Individual report

240032316

March 28, 2025

Word count: 1094

1 Individual contribution

1.1 Implemented requirements

Due to a deadline clash on my part I was unable to start work immediately, so the other developer implemented the client and a single-player version of the game which I then extended to multiple players (see Group Report: Team Working). Therefore, I was directly responsible for part 2 of the basic requirements, parts 1, 3-4 and 5 from intermediate, and parts 1 and 5 from advanced. Furthermore, I was responsible for any "multiplayer" components of other requirements that required communication between the clients, e.g I did not implement the logic in Game required for placing a fence, but I did implement the communication notifying other clients of this fence.

1.2 Design, development and management processes

1.2.1 Design

Before dividing responsibilities or writing code, my partner and I held two in-person meetings where we planned how to implement all basic, intermediate and advanced requirements. Our plan outlined the classes we would need and which class would be responsible for which requirement. 1 Our initial plan was good for high-level organisation of our code, but as the solution took shape and required more classes we found it difficult to align each class with a specific requirement as some classes accomplished nothing by themselves but were required for implementing another, e.g ServerConnection is required for Server but does nothing by itself. During production, we discarded the initial plan and focused on the requirements themselves (see Management).

1.2.2 Development

After I completed my section, I participated in daily pair-programming sessions designed to integrate these two sections. When integration bugs emerged, whoever was responsible for the section that crashed became the "navigator", explaining their part of the code and using breakpoints to determine whose section was ultimately responsible. The responsible developer for the bugged section became the "driver" and fixed the issue themselves, whilst the other "navigator" provided integration-specific suggestions and guidance.

1.2.3 Management

On a separate Google Doc, we split each requirement into smaller tasks and assigned them to each developer based on the section they corresponded to, e.g for implementing win/loss/draw statistics, I was responsible for tracking and sending a Player's statistics to the Leaderboard and my partner was responsible for displaying them. We could have used a more sophisticated project management tool (e.g a Git issue tracker), but as we were only two developers we found this method sufficient.

2 Individual challenges

2.1 Version control

We were required to use a non-bare Git for version control (origin) that was only accessable on lab computers. To work outside of the lab, I created a bare Github repository and added it as a remote to our non-bare Git. To upload work done outside the lab, I pushed to github, went to the labs, pulled from github and pushed to origin. This was a time-consuming process and I often forgot to push to origin, leading to merge conflicts when my partner pushed their work to origin. I frequently performed this process on my partner's machine during pair programming sessions, leading to my work being committed by them and producing a misleading Git log. At the time I did not have Jump server access so I could not SSH into origin from outside the university network. If I were to do this again I would have requested Jump service access far sooner.

2.2 Testing at the lab

Lab machines continued occupying a port after a program using it (e.g a localhost Server) was killed, leading to "Address already in use" errors when I tried to restart the program after making changes. This error did not occur on my personal

laptop, so I tested my section in isolation there. This approach proved impossible during pair programming which required a localhost Server to test the Client that my partner was implementing, meaning we had to wait for the port to be released or manually change ports which slowed down development. If I had VSC on my laptop, we could have pair-programmed and tested on my laptop to avoid this issue. In future projects, I will ensure I have the necessary tools on my personal laptop in case of situations like these.

3 Challenges during group work

3.1 Differing development methods

As my partner is on my degree programme, we took similar modules (in particular, CS5001 in Java) last semester and had a similar object-oriented programming style and professional experience implementing these ideas. Our degree program is small, so we knew each other well before this project and I felt we worked well together during pair programming sessions, communicated frequently outside of the lab, and contributed equally to the planning and execution of this project. Although I could not have asked for a better partner, we had some differences in our modes of development which had to be reconciled during pair-programming.

3.1.1 IDE vs terminal

My partner preferred using Visual Studio Code (VSC), whereas I preferred Vim and the terminal. As Vim is difficult to learn and use, we compromised by using Visual Studio Code for pair programming and I used Vim for individual work. This slowed down pair programming as I was unfamiliar with VSC and frequently had to ask my partner how to do something. This process taught me how to use VSC and I am now comfortable using it for pair-programming in future projects.

3.1.2 Differing programming conventions

Our code is a mixture of two programming conventions - my partner used camelCase for variable names but I used snake_case, I used name mangling to make some properties of objects protected or private and he did not, and we had different names for the same variables, e.g in the context of Grid, he used "dimX" whereas I used "length". This was a frequent source of integration bugs but were resolved quickly during pair programming as we could discuss the issue and decide on a common convention we would both use in this specific instance. In future projects, I will ensure all members agree on program-wide conventions before we started writing code.

3.2 Over-ambitious initial plan

Our initial plan included implementing every advanced requirement, and we produced highly modular and extensible code designed to achieve every requirement. As we approached the deadline, it became clear we would not have time for the more complicated requirements (e.g timer) and we had to cut them. Although this plan produced a solution that could easily be extended to meet these requirements, it was a waste of time and effort to implement classes and methods that were never used, e.g Player contained methods to store every Game that Player has played with the idea of implementing replays, which proved unecessary as we lacked time to implement replays. We should have focused our design on the basic and intermediate requirements first, then held a subsequent design meeting for advanced requirements based on our existing solution if we had time.

4 Appendix

CLIENT
−Basic 1: include "prospector" and student IDS
−Intermedite 3: require username first
    −Advanced 6: require password, await correct password from server,
only proceed if matching
    −Intermediate 1: include required username
−Basic 2: send an initialise request which joins an existing game with a game ID,
or start a new game if no ID provided and recieve new game ID
    −Intermediate 4: include optional size of grid
    −Advanced 1: include optional numbre of players
−Basic 4: user can change currently selected cell
−Basic 4: option to build fence, and send built fence to server
−Basic 4: input loop
    −Advanced 3: interruptable by timer
−Intermediate 5: return to initia
−Intermediate 5: leave current game on game close
    −Intermediate 5: inform server if client leaves
−Advanced 6: step through sent game history by acting as if client is playing both players

VIEW
−Basic 3: User friendly representation of grid
    −Intermediate 2: different colours
−Basic 4: Display currently selected cell
−Basic 4: take one character of input at a time
−Intermediate 5: leave current game key/button
−Advanced 3: display timer to user

SERVER
−Intermediate 3: send player object when login recieved
    −Advanced 6: refuse login requests if client supplies incorrect password
−Basic 2: recieve initialisation requests, look for existing game
or create a new instance of game, sends back game ID
    −Intermediate 4: recieve optional game dimensions
    −Advanced 1: tell client current players and maximum
−Basic 4: recieve fence requests, update game state, send updated game state to clients
−Intermediate 3: when game ends, update player's statistics with win/loss
−Advanced 3/4: Boot player from game if game says so

LEADERBOARD
−Intermediate 3: store dictionary of {player ID: player object}

PLAYER
−Intermediate 3: store wins, losses
    −Intermediate 1: store username
    −Advanced 6: store password
    −Intermediate 4a: store draws
−Advanced 5: store game objects of all games player

GAME
−Basic 2: initialise empty grid
    −Intermediate 4: pass optional grid dimensions
−Basic 3: store score

```
    -Basic 4a: update score if land claimed
    -Advanced 2: change score based on value of cell
-Basic 2: store current players in game
    -Advanced 1: host always starts. only start game when host plays.
    -Advanced 1: store maximum number of users in game
    -Advanced 1 (optional): send count of current number of players to client
-Basic 4: track turns
    -Advanced 3: tell timer when turn is up
    -Advanced 3: boot user from game when timer says timer is up
-Basic 4: build a fence
    -Basic 4a: check if fence completes a square of fences (no matter who owns fences);
claim land if yes, next turn if no
-Basic 5: check end conditions (all land claimed)
    -Intermediate 4a: check if game is draw
-Advanced 5: save turn actions to game history
-Advanced 6: once game is over, send game to all players

TIMER (component of server)
-Advanced 3: timer loops for each player in game
-Advanced 3: timer is paused when turn ends
-Advanced 3: if timer runs out, lose

GRID
-Basic 2: create empty grid of cells double -1 (e.g default 5x5 grid produces a 9x9 grid),
and set all uneven values of rows and columns as land
    -Intermediate 4: of dimensions optionally requested by game creator
-Basic 4: which cell is currently selected
    -Intermediate 2: send to view owners of cells
-Basic 4: change selected cell type to fence

CELL
-Basic 2: type can be none, land or fence. owner can be none, player 1 or player 2
    -Advanced 2: types can be regular, copper, silver, gold
```

Figure 1: Initial plan for implementing requirements.