Project report

1. Design

1.1Architecture

- the details of class hierarchy for your game state (model)

1. Game map: This includes the physical representation of the game map, including the size of the map, the layout of territories, and any other relevant details.

2. Territories: The Model would keep track of which territories are currently claimed by each player, as well as which territories are still available for claiming.

3. Unit: The Model would store information about each player in the game, including their names, budget, and other relevant statistics.

4. Region: The Model store information about each region in the game

5. Game logic: The Model would contain the rules for how players can claim territories, what actions they can take on their turns(Such as Relocate,Shoot,Move, etc), and how budget is calculated.

6. Game state: The Model would maintain the current state of the game, including which player's turn it is, how many territories each player has claimed, and any other relevant information.

7. Validation: The Model would validate player moves and ensure that they comply with the rules of the game, preventing any illegal moves or game-breaking scenarios

- the details of the display (view and controller), shown as graphical user interface

View

1. Game map: The View would display the game map and the territories that have been claimed by each player, including any visual cues to indicate which territories belong to which player.

2. Player information: The View would display information about each player in the game, including their names, budget, and other relevant statistics.

3. Turn order: The View would display the current turn order, indicating which player's turn it is and who will go next.

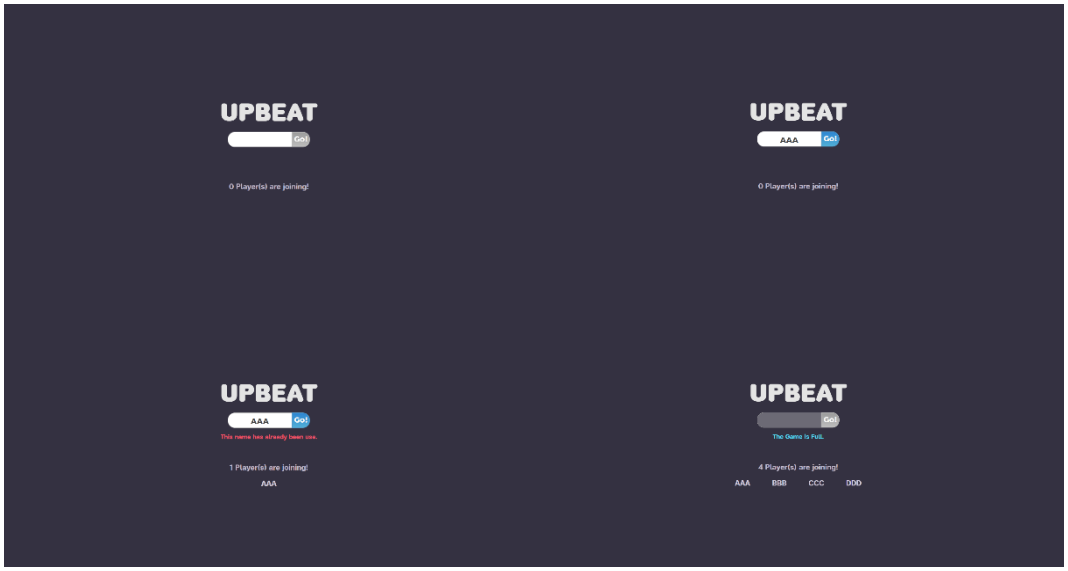
4. Game status: The View would display the current status of the game, including which player is currently winning, how many territories each player has claimed, and any other relevant information.

5. User interface: The View would provide a user interface that allows players to interact with the game, such as by selecting territories to claim, viewing rules and instructions, or accessing game options.

6. Game messages: The View would display messages to the user, such as notifications of game events (e.g. "It's Player 2's turn" or "Player 1 has claimed a new territory").

7. Game animations: The View could also include animations or visual effects to enhance the user experience, such as highlighting the territories being claimed or animating player avatars

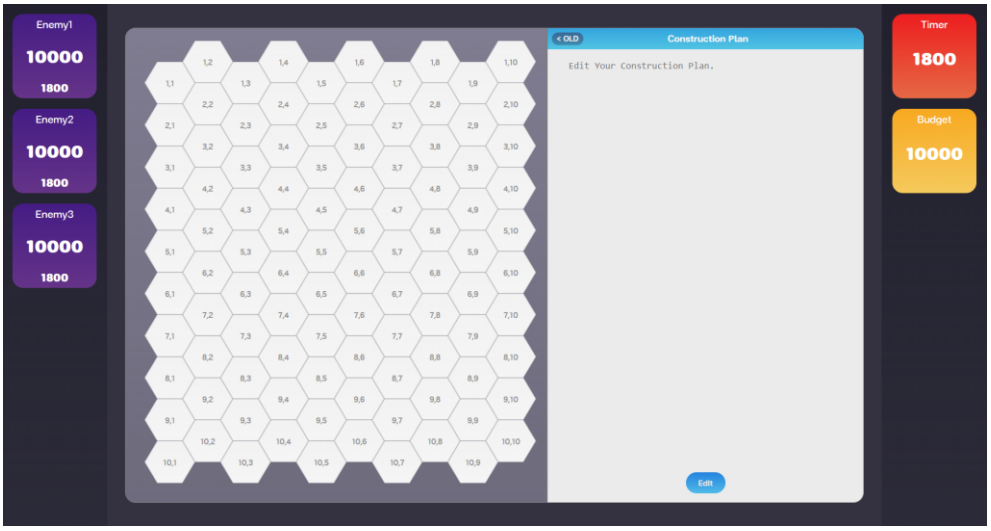
- An example of a complete UI.



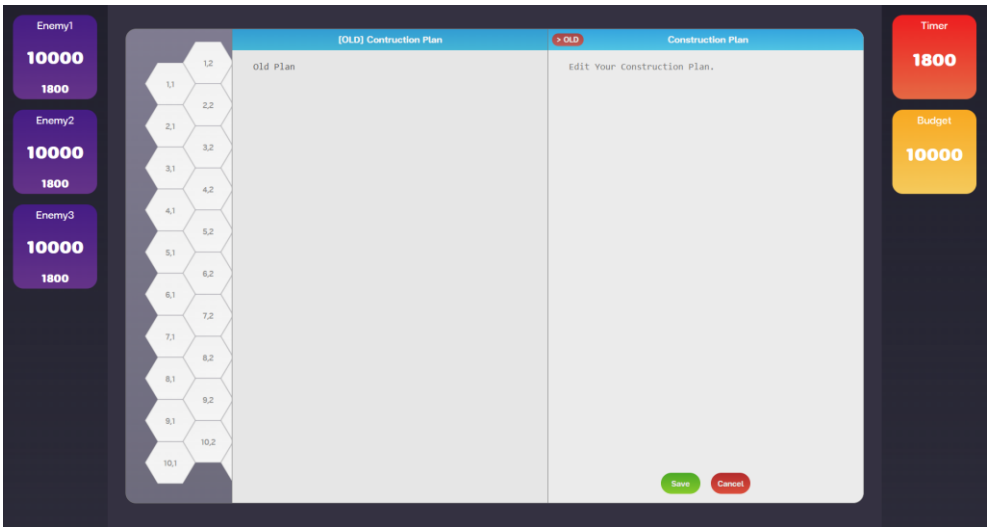
Homepage and Sign in System

(Cannot use an existing name.)

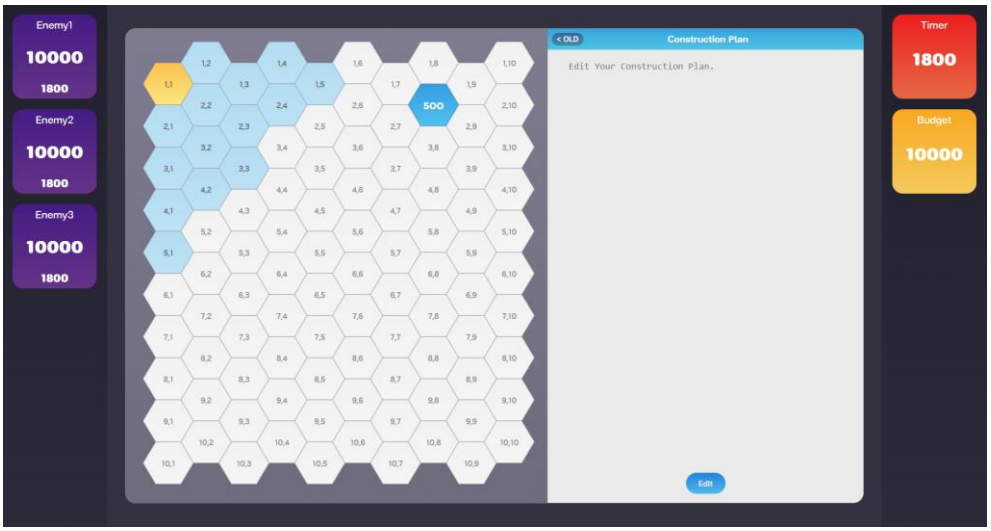
(Limited to a maximum of 4 players.)



Game Map



The Construction Plan



- In the table, white areas are not owned by player, while light blue areas indicate that they belong to player. When hovering the mouse over any cell, it will be highlighted in a darker blue color and the text will change from pair(x,y) to "Deposit" for that particular cell.

Controller

1. Game logic: The Controller would contain the logic for how the game is played, including how players can claim territories, what actions they can take on their turns.

2. Input handling: The Controller would handle player input, such as selecting territories to claim or executing other actions on their turns.

3. Game flow: The Controller would control the flow of the game, managing the turn order and advancing the game state based on player actions.

4. Validation: The Controller would validate player moves and ensure that they comply with the rules of the game, preventing any illegal moves or game-breaking scenarios.

5. Game state management: The Controller would manage the game state, including updating the Model with player actions and determining when the game has ended.

6. User interface updates: The Controller would update the View to reflect changes in the game state, such as displaying new territories claimed by players or updating player scores

A server-client implementation :

1. Server: The server would manage the game state, validate moves, and coordinate communications between the clients.

2. Clients: The clients would be responsible for rendering the game map and sending user input (e.g. moves) to the server.

3. Game state: The game state would include information about the current state of the game map, such as which tiles are claimed by which players.

4. Turn-based mechanics: The game would progress in turns, with each player taking a turn to make a move and the server validating the move before progressing to the next turn.

5. Claiming territory: The objective of the game would be for players to claim as much territory as possible, with the winner being the player who controls the majority of the map or shoots down all of the other player's citycenter.

some sample API endpoints that could be used for the server-client implementation:

POST /game/start - This route would initiate a new game, creating a new game state and assigning players to their respective starting positions.

GET /game/map - This route would return the current state of the game map, including which tiles are claimed by which players.

POST /game/move - This route would allow a player to make a move by specifying the tile they wish to claim. The server would validate the move and update the game state accordingly.

GET /game/status - This route would return the current status of the game, including which player's turn it is and the current score (i.e. the amount of territory claimed by each player).

POST /game/end - This route would end the game, calculate the final score, and declare a winner.

The input/output structures for these API endpoints, could include JSON objects with fields such as playerId, tileId, gameState, turnStatus, and winner.

Example of class:

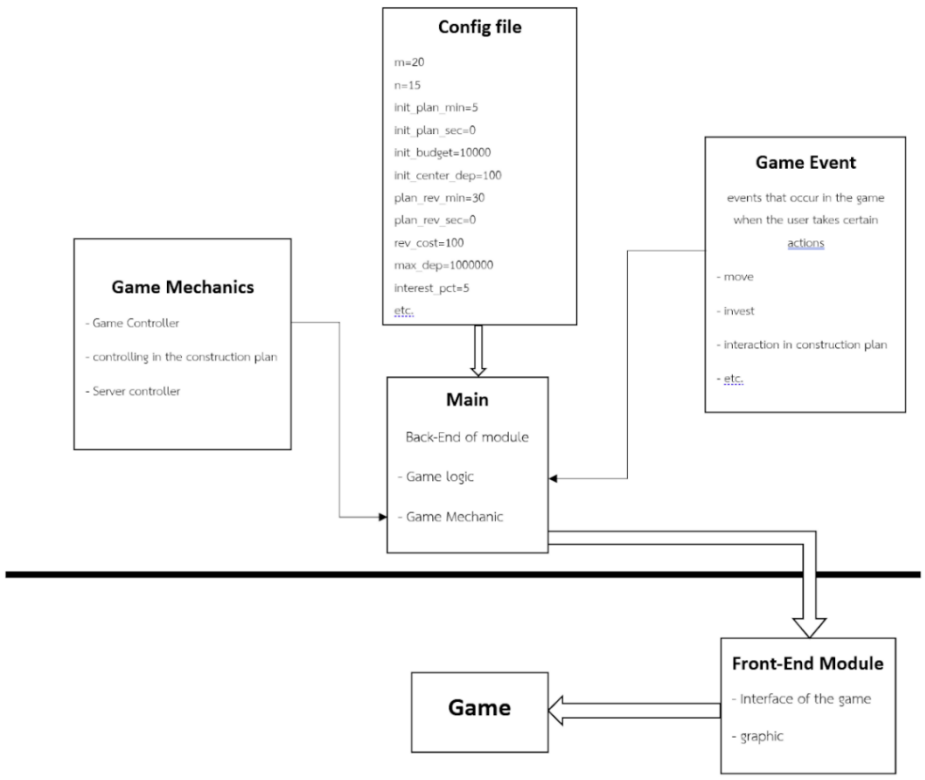
Model:

- Class Region
- Class Territory
- Class Unit
- Class Configuration
- Class Player
- Class PlayerManagement

View:

- Homepage
    - Sign-in field.
    - Show a list of joining players.
    - Show duplicate name rejection message.
    - Display a rejection message to accept more players when the number of players is full.
- Gameplay page
    - A Territory (game map) consisting of rows * columns Regions.
    - Each region shows its index and deposit.
    - Current/Revisiting Construction Plan window and Old Construction Plan window
    - Revisiting Time and Budget slots of our side.
    - Revisiting Time and Budget slots of the enemies.

Controller: As this game involves writing code to control units, the controller is therefore part of the parser and evaluator.



- key interfaces provided by classes, design patterns

- Player: This interface can define the methods that a player class or related classes should implement, such as move() , shoot(), and invest().

- Map: This interface can define the method that a map class or related classes should implement, such as printMapdata (),updateMap(),and checkifCityCenter().

- Game: This interface can define the method that a game or related classes should implement, such as startGame(),endGame(),and restartGame().

- discuss any important rep invariants maintained by classes

1. Construction plan :

a. Any variables that have not been used will always have a value of 0

b. After calculating the values in the construction plan in each turn, any variables that have a value in this turn will always have the same value in the next turn of the player.

2. Game Map: The game map class must maintain the correct representation of the game state, including the size of the map, the location of each player's unit, and any other important information required to determine the game state.

3. PlayerData: The Player class must maintain the correct representation of each player, including the player's name, the player's unit, and the player's current score.

4. UnitofPlayer: also in the palyer class ; must maintain the correct representation of unit, including its position on the map, the player who owns it, and any other attributes required to determine the state of the piece.

5. Move: The move class must maintain the correct representation of each move made by a player, including the starting position of the unit, the ending position of the unit, and any other information required to determine the outcome of the move.

6. Game Logic: The game logic class must maintain the correct representation of the game rules, including the rules for making a move, the rules for determining a winner, and any other rules required to play the game.

These rep invariants are essential for maintaining the correct representation of the game state and for ensuring that the game logic is executed correctly.

1.2 Code design

- data structures used

1. Arrays - To store the game map and keep track of the state of each territory.

2. HashMaps - To map players to their territories, and vice versa.

3. Queues - To implement the turn-based mechanics of the game and keep track of the order of players.

4. Linked Lists - To keep track of the moves made by each player during the game.

5. Sets - To keep track of territories claimed by each player and check for win conditions.

- discuss any tradeoffs you made between different design goals, such as between simplicity of code and efficiency of execution

1. Game map: a tradeoff between code simplicity and efficiency could be seen in the design of the game's map representation. One option is to use a simple 2D array to store the information about each region in the territory. This approach is simple to implement, but it may become inefficient as the size of the territory increases, because checking the state of each region in the map would require iterating through the entire array.

An alternative option is to use an object-oriented approach, where each cell in the map is represented as an object with properties such as its coordinates and owner. This approach makes it easier to manage the state of the region, but it also increases the complexity of the code, as there will be more objects to keep track of and more methods to write. The tradeoff between code simplicity and efficiency in this case would be between using a simple 2D array or using an object-oriented approach to represent the map.

2. In User interface: The user interface is another area where tradeoffs may need to be made between simplicity and efficiency. For example, using a simple text-based interface may be easier to implement, but using a more complex graphical interface may make the game more engaging and enjoyable for players.

3. Testing and debugging: there may be a tradeoff between the simplicity and efficiency of testing and debugging the game. For example, using simple code that is easy to understand may make it easier to identify and fix bugs, but using more complex code that is optimized for performance may make it more difficult to identify and fix issues.

1.3 Tool

- tools to use or plan to use for the design and implementation of the project

Team Management:

- GitHub

IDE

- IntelliJ idea
- JDK17
- gradle

UI Design

- Figma

Using GitHub is convenient for sharing programs among group members. It is used with IntelliJ IDEA (IDE) and JDK 17 for creating the project scripts. and use Figma to design UI for this game, and Figma for UI design

2. Testing

- how to test the various parts of the project ?

Parser-Evaluator

1. Testing stream with very long or very short input strings, including edge cases like empty strings or strings with only whitespace.

2. Testing stream with unusual or unexpected characters, such as non-ASCII characters, special characters like $ or #, or escape sequences like \n or \t.

3. Testing expressions with edge cases such as division by zero, undefined variables or functions, or invalid syntax.

Model

1. Test different map sizes: Test the model with different map sizes to ensure that it can handle different sizes and shapes of the playing field. Test the model with small, medium, and large map, and ensure that it correctly tracks each player's progress and updates the game state accordingly.

2. Test edge cases: Test the model with edge cases to ensure that it can handle unexpected scenarios. For example, test the model with a player attempting to make an illegal move or a player attempting to claim territory that is already owned by another player. Ensure that the model correctly handles these scenarios and enforces the rules of the game.

3. Test different strategies: Test the model with different player strategies to ensure that it can handle different playstyles. Such as Collect, Relocate, Shoot or the command like “nearby” and “opponent” .

- describe test will be used for testing this project

Player actions testing:

- Start game
- Restart game
- Exit game
- interaction in construction plan
- etc.

Event testing:

Test if the event that occurred in the game is correct or not(The event that occurred must be in accordance with the construction plan.)

- move
- invest
- shoot
- collect
- relocate
- etc.

Unit testing: This involves testing individual classes and methods to ensure that they function as expected.

Integration testing: This involves testing how different classes and components work together. This could involve testing how player movements and actions affect the game state.

Game play testing: This involves testing the game as a whole, including user interactions, game rules, and win/lose conditions. This could involve manual testing and automated testing using game bots.

Usability testing: This involves testing the game's user experience and user interface, including accessibility and ease of use.

- what did your group learn from testing, that you have not talked about in the last report?

We have learned about the testing process and how to create test cases for each method in various formats to check the functionality and ensure that it works correctly. This involves testing with different input values, By testing this project , developers can identify and fix any issues or bugs, improve gameplay experience, balance and fairness, performance and stability, player retention, and gain insights into player behavior.

3. Workplan

- explain the role of member(s) in your group responsible for this part of the project

Mr. Phumipaht

- parser-evaluator
- Backend developer

Mr. Phanarin

- UI Designer.
- Frontend Developer.

Mr. Teeraphat

- supporting on the design.
- tester

* Our group didn't change the division of work.

*The duration of work is based on the schedule specified in the project specifications

- what did our group learn from the process of dividing up the work

We have learned about breaking down the work into smaller parts, where each part has a responsible person to ensure that the work progresses and team members are not overloaded with work. This approach helps to improve work efficiency and prevents team members from being exhausted by taking responsibility for every part of the project. By having each team member responsible for their own assigned part, the project can progress smoothly and efficiently.