Assignment 2

Team number: CyberpunkHackingMinigame - Team 1
Team members

Name	**Student Nr.**	**Email**
Irakliy Marsagishvili	2637597	i.marsagishvili@student.vu.nl
Ivan Ivanov	2672214	i.g.ivanov@student.vu.nl
Maxim Abramov	2673452	m.abramov@student.vu.nl
Man Chung Stephen Kwan	2631645	m.c.s.kwan@student.vu.nl

Implemented features

ID	**Short name**	**Description**
F1	Timer	This game shall have a timer which limits the amount of time a player has to complete a stage.
F2	Buffer	The player shall be able to copy selected tiles from the matrix into the buffer.
F3	Code matrix	Code matrix is the central interactive object which shall contain tiles with values that the player should be able to collect and store inside the buffer.
F4	Goal sequence	The main goal of the game is to match tiles in the buffer to the given sequence.
F5	Tile selection	Each time the player selects a valid tile, the set of available tiles shall alternate between the vertical and horizontal lines containing the selected tile.
F6	Sequence completion	If a selected value is not a continuation of the given sequence, the progress of the sequence shall be temporarily stopped, until the player chooses the right value of the sequence. If the player completes the sequence, the game shall end.
F7	User-game interaction	A player shall be able to use their mouse to interact with the game (left click to select a tile).
F13	Quitting	The player shall be able to exit the game by clicking on the quit button.

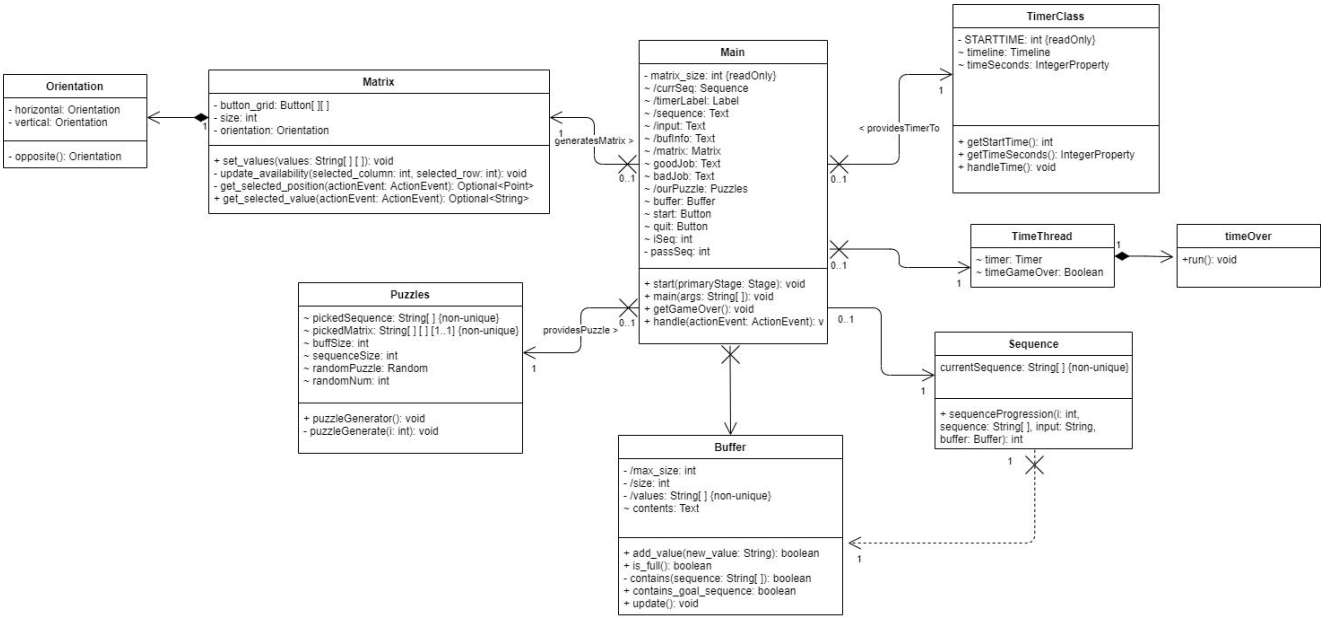
F15	Set of Puzzles	When the game has launched, the application shall generate a random puzzle from the set of pre-made puzzles.

Used modeling tool: Draw.io.

Class diagram

Author(s): Ivan Ivanov, Man Chung Stephen Kwan

This chapter contains the specification of the UML class diagram of your system, together with a textual description of all its elements.



https://drive.google.com/file/d/1f-I1zkbuhZ2bVRQzPbxvW9Wtahv3rWq2/view?usp=sharing (for readability)

Main

The Main class is the heart of the system and it connects to all other classes in our project. It is the entry point to the project. The Main class contains the code that creates the visuals in our game(the Start and Quit buttons, the Matrix, Timer, Buffer and Goal Sequence) and also the code which determines whether the player has won or lost the game.

1. Attributes (check about some with ta)
    a) *matrix_size: int* - When booting up the game the player will see the main playing field - the Code Matrix. This attribute sets the size of it.
    b) *timerLabel: Label* - A simple JFX label which contains the Timer in it.
    c) *sequence: Text* - A text node which contains the Goal Sequence which the player must match.
    d) *input: Text* - This node shows to the player the value in the Matrix node the player has pressed.
    e) *buffInfo: Text* - Here we show the size of the buffer.

f) *goodJob: Text* - A text node which contains the “You’re a Winner!” text when the player wins the game.
g) *badJob: Text* - A text node which contains the “Game Over!” text when the player loses the game.
h) *start: Button* - A JFX Button control. Once it’s been pressed the Timer gets started and the Code Matrix gets enabled.
i) *quit: Button* - A simple button which lets the user quit the game.
j) *iSeq: int* - Used for passing a value in sequenceProgression().
k) *passSeq: int* - This int is used for storing the value that determines the outcome of our game(Win or Game Over).

2. Operations
   a) *start(primaryStage: Stage): void* - This method acts as the main entry point for JavaFX application. It is called when the JavaFX application is started. The primaryStage parameter of type Stage is where the visual elements of a JFX application are displayed. In our case some of those elements are the Code Matrix, Timer and Buffer.
   b) *main(args: String[ ]): void* - This is another method that’s a part of JFX, it launches the application and can add command line parameters to it.
   c) *handle(actionEvent: ActionEvent): void* - This method handles some of the functionality in our game. It implements the JFX EventHandler interface. Its main function is to execute code which is associated with an event in our program(e.g. (add desc for 1 of them)).
3. Associations
   a) *Matrix* - The *Main* class uses *Matrix* in order to get the playable *Code Matrix* upon booting the game.
   b) *Puzzles* - *Main* uses *Puzzles* to snag a usable *Puzzle* and inserts it into *Matrix*.
   c) *Buffer* - The buffer is necessary in order to have one of the end-game conditions working.
   d) *TimerClass* - Used along with *TimeThread* to start a *Timer* and load it into the *timerLabel* attribute.
   e) *TimeThread* - Used along with *TimerClass*.
   f) *Sequence* - For the progress we’ve made in our sequence.

Matrix

This class’s main function is to generate the *Code Matrix*. Additionally it also fills out the matrix with values from the *Puzzles* class, disables the buttons which the user isn’t supposed to press at a given moment and provides the program with button coordinates and values.

1. Attributes
   a) *button_grid: Button[ ][ ]* - This makes a 2d Array of Buttons for the Matrix.
   b) *size: int* - Used for the size of the Matrix, value derived from Main.
   c) *orientation: Orientation* - Used for the orientation of the selectable tiles.(to remove?)
2. Operations

a) *set_values(values: String[ ] [ ]): void* - This method adds the values to the Code Matrix buttons. The values themselves are pre-made and contained in the Puzzles class.
b) *update_availability(selected_column: int, selected_row: int): void* - Disables buttons which are *not* the ones the user is supposed to press.
c) *get_selected_position(actionEvent: ActionEvent): Optional<Point>* - Returns the x,y coordinates of the pressed button in the Code Matrix, the event itself is registered through JavaFX's ActionEvent interface.
d) *get_selected_value(actionEvent: ActionEvent): Optional<String>* - Returns the values contained in the pressed buttons. Also through ActionEvent.

3. Associations
   a) *Orientation* - An enumerator that contains *vertical* and *horizontal,* used with *update_availability()* to have only the vertical or horizontal buttons near a previously chosen button be selectable for the next turn.

Puzzles

This class contains five handmade puzzles to be used in the game, handmade to not allow for any dead-end's to happen while playing the game.

1. Attributes
   a) *pickedSequence: String[ ]* - A string which passes the selected goal sequence.
   b) *pickedMatrix: String[ ] [ ]* - An attribute which passes the selected 2D array of values for the Code Matrix.
   c) *buffSize: int* - This attribute contains the pre-determined Buffer size for the selected puzzle.
   d) *sequenceSize: int* - Contains the size of the goal sequence.
   e) *randomPuzzle: Random* - Used for random selection of one of the puzzles, value is passed to puzzleGenerate().
   f) *randomNum: int* - Also used for the same function.
2. Operations
   a) *puzzleGenerator(): void* - This method generates a random number which is then passed as a parameter to puzzleGenerate() in order to select a puzzle.
   b) *puzzleGenerate(i: int): void* - This method provides the actual puzzle to the game.

Sequence

This class handles the logic behind our progress in the game's Goal Sequence and checks whether the buffer is full or the player can continue.

1. Attributes
   a) *currentSequence: String[ ]* - Holds the picked sequence from *Puzzles.*
2. Operations
   a) sequenceProgression(i: int, sequence: String[ ], input: String, buffer: Buffer): int - The method checks the user's input and its return value *pass* tells main if the user still has space in the buffer or not.
3. Associations
   a) Buffer - Checks if the buffer is full, triggers an end-game condition if it is.

Buffer

This class creates and handles the *Buffer* in the game. The *Buffer* acts as a limiter to the number of buttons the user can select in a game session.

1. Attributes
    a) *max_size: int* - Contains the maximum size of the buffer.
    b) *size: int* - Contains the current amount that the buffer can store at that moment.
    c) *values: String[ ]* - The values currently being stored in the buffer.
    d) *contents: Text* - The text our *Buffer* is showing.
2. Operations
    a) *add_value(new_value: String): boolean* - Adds at the end the latest value the user has selected.
    b) *is_full(): boolean* - Notifies when the buffer has filled up.
    c) *contains(sequence: String[ ]): boolean* - Parses a sequence and checks if it is contained in the buffer.
    d) *contains_goal_sequence: boolean* - Checks if the goal sequence is contained in the buffer.
    e) *update(): void* - Updates the buffer.

TimerClass

This class, together with TimeThread, handles the Timer in our game.

1. Attributes
    a) *STARTTIME: int* - The attribute contains the number of seconds the player will be limited to.
    b) *timeline: Timeline* - Used for animating the *Timer*.
    c) *timeSeconds: IntegerProperty* - Wraps the STARTTIME integer for use with the timeline.
2. Operations
    a) *getStartTime(): int* - Returns the time limit value.
    b) *getTimeSeconds(): IntegerProperty* - Returns the timeSeconds attribute. Main adds this value to the *Timer* label.
    c) *handleTime(): void* - Stops the timer when it reaches zero and generally updates the time in real time to the game with *getKeyFrames*.

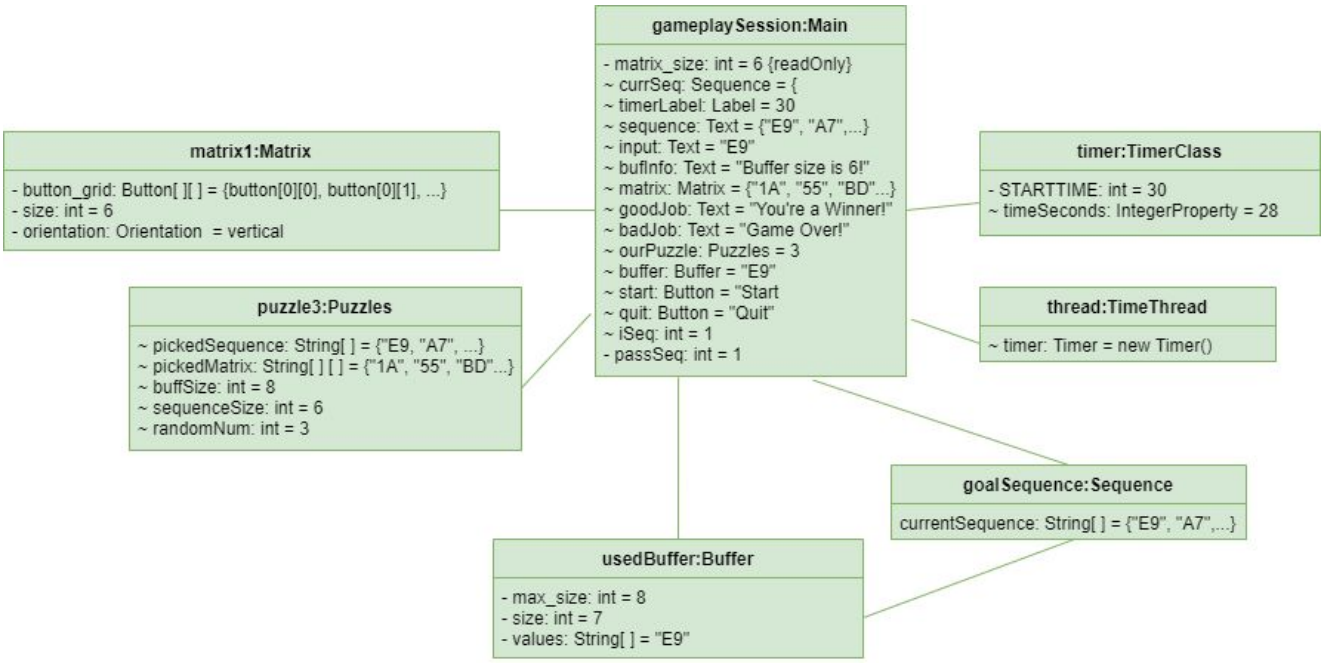
TimeThread

This class, together with TimerClass handles the Timer in our game.

1. Attributes
    a) timer: Timer - A *Timer* class object for creating the *Timer*.
2. Associations
    a) timeOver - A class that handles the cancelling of the timer thread.

Object diagram

Author(s): Ivan Ivanov



The system is in a state where the game has been started successfully, the user has pressed the Start button and pressed on a button in the first row of the Matrix.

Engine is the main source of control for the game. It is responsible for starting the game, creating the required JavaFX objects in a JFX scene and launching the game window. The currently used Goal Sequence is kept in *sequence*, the Matrix in *Matrix*(and its size in *matrix_size*), the textbox telling the user what the Buffer's current size is *buffInfo*.

Matrix has created a matrix and has filled it with values from *Puzzles*. Its attributes currently contain the following values - *button_grid* holds the JavaFX button controls, *size* passes the *Matrix* size from *Main* and the Orientation enumerator is currently set to *vertical* because the user has selected a button and the only selectable buttons are vertically in the row below it.

Puzzles has provided to *Main* a premade puzzle based on a randomly generated value. Currently the user is playing puzzle 3. *PickedSequence* and *PickedMatrix* contain the actual selected puzzle and its corresponding Goal Sequence. *BuffSize* and *SequenceSize* tell *Main* what the sizes of those two elements should be in the current game.

Buffer has provided some basic functionality to *Main,* like creating it, updating it, handling its capacity and checking if a sequence is contained in the buffer. In this state the buffer contains only one value, “E9” and it has 7 empty spaces left.

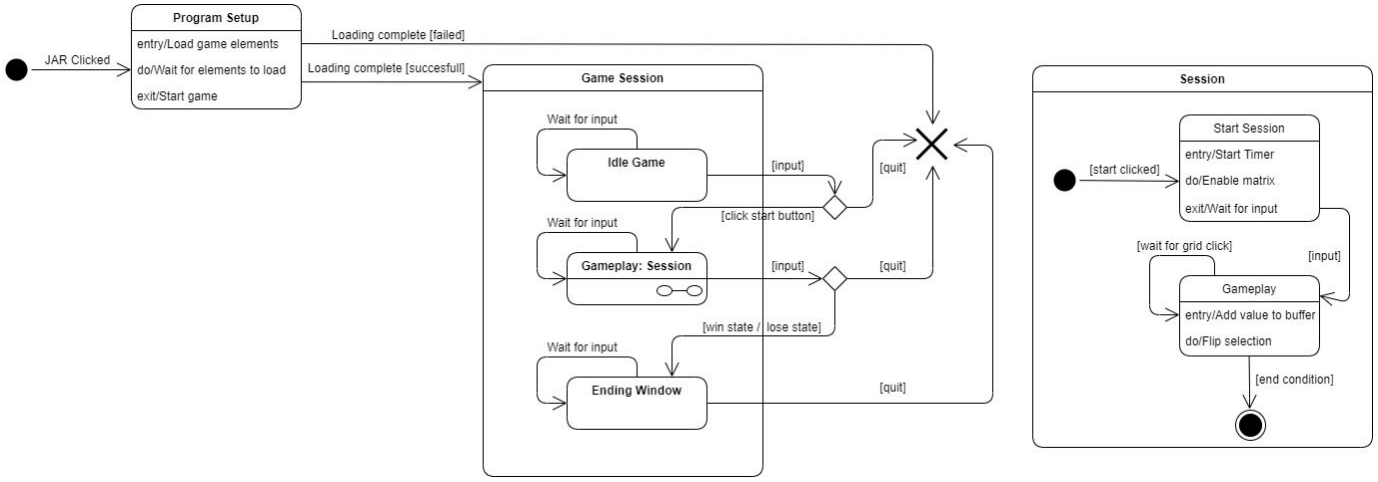
Sequence is monitoring the progress the user has made during gameplay. It tells *Main* if a Game Over/Win should be triggered or not.

TimeThread and *TimerClass* have started the timer that’s set for 30 seconds and a couple have passed.

State machine diagrams

Author(s): Ivan Ivanov

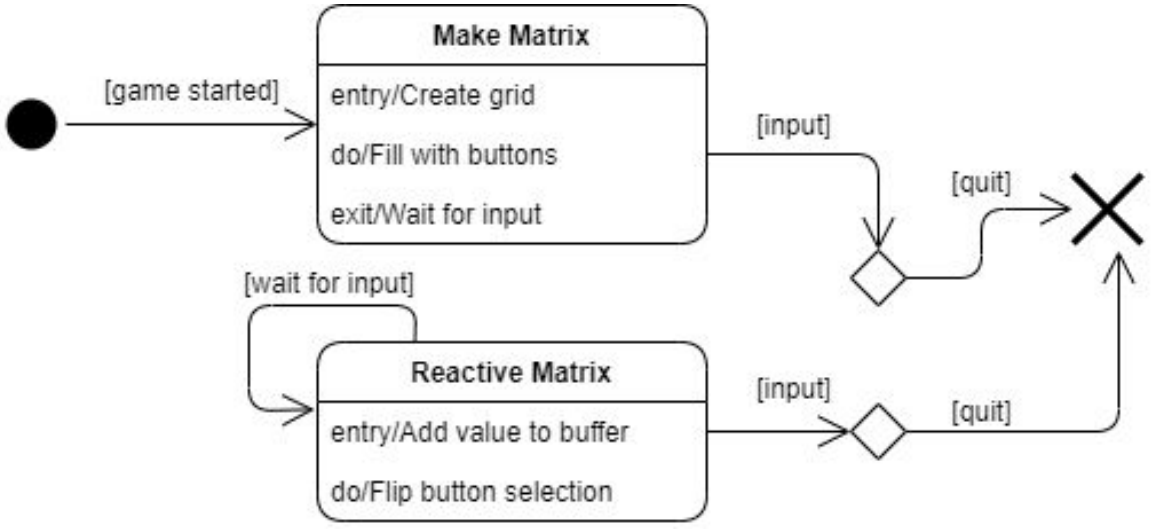
Main



https://drive.google.com/file/d/1t9_teuJx22LjdC8RjHWn5m41dNtnH8v5/view?usp=sharing

Our Main class is the heart of our game and the center of control. When booting up, the game is in Program Setup state. In this case Main’s start() method creates a JavaFX GridPane, which contains the visual elements of our game. Afterwards it makes the Timer, Start and Quit buttons, the Buffer and Sequence text boxes. After creation, Matrix fills up the GridPane with buttons and those buttons then with values like “E9”, “7A” and so forth from the Puzzles class’s random selection.
The game is now Idle and waiting for input from the user. After the user presses the Start button the Gameplay state commences. Here the Timer gets started and the user can select from any of the buttons on the first row of the Matrix, or he can simply press the Quit button to exit the game(present in all of the game’s states post “Program Setup”).
Here the gameplay loop continues with the user selecting a button from the available ones and the button’s value then gets added to the buffer. This Gameplay state continues until either the Buffer gets filled up, the Timer reaches zero, the user selects the correct buttons or presses the Quit button. If he doesn’t press Quit the game reaches the Ending Window state and depending on which of the three possible outcomes he reaches(full Buffer, Timer zero or correct selection) the user sees either a “You’re a Winner!” or a “Game Over!” message show up.

Matrix



The Matrix class has a couple of simple but critical to the game functions.
After the game window gets created, Main utilizes Matrix's constructor, set_values(), update_availability(), get_selected_position() and get_selected_value().
In the Make Matrix state the constructor creates the Matrix (in the form of a 2D Array of buttons) and adds it to the GridPane in Main. Set_values() also gets called from Main and it loads a puzzle at random from Puzzles.
Here the Matrix moves into the Reactive Matrix state, where the game waits for input. After a click update_availability updates the grid by disabling the buttons that the user isn't supposed to press and enables the ones he is supposed to.

Sequence diagrams

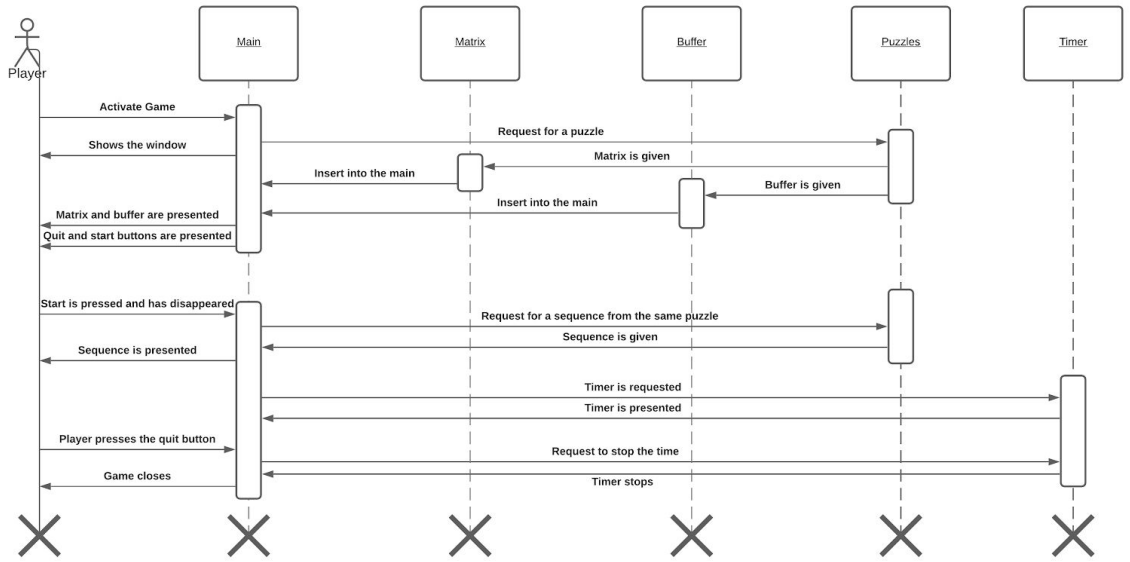
Author(s): Irakliy Marsagishvili, Maxim Abramov

This chapter contains the specification of at least 2 UML sequence diagrams of your system, together with a textual description of their elements. Here you have to focus on specific situations you want to describe. For example, you can describe the interaction of the player when performing a key part of the videogame, during a typical execution scenario, in a special case that may happen (e.g., an error situation), when finalizing a match, etc.

For each sequence diagram you will provide:

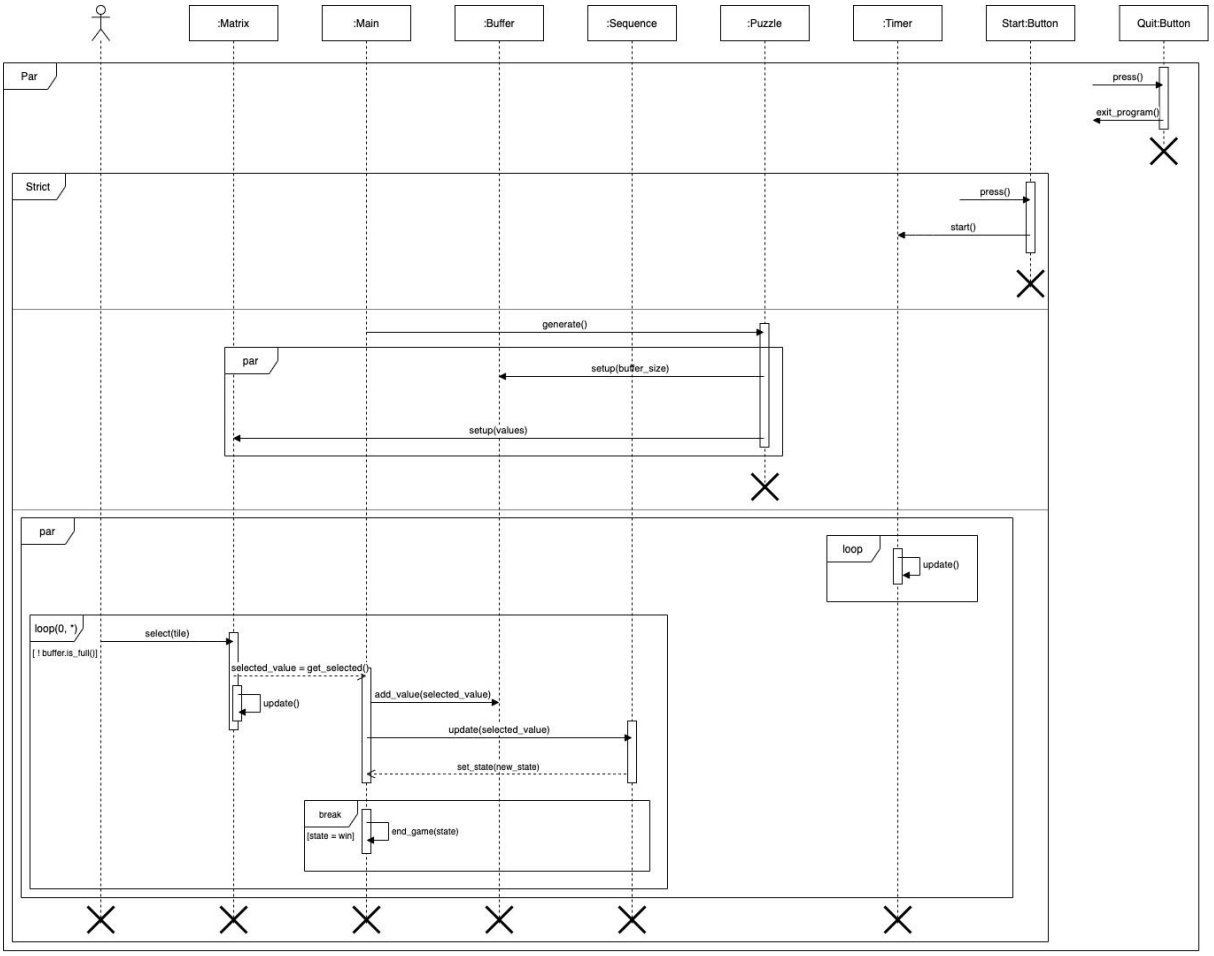
- a title representing the specific situation you want to describe;
- a figure representing the sequence diagram;
- a textual description of its main elements in a narrative manner (you do not need to structure your description into tables). We expect a detailed description of all the interaction partners, their exchanged messages, and the fragments of interaction where they are involved. For each sequence diagram we expect a description of about 200-500 words.
- The goal of your sequence diagrams is both descriptive and prescriptive, so put the needed level of detail here, finding the right trade-off between understandability of the models and their precision.

Title: Player decided to start and quit the game.



In this scenario, the Player turns on the Cyberpunk Hacking Minigame and during the activation of this game, the Main class sends the request to the Puzzles class to generate a random puzzle with Puzzle Generator, which generates a puzzle that is stored in a set of pre-made puzzles in the Puzzles Class. After the request has been sent, Puzzles class provides the information about the Matrix and the size of the Buffer to the Matrix and Buffer classes respectively. As soon as Matrix and Buffer classes receive that provided information, they will send it to the Main class, where the main logic and visuals are currently stored, so Matrix and Buffer would be visualised alongside the Quit button and the Start button. When Player decides to start the game, he/she presses the Start button, which disappears after it is pressed. Main class sends a request to the Puzzles class about the sequence of the puzzle that is made for that specific Matrix set. Puzzles class provides that information to the Main and now the Player can see the necessary sequence that has to be input into the Buffer in order to win the game. After the sequence is received, the Main sends the request to the Timer class to start the countdown timer. Timer class starts the timer and sends the information about it to the Main class, so the Player could see how much time is left. In this scenario, Player might decide to stop the game, because it seems too hard or Player has seen it before and wants to try other puzzles, so the Player presses the Quit button to stop the game. Main class receives the command and sends the signal to the Timer class to stop the timer. Timer class stops the time and sends that information back to the Main class. After that, the Main class stops the game and closes the window.

Player wins the game



In this scenario the player first launches the game, which triggers the construction of the ingame objects. From the start the player can see the start button, the amount of time they will have to complete the sequence, the size of their buffer, as well as the buffer itself, and the matrix of values to choose from. At any point during the gameplay the player can use the quit button, which exits the game and closes the application. The game starts as soon as the player presses the start button, which reveals the sequence the player has to collect to win and starts the timer countdown. Now the player can press on the matrix tiles to add their values into the buffer. Each time the player chooses a tile, the matrix updates the set of available tiles and the buffer shows the complete sequence of values the player has chosen so far. The sequence object in turn checks if the sequence of values stored in the buffer contains the goal sequence and updates the game state accordingly. In this scenario, the interaction is repeated until the player completes the sequence, in which case the game is in the win state, which allows the game to display the win screen.

Maximum number of pages for this section: 4

Implementation

Author(s): Irakliy Marsagishvili, Ivan Ivanov, Maxim Abramov, Man Chung Stephen Kwan

UML model to code implementation
We initially were figuring out how the classes correlate to each other if we use JavaFX, from there we sketched a simplified version class diagram for the features to start with. After we sketched out the simplified versions diagrams as well to connect the dots(object,state machine and sequence diagram) but it has to be refined yet. Then we moved to the next stage which is the code implementation, for every feature that we were able to make it work was being marked as valid and the attributes and methods of that class would be updated/elaborated on. By the time we finalized the implementations we evaluated it and refined the UML diagrams.

Concurrency problems
The hardest implementation was how to connect every single component into one game and make it work in concurrency, such as the timer that has to count down while no click events are happening, we managed to solve it by using JavaFX's framework component that is an animation API which provides a thread called Timeline in order to let it run in concurrency meaning that the timer will tick along while the game is remaining fully interactive.

Location of the main java class
Software-Design\src\main\java\game

Location for jar file to execute system
Software-Design\out\artifacts\software_design_vu_2020_main_jar

The link to the execution of the application and how to execute it
https://youtu.be/CmqItRkZV8o

Time logs

Member	Activity	Week number	Hours
Stephen	Meeting	3 to 5	34
Maxim	Meeting	3 to 5	34
Ivan	Meeting	3 to 5	34
Irakliy	Meeting	3 to 5	34
Stephen	UML	3 to 5	6
Maxim	UML	3 to 5	6
Ivan	UML	3 to 5	9
Irakliy	UML	3 to 5	9
Stephen	Code implementation	3 to 5	34
Maxim	Code implementation	3 to 5	34
Ivan	Code implementation	3 to 5	34
Irakliy	Code implementation	3 to 5	34
Stephen	UML description + fine tuning	3 to 5	4
Maxim	UML description + fine tuning	3 to 5	4
Ivan	UML description + fine tuning	3 to 5	4
Irakliy	UML description + fine tuning	3 to 5	4
		TOTAL	318