EGR 401/402 – Capstone Design and Presentation

Disputes and Diplomacy

By

Caleb Teasley

Christopher Trafton

Submitted To California Baptist University, College of Engineering
Professor Benjamin Sanders

1/15/2025

version 1.

Table of Contents

1 Problem Definition

1.1 Initial Problem Statement (as given by client)

The problem statement given to us by our clients is:

An application that can be used in a political science class to simulate international relations between students representing different countries.

The problem statement as we understand it is:

This is an application designed to simulate world interactions in a political science class. This app will put classmates into groups and assign them a country in which they will be in charge of. They will need to work to build their nation and military by using natural resources, trade, and conflict. This app will help give student insight into international relations between countries, and the different scenarios they find themselves in.

1.2 Client Suggestion Notes

- Encrypt messages used for sending game information to players or for saving the game to ensure that certain game functions cannot be exploited
- When developing the biome system, tiles that are close to each other should be more likely to be the same biome
- Use the client and server code from previous classes to establish multiplayer network
- Purchase or use home network for testing out multiplayer functionality

1.3 Client Interview Summary

1.3.1 User Classes (a.k.a., Personas)

1.3.1.1 Student/Player

1.3.2 Use Cases

1.3.2.1 Entertainment

1.3.2.2 Education

1.4 Background Research and Relevant Technology

1.4.1 Relevant Technology

There are many IDEs available for game development, but for our project we will be using IntelliJ. The game will be developed entirely with standard Java libraries such as JavaSwing. Other libraries like libGDX add more possibilities but would require a lot of development time just learning how to use the framework. For this project, JavaSwing will be able to do everything we need to accomplish our goals.

2 Objective Analysis

2.1 Problem Understanding

- Turn based gameplay and interactions
- Server connecting multiple student's devices together
- In game economy for resources and buildings
- UI that clearly represents gameplay
- Tile-based map with unique resources and advantages

2.2 Identify Two (or more) Potential Solutions

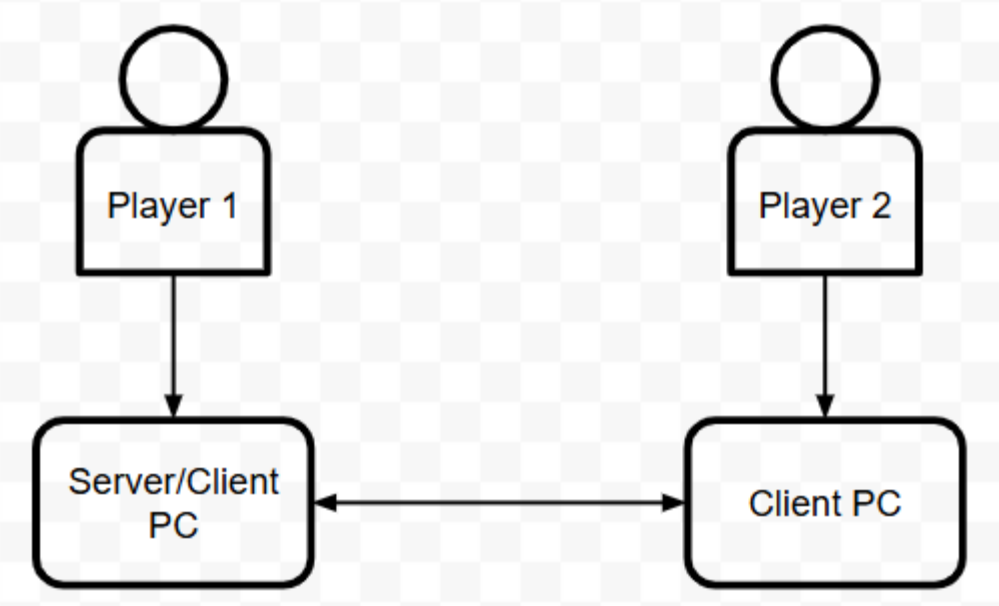
2.2.1 Desktop App that includes all above gameplay functionality

2.2.2 Web App that runs remotely on a server and is accessed through a search engine

2.3 Preferred Solution

The desktop app solution was selected because it is more familiar to us and provided more options for hosting and file storage. We will not have to rely on any external platforms by having the game run on the user's local machine.

2.4 Describe Solution Abstractions



- Player 1 uses a PC that has the server and client running on it
- Player 2 uses a PC that only has the client running on it
- Player 1 sets up and hosts the game
- Player 2 connects to the host
- Both players access and play in the same game

3 Development Methodology

3.1 Methodology

Game will run entirely within the confines of JavaSwing. Game development and versioning will be done via Github. Development structure will follow the AGILE sprint methodology.

3.2 Application of Methodology

Developers will push their code to the main branch whenever a new feature is completed. Developers will also pull any changes from the repository before making any new changes. This is done to make sure that local and online branches do not get desynced. Due to the team being very small, having multiple branches and testing environments is not necessary. Developers will push their code to the main branch whenever a new feature is completed. Due to the team being very small, having multiple branches and testing environments is not necessary.

3.3 Communication Mechanisms

3.3.1 Internal Communication Mechanisms (i.e., between team members)

- Main communication will be through a Discord group chat
- We will use GitHub to host our code so we can save and commit our changes
- A shared Google Drive to work on and store documents and presentations
- Jira Board will show us lists and graphs of work that needs to get done, is in progress, or is already completed

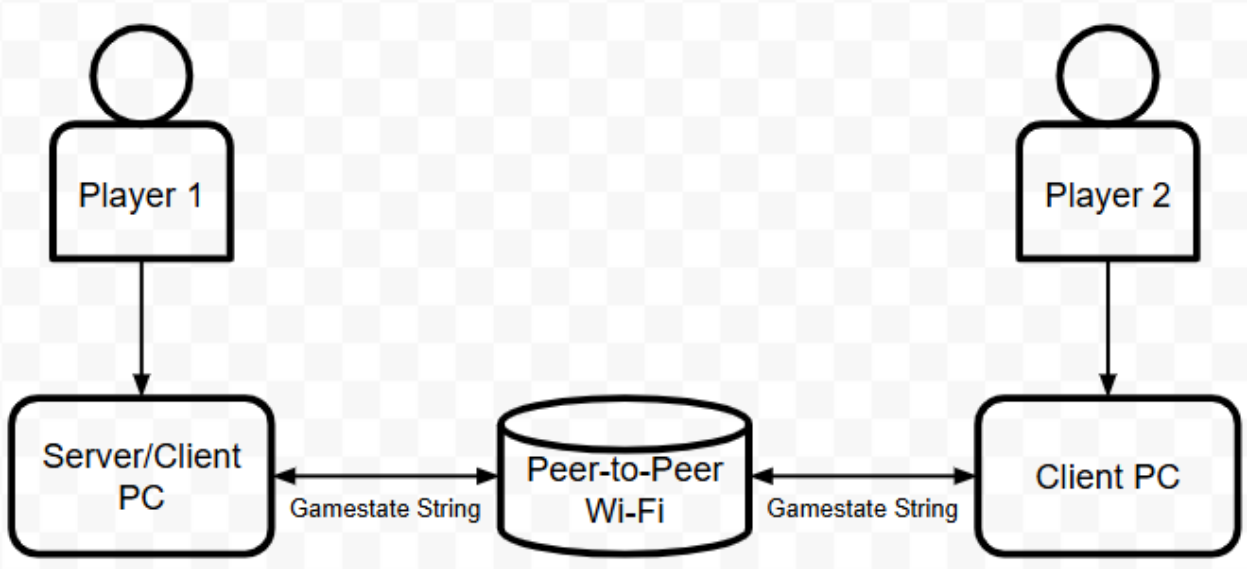
3.3.2 External Communication Mechanisms (i.e., with client, technical advisor, etc.)

- In class meetings
- Emails through Outlook

4 Requirements Specification (including Standards and Constraints)

- Any computer with a WIFI connection will be sufficient to run this software
- A network connection with Peer-to-peer connections enabled is required for multiplayer functionality
- Java is required to be installed on the computer to run the software

4.1 System Architecture



- Both players' PCs connect to a peer-to-peer Wi-Fi network
- Both players send strings of their gamestate to update the other player

4.2 SOFTWARE SPECIFICATION

4.2.1.1 Subsystem 1 - Game Board

- Creates game board and assigns the biome and resources

4.2.1.2 Subsystem 2 – User Interface

- Creates the UI windows along with all of their interactions and buttons

4.2.1.3 Subsystem 2 – Client/Server

- Creates the Server for the game to be run on two devices
- Connects the client to the server and sends gamestate info back and forth

4.2.2 Naming Conventions

- Classes – First-Letter Uppercase
- Objects and variables – Camel-case
- Functions – Camel-case
- Files – Words separated by underscore

4.2.3 Comment Methodology

Each major section of code should have a commented description. A major section includes class definitions, functions, threads, large loops. A specific style of comment is not required.

4.3 INTERFACE SPECIFICATION

- Main Menu - > New Game, Join Game, Quit
- New Game -> Input resources (Text Field), Nation Select (Dropdown menu), Start Game (function)
- Join Game -> Input host IP (Text field), Nation Select (Dropdown menu), Start Game (function)
- Quit -> Exit Application
- Start Game -> Game Board, Menu Bar
- Game Board -> Tile Select
- Tile Select -> Tile Menu

- Tile Menu -> Tile Info (data), Claim Tile (function), Build Menu, Train Menu, Command Menu
- Build Menu -> Build Building (function)
- Train Menu -> Train Military (function)
- Command Menu -> Move Military (function), Attack (function)
- Menu Bar -> Quit Menu, Nation Menu, Research Menu, Info Menu, Trade Menu, End Turn (function)
- Quit Menu -> Exit Application
- Nation Menu -> Nation Data (data)
- Research Menu -> Research Building (function), Research Military (function)
- Info Menu -> Military Statistics (data)
- Trade Menu -> Recipient (Dropdown menu), Give/Receive (radio button), Resource Values (Text field), Send (function)

4.4 Component Specification

4.4.1 Game Board

- Tiles/Map
- Mouse Tracker Panel

4.4.2 User Interface

- Main Menu
- Tile Menu
- Research Menu
- Nation Menu
- Trade Menu
- Construction Menu
- Military Menu
- Command Menu

4.4.3 Client/Server

- Player Handler
- Server Thread
- Client Thread

4.5 Data Specification

- Vectors (Tiles/Map, Military, Resources, Buildings)
- Tile Class
    - Contains information about a tile including owner, biome, military active there, resource values, buildings placed there, etc.
- Nation Class
    - Keeps track of all important data that a player needs such as resources they have, size of their military, what concepts they have researched, how many buildings they have
- Military Class
    - Defines a structure for military units (ranged, melee, siege) including their attack power and health

4.6 Algorithm Specification

4.6.1 Map Generation

The CreateBoard() method generates a game board by placing a specified number of tiles in a structured yet randomized manner. It starts with an initial tile at the center and iteratively attempts to place additional tiles in one of six possible adjacent locations. If a placement fails due to overlap or exceeding screen boundaries, it will try to place it at the next adjacent spot. If the process fails 6 times (each side is blocked), it retries with a different randomly chosen tile. Once all tiles are placed, each tile determines its neighbors, biome type, adjacency to water, resource rates, ownership, and available resources. This ensures a procedurally generated yet well-structured board layout.

4.6.2 Biome Generation

The generateBiome() method assigns a biome to a tile based on neighboring biome distribution and predefined constraints. It retrieves neighboring biome counts and assigns each a weight (count * 12), but penalizes any biome exceeding one-third of the total number of tiles. A total probability range is calculated, and a random selection within that range determines the biome assignment, favoring similarity to adjacent tiles while preventing biome overrepresentation. The selected biome's count is updated accordingly. If the selection falls outside expected ranges, an exception is thrown. This method ensures a balanced, natural-looking procedural biome distribution.

4.6.3 Attacking

This algorithm handles combat between military units from opposing factions within the game. It first categorizes units on a tile by their faction and stores them in separate vectors. Then, it initializes data models for displaying unit statistics and calculates each faction's total melee, ranged, and siege attack power along with corresponding health values. During combat, each unit type deals weighted damage to the opposing faction, depending on the attacking and defending unit classes (range, melee, siege),

reducing their total health accordingly. The faction with higher remaining health is declared the winner, and the defeated faction's units are marked as dead and removed from play. An audio cue is played based on whether the player's nation won or lost the battle.

4.6.4 Turn Management and Networking

This algorithm implements a server-client architecture for the game. The `ServerRunner` listens for incoming connections and assigns each player a `PlayerHandler`, which manages communication via sockets. Players can join the game, send moves, update the game board, and broadcast attack messages. The server maintains a synchronized list of active players and relays updates accordingly. The `ClientRunner` connects to the server, listens for messages, and processes updates such as board state, moves, and combat results. Clients send their actions when they reach the maximum allowed moves or when an attack occurs. This setup ensures real-time multiplayer interaction with synchronized game state updates. All messages between the server and client are sent as strings which can be parsed for a command and parameters.

5 Development Approach and Tools

5.1 Development Environment (IDE)

All development done in IntelliJ.

5.2 Source Control

We both work in the main branch, and push when we have an update with no errors, and stash then pull when someone else pushes to main.

5.2.1 Pushing to Main

We each work on separate instances of the project, and when one of us has code they would like to push to main, we first check to see if it will cause any errors or break anything. If not, then that person pushes their code to main.

5.2.2 Pulling From Main

Then the other person stashes their code changes that they have in progress, then they pull from main to get the updated version. Once that is done, they can unstash their changes and see if their new code has any errors, then continue working from there.

5.3 Hosting

All of our code and files will be saved and hosted on GitHub for each of us to push and pull from. Whereas our client/server will be hosted on a PC and Wi-Fi router.

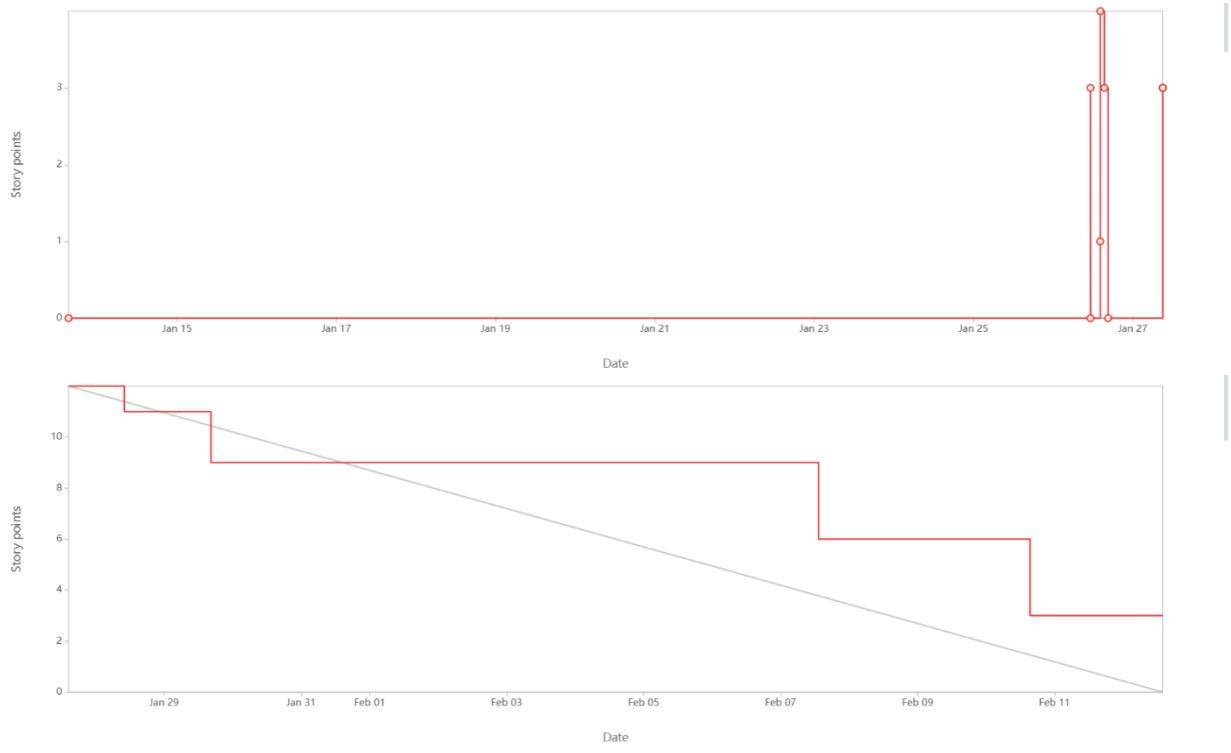
5.4 Release Process

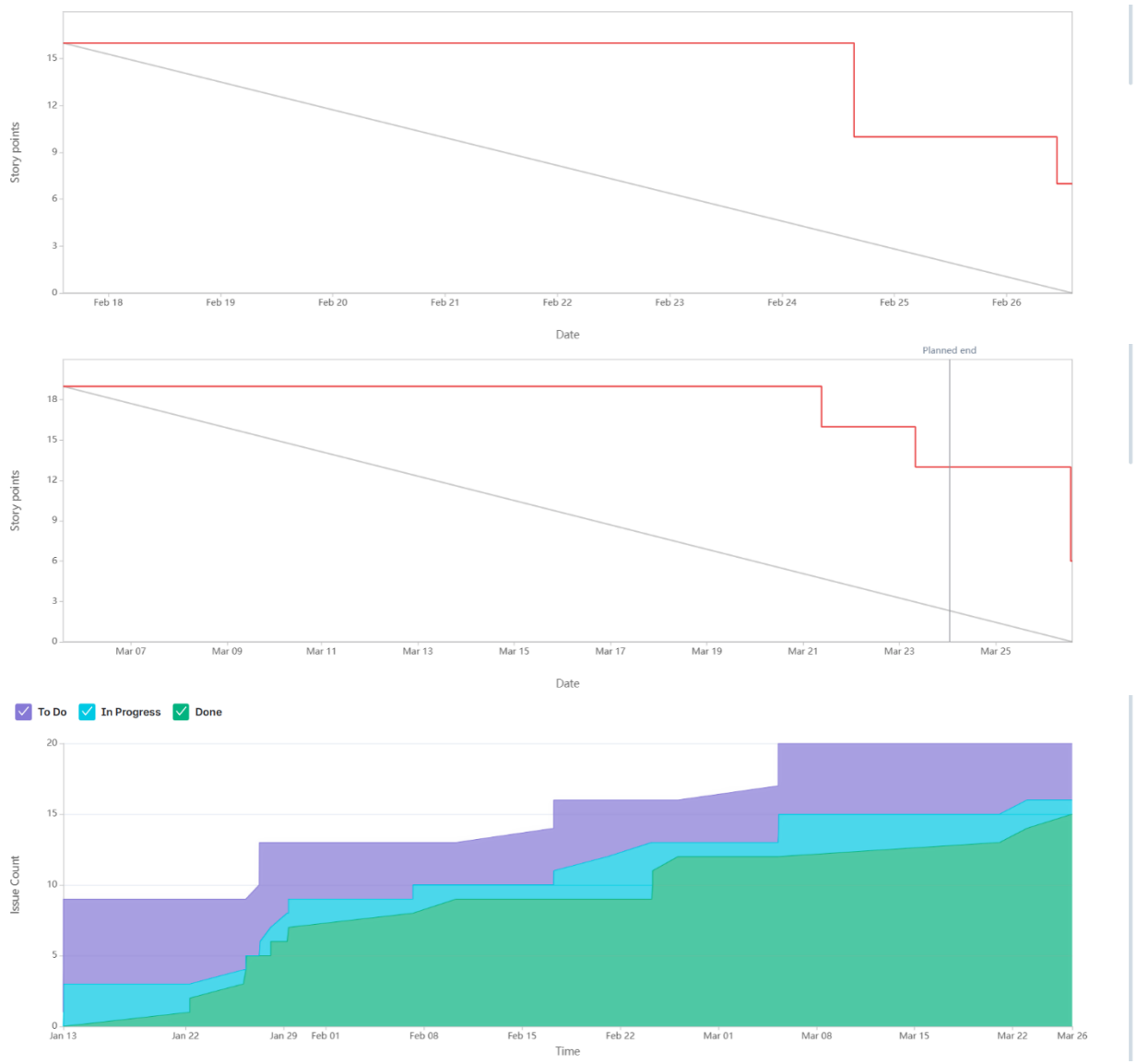
After new code is written, it will be checked in the current version to see if it causes any compilation errors. If not, then we check to see if it causes any runtime errors and that breaks functionality after the game has been compiled and is running. If no errors are found, then the code is pushed to mian for everyone else to pull from. If unknown errors occur and are fixed, then the person that wrote the code will submit and second push to fix them.

6 Project Schedule for EGR402 (weekly schedule)

All project planning is done through a Jira Board

6.1 EGR 402 Schedule History (Burndown Chart)





6.2 EGR 402 Schedule History Analysis

Starting off things went according to schedule, but eventually we got to a point where we went ahead of schedule. We would start developing features that were not initially planned for the sprint. There were also points towards the end of the project where we would plan to finish a feature, but would not be able to get it completely done, so it would get pushed to the next sprint. Overall, we eventually got most of what we wanted done for the project, except a few quality of life features we would have liked to do if we had more time.

7 BUDGET

7.1 ACTUAL COMPANY BUDGET

All development tools were free to use, so the expenditure for this project would only take employee salary into account. Hosting/publishing games can also be done for free in certain cases.

Calculation: Average Hourly Rate for a Game Developer in California * # of weeks * 9 hours/week

51.44 * 22 * 9 = $20,371.15

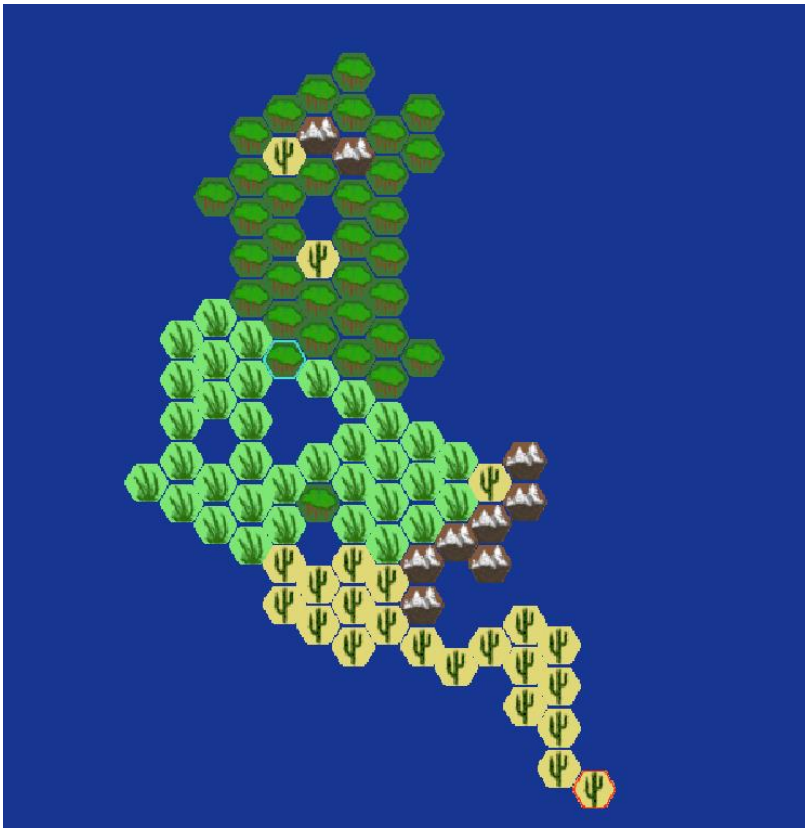
8 PROTOTYPING PROCESS

8.1 MAJOR COMPONENT PROTOTYPING

8.1.1 Game Board

The game board was initially developed separately from the rest of the user interface as it was a complicated window that had to be able to track mouse movements and presses. We started with building out the visual representation of the board using tiles so that we could see how the tile patterns were being randomly generated. After tile generation was at a point we deemed to be acceptable, we integrated different types of tiles that are differentiated by their biome, each having a unique representation. This allowed us to see that the board was being generated randomly both in shape and functionality. To integrate this component into the initial prototype, we figured out how to place the window with mouse tracking functionality as a subcomponent of the main game screen.

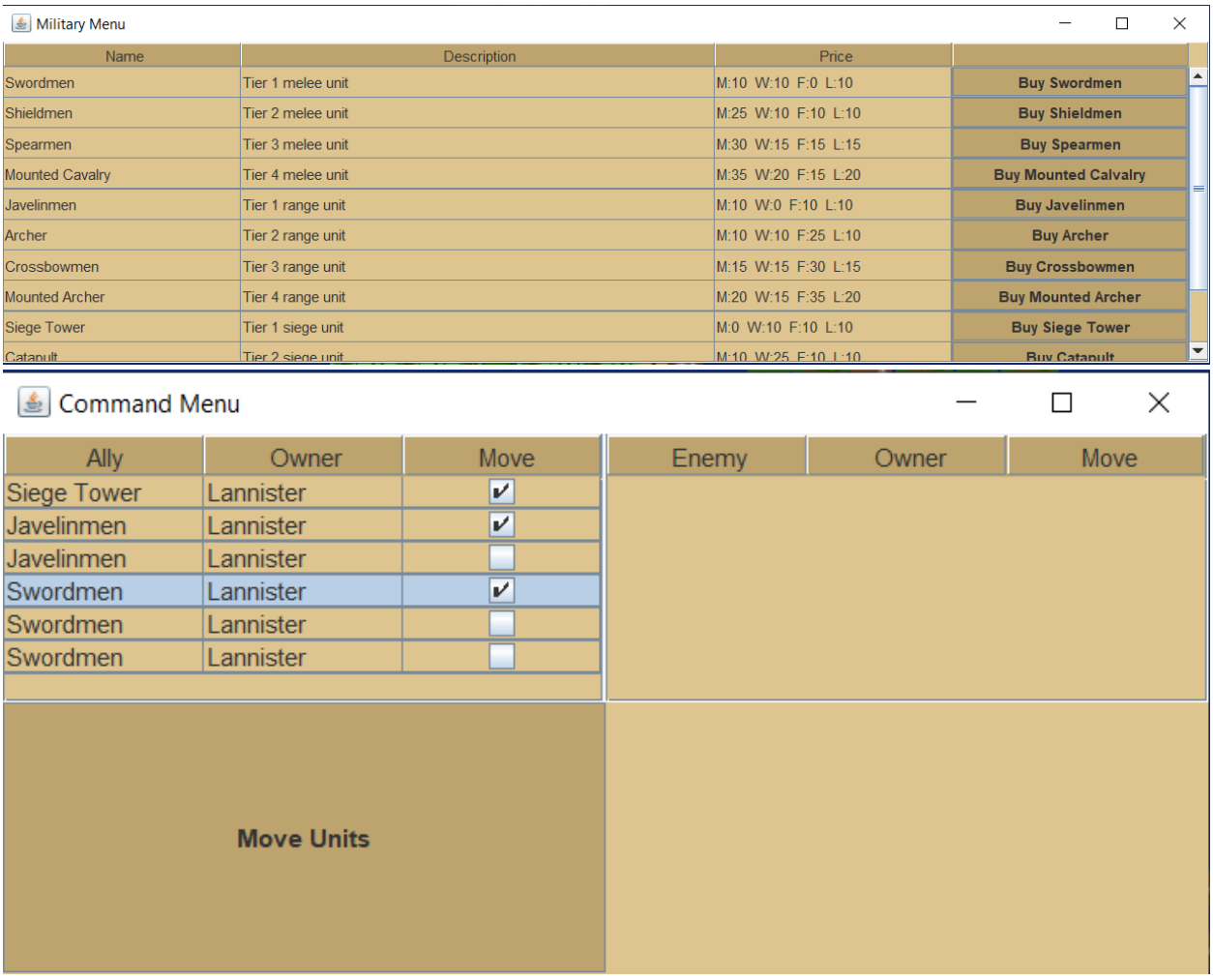
Game Board prototype:



8.1.2 User Interface

The UI was initially developed as a set of buttons attached to a menu bar that was placed at the top of the window. From there they were each given action listeners that would run a function every time the button is clicked. This would be the basic logic and design behind navigating the menu system. In the action listener functions a new pop-up window would be made with a combination of styling elements that the user can view or interact with. For example, on the buy screens, tables would be made with units and information, then have custom buttons placed inside the cells to allow for the interaction of buying something. These buttons would run a function that would decrease a user's resources and give them what they purchased. Other pop-up windows use text fields where the user can type in a string, which would be used in an associated action listener. Throughout the design process we would develop the basic layout of a pop-up window, then fill it with all of its UI elements, then added functionality and connect it to other windows and game features.

User Interface Prototype:



8.1.3 Client/Server

The network component of the project was developed as a new thread within the existing game structure. The backend network functionality was initially just used to connect two computers running the game, where they would just know the team that the user had joined. After this functionality was working, we expanded upon it, allowing the two connections to send information in the form of strings back and forth. We added the ability for the host to send the initial game board, which the other client would receive, parse, and rebuild the game board on their own instance. Using this same logic, whenever a user would make a change to the board by claiming a tile, adding/removing/moving military units, or placing buildings on a tile, this would be sent as a string that could be parsed and used to rebuild the game board with the updates. The final feature added using the network was live notifications of users being attacked by the other player, so they would know what was happening when it wasn't their turn.

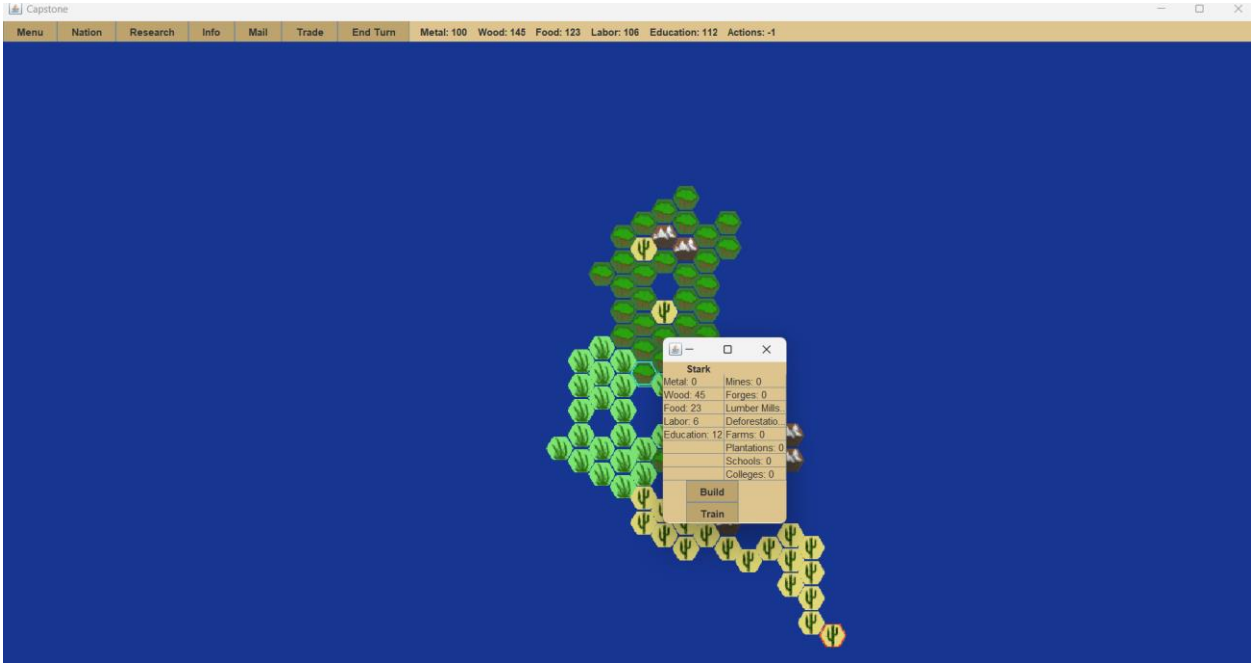
Server prototype output:

```
DND Server is running on port 12345
Stark has joined the game.
Lannister has joined the game.
Server: Requesting board
Client: Sending board
Server: Recieved board
```

8.2 Overall Integrated System Prototyping

Due to the way that JavaSwing works, integrating all of these separate pieces into a unified prototype was relatively simple. The user interface and menus served as the framework that the other pieces could be placed within. Other menus could also show up as separate windows on top of the current screen. The game board was able to placed as a panel inside of the main user interface window. The network was purely a backend protocol being ran on a separate thread, but that was still able to update and make changes to the main game window.

Complete prototype:



9 Testing and Verification

9.1 Major Component Testing/Verification

9.1.1 Game Board

The testing process was handled in a trial-and-error approach, where we would build out a specific function of the game board, then handle any irregularities or bugs that we discovered before trying to develop the next functionality. When designing the initial functionality of randomly generating blank tiles, we discovered that tiles would generate outside of the observable window (off the screen). This made it impossible to interact with that part of the board, so implemented restrictions that would prevent this from happening. When adding the functionality to randomly generate biomes, we spent time making sure that tiles close to each other were more likely to be the same biome but that there still a good amount of uniqueness every time a board is generated.

9.1.2 User Interface

A trial-and-error approach was used for testing the UI. When a new window is being developed, we would run the game and open the window to see how the UI elements look, then adjust the styling accordingly. We would repeat this process until we got a UI design that we liked. As for functionality, we would develop the action listeners and functions connected to buttons in the UI, then we would play test them in the game to see if their functionality is working properly. If they were not, then we would use debuggers and print statements to see where the function logic was going wrong, and if we were not linking something correctly or not passing a variably the right way.

9.1.3 Client/Server

Once again, a trial-and-error approach was used for testing the network component. Before continuing development, we would perform testing of each function to make sure that it was working as intended. Most of the network structure did not need much testing as it was working as intended from the beginning. Most of the testing process for this component was related to the parsing of messages sent between clients. The game board updates were sent by encoding the entire game board as a string that could be parsed. Due to the size and complexity of the game board, this became a large and tedious process to figure out where the parsing strategy was running into issues. After making many minor changes and testing to see the results, we were eventually able to reach a point where the strings were being parsed correctly, allowing for the effective updating of game boards between users.

9.2 Overall Integrated System Testing/Verification

Overall testing of the system was done by connecting two computers through the client/server, then by playing a game and testing all of the game functionality. We would each try doing the same process to see if the client/server is passing information correctly, then we would research, buy and train to see if the purchasing logic was working for both players. Then we would move units between tiles to see if both players can see the units move, and that no units were lost or duplicated in the process. We would attack tiles to see if the attack logic was working and would notify the other player. Then we would capture a tile to see if the other player gets the updated gameboard with the tile captured. Eventually we would have tried every possible UI interaction and feature and tested the connectivity to the other player.