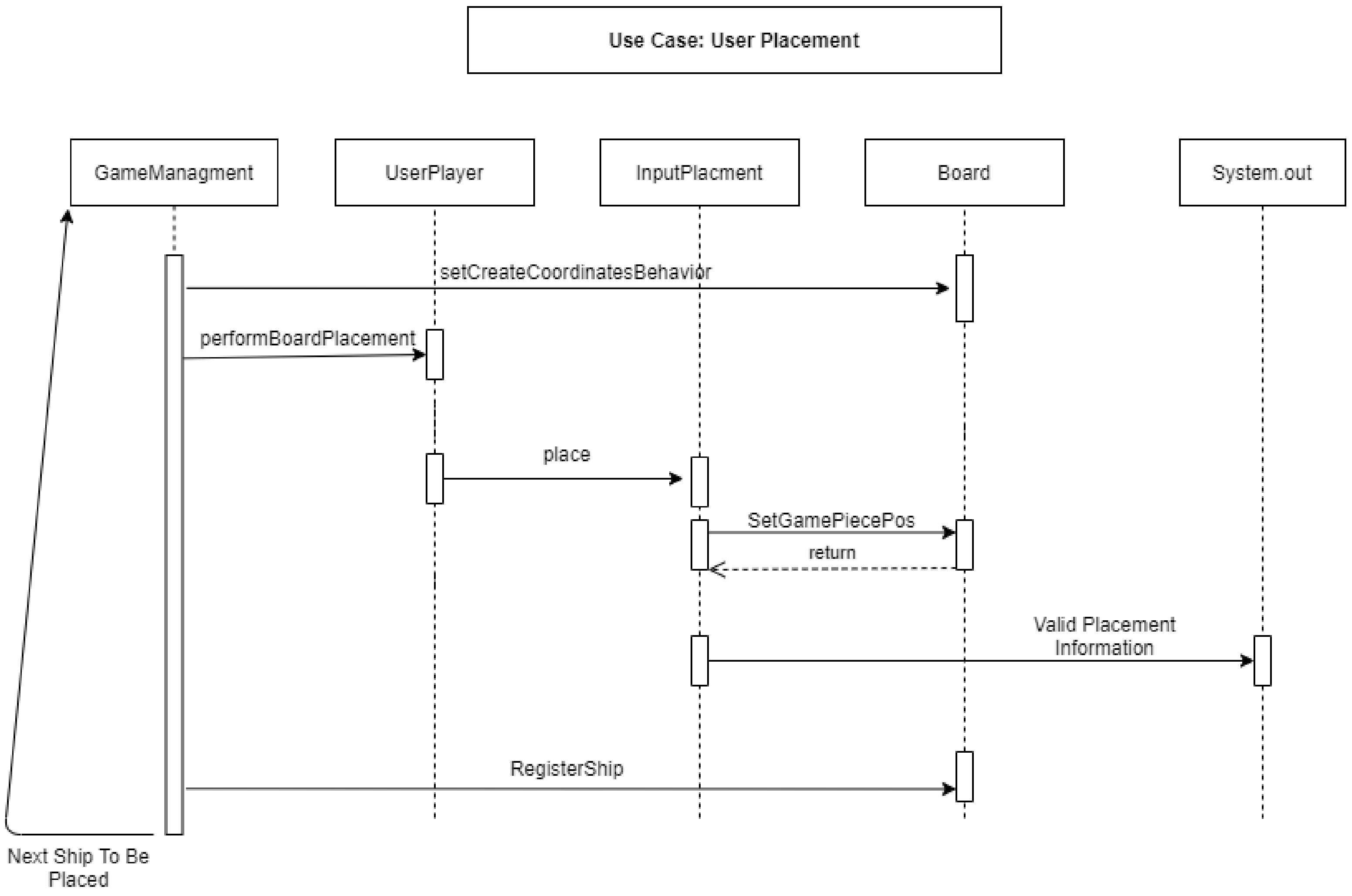
The ComputerPlayer overrides performTurn to implement a smart AI that has knowledge of their hits on enemy pieces.

Sequence Diagrams

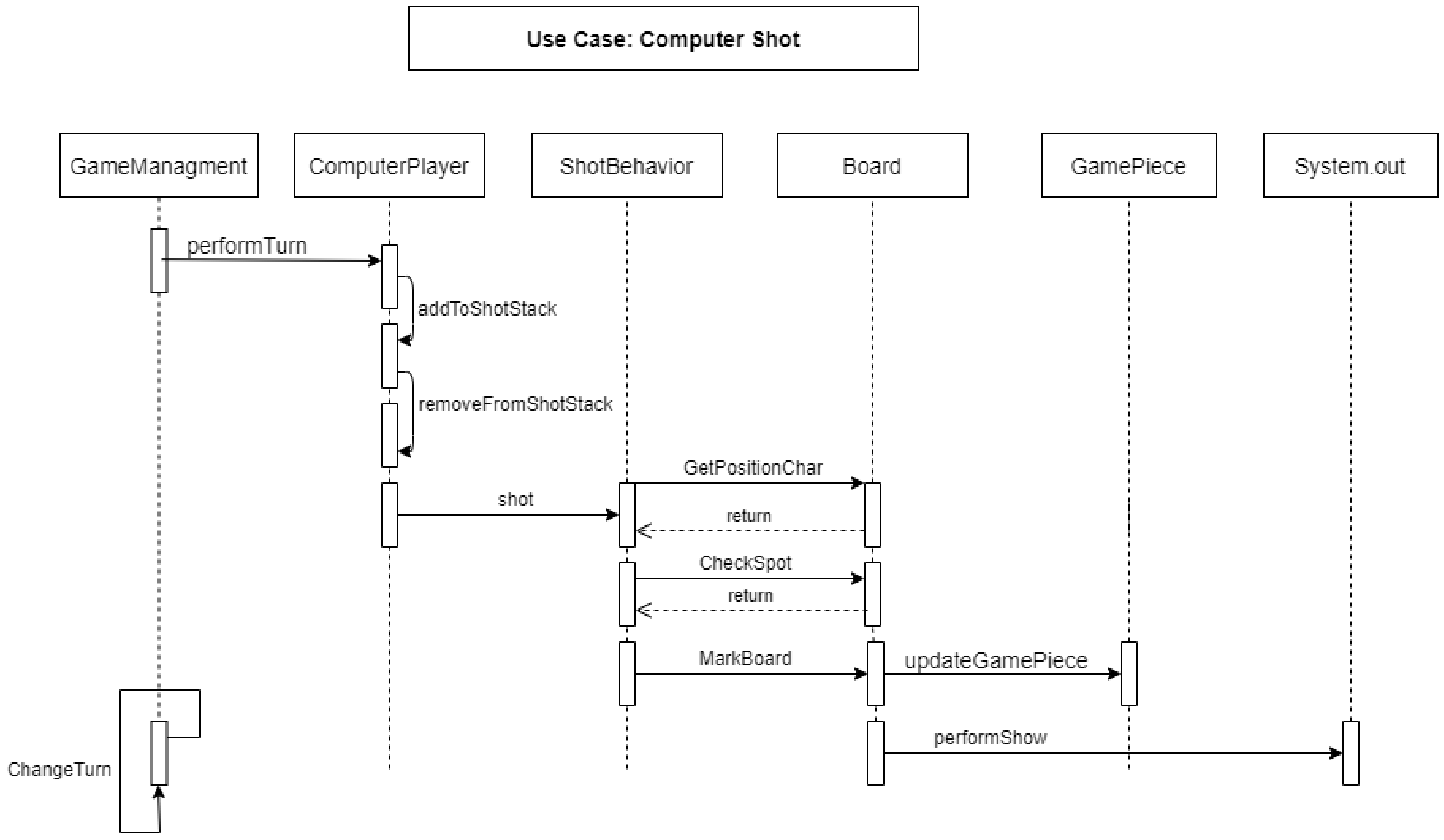
User Shot



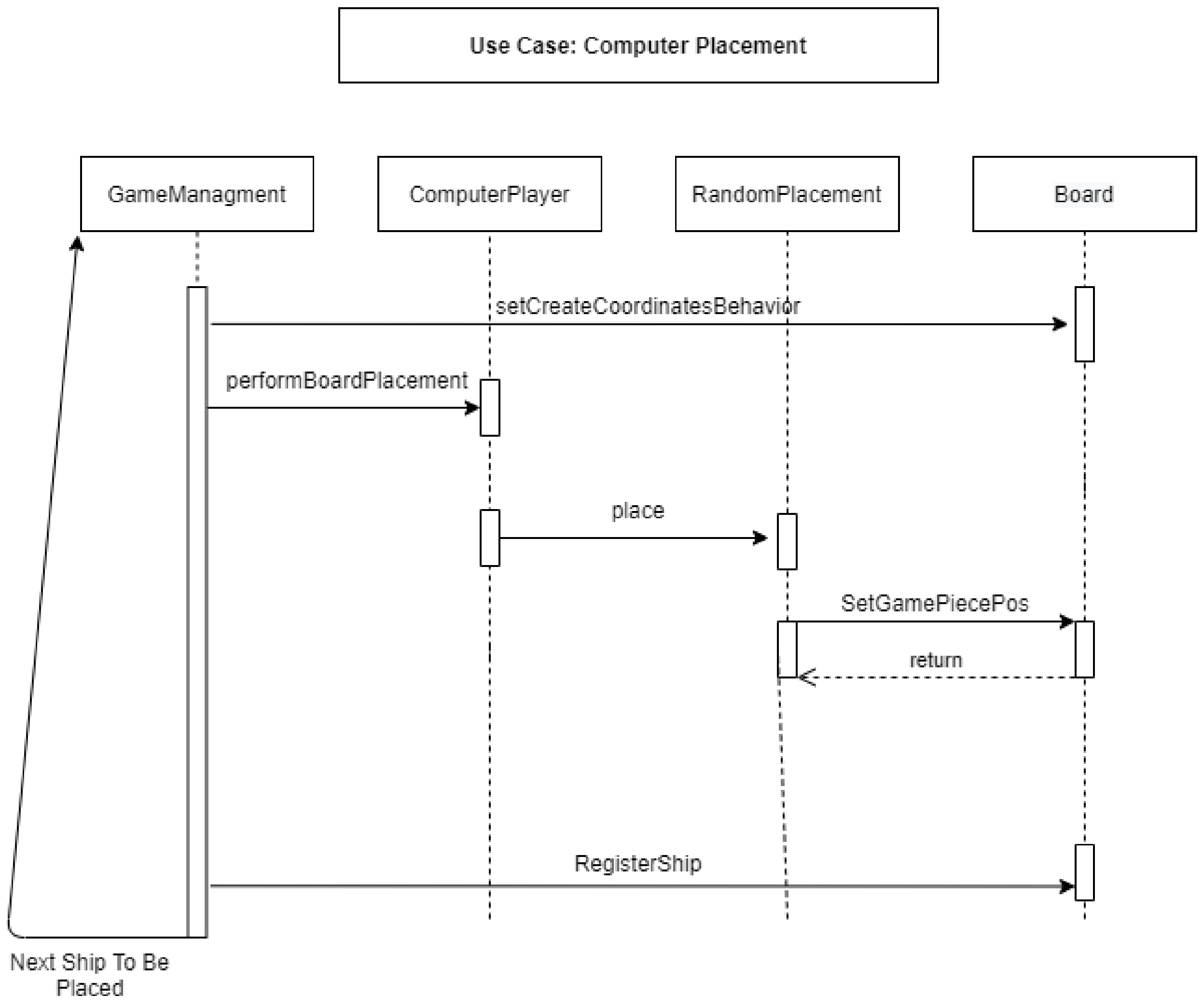
User Placement



Computer Shot



Computer Placement



Personal Reflections

Vanessa Reyes

This project was a great experience. I learned a lot of new skills such as Java, OOD, design patterns, software development, refactoring, and much more. I had no experience with Java beforehand so this project was a great start to learning about programming in Java as well as developing OOD principles. As well, I enjoyed the challenge that this project provided as our team and I had to learn how to keep up with changing requirements and consistently redesigning our code to be able to handle future features and requirements. I was able to gain new knowledge on how to code for the future and to make sure that the code I produce can be dynamic and handle change. Likewise, I am very grateful that I learned what refactoring is and how powerful it can be for programming and making the design of code efficient and easier to understand. Moreover, I felt that this project gave me a sense of how programming in industry could be with the use of XP and allowed me to gain experience with collaborating with an amazing team. This project was a great learning experience for me and I feel that I have become a better programmer.

Joseph Petrafeso-Crosby

While I think it is fair to say that this semester has been a challenge for everyone, I think doing this project has been rewarding every step of the way. Like other people in my group, I had no experience with Java or IntelliJ. I also had almost no experience with OOP (I just knew class objects but nothing about inheritance or polymorphism). I also really had no experience with VCS's. So having to learn these things certainly proved a challenge at first, and aside from VCS learning these things kind of just felt arbitrary. However, as we started getting into the design patterns unit of the class it became much more apparent as to why these things are so important for software development. Once I realized this value, I enjoyed starting each new milestone out by thinking about how we could apply new OO design principles or patterns we had learned about. I think while I was initially annoyed by the constantly changing requirements for each milestone I tried to embrace the XP approach to coding by just adjusting to the fact these changes were a given. Not only did this help us refactor and rethink how our code could better match up with OO design principles, I think this helped me to be better prepared for constantly changing requirements that are going to be natural in my future work environments.

Priscila Trevino

The experience I gained from this project will benefit me once I enter the industry because it made me grow as a programmer, teammate and I was able to learn new skills. I did not have any prior experience coding in Java or working with IntelliJ, so this project was a great way to learn the language, a new IDE, and how to apply different design patterns into our code. Learning about the different types of design patterns was valuable because it showed me that whenever we needed to change or add to our project, these design patterns helped our code be open to extension. It was great learning about refactoring because it made me realize that it helps improve our code by making it more readable, understandable, and clean. This project was a good refresher when it came to using GitHub because I was able to practice handling merge conflicts when pushing my work. I also learned how useful it is to use the Wiki feature on Github as it helped my teammates and I stay organized and focused on the project. I appreciate the experience that I got from this project and I am grateful for the hardworking team I was paired with.

Salvatore Pacifico

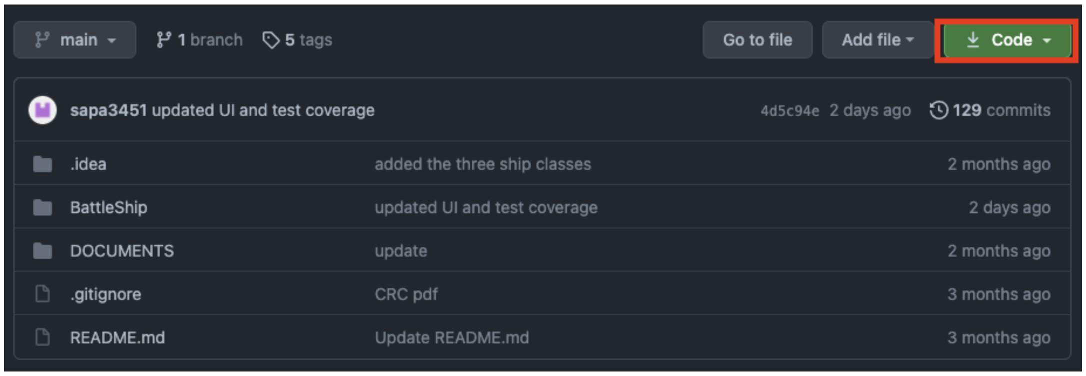
Working on this project has allowed me to gain fantastic skills when it comes to team communication, working with GitHub, design patterns, proper object-oriented design, design principles, refactoring, and more. Having no experience in Java beforehand, this project was a fantastic introduction to the language and really helped me understand the different design patterns we learned. It was nice to get more in-depth with IntelliJ too. Only previously using it in a few classes, being able to find the new tools such as the JUnit extension, the tool for analyzing code, and the automatic code styling it has was a great introduction into how powerful certain IDEs can be. Many of us were out of our comfort zone when using TDD and XP, as it was all new to us. I'm glad for the experience though, as TDD is pretty big in the industry and software engineering world. I was grateful to be paired with a team that worked hard and had great ideas for the project. A very rewarding project overall!

Appendix

Instructions for Installation

This project was based in IntelliJ. With that being said, these installation instructions are specific to IntelliJ.

1. Go to the Git Repository for Team20 (Repository link)
    1.1. When in the Git Repository, click on "Code"

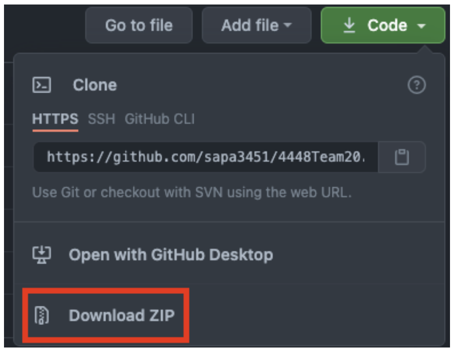


main · 1 branch · 5 tags · Go to file · Add file · Code

sapa3451 updated UI and test coverage — 4d5c94e 2 days ago — 129 commits

.idea	added the three ship classes	2 months ago
BattleShip	updated UI and test coverage	2 days ago
DOCUMENTS	update	2 months ago
.gitignore	CRC pdf	3 months ago
README.md	Update README.md	3 months ago

    1.2. Download the Zip Folder in the desired directory and unzip it



Go to file · Add file · Code

Clone

HTTPS SSH GitHub CLI

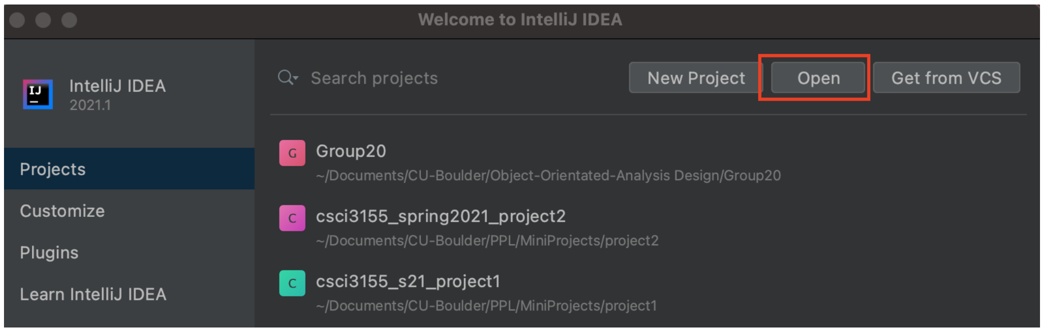
```
https://github.com/sapa3451/4448Team20.
```

Use Git or checkout with SVN using the web URL.

Open with GitHub Desktop

Download ZIP

2. Open IntelliJ
    2.1. Click on "Open" and locate the unzipped folder and open it



Welcome to IntelliJ IDEA

IntelliJ IDEA 2021.1

Projects · Customize · Plugins · Learn IntelliJ IDEA

Search projects · New Project · Open · Get from VCS

Group20
~/Documents/CU-Boulder/Object-Orientated-Analysis Design/Group20

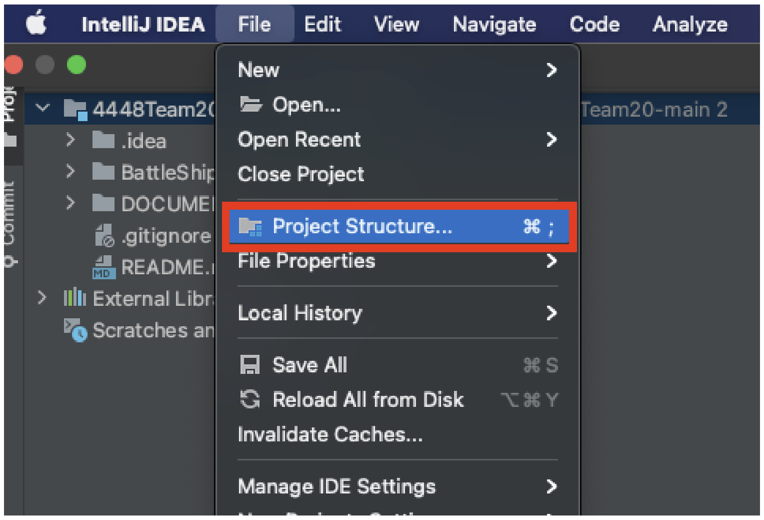
csci3155_spring2021_project2
~/Documents/CU-Boulder/PPL/MiniProjects/project2

csci3155_s21_project1
~/Documents/CU-Boulder/PPL/MiniProjects/project1

3. After the project opens, in IntelliJ:

   3.1. Go to the top right corner and open "File"

   3.2. Find and click on Project Structure

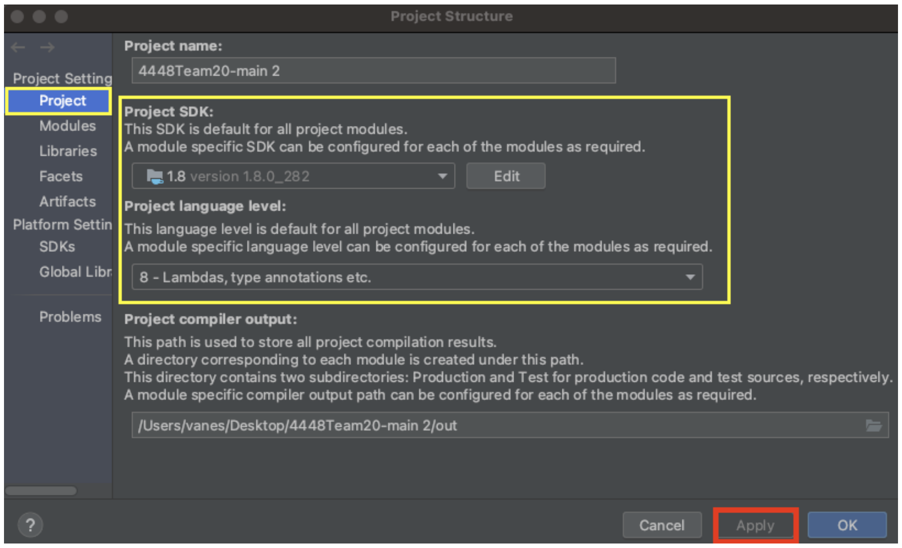


IMPORTANT: Our project is based in Java 1.8, and therefore needs the 1.8 version in order to run properly. If another version must be used, JavaFX must be installed as well as the corresponding JUnit libraries for the version. The instructions below are easiest and are for 1.8 Java. https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html

4. In Project Structure:

   4.1. Go to "Project" and make sure that "Project SDK is Java 1.8 and that "Project language level" is set to level 8

   4.2. Once set, click "Apply"



Project name:
4448Team20-main 2

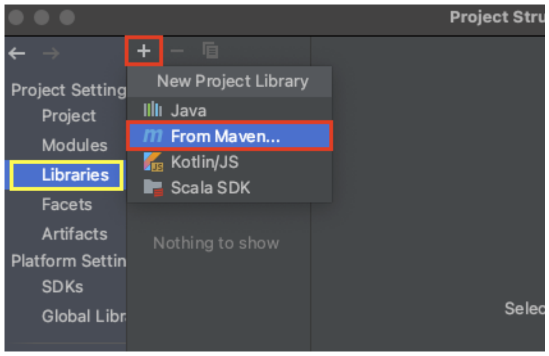
Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.
1.8 version 1.8.0_282

Project language level:
This language level is default for all project modules.
A module specific language level can be configured for each of the modules as required.
8 - Lambdas, type annotations etc.

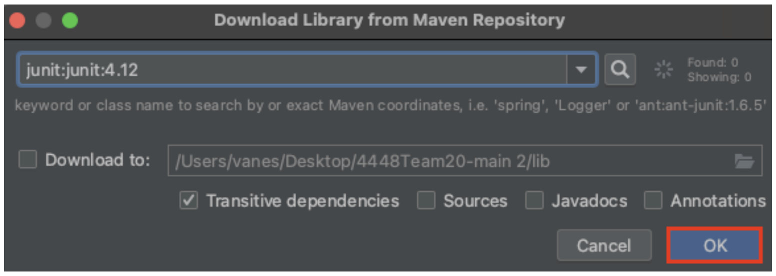
Project compiler output:
This path is used to store all project compilation results.
A directory corresponding to each module is created under this path.
This directory contains two subdirectories: Production and Test for production code and test sources, respectively.
A module specific compiler output path can be configured for each of the modules as required.
/Users/vanes/Desktop/4448Team20-main 2/out

4.3. Then we need to add specific JUnit libraries, go to “Libraries”, click on the “+”

4.3.1. Select “From Maven…”



4.3.2. Type in following library “junit:junit:4.12” and hit “OK”


Download Library from Maven Repository

junit:junit:4.12

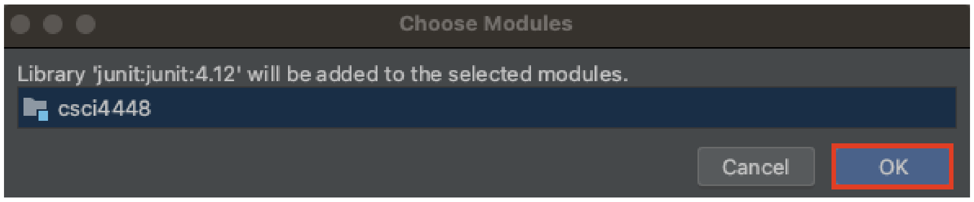
Found: 0 Showing: 0

keyword or class name to search by or exact Maven coordinates, i.e. 'spring', 'Logger' or 'ant:ant-junit:1.6.5'

☐ Download to: /Users/vanes/Desktop/4448Team20-main 2/lib

☑ Transitive dependencies ☐ Sources ☐ Javadocs ☐ Annotations

4.3.2.1. Make sure it is added to the csci4448 modules and click “OK”

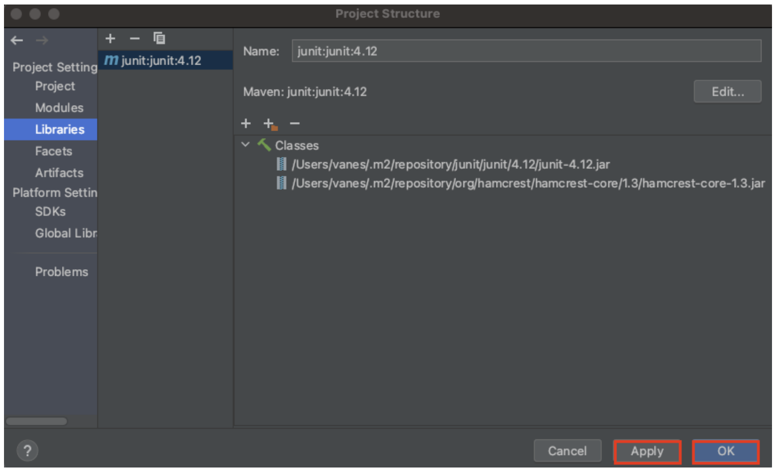

Choose Modules

Library 'junit:junit:4.12' will be added to the selected modules.

csci4448

4.3.3. Click on “+” again and add the following library with the same process above: “org.junit.jupiter:junit-jupiter:5.4.2”

4.4. Once libraries are added, click “Apply” and “OK”


Project Structure

junit:junit:4.12

Name: junit:junit:4.12

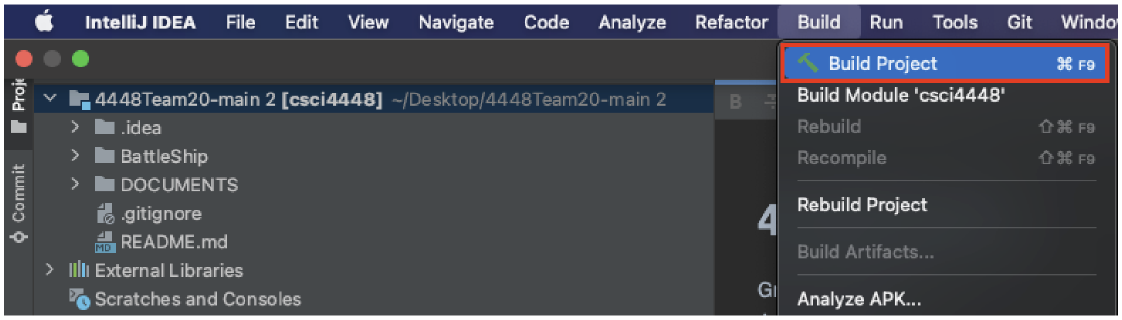
Maven: junit:junit:4.12

Classes

/Users/vanes/.m2/repository/junit/junit/4.12/junit-4.12.jar

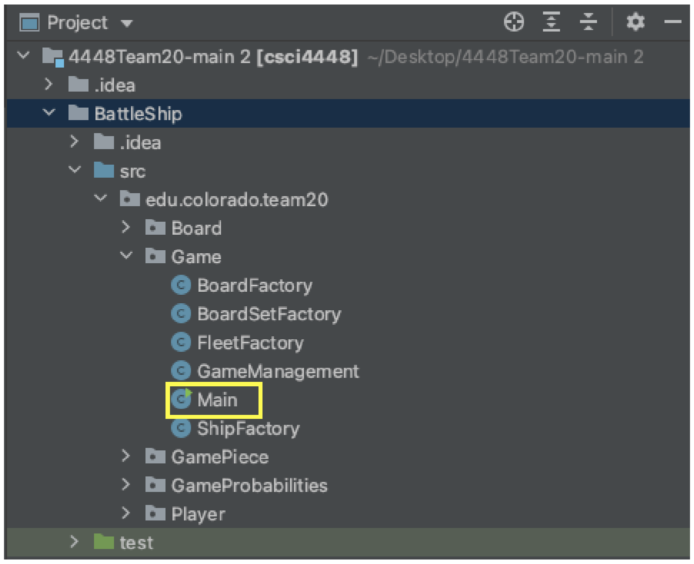
/Users/vanes/.m2/repository/org/hamcrest/hamcrest-core/1.3/hamcrest-core-1.3.jar

5. Now, go to the right-hand corner and click on “Build” and select “Build Project” to build the project



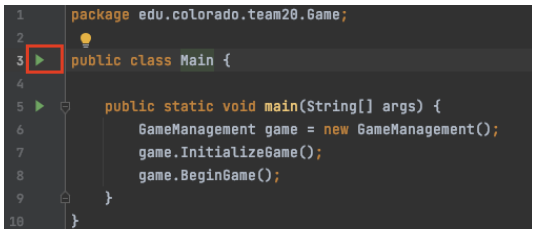
6. To see if the build was successful, navigate to the Main file through the file hierarchy to the left

6.1. 4448Team20-main → BattlesShip → src → Game → Main



7. Once the Main file is open, run Main by clicking on the green arrow and selecting “Run Main.main()”

8. You should get an output similar to the image below, showing our Game Menu, meaning the build was a success and you are set to go!

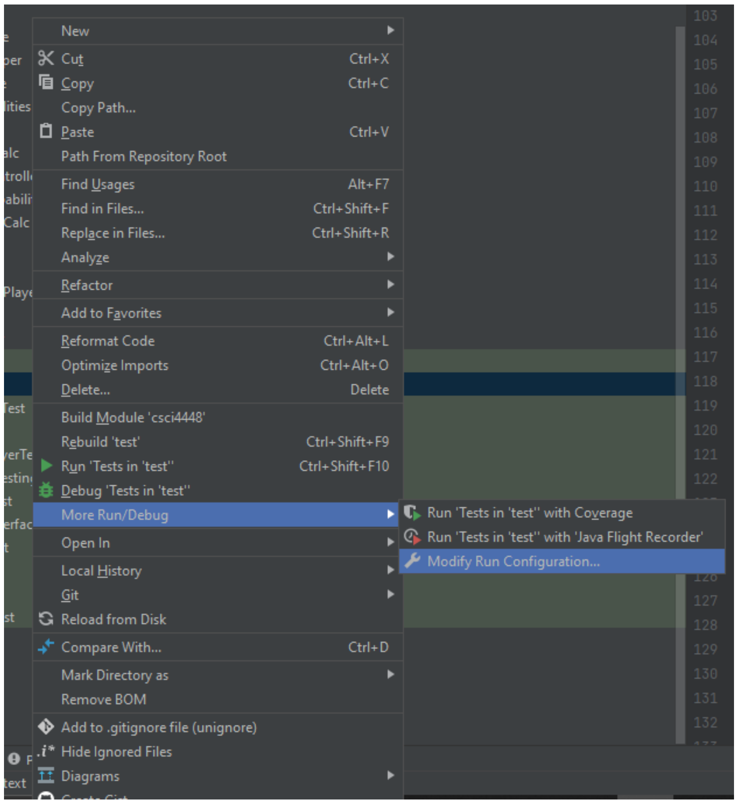


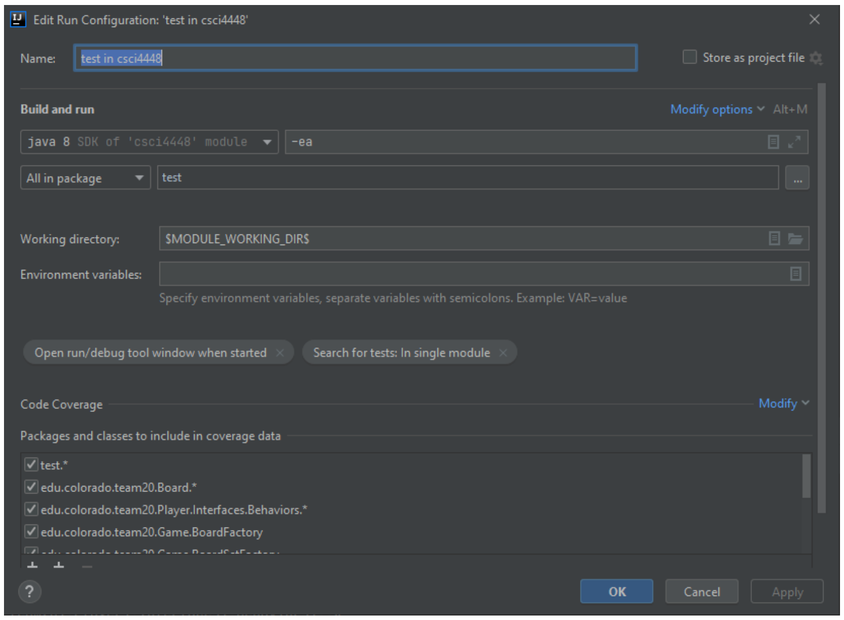
```
package edu.colorado.team20.Game;

public class Main {

    public static void main(String[] args) {
        GameManagement game = new GameManagement();
        game.InitializeGame();
        game.BeginGame();
    }
}
```

9. To run tests with coverage on your IDE, if needed, (coverage data is shown in the documentation above), right-click on the test folder within the Test directory, and click on Modify Run Coverage. Add and Exclude the following files/folders seen in the list below (*indicates a folder). Then, tests should be run with coverage of the full project code.

<u>Include:</u>

test.*
edu.colorado.team20.Board.*
edu.colorado.team20.Player.Interfaces.Behaviors.*
edu.colorado.team20.Game.BoadFactory
edu.colorado.team20.Game.BoardSetFactory
edu.colorado.team20.Game.FleetFactory
edu.colorado.team20.Game.ShipFactory
edu.colorado.team20.GamePiece.*
edu.colorado.team20.GameProbabilities.*
edu.colorado.team20.Player.*


New
Cut Ctrl+X
Copy Ctrl+C
Copy Path...
Paste Ctrl+V
Path From Repository Root
Find Usages Alt+F7
Find in Files... Ctrl+Shift+F
Replace in Files... Ctrl+Shift+R
Analyze
Refactor
Add to Favorites
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Delete
Build Module 'csci4448'
Rebuild 'test' Ctrl+Shift+F9
Run 'Tests in 'test'' Ctrl+Shift+F10
Debug 'Tests in 'test''
More Run/Debug
Open In
Local History
Git
Reload from Disk
Compare With... Ctrl+D
Mark Directory as
Remove BOM
Add to .gitignore file (unignore)
Hide Ignored Files
Diagrams
Run 'Tests in 'test'' with Coverage
Run 'Tests in 'test'' with 'Java Flight Recorder'
Modify Run Configuration...


Edit Run Configuration: 'test in csci4448'

Name: test in csci4448

Store as project file

Build and run

Modify options Alt+M

java 8 SDK of 'csci4448' module

-ea

All in package

test

Working directory: $MODULE_WORKING_DIR$

Environment variables:

Specify environment variables, separate variables with semicolons. Example: VAR=value

Open run/debug tool window when started

Search for tests: In single module

Code Coverage

Modify

Packages and classes to include in coverage data

- [x] test.*
- [x] edu.colorado.team20.Board.*
- [x] edu.colorado.team20.Player.Interfaces.Behaviors.*
- [x] edu.colorado.team20.Game.BoardFactory