Team Software Project Report

Monopoly 101

Dlo Bagari
Cathal Ahern
Adam Cunningham
Seán Bolton

For our project we were give the task of designing and implementing a multiplayer, server/client based, monopoly like board game. We formed groups of four. We were given three weeks to agree on a process, design and implement the game.

Tools

- Java Language
  -JavaFX
  -Networking sockets I/O
- IntelliJ - IDE
- Java Doc
- JUnit
- Graphic Design
  -Photoshop
  -Adobe Illustrator
- Git and GitHub – Version control/Project Management

We choose these tools for many reasons, either because we believed we would be using them on placement and hoped to learn more by using them in the project or because we believed they were the best choice in terms of usefulness and efficiency. We choose the Java language as it was the language each of us was the most familiar with. This saved time for all of us as we didn't need to spend much time trying to learn features to use in our work. The large selection of Java and JavaFX libraries available that was relevant to our work such as for the client/server and the GUI allowed us to more efficiently and effectively write code.

IntelliJ, which we were also familiar with, helped us to speed up our code writing thanks to its useful tools for code generation and debugging. IntelliJ also came with Java Doc and JUnit built in. Java Doc made use of automatically generating documentation from our code and making html files from the generated documentation. JUnit tools made it possible to quickly create and run tests.

Git and GitHub were very useful as both a project management and version control tools. We could access and add to a single version of our project and download it whenever we wished. This removed the problem of all of us having pieces of an incomplete project or multiple different versions of the project. This would have lead to confusion, mistakes and ultimately wasted time. It also meant we didn't have to constantly organise to meet in person to work on the project which saved us a lot of time.

Adobe Illustrator and Photoshop were used to create the images for our GUI. However, as none of us had very much experience with Photoshop it was very confusing and frustrating to use as we lost a lot of time trying to figure it out. On the other hand, Adobe Illustrator we found to be much more intuitive to use and this allowed us to easily and quickly create exactly what we wanted for our GUI.

Process

For this project we agreed to go with an agile process. Originally, we planned to work on the projects in distinct iterations with meetings and testing and the end of each iteration. The first iteration was to get us to the pass grade with only the basics of each feature implemented and each subsequent iteration was to increase our grade. When we realised that we had less time to work on the project than originally thought, we believed that this approach may no longer be a viable approach. Instead of focusing on distinct iterations we decided to focus on adding one competed feature to the project at a time which proved to be more effective and efficient for us.

In terms of work allocation it was decided that each team member would be given an area of the project to focus on. However, we wanted to keep everyone flexible and informed in all areas of the project. Each team member would help in other areas as needed. We believed that this would help reduce risks associated with team members being sick or missing. We also originally agreed that each team member would be responsible for their own documentation as they wrote code, however as code was

constantly being changed it became time consuming to also constantly change the associated documentation. In the end we agreed to work on all documentation at the end of the project together when the code was more likely to be stable.

Design and Implementation

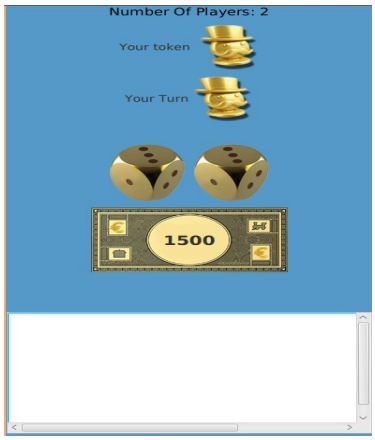
GUI:
We wanted to design the game so that it felt and looked like the original monopoly game as much as possible. We made this decision because player already were quite familiar with the rules and play style of the original game and because of this would not struggle with the mechanics we planned to implement. We agreed on a simple, 2D design for the GUI as a whole.

The first screen the players will see only contains two buttons, one to create a game and one to join a pre existing one.

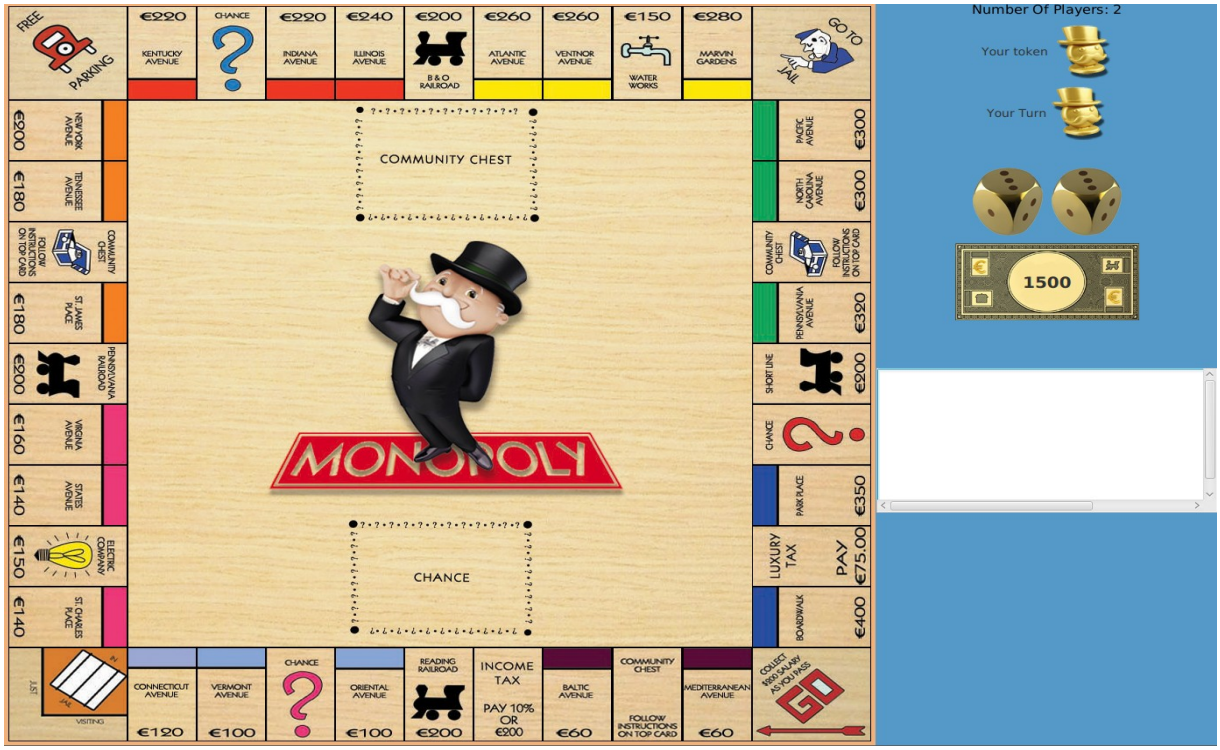
When a player has made their decision on the first screen they will be presented with a second, where they will chose their tokens to represent them on the board by clicking on the one they want.

The game will wait for eight players to join or the player who created the game to click start.

Once the game starts the players will be shown the third and final screen. On this screen, on the right, the player can see whose turn it is, interactive dice, their own funds and an event log.

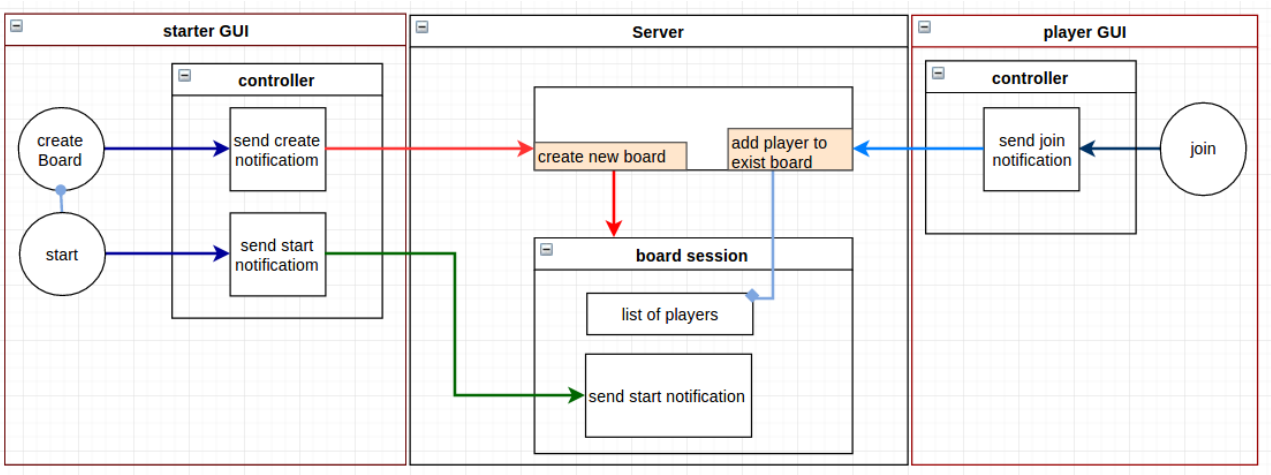


The dice will only appear when it is the player's turn and are clicked to receive the dice roll which will move the player's token on the board. When the player lands on a property the corresponding property card will appear allowing the player to decide whether to purchase or auction it. The name of the player that owns the property will appear below the tile. If a player lands on a card tile the card, community or chance, will appear in the appropriate card space on the board and wait for the player to accept the result. We decide to not allow the player to see the other players' data for simplicity as we did not believe we would have time to implement this. Below is the end result of the GUI design and implementation.



PreImageView:
This class is responsible for resizing player tokens on the board. If multiple players are on the same tile this class will shrink the tokens so that all players can be clearly seen on the tile. It will then resize the token as fewer players are on the tile.

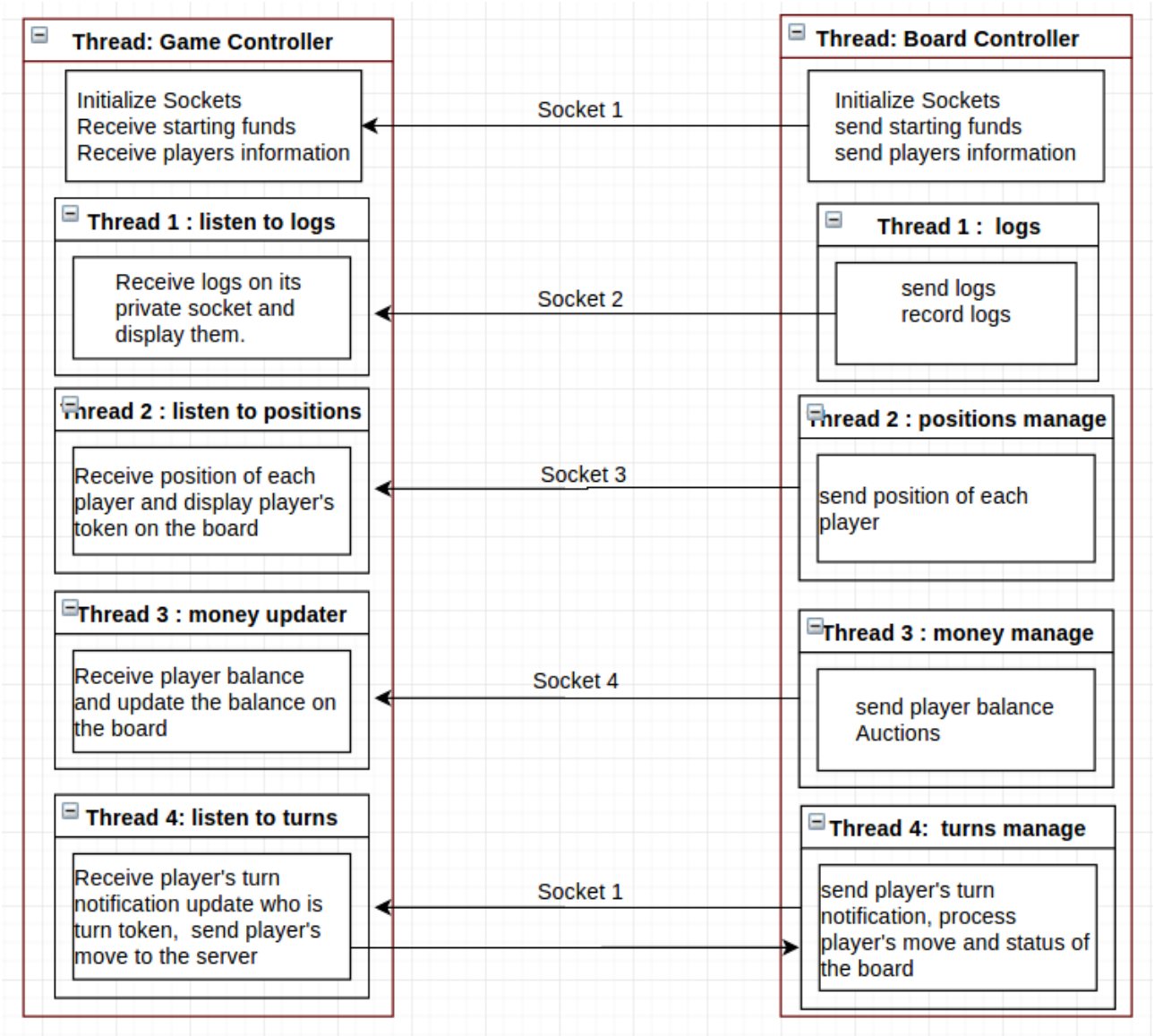
Server:
We decided to work with thread based servers. This allowed for multiple requests to be processed without stopping or slowing down games in progress. We decided on having one main server to act as a game lobby which would process clients as they connected. Based on the clients decisions this server would pass the client to an appropriate sub server (Board Controller) which hosted a unique game session. This sub server processed all requests for its hosted session. This allowed us to have multiple game sessions running at the same time and up to eight players in each. The main sever and the sub server can run on the same machine or on a seperate Java virtual machine. This gives the advantage of one game not interfering with another.



Port Generator:
We implemented this class to sequentially generate ports for our servers. This helps us to run multiple game sessions at the same time. This class is implemented as an enum class so that it can only have one instance at any one time.

Client:
Each client has four sockets to connect to a server with. We use four sockets to correspond with the four main aspects of the game: log, position, funds and board state. This allows multiple, unlimited requests to

be processed as quickly as possible without mixing communications. This gives a stable and smooth game for all players. This is represented in the diagram below. The client is also responsible for setting up games and receiving pre existing games and available game tokens and passes this information to the game controller which is a task. The game controller initialises the sockets and starts the threads.



Bank:

Each game session has its own unique bank. There is no communication between the player and the bank itself. If a player wishes to make a purchase, pay rent, etc. the board controller submits a job the bank which processes the request and returns the corresponding results. The bank can only process one request at a time. During this time the bank is locked and no other jobs can be submitted. All communication is done through the board controller (sub server). The back is responsible for keeping track of

all data in the game: player funds, properties, etc. It also initialises and keeps track of all properties including purchasing and mortgaging. It is responsible for all verification of these requests including verifying the player funds and if players already own properties. The bank also records all requests in a separate file for review after a game.

Dice:
The Dice class extends StackPane class which is a built in JavaFX library. It listens to mouse click events and returns a random number from one to six. This allows or interactive and animated dice.

Auctions:
When a player decides to pass on buying a property they landed on it gives other players the chance to purchase the property through an auction event. A window for each player will appear showing a countdown and the current offer on the property. Players will have the option to pass on placing any offer on the property or the option to increase on the current offer by $10. The last player to increase on the offer when the countdown reaches 0 will receive the property for that price.

Properties and Board Tiles:
Properties are data structures which contain integers which are status corresponding to owners, price, rent, etc. When a player lands on a tile these statuses are sent to the game controller (client). The game controller contains many options to run and who to run them on based on which

status it receives. E.g. -1 this property has no owner and is available to be purchased. Between 0 and 7 corresponds to who owns this property and player has to pay rent to this owner.

Cards:
Card is an abstract class from which all chance and community cards are sub classes of. Card contains an abstract method called process which is overridden and called on the sub class card the player receives after landing on a chance or community tile on the board.

Conclusion

In the end, we believe that the project was an overall success. We worked efficiently and effectively as a team with very few issues. We produced a playable and deployable product which was a close to the given specifications as possible in the allotted time. Our game has a stable server/client model and has had very few (if none) crashed or game breaking bugs during the course of our testing sessions.

There are still areas of the project that could be approved upon if given more time. For example, currently if a player leaves the game it cannot continue. To fix this we could implement a “heartbeat” with the client to prove they are still live to the server. If they time out, the server will remove them form the game or replace them with a bot. Some minor rules we were unable able to implement. Adding these would bring the game closer to an official, standard, monopoly game.

In conclusion, we believe this project has given us much insight into how a project in industry is planned, designed and implemented. This experience should help prepare us for our work placement and maybe even for going to the industry after graduating.