Software Development (202001064)
Programming (202001066)
January 2023

Programming Report – Exploding Kittens

Culda Alexandru Dan, s3022684, a.culda@student.utwente.nl

Overall Design

Hi there! Thanks for spending time grading my project. It was a ride, keeping in mind that I was a resitter (1) who did it alone (2) & took care of the protocol (3). I learnt a lot, especially about networking, and I am grateful for what I've accomplished.

Class Diagrams

Not needed, since I am alone.

A review of each requirement for the project

You should be able to play Exploding Kittens with at least 2 people over a network with a client & server application.

Correctly implemented. There can be even 4 players in the Game. The game handle them correctly and announces the winner in the end. This requirement spans over the entire project, from packages such as Client and Server to classes such as model.Game.

!VERY IMPORTANT!

You can find these lines in network.ClientHandler.

```
446                 case Protocol.PLAYMOVE -> {
447                     if (!timerActivated || dummyCounter == 1) {
448                         nopeCardsCounter = 0;
449                         resumeGameHasBeenCalled = 0;
450                         onCardPlayedStartNope(clientMsgSplitted);
451                     } else {
452                         if ( clientMsgSplitted[1].contains("nope") || clientMsgSplitted[1].contains("NOPE"))
453                             this.onNopeCardPlayed(clientMsgSplitted);
454                     }
455 //                      this.normalFlowOfGame(clientMsgSplitted);
```

If you want to play with a 7 seconds timer, keep it exactly like this.
If you want to play without it, comment 447 -> 454 and de-comment 455.

The client should be able to connect to a server, play a game and announce the winner in the end.

Correctly implemented. All cases are handled correctly, and the players are automatically removed from the game (Server closes the socket) when they exploded. Everything is handled correctly, from players deliberately sending "abort" commands to forced Terminal shutdowns. I stated this in the report, but, replying specifically to the *Report Template Guidelines*, this requirement spans over the Client, Server, Controller and Model classes. The Client connects to the Server (class) via ClientHandler. Everything is then moved to Controller, where the main loop of the game is (while(!gameHasWinner)).

The server should be able to host at least one game with 2 players, following the rules of Exploding Kittens, and determine the winner in the end.

This happens. The players can play over the network a full game of Exploding Kittens. The server also does input checking, such that it specifies if a player played a move even if it's not his turn, or played a wrong move (two example). During the protocol session, a 0-trust policy was adopted.

It is required to have a UI in the client in order to play the game. This may be a TUI, and it doesn't matter what it looks like.

My implementation is as a Textual User Interface (TUI).

Your application / codebase should be structured as an MVC application.

This section is touched in the upcoming explanation of the MVC pattern in my project. I have implemented this feature.

Your application should handle common exceptions and errors correctly during running (connection loss, invalid input)

The application handles common exceptions and errors. I have made custom exceptions that can be found in the network folder. As I said previously, the server checks users' input as all times.

Computer Player, You should have a computer player to play with. This player should at least only do valid actions.

I have sucessfully made a bot that can play moves. Simply put, this bot either plays a normal card (SHUFFLE, ATTACK, SKIP, SEE THE FUTURE) or draws if none of the aforementioned are in his hand.

Extra features

- Chat
    - A general chat was implemented in the server, when clients can write a message at any time, such that it is broadcasted to each player, even when the server has started the 7 seconds timer. For you to test this, you can type "chat <message>". Make sure to always type "chat" whenever you want to send a message to the server. I do not believe I deserve all 0.5 bonus points, because I have a vague impression that the chat should have been in a nice GUI where a function of 'Private messages' could also be implemented, but I do believe that at least some fractions of bonus points should be awarded.
- Lobby
    - The lobby is created upon the first player joining the server. Even though players cannot 'decide together to play a game of exploding kittens', my implementation of Lobby is more of a Waiting Room. That is, until the Server receives the specified number of players, the already-connected players can send messages for chat. Once the desired number of players is reached, the server automatically starts the game.
- Special Combos
    - Special Combos is a feature implemented in the server that extends the capabilities of Cat Cards to all Cards. If you want to play special combo, the format is:
        - 2-of-a-Kind: "play" <CARD> <CARD> <TARGET'S PLAYER NAME>. When this is played, the server automatically picks a card from the targeted player and puts it in player's hand (the one who played the 2 of a Kind).

        - 3-of-a-Kind: “play” <CARD> <CARD> <CARD> <TARGET’S PLAYER NAME> <DESIRED CARD>. If the desired card is in the target’s player hand, then the player who issued the command gets it. If not, all 3 cards are put into the discard pile.
    - **There are limitations**. No one can play EXPLODE, DEFUSE or NOPE.
        - EXPLODE, because that can only be drawn from the deck. At no point is a player having EXPLODE cards in their hand.
        - DEFUSE, because that is reserved only to instances when the players are exploding. It is the most valuable card in the game and I do not think anyone would trade it (especially 2 or even 3).
        - NOPE, because it is reserved only for playing NOPEs upon players issuing commands.
    - Server has all sorts of checks for these:
        - Targeted player does not exist (server checks by name)
        - Player has played two different cards (“play SHUFFLE ATTACK MARIA”)
        - Player gives only 4 arguments instead of 5 in a 3-of-a-Kind case.
    - I believe I deserve the full 0.5 bonus points for this feature.
- Protocol Maintainer
    - This is not an *implemented* extra feature, but I felt it was important to mention that I think I deserve 0.5 bonus points because I was the protocol maintainer, fact that can be checked by the TAs.
- Early Bird Submission
    - This is not an *implemented* extra feature, but I felt it was important to mention that I submitted the project before Wednesdsay, 31 January, 23:59. I believe I deserve 0.5 bonus points for this.

The Model-View-Controller Implementation

My project has two main folders: **local** and **network**.

The local folder contains the initial implementation of the Model View Controller pattern, covering the local game. That was used for laying down the foundation for the Controller and Model, as the View suffered great modifications.

Everything related to this project is in the network part. The network folder has many folders, as I will dissect them right now:

- Model, which contains everything related to the game logic, from checking each player’s move to initializing the starting deck. All model classes communicate with model.Game, which further on communicates with the controller.Controller. In model there is a class Player that has two implementations (those two classes extend Player): ComputerPlayer and HumanPlayer. The Deck is used for the deck pile, whereas the discard pile is just an ArrayList.
- Controller, which handles the main logic of the game in the start() method. There can be found an while statement ***while (!gameHasWinner),*** which basically handles the flow of the game. A wait-notify system has been used to wait for player’s input after a play has been made., where the wait happens in the abovementioned method and notify happens in the method that receives a command on the socket. The Controller, in my case, is in a class called GameController. As I said, this is where the

logic of the games moves after the server started. This is also where I implemented a wait-notify system.

- Client, which represents my Client who is connecting to the server. This has a view implemented, where everything that is prompted the Client's terminal (or Run window) is done through ClientTUI). Moreover, the ClientTUI processes whatever was inputted by the client. The class client.Client is used solely for communicating with the Server, from creating the connection, understanding whatever comes through the socket to shutting down.
- Server, which represents my Server who is accepting clients that are connecting to the server. The Server class is used for opening up the port and listening to it. If players connect, a ClientHandler instance is created for each client. From there onwards, the communication from the server to the client happens through ClientHandler, who has instances of Socket, BufferedWritter's, BufferedReader's and so on. The server folder also has a view part, which is used for printing the server log on the console (or Run window). Whatever is written there, it is made via calling methods from Server to ServerTUI.
- Protocol, a very important package, in which the Protocol class can be found. This was distributed by me and methods from it were called throughout the Server's and Client's implementation. Each time, for example, a Client wants to play a move, Client sends to ClientHandler a command preceding with Protocol.PLAYMOVE.

As such, let's see a normal flow of a sent command, in the middle of the game. A player wants to play a valid move (assume that). Talking about only classes: the client types something, and ClientTUI checks it and dissects it into commands. It then call classes from Client that do the respective command (*insert* for inserting the explode, *play* for playing a move and so on). Client sends the command on the socket to ClientHandler, which picks it up and sends it over for processing (still inside ClientHandler). The ClientHandler calls the appropriate method from Server, which then communicates the move to the controller. Controller calls playMove(String[] cmd) method from Game, which then checks if the move is valid and calls upon the appropriate methods.

User interface

The user interface has been adapted, through the help of a friend of mine, to accommodate the users that are not aware of the game's rules. For example, whenever a new player connects, he is granted with a message specifying the exact format of the command that he\she needs to type

```
Client started.

Attempting to connect to localhost/127.0.0.1:1234.
Connection to localhost/127.0.0.1 and port 1234 initialised.

SERVER: Welcome to the server. You have been put in a lobby. You are the first one.
SERVER: How many players do you want the game to have? Type "private" <from 2 to 5>
Type >>
```

In the above picture, the user is let known that he connected to the server, as well as the need of typing, for example, “private 2” if he desired to play against another player.

Another example of the mentioned idea is that, for example, if a player does not know how to play a *3 of a kind*, the server detects that prompts an informative text:

```
Type >> play rainbow rainbow rainbow catalina

SERVER: Did you want perhaps to play three of a kind?If so, format is <CARD> <CARD> <CARD> <TARGET PLAYER'S NAME> <DESIRED CARD>
```

Special TUIs or GUIs, with colors or other, more professional looks, were not implemented, due to the acknowledgment of this course’s goals: learning Java’s network and OOP concepts, not UI/UX.

Testing

Unit Testing

The chosen classes for unit testing were Game and Deck. Whilst Deck is not that complex, the generation of the deck of the Game has been tested throughouly. Two games were created, each with different number of players, such as the numbers of EXPLODE and special cards are correct. Checks were put in place, to check if each player gets the right number of cards and, for example, if each player has, at the starting point, a DEFUSE in their hand. The testing was conducted together with other classes, because there was a need to gather players (instances of Player) and decks (instances of Deck). All 7 implemented tests are passing, thus confirming the set-up of the deck given the amount of players is correct. The Game class was tested because it had the most important role in the game, as well as was the most complex class overall.

System Testing

Of course, the system, throughout its development, has been used an uncountable number of times. What are covered in this chapter are system tests made on the final product. As such:

- User interface: I have included a friend of mine in the test, asking him to play a game of Exploding Kittens. The outcome of the experiment was incredible. The friend got stuck at the first screen, where he needed to select the desired number of players into the game. As such, we tested each case of the game (need of reinserting the explode, need to pass a card as a response to FAVOR and so on). Now, the Game should have straight forward indications of the format of the commands, such as : "2 of a Kind format: ""<CARD> <CARD> <TARGET PLAYER'S NAME>" (whenever the player played, for example, a CATTERMELLON and a BEARD together).
- Gameplay mechanics: A lot of effort has been put into checking the game logic, especially whenever a player drew an EXPLODE but no DEFUSE was found in his/her hand. Among a lot of undesired behaviours, I could mention:
    - Incorrect handling of the remaining players after a player exploded;
    - Incorrect handling of the stacking of turns whenever an ATTACK was played on top of another ATTACK;
    - Incorrect handling of the announcing the winner of the game.

    - Incorrect handling of the SKIP card, where the server would still ad a Card in player's hand.
    - Incorrect handling of BOT's insertion whenever it draw an EXPLODE.
- Multi-Player Connectivity: A LOT of problems occurred on the network layer. Fortunately, because I did week 7's exercises, the setup of the sockets was easy. Linking it to the logic of the game was not. For example, I spent an entire day on a bug, trying to find out why the server receives an empty string (""), while the server did not send one (turned out I had an out.newLine() and out.flush() by mistake somewhere). Another example would be letting a player send a command whenever he needed to reply to a FAVOR, but not letting him to send a command when it was not his turn. Playing with indexes and making sure the ClientHandler instances match my logic side (I had a Player class for each ClientHandler, and the instances of Player's were used in the logic of the game) was not an easy job. I tested on a lot of variations, from clients sending commands with illegal number of parameters to clients sending commands when it was not their turn. Also, the server can handle any player that is disconnecting throughout the game.

Alternative flow testing could, in my case, be represented by various error handling mechanics put into place. There were made (by me) custom exceptions, in cases where, for example, the Server or the Client would be unavailable.

Concluding, the system has been rigorously tested, and has checks in place, among other aspects, for:

- Player playing a move when it is not his/her turn.
- Player playing an incorrect format move
- Player that needs to make a move disconnects
- Player that does not need to make a move disconnects
- Player playing invalid commands
- Player playing invalid moves (EX: Play DEFUSE; DEFUSE cannot be applied, but drawn by the server automatically if found in player's hand)

A DISCOVERED BUG THAT COULD NOT BE (EASILY) FIXED IN DUE TIME WAS WHEN A PLAYER EXPLODED. WHEN THE CURRENT PLAYER EXPLODES, THE PLAYER (ACTUAL HUMAN PLAYER) MUST TYPE IN A COMMAND AGAIN, SUCH AS "DRAW". ONLY AFTER THAT THE SERVER PROCEEDS. THE FLOW OF THE GAME CONTINUES NORMALLY, AND THE WINNER CAN BE ANNOUNCED. THIS HAPPENS ONLY IN THE VERSION OF THE GAME WHERE THERE IS A 7 SECONDS TIMER BEFORE EACH MOVE. WHEN THERE IS NO TIMER, THE GAME WORKS PERFECTLY.

Academic Skills Chapter

4.1 Time Management and Procrastination Avoidance

Being a resitter, the situation was different for me. Last year I passed the Programming exam with a high grade (8.2), but I failed at the networking part in the project. This happened due to personal circumstances, but my biggest mistake was not doing Week 7's exercises

completely. Nevertheless, this year the difference in how I tackled this project was notable different.

First (4) weeks were quite easy, as I was doing the exercises alongside my regular MOD6. I was doing the exercises, even though I was not supposed to, in order to remember the syntax of Java, mainly for the exam. Besides some problematic exercises that were not written in a clear and concise language, I did not face any issues. I kept my schedule that I submitted for Academic Skills up-to-date, and I tried as much as possible to follow it.

The upcoming weeks came with a drastic change in the workload, mainly because of the Mathematics exam (! Statistics are not easy) in Week 6. In regards to the project, I started to code since the evening it was introduced, making test classes, playing with various almost-stub implementations of the classes. The Model-View-Controller was implemented since day one in my project. This fact could be seen in the both milestones. I managed my time as best as I could, allocating more hours to study and almost none for leisure time. The Christmas Break was a relief, as I could stay with my parents a little bit. Week 7 through 10 were harsh. Having so many responsibilities project-wise on my shoulder, I needed to keep up with both the protocol and own implementation of the Game. I enjoyed a lot to offer help to my peers whenever I could, answering questions about the protocol and admitting my mistakes. The 9^{th} and 10^{th} week did not see my outside, as I was just coding for the project, fixing bugs (! Threads are problematic) and implementing even more of the rules of the network.

Time-management for the project, I would say, was well-thought from the start. As I said, I started created the folders since the day the students received the announcement. I worked towards both Milestones, as I knew I wanted to achieve a high grade by acquiring as many bonus points as possible. Week 5 and 6, the ones before the break, did not see me working much on the project. Week 7 got me studying for my MOD6 exams, while Week 8, 9 and especially 10 were dedicated totally towards the P-Project.

4.2 Strengths and Weakness

The Learning Journey, in my case, was… interesting. Weeks 1-6 did not teach me anything new. While the exercises were not mandatory, I still wanted to redo… at least some of them. And this decision benefited me. Nonetheless, I had a lot of moments in which I forgot Java OOP principles or syntax, but I remembered them as soon as I searched them on Google.

The biggest curve of the Learning Journey was in Week 7 and 8, when we had to do the networking exercises. Those were a real challenge, as last year I did not do them. While my strongest skills could still very much be in OOP principles, I learnt a lot about Servers, Sockets, Clients, ServerHandlers, ClientHandlers and so on. Participating in the lectures laid to a strong foundation of those principles, as I asked many (maybe too many) questions during the lectures. Being a student in my second year, I could see how easy things are if one would really understood them. Not seeing sockets and BufferedReaders and BufferedWritters for the first time were a real advantage. I remember last year, when I saw them in Horst C101, being totally lost. I am far from proficient now, but I could say that I am confident in my implementation of the network throughout the project. Checkpoint meetings, for me, were mostly about programming exercises, as I was (kind of) on track with

my schedule. I did not need any help towards it, as I knew my motivation throughout the module. Luckily, the motivation remained at a high level. I liked those checkpoint meetings, especially the TA. He was kind to me, making sure I am on track with the schedule with both my resit Programming course and the regular MOD6 one.

4.2 Checkpoint Meetings

The checkpoint meetings were of much benefit to me, mainly because I ‘clicked’ with the TA. Much about my schedule was not to say, as, I already said, I knew why I was doing and my motivation level was high throughout the module. Overall, the influence of the checkpoint meetings were positive, as I had a sense of having a support - a point of communication in case of demotivation or failure in some aspects. I could get valuable feedback from Codegrade and discuss it with the TA. The metacognition cycle helped me to keep track of my achievements, reminding me of what I had been accomplishing throughout the module.