Battle the Bot!

Niko Johns, Joanna Parker, Jordan Okada

Work Done

Niko: Created board, player, user, and npc classes
Jordan: Created playGame class, added more functionality to board and player classes. Created UML class diagram. Developed tests for playGame, board, and player classes.
Joanna: Created basic decision tree and random forest classes. Fully developed data and turn classes. Developed tests for those classes. Worked on the UML class diagram.

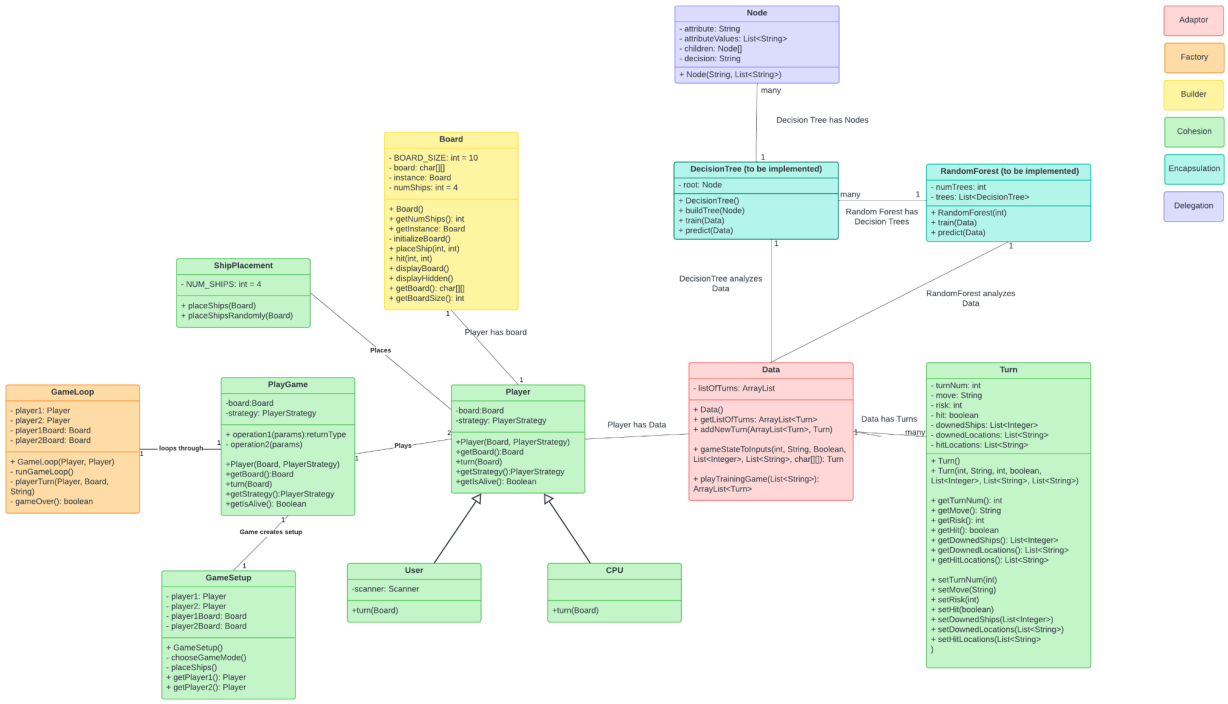
Changes and Issues

- Linking the learning algorithms and the game framework required defining a custom data structure to hold the necessary information. It also required slight modification to how hits and misses were printed so that the data class could differentiate between the two just based on board output.

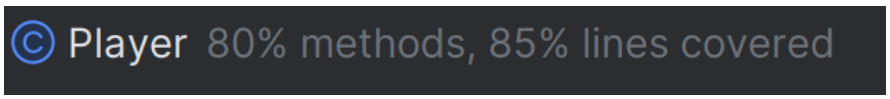
Design Patterns

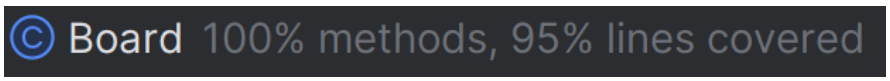
- *Adaptor*: the data class is the interface between the game and the random forest. It acquires the necessary information from the board and player classes and transforms it into input the random forest or decision tree will understand. This prevents significant modification of the learning algorithms or the structure of game play.
- *Factory and Builder*: We create new players and ships without using the 'new' keyword for each creation. This makes generating testing data much easier, as a loop can be written to play random players against each other.
- *Cohesion*: the game framework and learning algorithm framework are broken into classes focused on one task. This prevents repetitive code and makes the interface between the two frameworks simpler.
- *Encapsulation*: Users, or ourselves writing code, do not have access to the data structure of the learning algorithm. Certain parameters can be adjusted, but we cannot break the algorithm by accidentally modifying the data structure or passing in disallowed parameters.
- *Delegation*: Classes delegate irrelevant tasks to other classes. This prevents any one class from becoming too complex, so the code base is readable and easy to use.

UML Class Diagram



Test Coverage





BDD Scenarios

1. Singleton Pattern (Board Initialization)
    a. When a player starts the game, a new board instance is created with all cells initialized to water (‘~’).
2. Strategy Pattern (Player's Turn)
    a. When a player takes their turn, they choose a cell to attack on the opponent’s board. The system updates the board based on whether it was a hit or a miss.
3. Placing Ships
    a. When a player chooses a row and a column to place a ship, the ship is placed on the selected coordinates if not already occupied.
4. Hitting a Ship

    a. When a player selects a coordinate that contains the opponent's ship, the cell changes to indicate a hit 'X'. The number of ships the opponent has decreases by one.
5. Game Setup
    a. When the player chooses the game mode Player vs CPU, the player is prompted for information to fill out their board and the other board is generated randomly.
    b. When the player chooses the game mode Player vs Player, both players are prompted for information to fill out their board.

Plan for completion

In our next iteration we will:

- Implement ships of different sizes
- Create a UI design
- Train a random forest, and use it as the computer opponent
- Make the random forest class more robust (Adding a depth option as well as number of trees)

Repository Link