Software Development (202001064)
Programming (202001066)

Programming Report – Exploding Kittens

Ugnius Tulaba, s-3274519, u.tulaba@student.utwente.nl
Ervinas Vilkaitis, s-3251942, e.vilkaitis@student.utwente.nl

2024/02/02

Overall Design

1.1. *Class Diagrams*

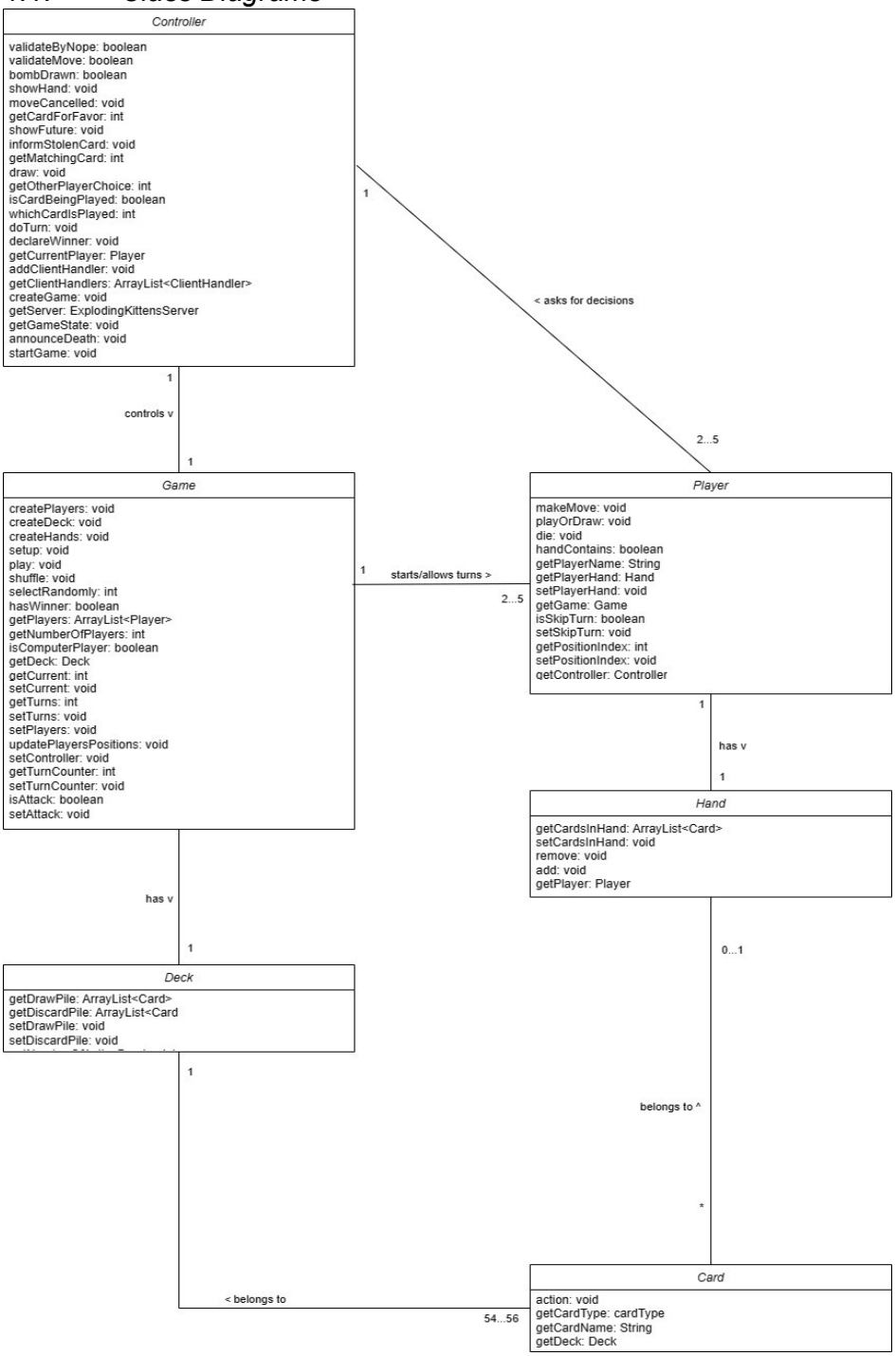

Figure 1: Partial class diagram for basic game logic

Figure 1:

In the class diagram above, you can get a basic understanding of how our game was constructed, what is the main logic, main relations between classes. The Controller is the class that is runnable by a user. It can be a local controller, or a server controller, which allows the user to play both local and network-based games. However, our game does not care what type of game is being played, that is the way we constructed it. The first step, controller creates a game with parameters received from the user, and from that point the main cycle of turns is run by the game class, which creates the players, the deck, the hands, and the cards through the deck.

All that is needed for gameplay is constant communication between the player class and the controller, because all user-related decisions that the player requires to proceed with its turn is acquired from the controller. When controller is called, depending on which type of controller it is, it has algorithms to find exactly what the human-player wants to do (by simply receiving input from console, or through communication via server).

It is not reflected in this class diagram, but the class 'Class' is only the interface that is shown here. There are nine different subclasses such as 'AttackCard', 'DefuseCard', 'RegularCard', etc. We did not see the point of illustrating this in the game logic, but each subclass is different in its 'action()' method, which depending on the card has different functionality.

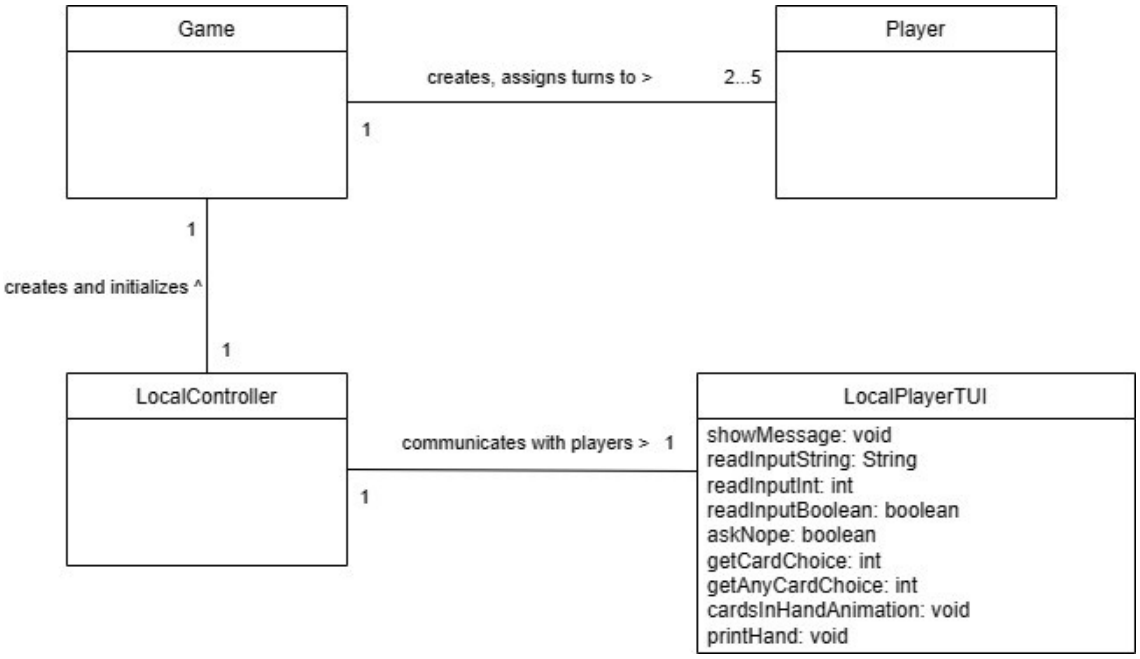

Figure 2: Partial class diagram for local gameplay

Figure 2:

This class diagram shows the main structure of local gameplay. It is only partial because not classes related to game logic are illustrated, but those can be seen in Figure 1 (you can treat Figure 1 as an extension of this diagram, just with 'LocalController' replacing interface class 'Controller').

Local gameplay is based on playing Exploding Kittens on one terminal console. This can be done by sharing the same console window between different human-players while passing around the device, or just challenging a computer player by yourself.

Local gameplay is using textual user interface, which can be found in the class 'LocalPlayerTUI' with its main functionalities being communicating information out to the player and receiving input from them.

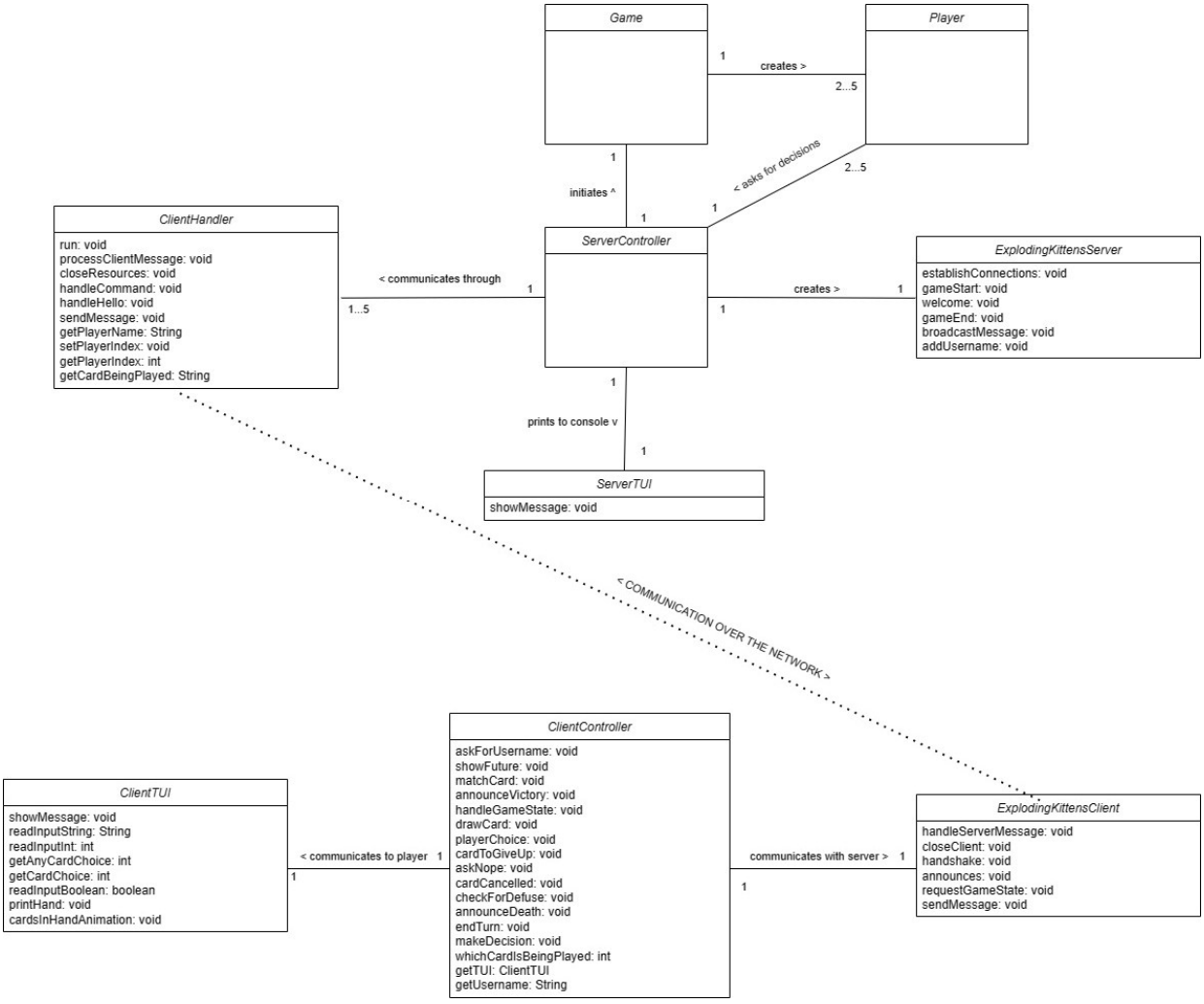

Figure 3: Partial class diagram for network gameplay

Figure 3:

This class diagram shows the structure of a network-based gameplay of Exploding Kittens. As you may notice, 'ServerController' is in a similar position as a 'LocalController' would have been if it

was a local gameplay. That is because it does the same things as the local controller, but to get input, decisions, etc. it is using server communication.

The Server 'ExplodingKittensServer' is responsible for creating server, establishing connections with the clients and each of these connections is being associated with a client handler 'ClientHandler' class, which is its own thread, and even has an additional thread to constantly read for messages coming from its client. The same applies for server's messages that need to be communicated to the clients 'ExplodingKittensClient'. The controller gets a specific client handler to send a secific message and expects a reply.

For network-based gameplay, the 'ServerController' class has to be run with specific program parameters discussed in the README.md file. Depending on how many clients are expected to connect to the server, the server waits until all of the clients are connected and ready, and then starts the game. Game then following the basic logic handles turns and 'ServerController' is being asked to deliver client's choices and decisions while also broadcasting what is being done to the other clients.

Now from the client's perspective, a client needs to run 'ClientController' class with specific program arguments, then is asked to provide his/her username and will be connected to the server.

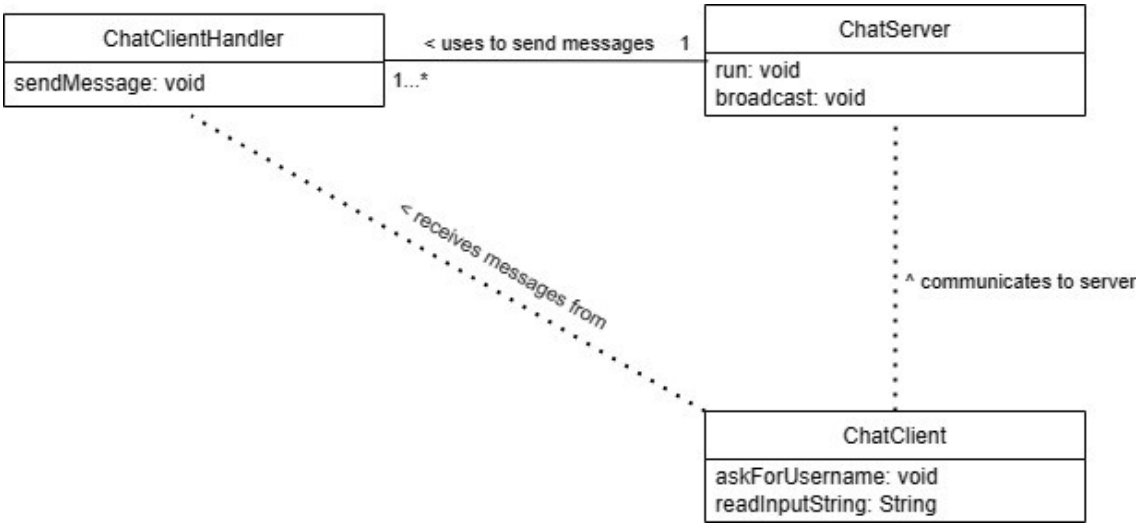

Figure 4: Class diagram for chat

Figure 4:

This class diagram represents the functionality of chat. It is part of our project, and is fully functional, however as you can see it is structured in a way that it functions using a separate server configuration. 'ChatServer' needs to be run, then however many 'ChatClient' instances can connect, and communicate with each other as in classical chat room. This structure is separate from the game, so whoever's hosting the game server, should also host the chat server, and have two terminal working at the same time.

From client's perspective, the client connects to game server through 'ClientController' class and then runs a 'ChatClient' class, which opens in a separate console. At any point while playing the game, the client can switch between these consoles and send or receive text messages from other players.

Lastly, chat can be used separately without even playing the game, which makes it functional regardless of the state that the game is in, even if it breaks down.

2.1. Requirements Implementation Map

In developing the Exploding Kittens application, the primary goal was to make sure that code follows all the required functionality. Every specification was implemented efficiently, and a smooth and engaging gaming experience was accomplished. Below every requirement is dicussed: the way in which it was achieved, and which classes were used.

Network gameplay - application supports playing Exploding Kittens over a network for at least two players.

Network gameplay in our project is sufficient and can host up to five clients, or up to four clients and a computer player. This is the most important and biggest requirement, which inolves almost all classes of the project. First of all, server is created using ServerController class, which evokes ExplodingKittensServer class. Clients can join using ClientController class and are distributed to a list of instances of ClientHandler class. When all intended clients connect and the handshake is completed, an instance of the class Game is creating passing arguments that are needed for the Game class to create instances of Player class, Deck class, and Hand class. Moreover the game deck consists of cards creating using Card interface class (our project has nine different card classes, which follow Card class interface). The game is set and ready to be started.

From that point, the game is running in a constant loop inside Game class, which sets turns to be taken by players until there is only one player left, who is considered to be the winner. In that case the game ends. Player class has main methods such as makeMove() and playOrDraw(), using which player takes turns. Game logic is based on constantly receiving input from players and making ceratin actions. That is achieved by Player class evoking methods in ServerController, which then send specific commands to the current player, and waits for the answer from ClientServer. Answer is proccessed and returned to Player class, which takes neccessary actions.

Client and Server Functionality - the client connects to a server to play the game, and the server hosts the game, ensuring smooth gameplay and announcing the winner.

Client and Server functionality is achieved using classes ServerController, ExplodingKittensServer, ClientHandler, ClientController and ClientServer. ExplodingKittensserver ensures to establish and keep track of connections to clients through a list of instances of ClientHandler class. ClientHandler class implements Runnable class, which makes it a separate thread from the main ServerController thread. Thus input from every client is constantly being read and processed by their own threads.
ServerController controls gameplay, processes and acts upon client input. This ensures smooth gameplay and in case when only one player is left alive, announces the winner and ends the game.
Keep in mind that the game logic itself is inside classes related to basic game logic: Game, Player, Deck, Hand, Card (AttackCard, DefuseCard, NopeCard, etc.). Besides, user input is received using ClientTUI class, which asks for specific input and handles in case it is invalid.

User Interface - included a user-friendly textual interface, making the game accessible and enjoyable for all players.

User Interface in our project is of textual type. Everything that needs to be communicated to a player is sent to his console in the form of text. The same as the output, the input needed from the player is received in textual for from the player. The player depending on the input needed, maybe be asked to provide answer in the form of boolean (his answer will be refactored to true or false values). This is achieved by asking the player to type in either 'y' letter for 'true' value, and 'n' letter for 'false'. Invalid inputs are being handled by informing the player that it is not valid, and getting him another chance to give input. The input will be constantly asked for until it is considered to be valid (either 'y' or 'n'). In a similar way integer input is handled, when a player is asked to type in a number. In most cases that number represents the index of another player, or the index of a card in his hand. Lastly, string input receives the whole line of text as a parameter, but the only place where this is used in the project is in the begginning stages, when player is asked to provide his/her nickname that will be used for the game.

PlayerTUI class is an interface with all of its methods directed to achieve textual user interface fucntionality. LocalPlayerTUI class implements it and is used in local gameplay. However, during network-based gameplay, PlayerTUI interface is not implemented in ClientTUI, because it differs significantly from local gameplay.

Structure - the entire codebase follows the MVC structure, ensuring a clean, organized project.

Structure of our project was constructed following MVC. MVC stands for Model-View-Controller, which is essentially a way of structuring your code so that the model holds all of the classes related to resources needed for the game such as Game, Player, Deck and Card. View package holds classes related to User Interface, and Controller package holds a Controller class, which connects the game logic and User Interface.

Our project is divided into four packages: client, exploding_kittens, local, server, and tests. Each package with the exception of tests, is structed following MVC. Package has subpackages: model, view, controller. In some packages model is not included, because it is not needed for example in case of a client. Client does not need any game logic resources to be able to play the game. Additionally, packages client and server have subpackages named chat, which include classes neccessary for chat functionality. The rest of the classes that are not in any of the MVC packages are game server related.

Error Handling - built in error and exception handling for a stable gaming experience, addressing potential issues like connection loss.

Error handling in our project consists mainly of exceptions used for invalid user input, and connection loss handling. In all of the cases (unless a bugg arises) when a user provides invalid input, it is examined and the user is asked again for valid input with some additional information given of what exactly is expected of it. Connection loss is handled in a way that that player is considered to be dead (having exploding from an Exploding Kitten) and the rest of the players are informed of this event. However, we find that in many cases when a player is in the middle of an action (while playing Attack card, Favor card, or just overall when a server is awaiting for some input) the game still crashes in those cases. Connection loss is 100% covered when it is that player's turn, and server is awaiting for his decision whether he wants to play a card or draw. If the player loses connection in that instance, no errors occur and gameplay is consistently smooth for the rest of the players.

Computer player - a computer player is available for solo gamers, capable of making valid game moves.

Our project has a class Computer, which extends Player class and covers the same functionality as the human player has. Our computer players is named 'RoboTukas' and is able to make all of the valid moves, play each card. When it is its turn, a random number between 1 and 0 is picked. In case the number is 1, a computer is considered to have made a decision that it wants to play a card. Then again, it picks a card from its hand in a completely random way. That card is then played. In case that number is 0, the computer player draws a card and ends its turn. In case the drawn card is an Exploding Kitten, computer plays a defuse if it has one, and puts the Exploding Kitten on the top of the draw pile for the next player to drawn

Card classes such as AttackCard, FavorCard, SeeCard and others have specific functionality in case the player that is playing them is an instance of a Computer class. For example, when a FavorCard is played by a human player, FavorCard class evokes controller methods such as getPlayerChoice() which asks the player for input on who does that player want to attack. Then, a method getCardForFavor() is evoked, which returns an integer index of the card in the victim's hand that will be given up. Now when FavorCard is played by a computer player, the card automatically randomly picks another player and evokes the method getCardForFavor() which then returns that player's choice. We found that this way of implementing computer player worked best, and ensured the least bugs and game crashes.

Chat (additional feature) - to enrich the gaming experience, a chat function is integrated, allowing players to communicate with each other during the game.

Chat was implemented in our project as one of the additional features. It can be found in client and server packages, in subpackages called chat. Client has a class ChatClient, which asks the user for a nickname to be used, and establishes a connection with the server. Server has ChatServer class and ChatClientHandler class. Server establishes connections with clients and is constantly reading for input. Whenever a client sends a textual message, it is broadcasted to the rest of the clients with the nickname in the front indicating which player sent a message.

Chat in a way is separate from our game, because it uses a separate server connection and can actually be used even when there is no game in place being played. We have made such a decision to implement it this way, so it does not interfere with the gameplay and is essentially pretty simple. The client is expected to have two consoles open, one for the gameplay and one for the chat. He is thus free to switch between these consoles or split his screen to be able to interact with both at the same time. The one flaw we see with this type of implementation is that we were not able to find a way to sync the nickname that a player is using in the game with the one he may be using in chat, because he is asked to provide both separately and independently. This is left for the client to ensure that he is using the same nickname in both, so that other players can identify which player this actually is.

3.1. *Model-View-Controller Implementation*

Our project is based on MVC design pattern, which helps with code maintenance if one will need to be done in the future and enhances understanding not only of the testers or developers who first see the code, but also the developers while building a bigger project. The main idea behind is that Model consists of all classes related to the bare project resources that are manipulated to achieve functionality. View consists of classes related to User Interface, communication with the user including output and

input information. Lastly, Controller consists of classes related to Controller of the project. This is the class that connects the resources (Model) and User Interface (View).

In our case, our project is divided into four main packages: client, exploding_kittens, server, and local. Each will be covered separately in hopes of better and clearer understanding:

Client:

Package consists of subpackages chat, controller, view and Client interface class with ExplodingKittensClient class. Client interface is a class following all the guidelines needed to fulfil the protocol used in our project. ExplodingKittensClient implements this class. ExplodingKittensClient is connected to ClientController class, by assigning it to a private variable in the constructor. ClientController class creates instances of ClientTUI and ExplodingKittensClient in its constructor thus ensuring that the client is connected to the main server connection and the client's textual user interface. ClientTUI does not have any associations with the other classes because it only returns values to ClientController, which evokes its methods.

In the subpackage chat, ChatClient is a class by itself and does not have any associations with the other classes in this package because it is a separate process.

Exploding_kittens:

This package contains subpackages model, view, and has an interface class Controller. In the model subpackage, all the neccessary game logic classes are based. Game class creates the game and has the main game loop, when the game is started. It has a variable with the value of Controller, whether it is a LocalController or ServerController (depending if the game is local or network-based). It is set through a setter setController(). Game class creates an instance of Deck class, which holds all of the cards of the game. It has a public access point through a getter getDeck() and has a variable with its value. Game also creates Hand class instances for each Player and assigns it to the Player class but has no direct association with Hand class. All of the players are placed inside a list and can be accessed through a getter getPlayers().

Hand class has a variable with the value of an instance of a Player class, which it 'belongs to' and is declared in the constructor through parameters. Deck class creates specific amount of instances of different Card classes such as AttackCard, NopeCard, DefuseCard and places them in a list (draw pile). Then, when hands are being creating, the cards are taken from this draw pile and added to list in the hands, which is considered to hold all the cards that a specific player has in his hand. Thus those cards are removed from the draw pile.

Player class has variables with values of the Game and Controller classes, which are assigned in the constructor. Also Player class has getters for both of these classes getController() and getGame(). Card classes have variables to hold Deck value, however it is not used since card classes have only one method called action() which receives Player class as an argument and is able to access the deck through getGame() and getDeck() getters.

Local:

This package consists of controller and view subpackages. Controller subpackage holds LocalController, which has associations with Game class and LocalPlayerTUI, the latter being in the view subpackage. Both of them are declared in the constructor of this class. LocalPlayerTUI does not

have any access to any of the other classes because it only returns values when LocalController evokes its methods in order to ensure successful and structured communication with the players.

Server:

This package consists of ExplodingKittensServer class, ClientHandler class, Server class, which is an interface meant to be used to fulfil the protocol and subpackages chat, controller, and view. Chat subpackage has ChatServer class and ChatClientHandler. ChatServer initializes server connection and essentially hosts the server for chat functionality. Each connected client is associated with ChatClientHandler class and is placed in a list with all chatClientHandlers.

Controller subpackage has ServerController class which has an instance of ExplodingKittensServer class through which server connection is maintained. This class also holds the list of instances of ClientHandlers class, which essesntialy run in separate threads. ServerController has an instance of Game class, because it creates it and assists it with providing user input through TUI needed for the resources of the game logic to ensure functionality. It also has a getter getServer() which connects the server with ClientHandler to fulfil handshake. ClientHandler has a variable associated with ServerController, because it needs to invoke methods of the controller upon specific messages received from the client.

ServerController also has an instance of ServerTUI class used to communicate the status of the server and the game to the server's console.

4.1. User Interface

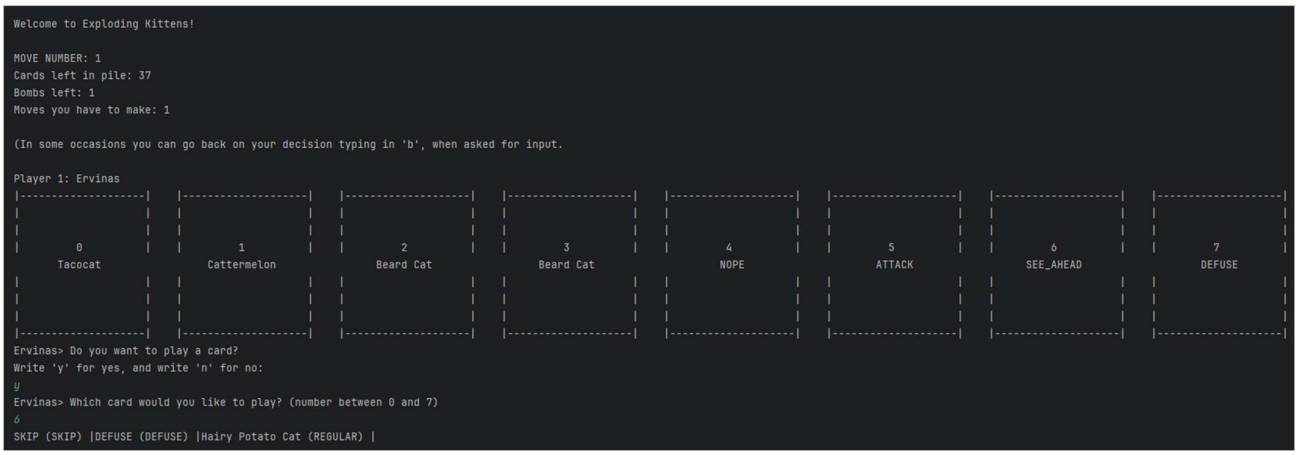

Figure 5: beginning of local “Exploding Kittens” game TUI

Figure 5:

In the provided screenshot above we can see a TUI of a beginning of a locally created “Exploding Kittens” game. At the very top we see a greeting “Welcome to Exploding Kittens!” indicating that the game is starting. Below we see what move of a game is being played, how many cards are left in the pile, how many bombs are left in the pile and how many moves a current player has to make. Next to “Player 1” we can see what player is about to make a move, in this case its “Ervinas” move. Below that we see cards display in textual interface with its name and index above it. Then the player is asked if he wants to play a card with possible answers of how he can answer the question using textual (String) input. After that he’s asked to provide another textual input (Integer) in order to determine which card does the player wants to play and action is executed and if needed text is displayed regarding the provided answer of a player.

Figure 6: use of a “Nope” card in a local game TUI

Figure 6:

In the provided screenshot above we can see the process of a “Nope” card being played. We can see that player “Ugnius” wants to play action card “Shuffle”. After that another player in this case “Ervinas” is being asked if he wants to play a “Nope” card if he types “y” a “Nope” card is being played. After “Nope” card is played it is displayed that action card is being cancelled. Therefore, the action of a action card is not performed and it is removed from players’ hand.

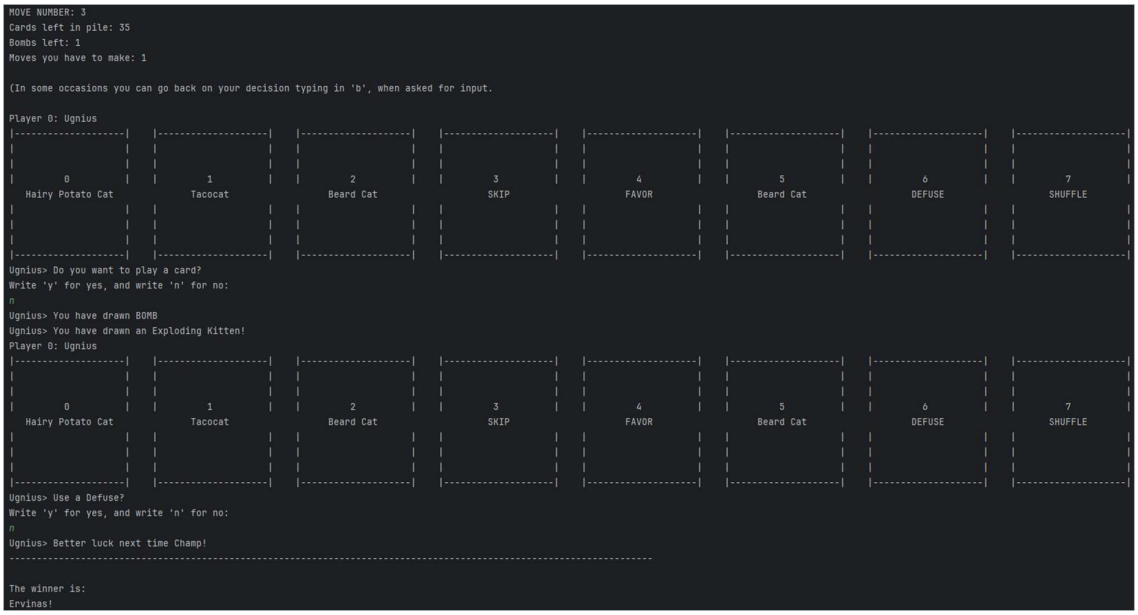

Figure 7: the end of a local Exploding Kittens game

Figure 7:

In the provided screenshot above we can see what is displayed when the game end and winner is anaunced. In this case when player “Ugnius’ drawn the “Bomb” he didn’t want to play defuse card which led into player “Ervinas” winning the game. When game is finished the winner is anaunced by “The winner is: “ and the name of the winner below.

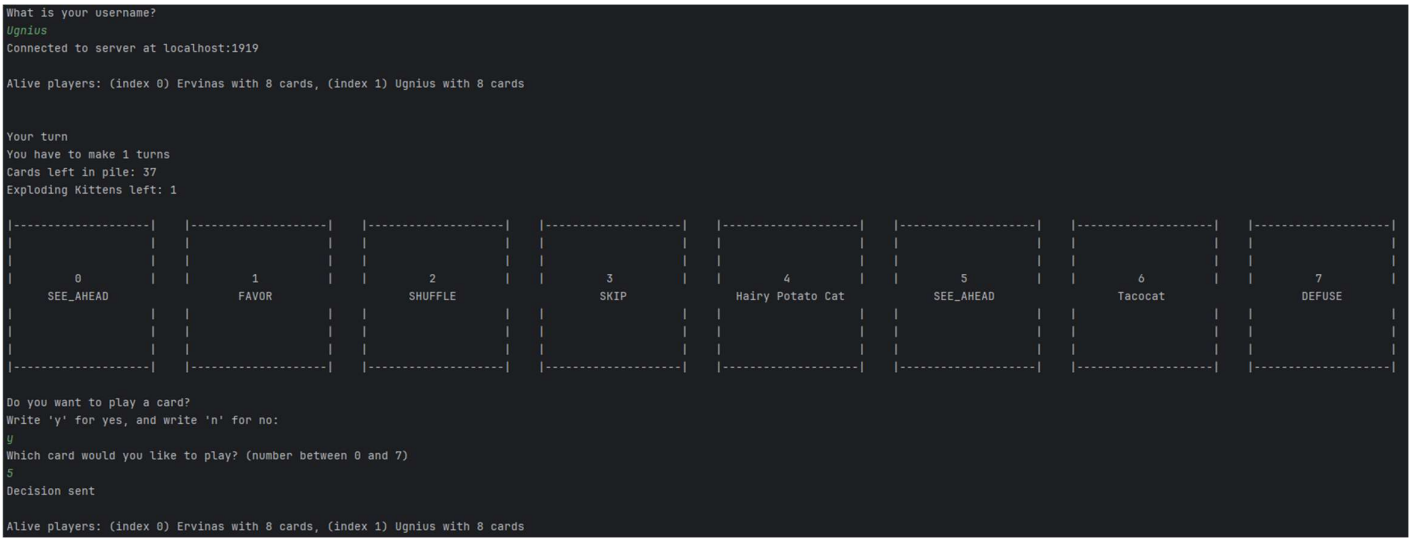

Figure 8: the start of a server Exploding Kittens game

Figure 8:

In the provided screenshot above we can difference between playing the game localy and on the server. It's basically the same, however some differences are: server first asks for a username of a player. After player types "username" it is also displayed to what server the player is connected. The rest TUI is the same except at the end we can see "desicion sent" as well as display of active players and how many card does each have. Additionally, at the beginning of a game played through server one players' TUI looks like provided in figure 8 the other ones' looks like provided in figure 9.

```
What is your username?
Ugnius
Connected to server at localhost:1919

Alive players: (index 0) Ugnius with 8 cards, (index 1) Ervinas with 8 cards

Player Ervinas is making his/her turn.

Alive players: (index 0) Ugnius with 8 cards, (index 1) Ervinas with 8 cards

Player Ervinas is making his/her turn.
```

Figure 9: the start of a server Exploding Kittens game

Figure 9:

In the provided screenshot above we can the difference between figure 8. It's also the beginning of a game, however, since in this case first player to make a move is not the one provided in figure 9, he has to wait for the other player to make a move. This is indicated by "Player Ervinas is making his/her turn.". After other player makes a turn then the same TUI appears as for the player in the figure 8.

Testing

2.1 Unit Testing

For software to be reliable, maintainable, and of high quality, testing game "Exploding kittens" code is essential. It enables safe code rewriting, early problem detection, and functionality verification for developers. Tests act as useful documentation, instructing novice developers on how to use the code. Code quality is increased overall by this strategy, which promotes more modular and maintainable code design.

Our tests were successful because of careful planning, we were able to provide a broad coverage of conventional situations and edge cases by utilizing JUnit to generate thorough test cases for each individual method. We were able to closely examine the behavior and functioning of our code separately.

Further down we have information which describes tests of our developed game for classe “Game” accordingly named “GameTest”.

“GameTest” puts the game's setup procedures to the test to the fullest, testing things like making players with the right position indices and nicknames, starting the game deck with the right amount of cards, and giving out hands to players who have the right number of cards. It also assesses important game logic, including identifying the existence of a winner under particular conditions, and utility functions, like choosing a random number within a range. “GameTest” guarantees that the game performs as intended. Additionally, “Game” class was tested in isolation to ensure modularity and independence of components. Furthermore, there’s each test looked into individually.

TestCreatePlayers - test verifies the game's ability to create players correctly. It checks that the setup method initializes a list of players that is not null, contains the correct number of players (3 in this case), and that each player has the expected nickname and position index. This ensures that the game can handle player setup correctly, including assigning unique identifiers and positions which are critical for game interactions and turn management.

TestCreateDeck - test focuses on validating the creation of the game deck during the setup phase. It ensures that after calling the setup method, the game deck is not null and contains the expected number of cards (30 cards, based on the assumption that the number is adjusted for the number of players). This test is crucial for confirming that the game begins with a complete set of cards necessary for play, adhering to the rules and ensuring fairness.

TestCreateHands - test checks that each player's hand is correctly initialized with the right number of cards (8 cards in this context). By ensuring that the setup method properly distributes the correct amount of cards to each player, this test confirms the game's capability to prepare players for the game, reflecting an essential step in starting the game under the correct conditions.

TestSelectRandomly - purpose of this test is to assess the functionality of selecting a random number within a specified range, which is a critical component in games for ensuring unpredictability and fairness in various game mechanics. The test verifies that the method returns a value that falls within the expected range (0 to 4 in this scenario), confirming the method's reliability for game operations that depend on randomness.

TestHasWinner - test evaluates the game's logic for determining if a winner has emerged. Initially, it asserts that the game should not identify a winner right after setup, reflecting the game's starting state. The test then simulates a scenario where all but one player are eliminated, and checks if the game accurately recognizes the presence of a winner. This test is vital for confirming the game's end-state logic, ensuring that the game can correctly conclude and declare a winner based on the rules.

Overall, “GameTest” class ensures that game runs smoothly without any unexpected errors or bugs in the “Game” class. The class “Game” is arguably the most important class, since it has the whole games’ logic which is crucial for the success of a well working game. Lastly, test coverege isn’t 100%, however it’s a big margin of that.

2.2 System Testing

To begin with, system testing was done by the authors of the project itself. Main goal at the beginning was to test everything that is related to the game logic, thus the game would work regarding to the rules. Firstly, we faced challenges while testing the game logic, which were to make all the cards to be

working as it should and doing actions that it is supposed to do, thus just by trying to play the game every error we would get meant that we have to implement something in the code. Main cards that we had most issues with was “Atack” and “Nope”. With the “Attack” card we’ve faced several problems such as not asking for the input of what player to attack, turns didn’t stack, after the action wrong players were asked to play his move. However, by implementing the code and changing the way “Attack” class worked issues were overcome. Lastly, for the game logic to fully work we were stuck with debugging “Nope” card algorithm. It started with the issue of asking all the players if they want to play “Nope” card. When we sorted the issue, the following one was to register the first player who said that he wants to play “Nope” card. Also, stacking “Nope” cards, as well as some actions gone missing. When we got sorted “Nope” card algorithm the game logic was working, and the game could be played locally.

In the last phase of our system testing, we concentrated on enhancing the robustness of our game by meticulously testing and handling incorrect player inputs, a critical aspect to ensuring a seamless user experience. Since our project has three inputs: Integer, Boolean, and String, by playing many games over and over again we managed to build up the code, thus it is able to handle wrong, unexpected inputs from players and give guidelines for the player to choose the right input for the game to work smoothly. Lastly, the main two exeptions are called “BackInputException” and “BooleanReturnException”. These exceptions were specifically designed to catch and manage incorrect inputs effectively.

To summarize, we encountered many bugs and errors while system testing regarding the wrong inputs or game logic, however everything is covered with exceptions mentioned in the paragraph before and implementations of game logic algorithms.

Academic Skills Chapter (Ugnius T)

4.1 Time Management and Procrastination Avoidance

To begin with, the weeks 2-4 helped me to manage my time better, which lead to better execution in weeks exercises – programming, system design and math, as well as gaining more knowledge in the same period of time compared to the first module. The plan and selected study material for each week helped to build theoretical knowledge and turn it into practical. Furthermore, doing the “Assignment of Academic Skills” I’ve found self-regulation mechanism called “Pomodoro technique”, which turned out to be super beneficial for increasing productivity and helping to keep procrastination at bay. Additionally, self-reflection helped me to understand what kind of subject I need to look into more and where to improve my planning to achieve even more efficiency in learning. Lastly, “Assignment of Academic Skills” equipped me with the skills to safeguard myself from becoming overwhelmed by a heavy workload, while simultaneously enabling me to accomplish more.

I proactively modified my techniques when my time management strategies did not produce the desired outcomes. I looked at changes for improved results because the Pomodoro approach and organized planning initially worked well for me. In order to handle unforeseen difficulties and workloads, I experimented with changing the attention intervals and including more flexible planning strategies. Even in the face of challenges, I was able to retain my productivity and efficiency because to this process of ongoing assessment and modification. I was able to get through periods of dissatisfaction by being adaptable and willing to change my strategy. This improved my ability to manage time and tasks and showed how important flexibility is to long-term success in both academic and personal efforts.

During the final two to three weeks of the project, it was critical to implement advanced time management techniques. Crucially important was the foundation built throughout the seven weeks of

prior knowledge acquisition. We implemented strict scheduling and methodically assigned jobs, giving priority to those that needed to be finished. This allowed us to maximize output. During this time, there was a lot of teamwork, rapid problem-solving, and frequent meetings to maintain alignment. By strategically allocating our time and striking a balance between task urgency and importance, we were able to effectively apply the theoretical knowledge we had learned in a real-world setting. This painstaking preparation and execution under strict time constraints highlighted the importance of efficient time management for project success and showed our capacity to adjust and perform well under duress.

4.2 Strengths and Weaknesses

My strongest abilities are system design and programming, which demonstrate a firm grasp and implementation of module principles, according to an analysis of my learning journey. On the other hand, I need to improve my knowledge in mathematical subjects and sophisticated theoretical ideas. In terms of academic abilities, efficient planning and the application of the metacognitive cycle were credited with fostering strengths in time management and self-regulated learning. I am aware, though, that I still need to improve my ability to identify flaws and get peer criticism. The checkpoint sessions were essential in helping me reevaluate and enhance my learning method by providing insightful feedback. I value the results of these sessions and want to get better by aggressively seeking out and using peer input more skillfully, so that future learning will be more all-encompassing and well-rounded.

4.3 Checkpoint Meetings

The checkpoint meetings with my mentor helped steer my progress during the course. These meetings gave me a priceless chance to think and get feedback, which had a direct impact on my development path. They made sure I followed the metacognition cycle, which allowed me to evaluate my knowledge and abilities every two weeks. The input I got during these sessions was really helpful, it not only confirmed my judgments of myself but also provided fresh viewpoints and methods for development. By integrating my mentor's suggestions into my learning plan, I was able to modify my approach as needed, which improved my productivity and understanding. Throughout the course, this mentoring element was essential to helping me stay focused and adaptable while also achieving my learning objectives.

Academic Skills Chapter (Ervinas V)

This chapter must be written **once for each member of the group**. These are individual reflections on the Learning Goals of Academic Skills.

5.1 Time Management and Procrastination Avoidance

To begin with, the first four weeks of the program greatly strengthened my ability to manage my time, which in turn improved my performance on weekly assignments involving programming, system design, and mathematics. The scope and depth of my knowledge increased significantly over this time, surpassing what I had gained in the first module. I was able to give each subject the proper amount of time thanks to the organized preparation and implementation of my study program, which guaranteed a balanced approach to theoretical and practical learning. This effective use of time promoted a more fruitful and structured study habit in addition to helping students grasp difficult subjects at a deeper level.

Upon realizing that my first-time management tactics were not producing the desired outcomes, I proactively revised my approach to improve efficiency. I experimented with changing the duration of my concentrated work sessions and incorporating more elastic planning strategies because I saw the advantages of organized but flexible planning and wanted to be better prepared for unforeseen obstacles and workloads. In spite of challenges, I was able to sustain high levels of productivity and effectiveness thanks to this practice of ongoing assessment and modification. I overcame times of poor performance by being flexible and open to changing my approach, which greatly improved my ability to manage my time and tasks.

During the final weeks of our project, it became critical to implement advanced time management techniques. The foundation built over the first seven weeks of intensive training came in very handy. We assigned work in a methodical manner, giving priority to those that were essential to do, and we implemented strict scheduling. This method really increased our productivity. Strong teamwork, prompt problem-solving, and frequent meetings to make sure everyone was in agreement characterized the time. We were able to effectively use our theoretical knowledge in the real world by managing our time well and striking a balance between the importance and urgency of the tasks at hand. Our careful preparation and timely execution highlighted how important time management is to meeting project milestones and showed that we can adapt and perform well under duress.

5.2 Strengths and Weaknesses

It's clear from looking back on my learning journey that my areas of strength are programming and mathematics, where I've shown a solid understanding and skillful application of topics. But system design stands out as a crucial area that needs work, indicating a weakness in my comprehension and implementation of these ideas. My proficiency with time management and my ability to use the metacognitive cycle to engage in self-regulated learning are two of my strongest academic traits. Nonetheless, I believe there is still much space for improvement in terms of applying peer critique wisely and correctly recognizing one's own shortcomings. Checkpoint sessions have been very helpful in reviewing and improving my learning strategy and in giving me critical input for advancement.

5.3 Checkpoint Meetings

My learning path was greatly influenced by the checkpoint conversations I had with my mentor. These exchanges were essential for getting helpful criticism and considering my development, which had a big influence on how I approached studying. By rigorously adhering to the metacognition cycle, I was able to assess my learning process every two weeks. My mentor's observations were crucial in helping me refine my self-evaluation by providing direction and clarity in areas that required work. I adjusted my study techniques to better achieve my goals by using the advice from these conversations, resulting in a more focused and efficient learning process. This mentorship was crucial in helping me stay on track with my academic objectives and in encouraging a flexible and adaptable strategy for conquering obstacles.