Assignment 3

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

Summary of changes of Assignment 2

*Author(s): Ivan Ivanov*

- Moved Class, Object, State Machine and Sequence diagrams to a landscape orientated page for clarity.
- Added missing <<enumeration>> in Class diagram, changed colors in Object diagram and removed Main's object name.
- Added descriptions of the classes associated with Main in the beginning class diagram description.
- Elaborated upon the design decisions we took in relation to the project and the class diagram.
- Elaborated additionally upon the Object diagram's description and how it came to be.
- State Machine: Changed some names of window-states and fixed the improperly used "wait for input" activity. Connected the two states in the Matrix diagram and changed some names to nouns.
- Fixed some rough sentences in Implementation and the Sequence diagram description.
- Added a list of upcoming in Assignment 3 features and elaborated further upon that.

Application of design patterns
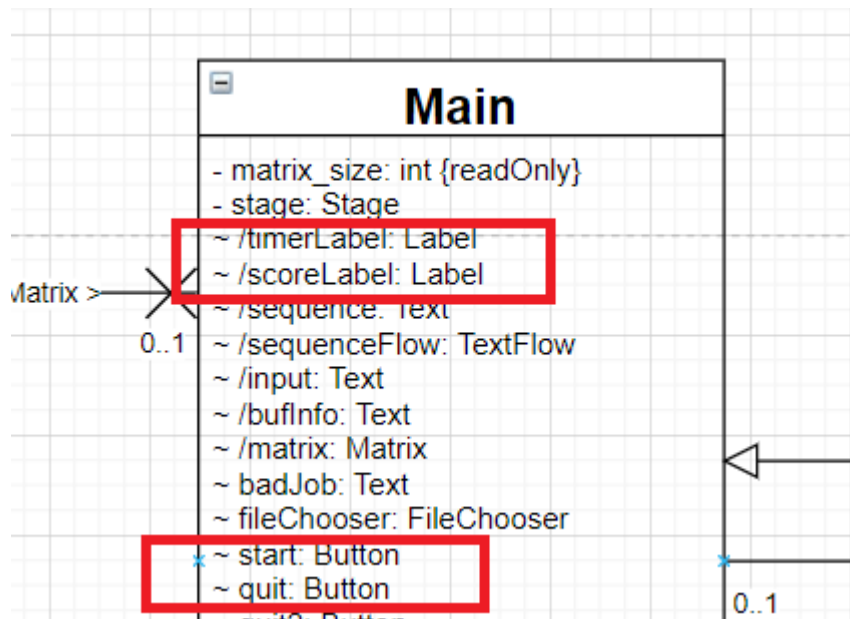
Author(s): Stephen Kwan
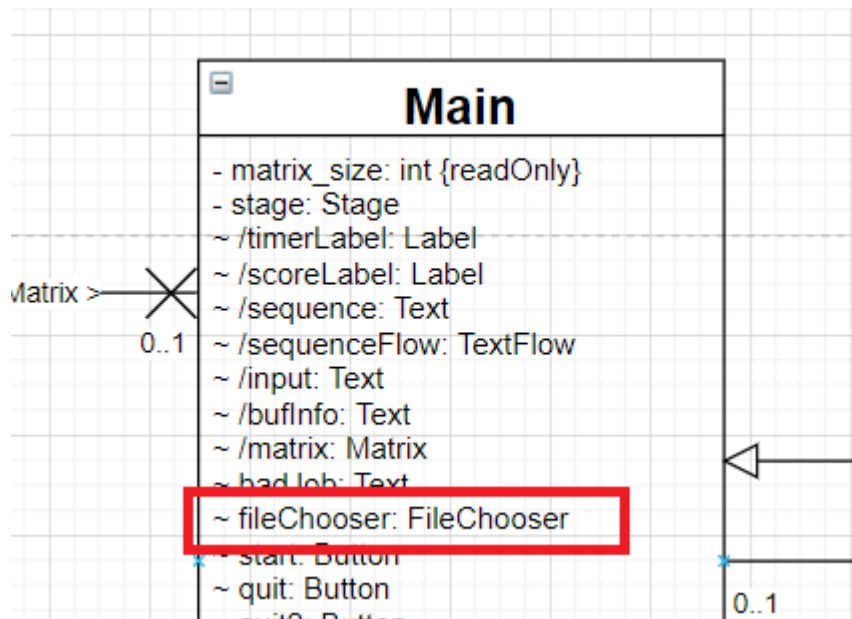
Utility Fig1.0 DP1.
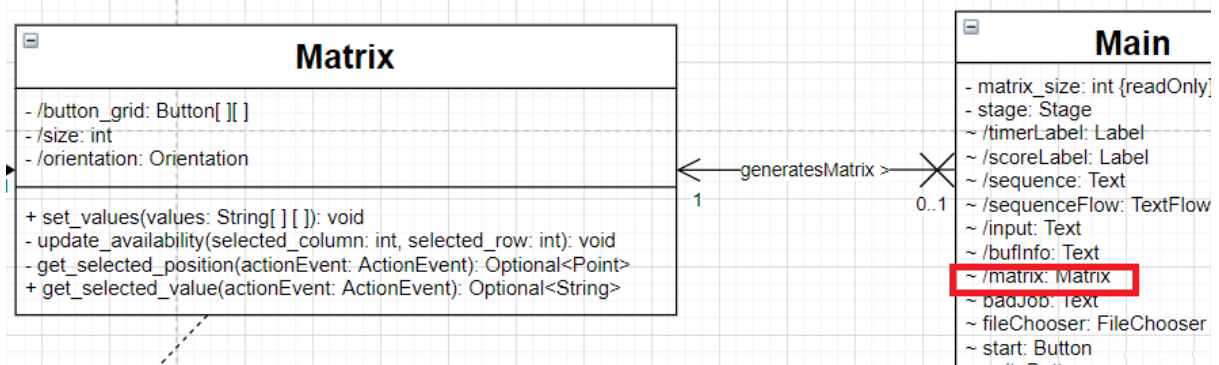
Utility Fig1.1 DP1.

	DP1
Design pattern	Singleton pattern
Problem	Merely one object can exist from that one class.
Solution	Instantiate one object that is global with the keyword new <Class>;
Intended use	See Utility Fig1.0 DP1. See Utility Fig1.1 DP1.
Constraints	If the demand is high in the future in terms of instantiating more objects of that class, it would be hard to maintain since parameters are hard coded.
Additional remarks	-

Utility Fig1.0 DP2.

	DP2
Design pattern	Factory pattern
Problem	UI elements that exceed the amount of one. Such as button and error prone to typos such as a wrong value in the parameter
Solution	By providing the same method for the subclasses that is inherited from the main class. In order to slimmer the chance of typos in the parameter, initialize a PUBLIC STATIC FINAL data type for this instance meaning that core adjustments are merely made inside the object's class.
Intended use	See Utility Fig1.0 DP2.
Constraints	-
Additional remarks	-

Class diagram

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

Orientation
<<enumeration>>

- horizontal: Orientation
- vertical: Orientation

- opposite(): Orientation

Matrix

- /button_grid: Button[ ][ ]
- /size: int
- /orientation: Orientation

+ set_values(values: String[ ][ ]): void
+ update_availability(selected_column: int, selected_row: int): void
+ get_selected_position(actionEvent: ActionEvent): Optional<Point>
+ get_selected_value(actionEvent: ActionEvent): Optional<String>

Main

- matrix_size: int {readOnly}
- stage: Stage
~ /timerLabel: Label
~ /scoreLabel: Label
~ /sequence: Text
~ /sequenceFlow: TextFlow
~ /input: Text
~ /bufInfo: Text
~ /matrix: Matrix
~ badJob: Text
- fileChooser: FileChooser
~ start: Button
~ quit: Button
~ quit2: Button
~ openButton: Button
~ amountOfCorrecetValues: int
~ ingame: VBox
~ timeUp: VBox
~ scenes: VBox
~ root: StackPane
~ initTimeLine: Timeline
~ goal_reachable: boolean
~ /updateSequence: String[]

+ start(primaryStage: Stage): void
+ main(args: String[ ]): void
+ ingameScene(): VBox
+ timeUpScene(): VBox
- configureFileChooser(fileChooser: FileChooser): void
+ handle(actionEvent: ActionEvent): void
+ handleScore(): int

TimerClass

- STARTTIME: int {readOnly}
~ timeline: Timeline
~ /timeSeconds: IntegerProperty

+ getStartTime(): int
+ getTimeSeconds(): IntegerProperty
+ handleTime(): void

Sequence

~ /sequenceNew: String[ ]
~ /colourSequence: TextFlow

+ sequenceProgression(i: int, sequence: String[ ],
input: String, buffer: Buffer): SequencePassState
+ arrayRemove(sequence: String[ ],
count: int): String[ ]
+ getGameOver(quit: Button): void
+ getWinner(quit: Button, score: int): void
+ colourfulSequence(value: String): void
+ uncolourfulSequence(): void

SequencePassState
<<enumeration>>

- nothing: SequencePassState
- pass: SequencePassState
- winner: SequencePassState
- loser: SequencePassState

CustomPuzzle

~ /customMatrix: String[ ][ ]
~ /goalSequence: String[ ]

+ loadPuzzle(file: File): void

Buffer

- /max_size: int
- /size: int
- /values: String[ ] {non-unique}
~ contents: Text

+ add_value(new_value: String): boolean
+ is_full(): boolean
- is_reachable(sequence: String[ ]): boolean
+ unreachable(sequence: String[ ]): boolean
+ update(): void

Puzzles

~ /matrix_size: int
~ /pickedSequence: String[ ] {non-unique}
~ /pickedMatrix: String[ ][ ] [1..1] {non-unique}
~ /buffSize: int
~ sequenceSize: int
- rng: Random
- value_set: String[ ]

+ puzzleGenerator(): void
+ generate_random_game(matrix_size: int,
sequence_size: int): void

generatesMatrix >
< providesTimerTo
< providesPuzzleTo
< providesPuzzleTo

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 it. **Main** uses the **Matrix** and **Puzzles** classes to get a *Matrix* whose values have been randomly selected from a pre-set list. Additionally it uses **Buffer** to get a buffer to be used in the game and the **TimerClass** to have an animated timer above the matrix. **Sequence** does the logic for the Goal Sequence indicator and **CustomPuzzle** provides an alternative in the form of custom-made puzzles.

Main class contains the code that creates the visuals in our game(the *Start, Quit* and *Open* buttons, the *Matrix*, *Timer*, *Buffer* and *Goal Sequence*), also the handler code for the final achievable score and for achieving an endgame state.

1. Attributes
    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) *Stage: Stage* - Used for passing primaryStage outside of the start() method.
    c) *timerLabel: Label* - A simple JFX label which contains the *Timer* in it.
    d) *sequenceFlow: TextFlow* -
    e) *scoreLabel: Label* - A label that contains the amount of points the user has scored in the game.
    f) *sequence: Text* - A text node which contains the *Goal Sequence* which the player must match.
    g) *input: Text* - This node shows to the player the value in the *Matrix* node the player has pressed.
    h) *buffInfo: Text* - Here we show the size of the *Buffer*.
    i) *badJob: Text* - A text node which contains the "Game Over!" text when the player loses the game.
    j) *fileChooser: FileChooser* - Provides the popup menu for choosing a txt file.
    k) *start: Button* - A JFX Button control. Once it's been pressed the Timer gets started and the Code Matrix gets enabled.
    l) *quit: Button* - A simple button which lets the user quit the game.
    m) *quit2: Button* - A secondary quit button used in the *TimeUp* screen(done to get around some limitations in JFX).
    n) openButton - A button used for triggering the FileChooser menu for opening text files.
    o) *amountOfCorrectValues: int* - Used for passing a number of the correct values.
    p) *Ingame: VBox* - All VBox and StackPane objects are used to setup the UI.
    q) *timeUp: VBox* - Used for the *TimeUp* screen.
    r) *scenes: VBox[]* - To set up the UI.
    s) *root: StackPane* - To set up the UI.
    t) *initTimeLine: Timeline* - For the timer in our game.
    u) *goal_reachable: boolean* - Set as either true or false depending on whether the user can complete the sequence.
    v) *updateSequence: String[ ]* - *UpdateSequence* contains a new version of the *Sequence*, that had the first value of the previous sequence removed.
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)).
d) *ingameScene(): VBox* - Sets up the main UI elements in our game.
e) *timeUpScene: VBox* - Contains the TimeUp pop-up window.
f) configureFileChooser(fileChooser: FileChooser) - This method is used for configuring the FileChooser to force the user into selecting a .txt file.
g) *handleScore(): int* - Returns the score the user has achieved after playing the game, based on the amount of seconds that have passed.

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 to start a *Timer* and load it into the *timerLabel* attribute.
e) **Sequence** - For the progress we've made in our *Sequence*.
f) **CustomPuzzle** - Provides code for loading a custom .txt file into the *Code Matrix.*

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, provides the program with button coordinates and values and triggers the value highlighting in the *Goal Sequence.*

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.
d) *was_selected: Boolean* - For keeping track of whether a button has been selected or not.

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.
b) *Sequence* - Accesses the sequence in order to change its color.

Puzzles

This class generates a randomized puzzle for our game.

1. Attributes
   a) *matrix_size: int* - For the size of the *Matrix*.
   b) *pickedSequence: String[ ]* - A string which passes the selected goal sequence.
   c) *pickedMatrix: String[ ][ ]* - An attribute which passes the selected 2D array of values for the Code Matrix.
   d) *buffSize: int* - This attribute contains the pre-determined Buffer size for the selected puzzle.
   e) *sequenceSize: int* - Contains the size of the goal sequence.
   f) *rng: Random* - Used for random generation.
   g) *Value_set: String[ ]* - Contains all the values a matrix can have, used for generating a random game.
2. Operations
   a) *puzzleGenerator(): void* - Works along with *generate_random_game()* to generate a puzzle with randomized values for the *Matrix*. Feeds the above mentioned method with the size of the matrix and a randomized goal sequence size.
   b) *generate_random_game(matrix_size: int, sequence_size: int): void* - Generates a randomized matrix and goal sequence based on a given size of the matrix and a randomized value for the sequence. The method gets its values from a given set stored in *value_set.*

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 and clears up the *Goal Sequence* from used values., also contains the UI elements for the Win and Game Over screens.

1. Attributes
   a) *sequenceNew: String[ ]* - Holds the picked sequence from *Puzzles.*
   b) *colourSequence: TextFlow* - An object used for coloring in red the values of the sequence which the mouse is hovering on.
2. Operations
   a) *sequenceProgression(i: int, sequence: String[ ], input: String, buffer: Buffer): SequencePassState* - 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.
   b) *arrayRemove(sequence: String[ ], count: int): String[ ]* - Removes the used value from the *Goal Sequence.*
   c) *getGameOver(quit: Button): void* - Contains the Game Over UI elements, triggered upon reaching a Game Over state.
   d) *getWinner(quit: Button, score: int): void* - Contains the Win UI elements, triggered upon reaching a Win state.
   e) *colourfulSequence(value: String): void* - Used for coloring a goal sequence value in red when the mouse hovers over a button that contains the same value.
   f) *uncolourfulSequence(): void* - Used for removing the color from a goal sequence value when the mouse doesn't hover over the button anymore.
3. Associations
   a) *Buffer* - Checks if the buffer is full, triggers an end-game condition if it is.
   b) *SequencePassState* - An enumerator containing the values *nothing, pass, winner, loser* for triggering a win, game over state or for simply continuing the game.

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) *is_reachable(sequence: String[ ]): boolean* -
    d) *unreachable(sequence: String[ ]): boolean* - Returns the opposite of *is_reachable()*.
    e) *update(): void* - Updates the buffer.

CustomPuzzle

The **CustomPuzzle** class is used by **Main** to load an user-made .txt file into the *Code Matrix* with the *loadPuzzle()* method.

1. Attributes
    a) *customMatrix: String[ ][ ]* - Contains the custom matrix loaded from a .txt(after the execution of the loadPuzzle() method.
    b) *goalSequence: String[ ]* - Contains the custom goal sequence loaded from a .txt(after the execution of the loadPuzzle() method.
2. Operations
    a) *loadPuzzle(file: File): void* - Scans the user-provided .txt for the *Matrix* he has provided and also for the *Goal Sequence* that's contained on the row beneath the *Matrix*.
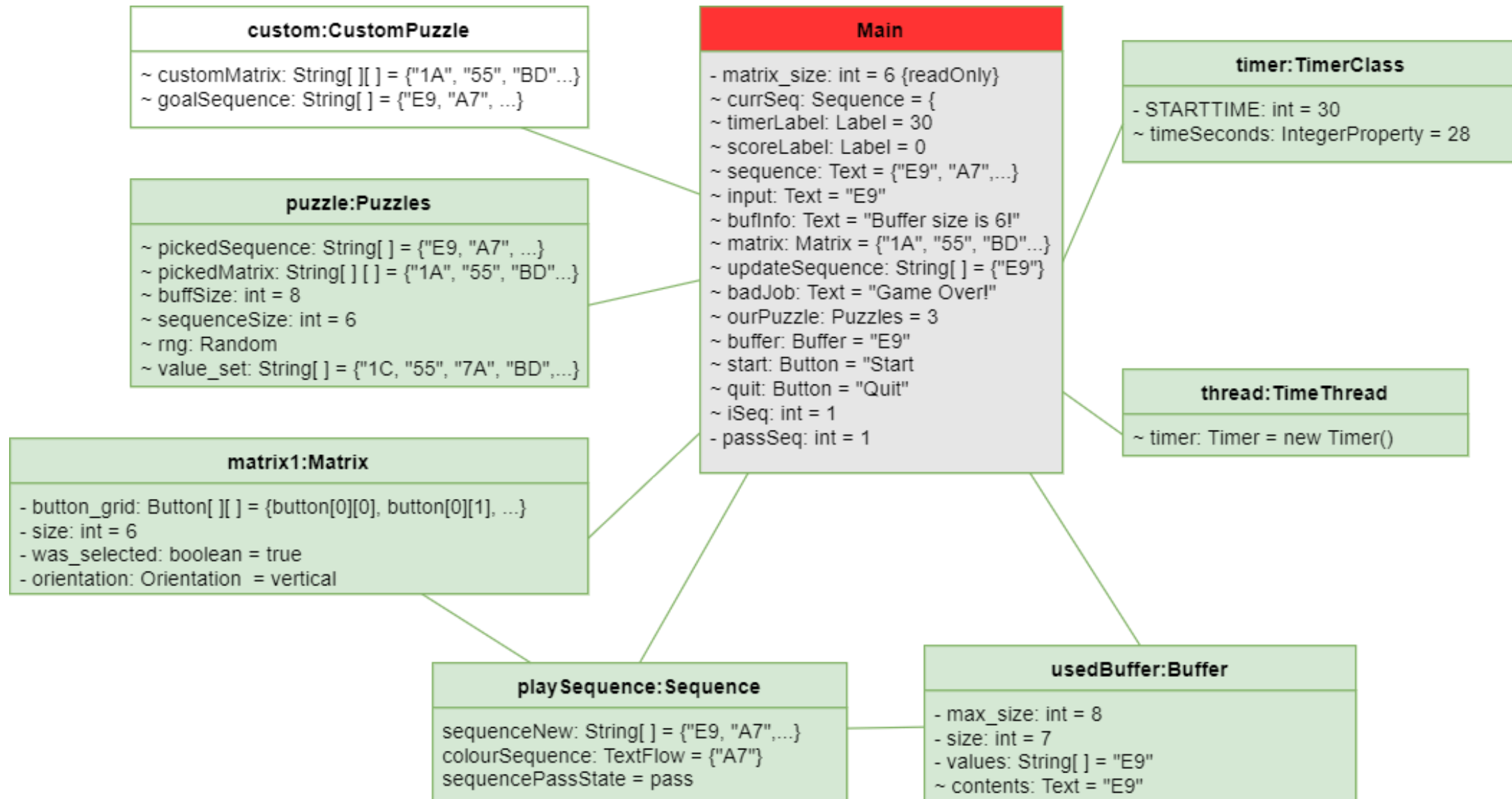
TimerClass

This class runs 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*.

Object diagram

Author(s): Ivan Ivanov

custom:CustomPuzzle

~ customMatrix: String[ ][ ] = {"1A", "55", "BD"...}
~ goalSequence: String[ ] = {"E9", "A7", ...}

puzzle:Puzzles

~ pickedSequence: String[ ] = {"E9", "A7", ...}
~ pickedMatrix: String[ ] [ ] = {"1A", "55", "BD"...}
~ buffSize: int = 8
~ sequenceSize: int = 6
~ rng: Random
~ value_set: String[ ] = {"1C", "55", "7A", "BD",...}

matrix1:Matrix

- button_grid: Button[ ][ ] = {button[0][0], button[0][1], ...}
- size: int = 6
- was_selected: boolean = true
- orientation: Orientation = vertical

Main

- matrix_size: int = 6 {readOnly}
~ currSeq: Sequence = {
~ timerLabel: Label = 30
~ scoreLabel: Label = 0
~ sequence: Text = {"E9", "A7",...}
~ input: Text = "E9"
~ bufInfo: Text = "Buffer size is 6!"
~ matrix: Matrix = {"1A", "55", "BD"...}
~ updateSequence: String[ ] = {"E9"}
~ badJob: Text = "Game Over!"
~ ourPuzzle: Puzzles = 3
~ buffer: Buffer = "E9"
~ start: Button = "Start
~ quit: Button = "Quit"
~ iSeq: int = 1
- passSeq: int = 1

timer:TimerClass

- STARTTIME: int = 30
~ timeSeconds: IntegerProperty = 28

thread:TimeThread

~ timer: Timer = new Timer()

playSequence:Sequence

sequenceNew: String[ ] = {"E9", "A7",...}
colourSequence: TextFlow = {"A7"}
sequencePassState = pass

usedBuffer:Buffer

- max_size: int = 8
- size: int = 7
- values: String[ ] = "E9"
~ contents: Text = "E9"

This object diagram represents the classes in our class diagram in a state where a new game has been started successfully. In such a case the user has pressed the *Start* button and pressed on a button in the first row of the *Code Matrix*. Many(though not all) of the attributes and variables from the class diagram are represented here. In this state, only some of them would contain data or a value in them(like the string *pickedSequence* in **Puzzles**) while others would not yet be activated or used as an alternative option only(**CustomPuzzle,** which gets loaded up only when using the *Open* button).

Main 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 random 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 *was_selected()* memorizes if a button has been selected already. 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.

CustomPuzzle is a class used in an alternative state of the program where the user has selected a .txt with a custom matrix. *CustomMatrix* would in this case contain the first six rows of the .txt and *goalSequence* would have the last row which is the *GoalSequence*. This class is colored in white as it is not used at this specific state of the program. It would be used if the user were to pick a text file.

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.
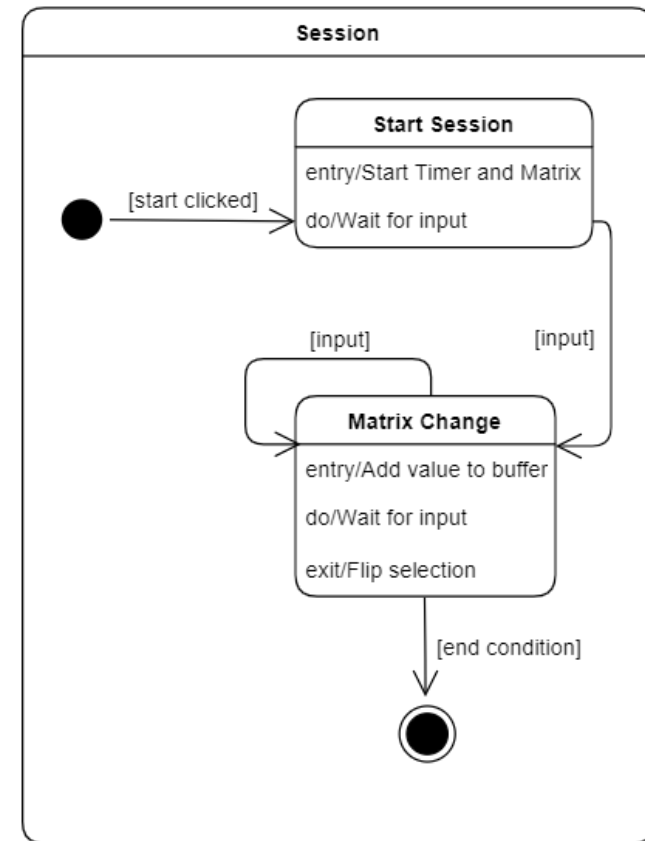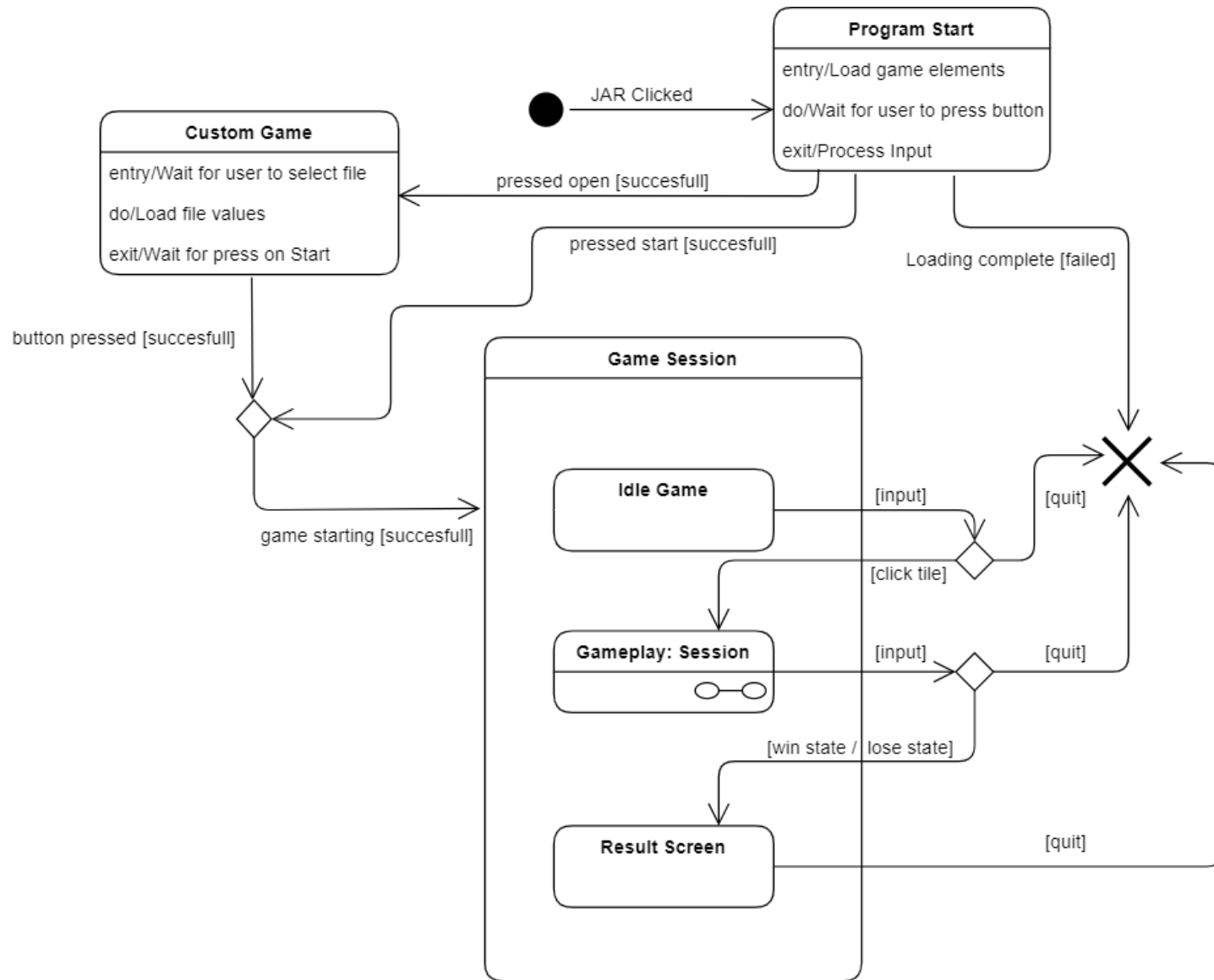
Sequence is monitoring the progress the user has made during gameplay. It waits on **Main** to request the *getWinner() or getGameOver methods for* a Game Over/Win screen.

TimerClass starts the timer that's set for 30 seconds and a couple have passed.

State machine diagrams

Author(s): Ivan Ivanov, Irakliy Marsagishvili

Main

Program Start

entry/Load game elements

do/Wait for user to press button

exit/Process Input

JAR Clicked

Custom Game

entry/Wait for user to select file

do/Load file values

exit/Wait for press on Start

pressed open [succesfull]

pressed start [succesfull]

Loading complete [failed]

button pressed [succesfull]

game starting [succesfull]

Game Session

Idle Game

[input]

[quit]

[click tile]

Gameplay: Session

[input]

[quit]

[win state / lose state]

Result Screen

[quit]

Session

Start Session

entry/Start Timer and Matrix

do/Wait for input

[start clicked]

[input]

[input]

Matrix Change

entry/Add value to buffer

do/Wait for input

exit/Flip selection

[end condition]

Our Main class is the heart of our game and the center of control. When booting up, the game is in Program Start 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, Open and Quit buttons, the Buffer and Sequence text boxes. After creation the game waits for the user to either press on Start or Open. If he's pressed on Start the Matrix gets filled up with values like "E9", "7A" and so forth from the Puzzles class's random generation.

If he presses on Open the game waits for him to choose a text file and after choosing the file the user would need to press on Start, same as in the previous case.

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 gets either a "You're a Winner!" or a "Game Over!" message to show up.
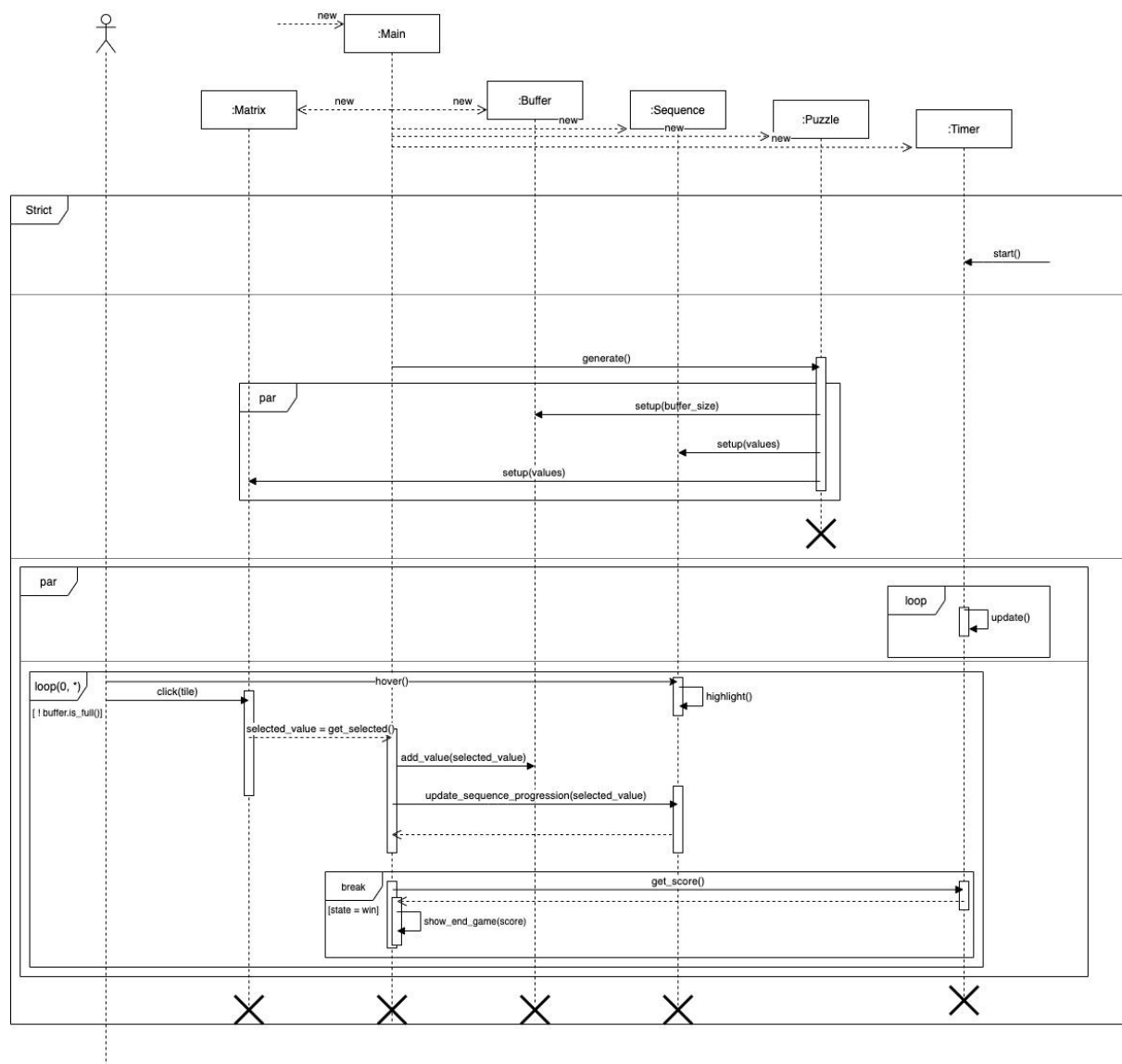
Matrix
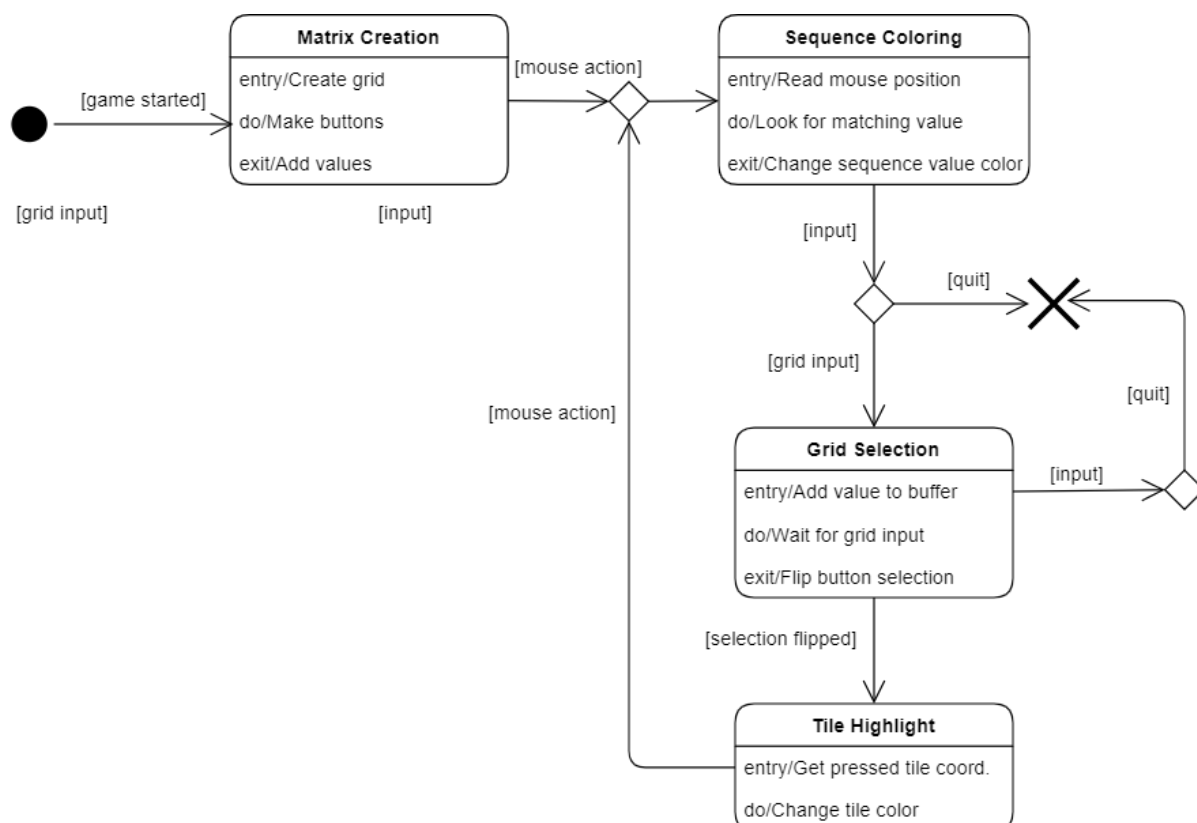
Our **Matrix** state machine diagram is a little different this time around. Now, our project has two new features that are relevant to the diagram - Sequence Highlight and Tile Highlight. In this new version of the class the following happens: The program creates a matrix grid, fills it up with buttons and adds the values. Afterwards it waits for the mouse to go over a button.

When it does, the value in the Goal Sequence that is the same as the button's own value gets its color changed to red and vice-versa for when the mouse isn't over the button anymore. After the user clicks a button and the grid of buttons gets updated for the currently available ones, the button(or tile) which has been pressed gets its color changed to blue. After the color gets changed the system finishes this loop and goes back to waiting for the mouse to hover over a tile. In any case, the user is also free to press the quit button and end the game prematurely.

Sequence diagrams
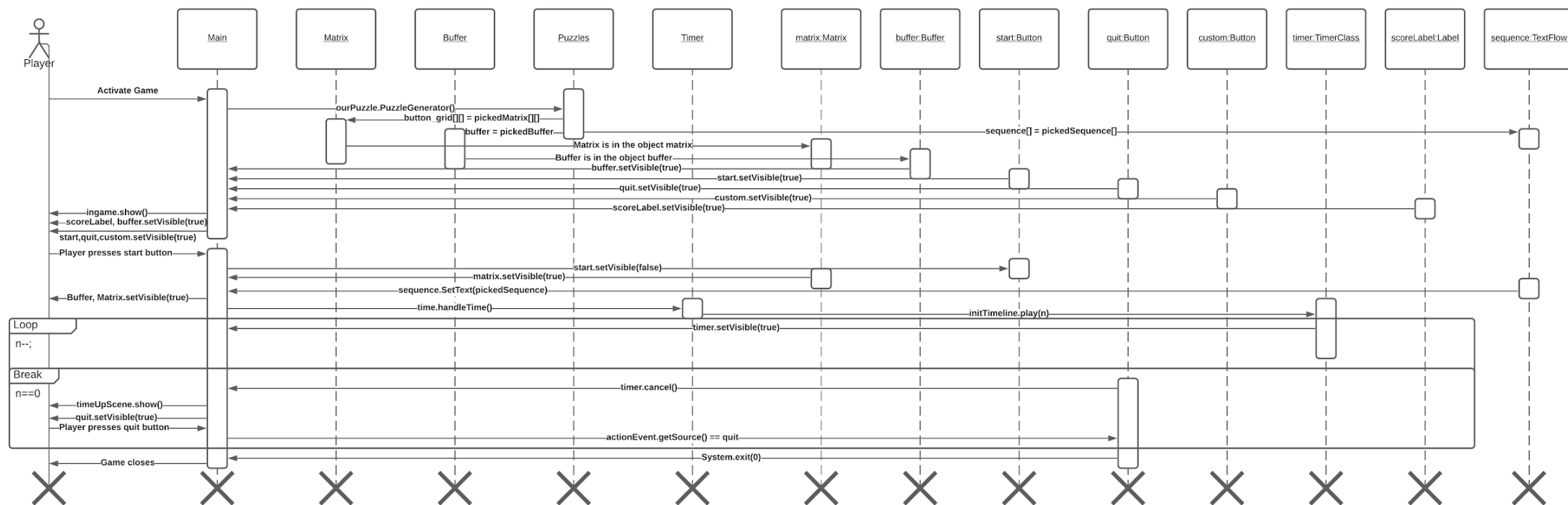
Author(s): Irakliy Marsagishvili, Maxim Abramov

Player wins a randomly generated game

In this scenario the player first launches the game, which triggers the construction of the ingame objects. When the player starts the game, the matrix and the goal sequence are revealed and the timer starts counting down. The faster one can construct the sequence, the more points they get. 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 with the player's score.

Player runs out of time

Player

Main | Matrix | Buffer | Puzzles | Timer | matrix:Matrix | buffer:Buffer | start:Button | quit:Button | custom:Button | timer:TimerClass | scoreLabel:Label | sequence:TextFlow

Activate Game

ourPuzzle.PuzzleGenerator()

button_grid[][] = pickedMatrix[][]

buffer = pickedBuffer

sequence[] = pickedSequence[]

Matrix is in the object matrix

Buffer is in the object buffer

buffer.setVisible(true)

start.setVisible(true)

quit.setVisible(true)

custom.setVisible(true)

scoreLabel.setVisible(true)

ingame.show()

scoreLabel, buffer.setVisible(true)

start,quit,custom.setVisible(true)

Player presses start button

start.setVisible(false)

matrix.setVisible(true)

sequence.SetText(pickedSequence)

Buffer, Matrix.setVisible(true)

time.handleTime()

initTimeline.play(n)

Loop

n--;

timer.setVisible(true)

Break

n==0

timer.cancel()

timeUpScene.show()

quit.setVisible(true)

Player presses quit button

actionEvent.getSource() == quit

Game closes

System.exit(0)

In this scenario, the Player turns on the Cyberpunk Hacking Minigame and during the activation of this game, the Main class sends a request to the Puzzles class to generate a random puzzle with Puzzle Generator. 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. Puzzles class also sends the sequence into the TextFlow object, where the sequence will be stored for visual presentation, but will not be visible yet, until the Start button is pressed. 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 Buffer would be visualised alongside the Quit button, the Start button. The Open (custom) button and the Score Label. 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 TextFlow object sequence, which already has the sequence that is generated for that specific Matrix set, to make it visible. Now the Player can see the necessary sequence that has to be input into the Buffer in order to win the game and the Matrix set that the Player has to play on. After the sequence is received, the Main sends the request to the Timer class to start the countdown timer. As soon as the timer starts, which has been set to n seconds, the Player might decide to think about the solution without touching the matrix first and may have a tough time solving this puzzle before time runs out. So if those events shall happen, the time will run out, when n is going to be equal to 0. When time runs out, the new scene shows up, that has a text "time is up!" and the quit button is presented, so when the Player presses the button, the program would shut down.

Implementation

Author(s): Stephen Kwan, Ivan Ivanov

UML model to code implementation

At first, we spent a few days entirely on assessing the needs of this second version of our project. Initially, we discussed how the new features would get implemented and how JavaFX can handle the features we're building.. From there improved upon the class and object diagrams to use as a baseline for the extra features that we need to do. After the creation of the diagrams we discussed again how we need to approach our project. Then we moved to the code implementation stage. We started with a very basic version of the new features(file loading was done first, then sequence color change and button highlighting). By the time we finalized the implementation we evaluated it and refined the UML diagrams according to the final Assignment 3 product we had.

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.

Timer problem

We detected a bug that the timer was not showing the time up scene(only in the console), however, the fix costed quite some time since we approached the problem by alternating the first scene(called ingame scene where all the main game elements and logics are stored) into the second scene(time up scene where the time up elements are shown such as text that indicates "time up" and the quit button),thus we reorientate the main code drastically and removed the class: TimerThread since it does not serve any purpose anymore.

Location of the main java class
Software-Design\src\main\java\game
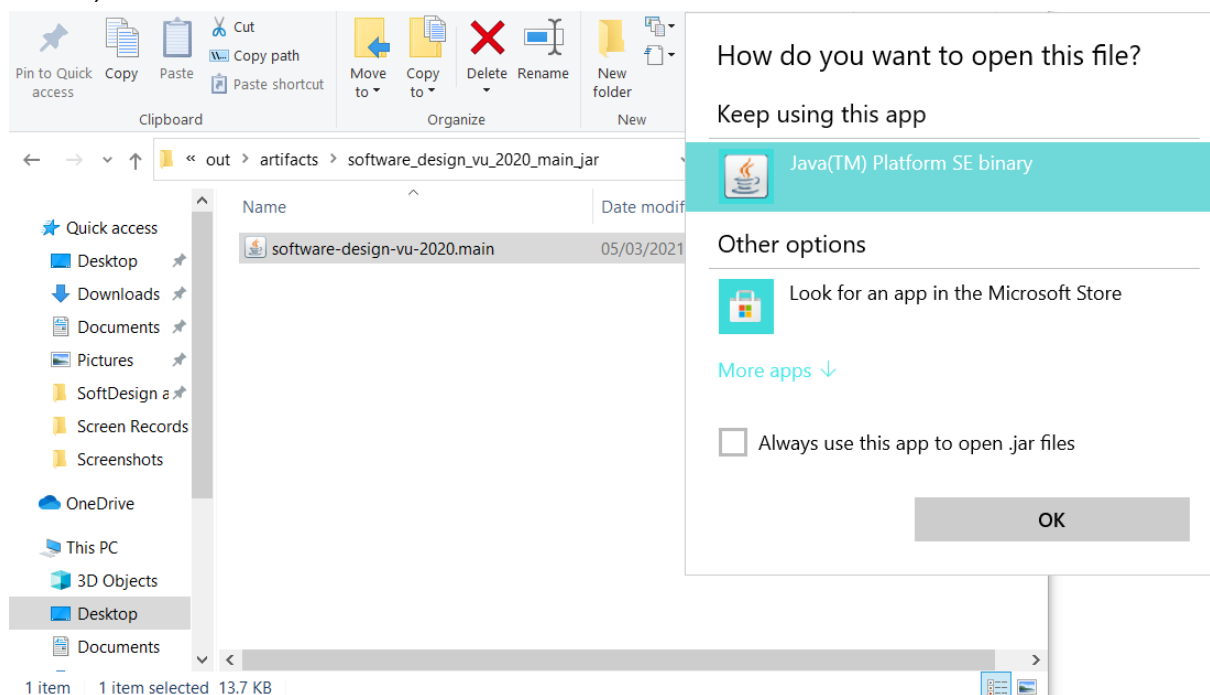
Location for jar file to execute system
<root>\out\artifacts\Software_Design_jar\Software-Design.jar
<root>\build\out\artifacts\Software_Design_jar\Software-Design.jar

The link to the execution of the application and how to execute it
https://youtu.be/f8OWo-krWXs

Right click on the file and press Open With and Choose Java (Windows) (If Java isn't on default)

Time logs

Team number		1	
Member	**Activity**	**Week number**	**Hours**
Stephen	Meeting	6 to 8	12
Maxim	Meeting	6 to 8	12
Ivan	Meeting	6 to 8	12
Irakliy	Meeting	6 to 8	12
Stephen	Assignment 2 Feedback	6 to 8	3
Maxim	Assignment 2 Feedback	6 to 8	3
Ivan	Assignment 2 Feedback	6 to 8	3
Irakliy	Assignment 2 Feedback	6 to 8	3
Stephen	Code Implementation	6 to 8	28
Maxim	Code Implementation	6 to 8	28
Ivan	Code Implementation	6 to 8	28
Irakliy	Code Implementation	6 to 8	28
Stephen	Assig 3 UML + fine tuning	6 to 8	6
Maxim	Assig 3 UML + fine tuning	6 to 8	6
Ivan	Assig 3 UML + fine tuning	6 to 8	6
Irakliy	Assig 3 UML + fine tuning	6 to 8	6
		TOTAL	196