CamIship: Final Design Document

Arpit Sheth, Jeannie Fu, Sahitya Mantravadi, Taylor Schoettle

SYSTEM DESCRIPTION:

The Key Idea:

We will implement a Battleship platform with an intuitive user interface and an artificially intelligent player.

The Key Features:

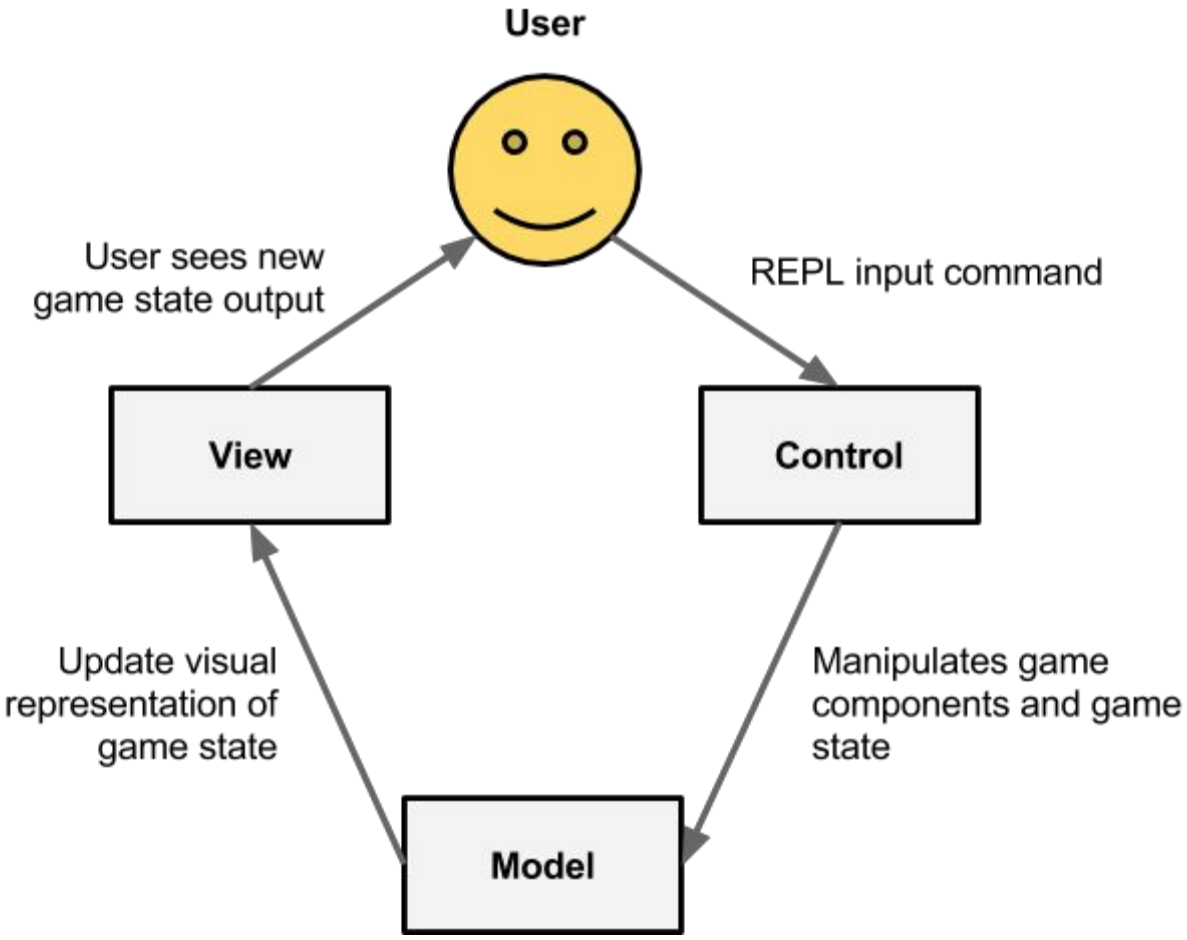
- AI Design
  - Initial placement of ships (will not touch)
  - Making guesses for turn based on current hits
- User Interface
  - Initially text based that takes in commands based on a specific set of rules
  - Text based visualization of the current game state for a given player
  - Potential for GUI extended functionality

The Description:

This project creates the Battleship game with a multiplayer and artificial intelligence component. There is a text based interface responsible for handling user input and output based on a given set of rules. The current game state for each player is rendered in the command line per turn as well. The artificial intelligence component of this game is based on a heatmap of the entire game grid, where each tile is given a “score” that can be influenced by the “scores” of its neighboring tiles. The AI selects the tile with the highest score in the entire grid for its turn.

The Architecture:

Our design of the Battleship game was made using the **Model-View-Control architecture**. The **Model** of the game is the data structures and states of different game components such as the grid map, battleship layouts, and players (including the game AI). The **View** of the game is the rendering of the game state, and especially the grid map of where a player has shot, where they missed, and where they hit a battleship. The View is implemented as a text based interface in the command line. The **Control** of the game consists of interactions between the various game components, such as players, battleships, and the grid map, given an input/output stream of commands from the user.



The System Design:
At the highest level, is a main module, called Gameplay, that runs a REPL, initializes the game, and stores information on the players and game state. The user passes commands into this module which then changes the game state accordingly. If the player is actually an AI, then instead of prompting the user for a command, it does a computation to make the best move with the current game state. The Gamestate module is used to represent the grid itself and keeps track of hits, misses, and boat placements. The Gameplay module depends on the Gamestate module since the two players grids are included in the game state.

The Gamestate module is responsible for keeping track of information about the gamestate and grid and providing methods to interact with the grid. Each player has information about his own grid, and is able to access his opponent's grid to print its current state. There is a function (display_gamestate) which allows for the user to see the placement of ships on his own grid and the hits/misses of the opponents. Each tile of the grid is represented by a (terrain*tilestate) pair, which explains what is located at a certain location and whether the opponent has hit that location. Our fleet variable is a ship list.

The AI module is responsible for the functions that the AI players will use to compute their next move based upon the current game state. We have implemented two versions of the AI to make moves.

AI's Placing Ships Function:
The AI places the ships using the following method, and helper functions are used to make sure that they are valid placements. The AI decides to place ships more or less at random. This forces the player to not have any method of figuring out where the ships are besides just guessing. The complex part of the algorithm comes when the AI is about to place a ship adjacent to another ship. If this event occurs then 50% of the time the AI finds another spot to put the ship and the other 50% the AI just places the ship there. This causes the less of the AI's ships to be touching but for it to still be a possible event.

Easy AI:
The easy AI takes in a gamestate and generates a random coordinate to move at. The random coordinate is generated by first ensuring it is within the bounds of the gamestate grid. Next, it ensures that the proposed coordinate has not already been acted upon (either with a Hit or Miss). If these properties are ensured, then the proposed random coordinate is a valid coordinate, and that is where the AI will attack. Otherwise, the AI tries to generate another coordinate to attack on by repeating the process.

Hard AI:
The hard AI takes advantage of mutability in order to generate a new move based on the success of its attack history. The AI takes in a gamestate and manipulates a record for best move data, which tracks important information about the AI's attack history and proposed next moves.

```
type best_move_data = {
  mutable first_hit : coord option;
  mutable next_moves : coord list;
  mutable last_move : coord option
}
```

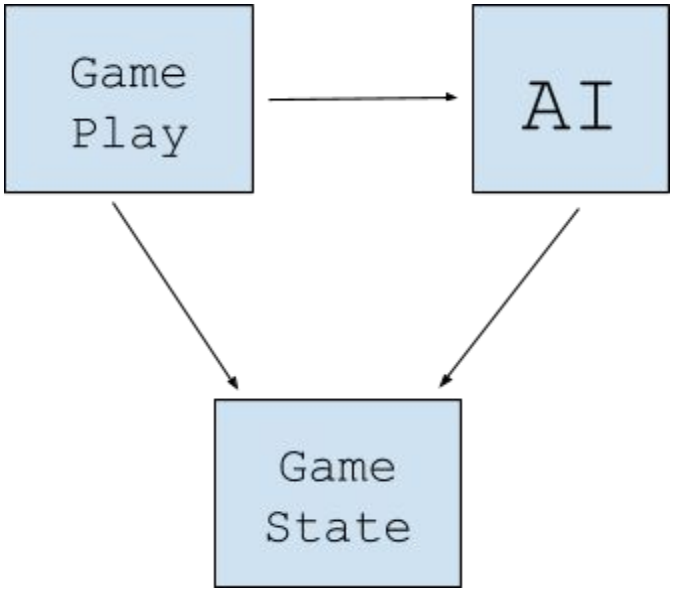
First hit represents the coordinate at which the first hit streak started.
Next moves is a list of coordinates that the AI can choose from as its next possible best moves.
Last move is the coordinate where the AI last attacked, so it can see if its attack was a hit/miss.

First hit is None if the AI is not actively targeting a ship, thus it should attack at a random coordinate. Last move will only be None when it is the AI's first turn in the whole game, thus it should attack randomly for its first turn. If next_moves is empty, then first_hit should also be set to None because it means that the AI is not actively targeting a ship.

However, if last_move is a Hit, then first_hit should be Some coord, that represents the first time a ship was targeted. When first_hit is set to Some coord, next_moves should be populated with up to four neighboring directions from the first_hit coord. These four coordinates should represent one coordinate unit that is in the up, right, down, and left directions. However, before next_moves is set to these possible coord values, they should be checked to see if they are valid (within bounds of the game grid and Empty (not acted upon yet)).

If last_move is a Hit and next_moves is not an empty list, then it means the AI is currently acting on a targeted ship. Therefore, the AI should pick an element (which is a coord) from the next_moves list. If the AI attacks there and it misses, then the AI should remove that coordinate from the next_moves list and pick another element. If the AI makes a Hit at the coordinate, the AI can now determine the direction of the ship based on its last_move and first_hit. Once the direction is determined, any directions in next_moves that are not consistent, are removed. For example, if the direction is Up, then Left and Right direction coordinates should be removed, but the Down direction coordinate can remain. Finally, a new coordinate that is consistent with the direction of the ship is added to next_moves so that the AI can attack in the next coordinate.

Module Dependency Diagram:



The Module Design:
See interfaces.zip for .mli files

The Data:
We need to store data relating to the game, which consists of playerstates and gamestates. The player state is stored as a record containing the first player's name, the second player's name, and the current player. The gamestate is stored as a tuple of sides, where each side consists of

a grid and a fleet. The grid will consist of a list of lists of tiles. Each tile is a pair of terrain type and tilestate type. The terrain can be either part of a ship or water, and the action can be a hit, miss, or not done yet. In this way, we can show each player either the entire grid (when they view their own grid) or just the actions portion of the grid (when they view their opponent's grid).

The External Dependencies:
The game requires external dependencies primarily for the REPL feature. It requires the Str module in addition to standard modules such as Pervasives, List, and String. From the Str module, the Str.split function was used to parse the different components of the input commands entered by the user.

The Testing Plan:
We first created a game of battleship usable with internal commands. After we tested this and ensured our basic functionality works, we extended this out to an AI vs. human game.

First, we implemented a grid since this is what the rest of the game would be interacting with. We first ensured that we were able to correctly place all ships on the grid. We then tested our grid by performing moves and making sure that the grid could recognize hits/misses and sunken ships. We also tested that our grid could recognize that all ships have been sunk and the game is over.

Once this was done, we tested our main game module. We made sure that our user interface was working correctly, by testing that the player could place ships and make moves that are recognized by the game module.

Finally, we tested our AI by making sure that it could intelligently place ships and make turns. This was be done by play-testing an AI to see if reasonable moves were made and if the AI could win in a reasonable amount of time.

Division of Labor

The group met at minimum three times per week in person, along with remote individual contributions. Overall, each member contributed about 15 hours per week per person on the project. This includes technical and non-technical contributions, and each member contributed significantly to the technical aspects of this project. No member was assigned a strict non-technical role.

Arpit Sheth
I helped with designing the interface, especially for gamestate. We worked on this together in person and received feedback from the TA. I also helped with the REPL, especially in developing the interp_input process, working alongside Sahitya.

My main individual contribution was in creating display_gamestate for the string representation of the gamestate. Because the other parts of the game heavily depend on the string representation of the game board (especially in making testing easier), I also set up the testing suite with a large number of reusable variables. The other group members followed my format in creating their own unit tests for their functions.
Finally, I helped develop the easy and hard AI's with Jeannie.
I also took on an unofficial project management role by acting as a liaison between group members to discuss proposed design/interface changes, especially in the beginning.

Jeannie Fu
I also helped design the interface. In terms of implementation, I created the turn function (and its helper functions and test cases) by myself, and worked with the repl who . I also created the random AI and smart AI with Arpit. Additionally, I edited all of the specifications at the end.

Taylor Schoettle
My main individual contribution was in the placing ships phase of the game, both for the human player and the computer player. The nice part about this was that if was rather separate from other parts of the code so I didn't need to worry about interfering with other people's work. I also helped with general design, user interface, and debugging.

Sahitya Mantravadi
Sahitya designed the REPL for the attack phase. This included getting user input, interpreting it, and switching off between players turns. She also made the user experience much better by providing instruction and making the game very clear.