4Dimensional1D Games

Battleship Game

Final Project Documentation

Members:
Harry Ainsworth
Justin Velvick
Kaden Ostendorf
Luke Phillips

Project Description:

We have designed and developed a modified version of the classic Battleship game. Battleship is a two-player strategy guessing game. It is played on ruled grids which each player's fleet of ships are marked. The other player's fleet location is concealed unless hit. Players alternate shooting shots at the other player's ships, and the objective of the game is to destroy the opposing player's fleet.

Players will decide where their ships start before taking turns firing back and forth at the other players board. After the game is initialized several new features have been implemented. The fleet can be moved and when a player decides to move the fleet, it is possible that they run into mines, or a powerup. If they run into a mine, their ship will be damaged. If they run into a powerup, they will get a powerful weapon that hits more blocks than the regular single shot.

Development Process:

For this project, our team followed **Extreme Programming (XP)**, a type of agile software development that aims to maximize new features being introduced, while minimizing new bugs being introduced. As an agile method, it helped us adapt to constantly changing requirements; combined with the programming practices we followed and the design patterns we used, we were able to future-proof our code to prevent these changing requirements from requiring major redesigns.

One practice of XP that we followed was **test-driven development (TDD)**: every feature we wrote had tests written beforehand (if applicable), and we ran our test suite frequently, to ensure that new code did not introduce bugs to old features. An issue we ran into quite early on with

TDD was that GUI and graphics were very hard to write unit tests for; due to the nature of what graphics are, we ended up having to test our running game with the GUI from time to time to make sure new bugs didn't appear on that front. Another issue we ran into was writing tests for elements of the game with a degree of randomness (e.g. mine placement); our solution was to introduce methods to recreate the random behavior minus all the randomness (in the case of mines, this took the form of a method to place a mine at a certain coordinate).

Another aspect of XP that we followed was **pair programming**. Although some code was written alone, the vast majority of our codebase was written while we were on Discord calls with each other; when not coding ourselves, we provided feedback to whoever was coding, as well as suggestions for how we could improve the code being written. This certainly helped catch a few issues.

We also practiced **continuous integration**. Periodically while coding, we would push to our GitHub repository; while doing that, we'd also pull from GitHub, resolving any merge conflicts and ensuring our code all worked together. Besides that, we also did a significant amount of **refactoring**: a very good portion of our code wasn't particularly good initially, so we made sure to go through our codebase every so often, look for poorly written code, and try to improve it in a way that would minimize necessary changes once requirements changed. This made our development process significantly more efficient than it would have been otherwise - we didn't end up having to change the entire codebase every two weeks.

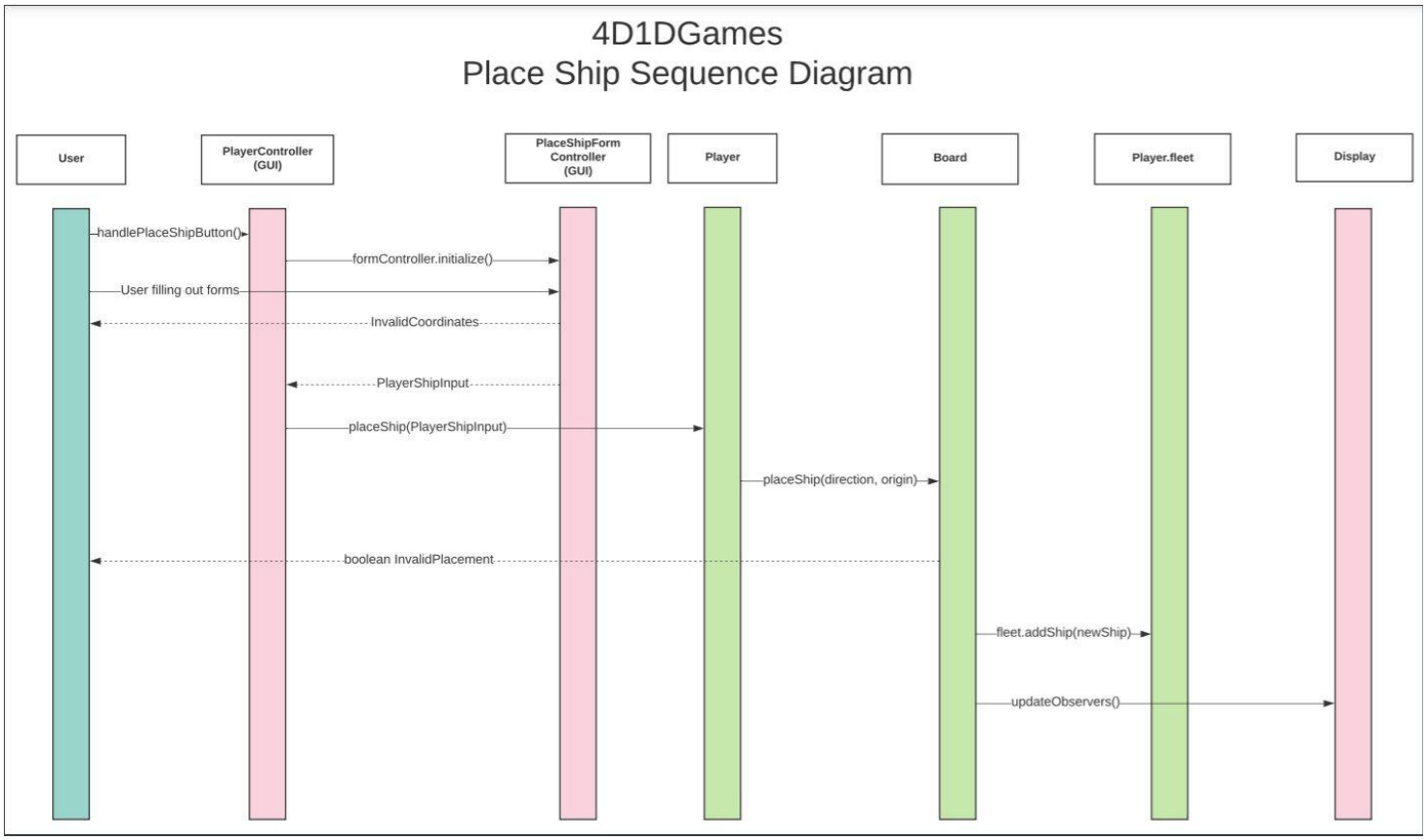
Requirements and Specifications:

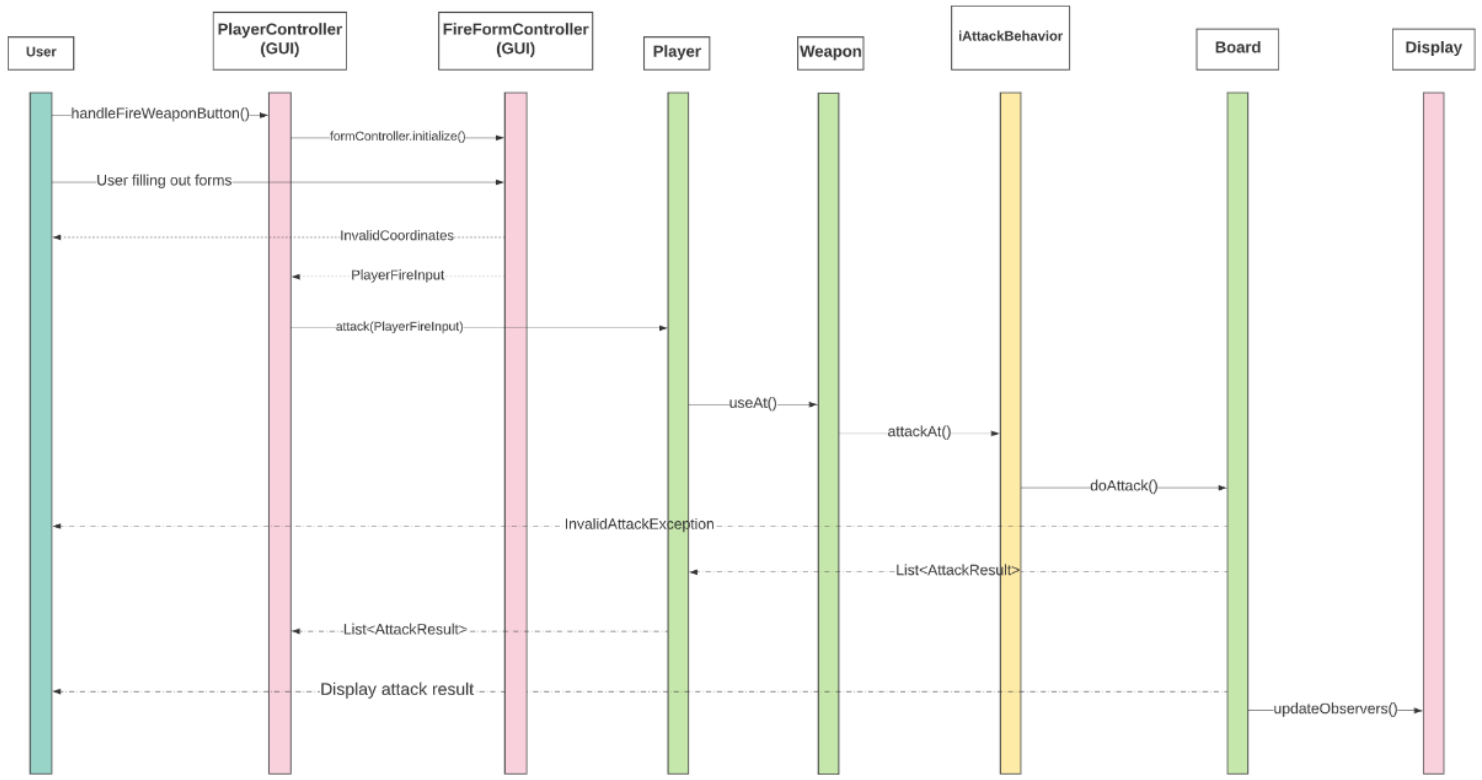
User stories for the standard battleship game:

- As a player, I would like my enemy to not be able to see my board during his turn, so that I can be sneaky and strategic.
- As a player, I would like tiles that I have already attacked to be differentiated, so that I can keep track of my attacks.
- As a player, I would like the game to regulate player turns and tell the players when the game is over, so that the game is fair and players must adhere to the rules of play.
- As a player, I would like a graphical user interface to interact with and see where my ships are

Advanced game mechanics and ship movements:

- As someone who wants to keep their enemy on their toes, I would like to be able to move my fleet, so my opponent never knows for sure where to shoot next.
- As a player who likes to be more sneaky, I would like a ship that goes underwater and cannot be hit by conventional weapons, so that my opponent is forced to think about using other types of weapons against me.
- With submarines in the game, I would like to be able to hit and sink that new type of ship, so that I can feasibly win the game.
- As a player, I would like mines to be set to make my opponent more wary of moving their fleet
- As a player, I would like to be able to fully sink a ship by only taking out the “Captain’s Quarters” due to removing the central command of the ship.
- As someone who likes replay value, I would like each game to look and feel different, so I don't get bored and quit.


4D1DGames
Place Ship Sequence Diagram
User
PlayerController (GUI)
PlaceShipForm Controller (GUI)
Player
Board
Player.fleet
Display
handlePlaceShipButton()
formController.initialize()
User filling out forms
InvalidCoordinates
PlayerShipInput
placeShip(PlayerShipInput)
placeShip(direction, origin)
boolean InvalidPlacement
fleet.addShip(newShip)
updateObservers()


4D1DGames
Fire Weapon Sequence Diagram
User
PlayerController (GUI)
FireFormController (GUI)
Player
Weapon
iAttackBehavior
Board
Display
handleFireWeaponButton()
formController.initialize()
User filling out forms
InvalidCoordinates
PlayerFireInput
attack(PlayerFireInput)
useAt()
attackAt()
doAttack()
InvalidAttackException
List<AttackResult>
List<AttackResult>
Display attack result
updateObservers()

Weapons upgrades:

- As an experienced player, having more than one weapon choice would make the game interesting. Additionally, I would like to be able to choose which weapon to use each turn in order to play out my own strategy.
- As a player, I would like the chance to earn weapons upgrades by moving my fleet onto special upgrade tiles.
- As a player, I would like variety in my weapon choices and more strategic options.
- As a player, I would like the sonar pulse to display my board for only one turn so the advantage is only momentary and does not last the entire game
- As a player, I would like to fire a penetrating space-laser weapon that has the ability to attack underwater submarines, which the normal attack weapon cannot reach.
- As a player, I would like to have a certain probability of gaining a weapon from a weapons upgrade that takes out a cross-shaped section of the board where I fire it.
- As a player, I would like to have a certain probability of gaining a weapon that fires on ALL tiles on the board, winning me the game instantly.

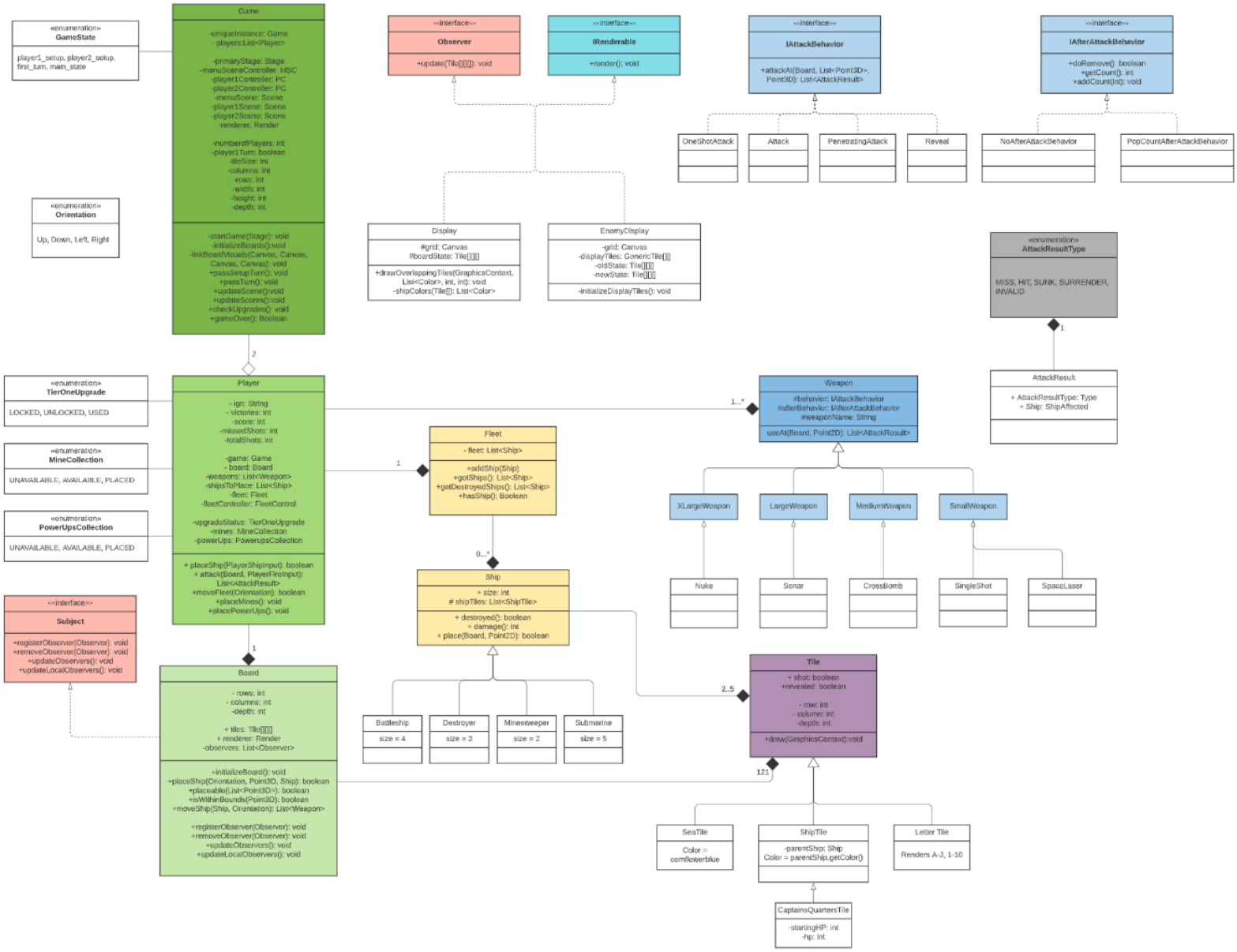
Architecture and Design:

Project Structure/Organization:

- Our project is divided into **packages** which contain related classes. For example, the "ship" package contains all the concrete ship classes along with the abstract class Ship.java itself. This package also contains other things relating to ships such as the Fleet.java and FleetControl.java. Some packages have more packages within them to create a "folder-file" type relationship where a hierarchical relationship is established.
- More broadly, we have two main forks in our project tree: **Main** and **Test**. Main contains all the .java files, the actual uncompiled logic of our program. Test contains all our test classes that check our current code against retroactive requirements to be sure nothing from before was broken from adding additional logic.

Test Driven Development Using JUnit 5:

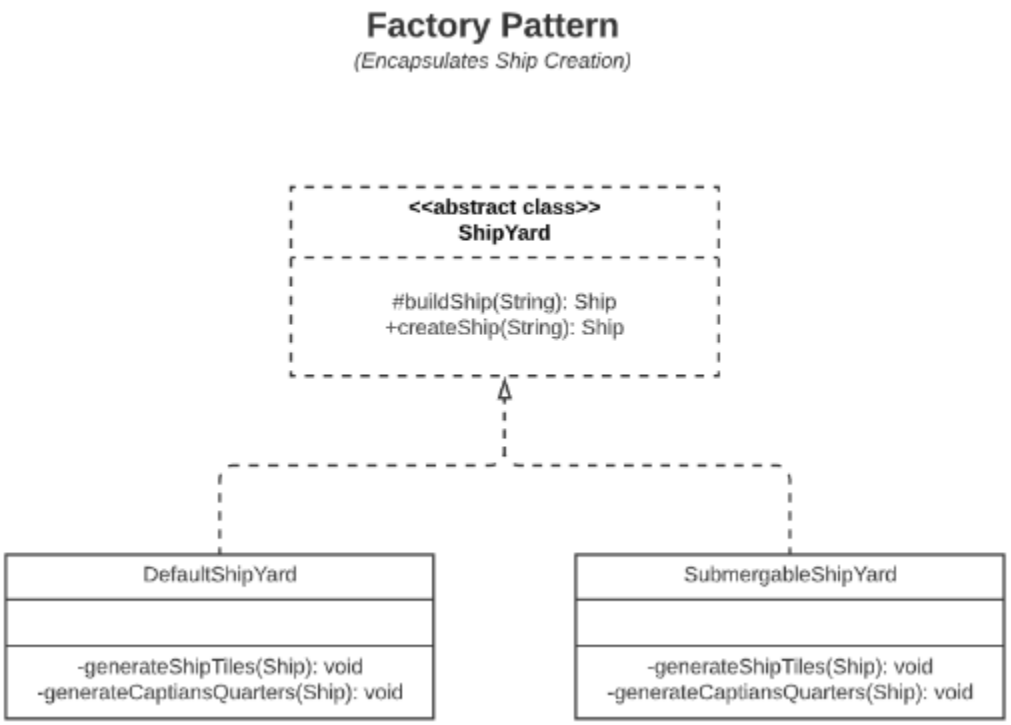
- When new requirements are given, we first as a team brainstorm what design patterns or data structures would be appropriate. Once we have a basic idea of how we want to fulfil these new requirements we first create failing tests that call blank methods. Making these failing tests pass in turn will most of the time also automatically fulfill the requirements laid out with no extra work needed.
- With a huge net of tests already built, further along the development process you can feel secure knowing that anything you add can have our test suite check if any previously added logic or features broke in unforeseen ways.



Business Layer:

- Take away the GUI along with Testing, and what remains is the heart of our project: the logic. We have a **Game.java** class that handles all higher level game management such as when to pass the turn or deciding to restart the game or not. This class implements the **Singleton Design Pattern**, and as such, only one instance of Game can exist at a time in our application.

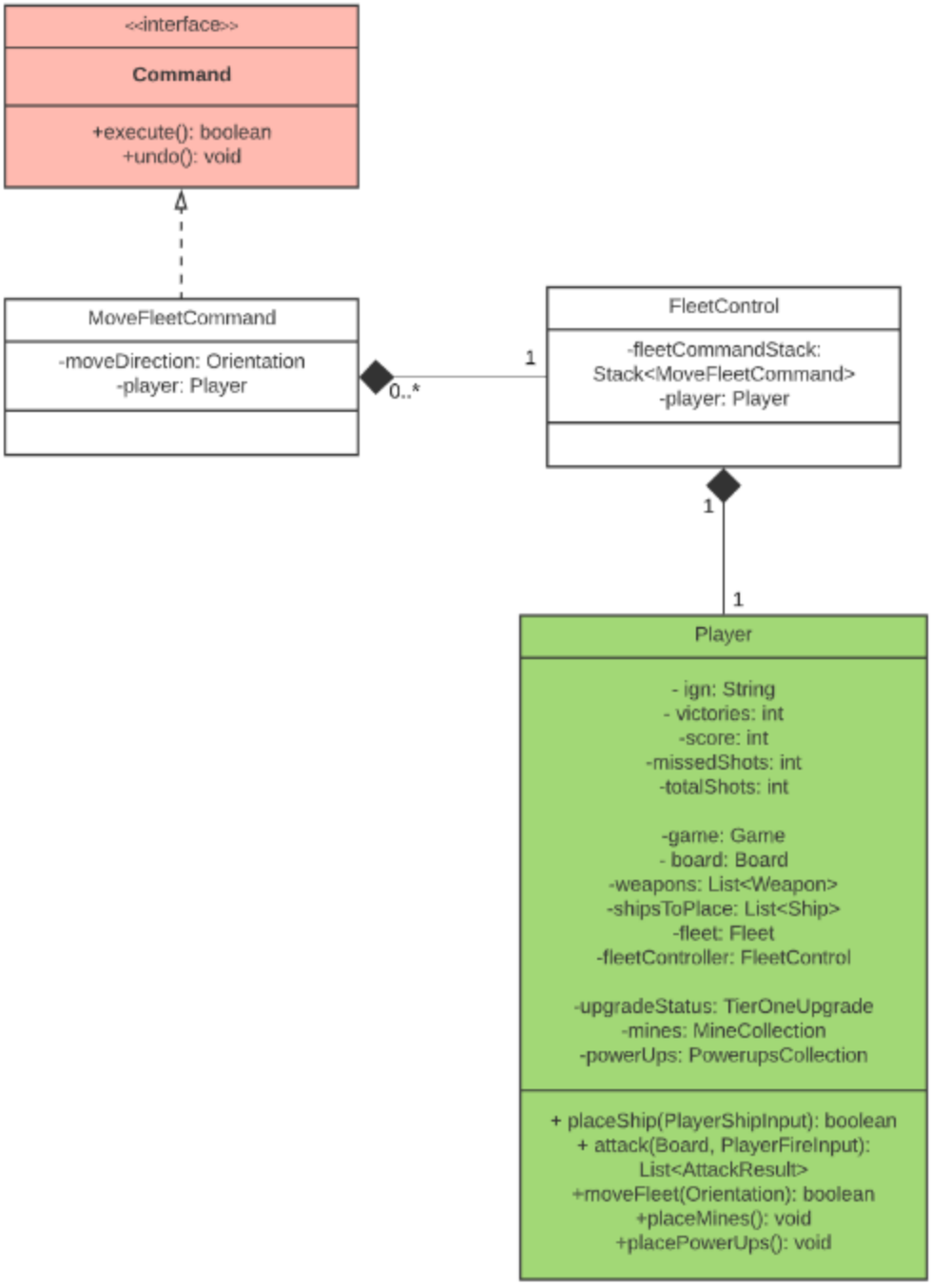
- The **“Tile”** object is the foundation of all “tangible” objects in our code. A ship for example, is made up of ShipTiles which naturally extend from Tile. A game board is blank at the start of a game with nothing but SeaTiles until each player places all of their ships. Once that happens, randomly generated MineTiles and PowerUpTiles will spawn across the board, replacing the SeaTiles that once stood in those spots before. All tiles belong to a **Board** object which implements the Subject side of the **Observer design pattern.** The board handles checking if ship placement, movement, and attacks on that board are valid or not.
- A ship is created via a **ShipYard**, which implements the **Factory design pattern**. The ShipYard encapsulates everything related to making a ship: creating it’s ShipTiles, creating the CaptiansQuartersTile with appropriate health to match and deciding where that should be located on the ship, etc.



- To attack the enemy, the player must use a **weapon**. Weapons are a little more complicated as they implement the **Strategy design pattern**. Following the pattern, weapons have injectable behaviors at runtime. For our weapons, this means two distinct behavior types: IAttackBehavior and IAfterAttackBehavior. Each has a few choices that a weapon could choose to have, such as the reveal attack behavior for weapons like “Sonar Pulse” to reveal tiles instead of damaging them, or the penetrating attack behavior for weapons like the “Space Laser.” As for after attack behaviors, they handle weapon counts after use. A weapon with infinite use like “Single Shot” will have a “NoAfterAttackBehavior" object which simply does nothing when called. For weapons with a finite amount of uses like “Sonar Pulse”, a pop count behavior is added so that the total usages remaining gets decremented after use. The point of these behaviors is to simplify code that can handle any weapon with any behavior, and can treat every weapon the exact same.
- Players can also **move their fleet** as many times as they want during their own turn. Before passing the turn they can also undo any movement they ordered that turn. This undo mechanic is encapsulated in a **Command design pattern**. A player object has a FleetControl “remote” that can have a moveFleet() method called, passing in a direction. That is all the player knows about, a remote with two buttons: move and undo. Under the hood, there's a stack that stores MoveFleetCommand objects that store the direction used for each time the player invoked move fleet. Undo simply pops from the stack and calls the player’s moveFleet() method except this time, passes in the opposite direction to simulate an “undo” effect.

Command Pattern

(Encapsulates undo stack)



Added Technology (GUI):

- Early on, Tile extended the JavaFX Canvas class, which meant that each Tile was it's own little 40x40 pixel canvas. This lead to a lot of JavaFX related headaches, so Tiles no longer extend from Canvas, and instead we have one giant canvas for each 11x11 grid that individual objects which implement **IDrawable** can draw themselves on.
- We have two classes to encapsulate the idea of a display: **Display and EnemyDisplay.** Both classes have all the code necessary to update what the player sees on the screen and they can both handle what they should and should not see. EnemyDisplay filters out what gets rendered based on whether or not the tile has been shot or revealed as the enemy player should not be able to see where your ships are unless they hit one. Both classes implement the Observer side of the **Observer design pattern**, waiting for data from Boards (the Subjects) to update them.
- When JavaFX forms are spawned, they create accompanying **controllers**. These controllers handle events that the user may cause to happen such as button clicks or form submissions. Controllers then call various methods belonging to the business layer to decide what logic runs after each handler interrupt. We pondered implementing the Facade design pattern for our business layer to decouple from these controllers, but time and inexperience got the best of us and we were unsure how to go about creating this relationship in a clean way.

Personal Reflections:

Harry: Over the course of this class and this project, I really enjoyed the opportunity to learn how to apply object-oriented design patterns to an actual functional project. I have had some experience using design patterns in the past, but did not get to do much in terms of actually applying these concepts in code. This was a great experience to build my practical programming skills before I start working full time as a software engineer once I graduate this spring.

Justin: One challenge was making enough tests to cover all possible interactions. Even after having near 100% coverage, once we implemented the GUI we became aware of several bugs that resulted from interactions we just didn't think of happening. You can never have enough tests. By far my personal biggest mistake this project was deciding to have tiles store their location. As a naive OO programmer before having taken this class, this seemed like a harmless idea. In retrospect, after everything we have learned in this class, tightly coupling tiles to their location on the board was a mistake and lead to so many headaches further on in development. Another challenge I couldn't solve in time was how to visually represent submerged objects below surface level objects, this one requirement exposed bad decisions relating to Board, Game, and Tiles. I can't reiterate enough how much I learned in this class from messing up. A joke I made to my group mates throughout the semester was "A prerequisite for this class **is this class.**" I sincerely wish I could take this class again as it was the most practical coding class I have taken yet and I feel that this class prepared me most for possibly stepping into industry. I even learned life lessons in this class relating to change. Seeing all of our classmates who complained endlessly on piazza about unexpected/unforeseen change really opened my eyes. I wish those programmer's supervisors luck when they get hired. Change is inevitable, accept it, love it, and ruthlessly prepare for it.

Kaden: This project was honestly one of the most valuable experiences I think I've ever had in the CS department. The iterative development cycle that was used seems a lot more accurate to real-world software development, and I feel like it really helped me become more prepared for pursuing software dev (and particularly game development) as a career. I'd just like to say thanks to all of the staff for this class - the feedback we got every two weeks was really helpful in deciding where we wanted to go, and it definitely directed me towards being a better programmer.

Luke: This class has been very beneficial and helped expand my horizons in the computer science field. I have enjoyed learning about the development process and believe this class will be one of the most applicable towards my career going forward. I made my fair share of mistakes and have had ample time to correct those mistakes. In the future, I will be able to detect future problems much more sooner. It was also nice having such a cohesive structure to keep track of our progress and benchmark important milestones. I am confident that I can be a valuable and efficient group member in the future when working on larger projects. My coding skills have greatly improved from the beginning of class when I had next to no experience with Java.