Assignment 2

Team number: CyberpunkHackingMinigame 12
Team members

Name	Student Nr.	Email
Omer Faruk Cakici	2665502	omcakici99@gmail.com
Xiaoxuan Lu	2661045	junjunlujun72@gmail.com
Yasir Uçkun	2666346	ahmetyuckun@gmail.com
Ying Ying Ma	2686767	yingyingma@hotmail.nl
Emre Furkan Guduk	2667887	emregdk2853@gmail.com

This document has a maximum length of 15 pages (excluding the contents above).

Implemented feature

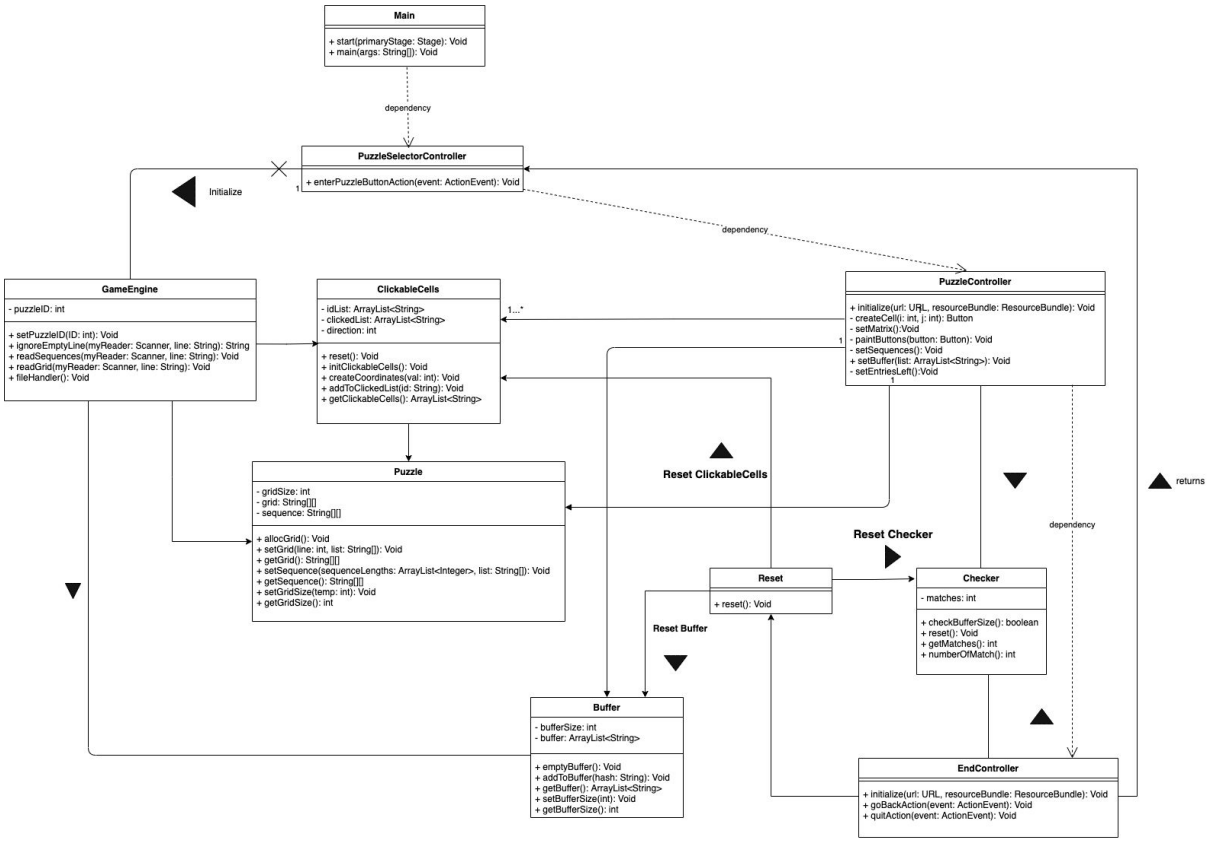
ID	Short name	Description
F1	Puzzle Choice	The system shall support different values in the tiles, sequences, and buffer in the game. The player is allowed to choose from 40 different puzzles before the game begins. The player can type a number between 1-40 to choose the files in the choosing interface. Format of file: ● The first line contains the buffer size (4-7) ● The matrix size (5x5 or 6x6) ● The sequence lists (1-4)
F2	User Interface	The user interface is a graphical user interface. The player can type and click on the user interface. There are basically three windows: the puzzle choosing window for the player to type to choose the puzzle, the game playing window for the player to click the tile and the end window displaying the result. There are three main parts showing when the game begins: code matrix, buffer, sequence list. Players pick tiles by clicking on the tile. When the tile is picked, the tile value will show on the buffer.
F3	Tile Picking	Starting in the first row, the player can pick a tile. Next, the player can pick a tile in the same column as the first tile. The game keeps alternating between rows and columns in a similar fashion until the game is over. When the player chooses a tile, the tile will be updated to the buffer. There will be entries left on the buffer to remind users how many tiles left they can pick. The system disabled the tiles that are not

		allowed to click on, and the user can only click on tiles that are allowed each turn.
F4	End/Outcome	**End** The game ends when the buffer is completely filled(entries left is 0) or all sequences in the list are found within the buffer size. The number of sequence lists found will be shown to the player when the game ends. **Win** The player will win when he finds at least one sequence list at the end of the game. The system will display a winning message to the player that says “Congratulations”. Moreover, the number of sequence lists found will be displayed to the player. The end screen will also show the player a happy cat picture as a reward. **Loss** The player will lose when he finds no sequence list. The system will display a losing message to the player that says “Game over. You couldn’t find any sequence.”.Also the end screen will show the player a sad cat picture to convince him/her to do better next time.
F5	Restart	**Restart/Puzzle page/Exit** After the game, the player has three options: 1. Restart the game with the same puzzle. 2. Return to the puzzle picking pages to pick another puzzle. 3. Exit the game.

Used modeling tool: diagrams.net

Class diagram

Author(s): all



Main

The **main** class initializes the game. It loads puzzleSelector.fxml to initialize PuzzleSelectorController and start the whole system.

Operations

- **start(primaryStage: Stage): Void** - loads puzzleSelector.fxml to invoke PuzzleSelectorController.
- **main(args: String[]): Void** - launches the application.

Associations

The **main** class has a weak association with **puzzleSelectorController**. The relationship between them is a dependency association**,** meaning that main depends on puzzleSelectorController. In *start*(), puzzleSelector.fxml is loaded. The puzzleSelector.fxml file then invokes PuzzleSelectorController.

PuzzleSelectorController

The class enables the player to enter the number of the puzzle he wants to play.

Operations

- **enterPuzzleButtonAction(event: ActionEvent): Void** - read the input (puzzleNumber entered) from the player.

Associations

- There is a unidirectional association between **PuzzleSelectorController** and **GameEngine**. PuzzleSelectorController can process methods of a GameEngine object. One PuzzleSelectorController initializes one GameEngine object.
- There is a dependency association between **PuzzleSelectorController** class and **puzzleController**. The PuzzleSelectorController object invokes the PuzzleController object by loading puzzle.fxml.

PuzzleController

PuzzleController initializes the game interface to the player. It sets the matrix, sequences, buffer, and entries left in the game.

Operations

- **initialize(url: URL, resourceBundle: ResourceBundle): Void -** by overriding this function, we aim to execute some functionalities at the moment when puzzle.fxml is called.
- **createCell(i: int, j: int): Button -** Create a button cell with the right ID and text then return it
- **setMatrix(): Void** - If the matrix is 6 by 6, this function adds one column and row to the gridview, then it fills the matrix with the cells returned from createCell() method.
- **setSequences(): Void** - This function sets the text of the label to show the sequences
- **setBuffer(list: ArrayList<String>): Void** - This function sets the text of the label to show the buffer.
- **setEntriesLeft(): Void** - This function calculates the remaining space in the buffer and displays it by setting the text of the label.

Associations

- **PuzzleController** has four unidirectional associations with **Puzzle, Buffer, ClickableCells, and Checker**. The PuzzleController object can call the methods of the Puzzle object, and it contains methods in Buffer, ClickableCells, and Checker classes. Every PuzzleController can have one or several ClickableCells.
- There is a dependency association between **PuzzleController** and **EndController**: PuzzleController loads end.fxml to invoke the EndController object.

GameEngine

GameEngine takes care of the initialization of the game. It reads the puzzle corresponding to the puzzle number.

Fields:

- **puzzleID: int -** A private variable which saves the player's puzzle choice.

Operations

- **setPuzzleID(ID: int): Void** - This function sets the *puzzleID* by taking into consideration the input taken from the player. Later on, *puzzleID* is used to pick the corresponding .txt file in order to initialize the game.
- **readSequences(myReader: Scanner, line: String): Void** - It reads sequences from the given .txt file and saves them in *line*.
- **readGrid(myReader: Scanner, line: String): Void - It fileHandler(): Void** - It reads the lines of the *grid* from the selected .txt file and it saves them in *line*.

Associations

- There is a unidirectional association relationship between the **GameEngine** and the **ClickableCells** - **GameEngine** can process methods of a **ClickableCells** object. **GameEngine** calls two methods of **ClickableCells** in order to initialize and fill up the *idList* under **Puzzle**.
- There is a unidirectional association relationship between the **GameEngine** and the **Puzzle** - **GameEngine** can process methods of a **Puzzle** object. Here, *readGrid* and *readSequences* under **GameEngine** serve as helper methods in order to fill up the corresponding variables under **Puzzle,** namely *grid* and *sequence*.
- There is a unidirectional association relationship between the **GameEngine** and the **Buffer** - **GameEngine** can process methods of a **Buffer** object. In this relation, the *fileHandler* method under **GameEngine** takes care of filling up the *bufferSize* under Buffer.

ClickableCells

ClickableCells represent the allowed cells on the *grid* which can be used to interact with the *grid*. This class controls the flow of the game by limiting the number of interactable cells in the *grid* and by setting only certain parts of the *grid* as functional. **ClickableCells** cannot exist alone, as they can only live and die with the **GameEngine**.

Fields:

- **idList: ArrayList<String> -** A private variable where the coordinates of the clickableCells in the current move are saved in string format.
- **clickedList: ArrayList<String> -** A private variable where the previously clicked cells' coordinates are saved in string format.
- **direction: int -** A private variable which enables the system to decide whether the next clickableCells should be taken from a row or a column.

Operations

- **reset(): Void -** A helper function which enables the system to reset the content of the clickableCells object.
- **initClickableCells(): Void -** A helper function which initializes the clickableCells. Basically, it initializes the *idList* with the first row of the *grid* and sets the *direction* value to 0.
- **createCoordinates(val: int): Void -** It builds a coordinate system so that the clicked cells can be saved as coordinates instead of values. It takes an int input and turns it into a coordinate value.
- **addToClickedList(id: String): Void -** A helper function to append an coordinate value into the *clickedList*.
- **getClickableCells(): ArrayList<String> -** A helper function to fetch the content of the private variable *idList.*

Associations

There is a unidirectional association between **ClickableCells** and **Puzzle**. **ClickableCells** calls a method of **Puzzle** in order to get the *gridSize*.

Puzzle

Puzzle is one of the main components in the game. It structures the game components –grid, sequence– and allows interactions on the *grid* from the user. Additionally, Puzzle works directly with the ClickableCells to determine the current action's validity.

Fields

- **gridSize: int -** A private variable that's saving the dimension of the *grid*. In all cases, the *grid* is an n x n matrix, which is why it is used to read the correct number of lines while reading the *grid*'s content from the .txt file.
- **grid: String[][] -** The *grid* –private variable– is a 2D matrix where the system saves the values on the *grid*. All the values in the *grid* are saved in String format.
- **sequence: String[][] -** The *sequence* saves the correct sequences which should be found by the player.

Operations

- **allocGrid(): Void -** A helper function which allocates memory for the *grid*.
- **setGrid(line: int, list: String[]): Void -** The setter function for the *grid*.
- **getGrid(): String[][] -** The getter function for the *grid.*
- **setSequence(sequenceLengths: ArrayList<Integer>, list: String[]): Void -** The setter function for the *sequence*. It takes 2 parameters –*sequenceLengths*, *list*– and creates sequences by splitting the second parameter *list* into small chunks with regard to the *sequenceLengths*.
- **getSequence(): String[][] -** The getter function for the *sequence*.
- **setGridSize(temp: int): Void -** Sets the *grid*'s size.
- **getGridSize(): int -** The getter function for the *gridSize.*

Checker

Checker is responsible for determining the result of the game.

Fields

- **matches: int** - This variable is used to return the amount of sequences found in the buffer.

Operations

- **checkBufferSize(): boolean** - This function compares whether "getBuffer().size" is equal to getBufferSize() - if it is then return "True" otherwise it returns "False".
- **reset(): Void** - This function resets variables and matches to zero. That means when the user restarts the game, the variables are reset.
- **getMatches(): int** - When calling this function it returns a variable win.EndController Class uses this function to determine the result of the game.
- **numberOfMatch(): int** - This function calculates the number of matches.

Associations

- There is a unidirectional association between **PuzzleController** and **Checker**. PuzzleController invokes *setWin*() and *numberOfMatch*() functions in the Checker object.
- There is a unidirectional association between **Checker** and **EndController**.
- There is a unidirectional association between **Checker** and **Reset**.

EndController

EndController gives the user several choices at the end of the game.

Operations

- **initialize(url: URL, resourceBundle: ResourceBundle): Void** - This function simply initializes when this class calls. It checks whether the user won or lost the game and calls **Reset** if the condition is true, to reset **Buffer**, **Checker** and **Clickable** cells.
- **goBackAction(event: ActionEvent): Void** - At the end of the game this function loads the puzzleSelector.fxml file into loader and goes back to PuzzleSelectorController.
- **quitAction(event: ActionEvent): Void** - This function quits the events named by “getSource, getScene, getWindow”.
- **tryAgainAction(event: ActionEvent): Void** - At the end of the game, if the user presses the “tryAgain” button then this function calls. It loads the previous data into the puzzle.fxml file. Puzzle.fxml files initialize **PuzzleController**.

Associations

There are two unidirectional associations between **EndController** and **PuzzleSelectorController** and between **EndController** and **Reset**.

Reset

Reset handles resetting values at the end of the game.

Operations

- **reset(): Void** - It invokes three methods in three different classes: *emptyBuffer*() in the Buffer class to remove all the elements inside, *reset*() in ClickableCells to clear *idList* and *clickedList* and *reset*() in the Checker class to assign *win* and *matches* to zero.

Associations

There are three unidirectional associations between **Reset** and **Checker**, between **Reset** and **Buffer** and between **Reset** and **ClickableCells**.

Buffer

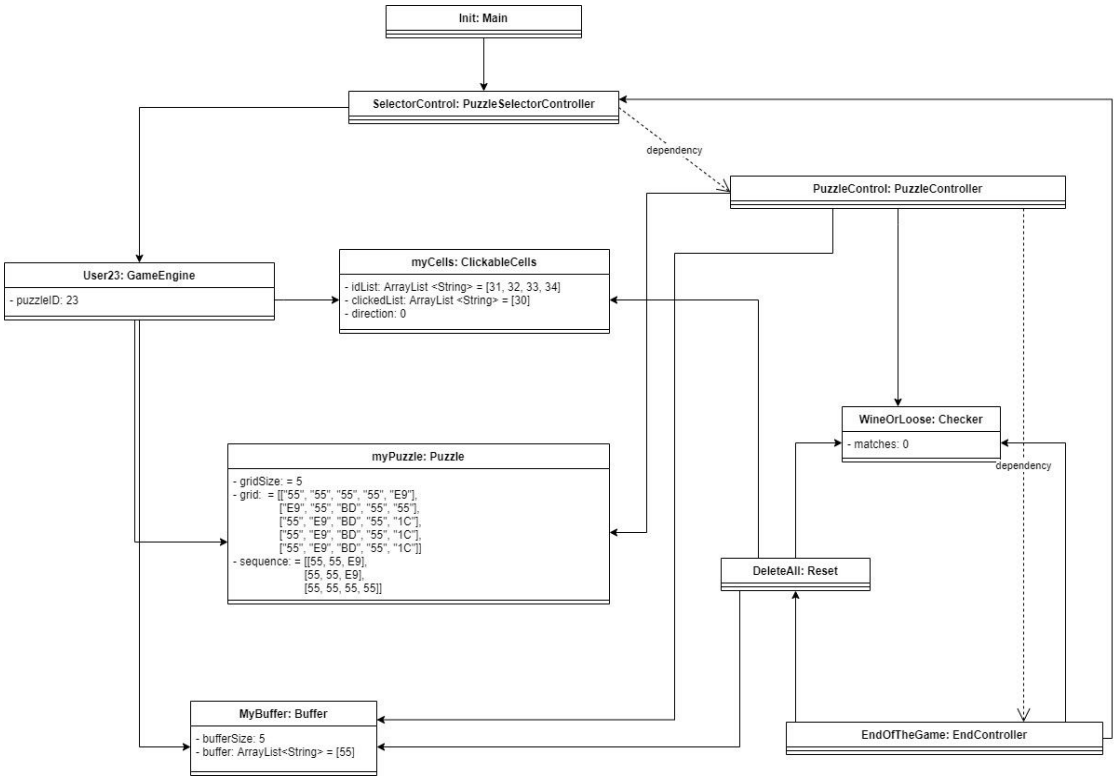
Buffer handles the operations done on the buffer, which contains the values in the cells selected by the user.

Operations

- **emptyBuffer(): Void** - This function works only at the end of the game to reset the buffer.
- **addToBuffer(hash: String): Void** - This function works when the sequence is found and it adds to the buffer.
- **getBuffer(): ArrayList<String>** - This function returns the buffer and displays how many sequences have been found.

Object diagram

Author(s): Omer Faruk Cakici



This object diagram represents the snapshot of the game after the click of the “30” cell on the *grid*. It shows that “3” -> third column and “0” -> first row.

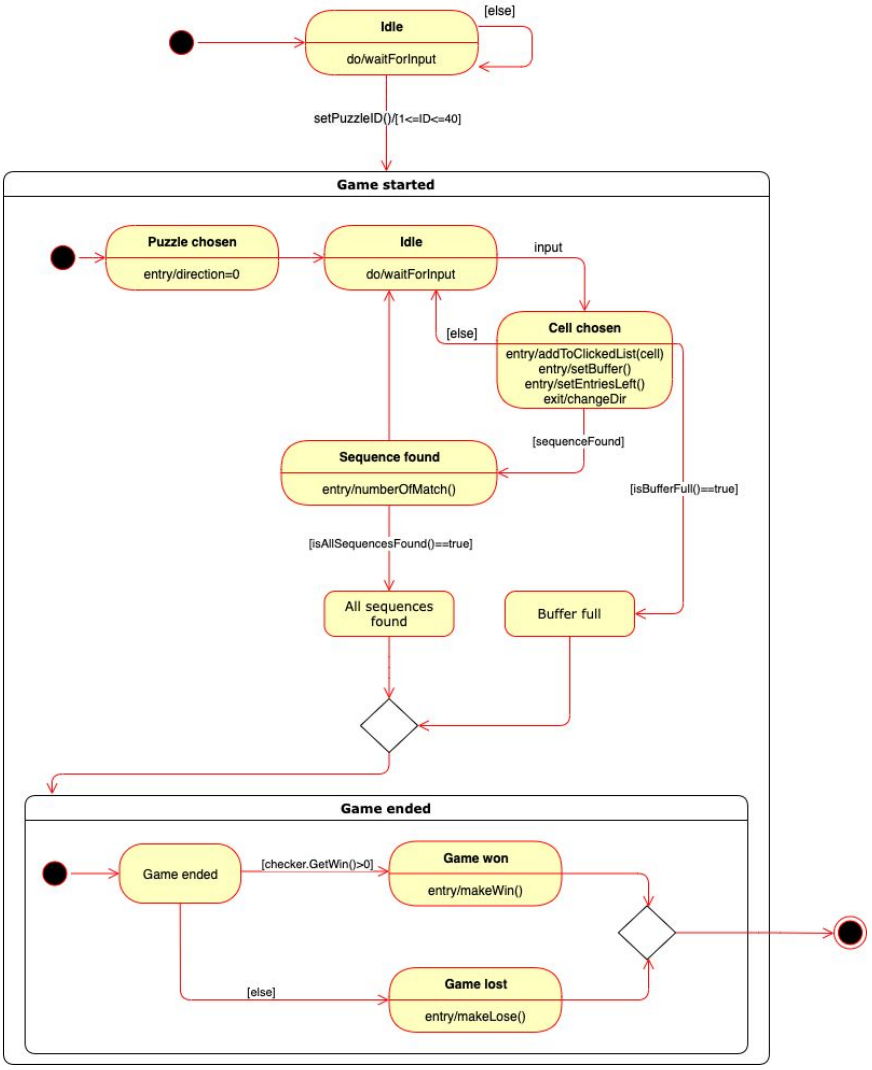
- **Init:Main:** GameStarts from the main class object and Main class does not have an object, it simply calls puzzleSelector.fxml and that scene builder file creates all the objects for us.
- **User23**: GameEngine object asks the user for the puzzleID and the user enters “23”.
- **myCells**: **ClickableCells** creates three objects:
    - **clickList:** shows which buttons were clicked by the user.
    - **idList:** shows what button coordinates can be clickable on the next move.
    - **direction:** determines whether a user can choose cells horizontally or vertically. If it is equal to 0, the direction is horizontal, else it is vertical.
- **myPuzzle: Puzzle** creates three objects:
    - **gridSize:** shows the *grid* size of the *grid*.
    - **grid:** shows what the *grid* contains. It will be the same in every move.
    - **sequence:** shows which sequences the user needs to find.
    - **myBuffer: Buffer** creates two objects:
    - **bufferSize:** holds the buffer size.
    - **buffer:** holds matched sequences.
- **PuzzleControl: puzzleController** creates no objects.
- **SelectorControl: PuzzleSelectorController** creates no objects.
- **WinOrLose: Checker** contains a *matches* object set to zero at this moment of the game, representing that the game has not been won.
- **EndOfTheGameL: EndController** creates no object.
- **DeleteAll: Reset:** creates no object.

State machine diagrams

Author(s): Ying Ying Ma

In this section, state machine diagrams will be presented to describe the states of the system.

Overall system:



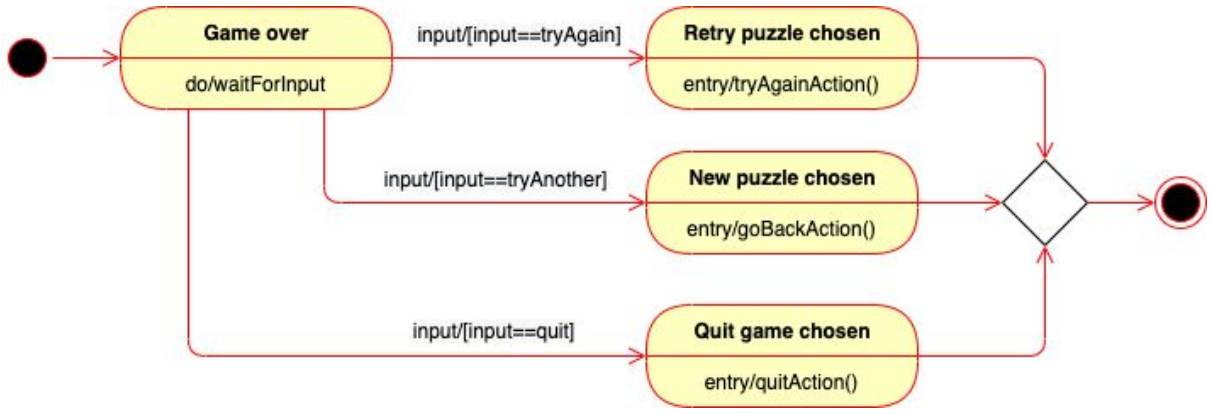
This state machine diagram is a description of the overall system. The initial state is the composite state ‘Idle’, where the activity ‘waitForInput’ is executed. This state transitions to the composite state ‘Game started’ once the event ‘setPuzzleID()’ has occurred, if the guard condition [1<=ID<=40] is true. Else, the user is prompted to enter another input, depicted by the transition from the ‘Idle’ state to itself.

Once the composite state ‘Game started’ has been entered, the system is in the ‘Puzzle chosen’ state, where the activity ‘direction=0’ is executed. From ‘Puzzle chosen’, there is a transition to another ‘Idle’ state, in which the activity ‘waitForInput’ is executed. Here, this input to be entered is the cell which will be picked from the matrix and added to the buffer. After the event ‘input’ has been triggered, the system transitions from ‘Idle’ to ‘Cell chosen’. Now, the entry activities ‘addToClickedList(cell)’, ‘setBuffer()’ and ‘setEntriesLeft()’ and exit

activity ‘changeDir’ are executed. From ‘Cell chosen’, three transitions are possible. Firstly, if the guard condition [isBufferFull()==true] is true, the system transitions to the state ‘Buffer full’. Secondly, if the guard condition [sequenceFound] is true, the transition from ‘Cell chosen’ to ‘Sequence found’ takes place. Here, the activity ‘numberOfMatch()’ is executed. From ‘Sequence found’, a transition to ‘All sequences found’ occurs if the guard condition [isAllSequencesFound()==true] is true. If this condition is false, a transition from ‘Sequence found’ back to ‘Idle’ occurs. If neither of the two conditions are met, the transition from ‘Cell chosen’ to ‘Idle’ occurs, creating a loop in which the user is prompted to provide input and choose cells to add to the buffer until one of the conditions is met. From ‘All sequences found’ and ‘Buffer full‘, transitions to a diamond symbol exist, leading to another composite state ‘Game ended.

The composite state ‘Game ended transitions to ‘Game won’ if the guard condition [checker.GetWin()>0] is true. In ‘Game won’, the entry activity ‘makeWin()’ is executed. If the guard condition is false, the system transitions to ‘Game lost’, where the entry activity ‘makeLose()’ is executed. Both ‘Game won’ and ‘Game lost’ transition to a diamond symbol, which leads to the end of the state machine.

Game over:



This state machine diagram is a description of the system when the game has either been won or lost. The initial state is the composite state ‘Game over’, where the do activity ‘waitForInput’ is executed. The user is given the option to retry the same puzzle, to try another puzzle and to quit the game. From ‘Game over’, three transitions exist, all of which require the event ‘input’ to be triggered. If the guard condition [input==tryAgain] is true, the transition to ‘Retry puzzle chosen’ occurs and the entry activity ‘tryAgainAction()’ is executed. If the guard condition [input==tryAnother] is true, the transition to ‘New puzzle chosen’ occurs and the entry activity ‘goBackAction()’ is executed. If the guard condition [input==quit] is true, the transition to ‘Quit game chosen’ occurs and the entry activity ‘quitAction()’ is executed. All three states transition to a diamond symbol, which leads to the end of the state machine.

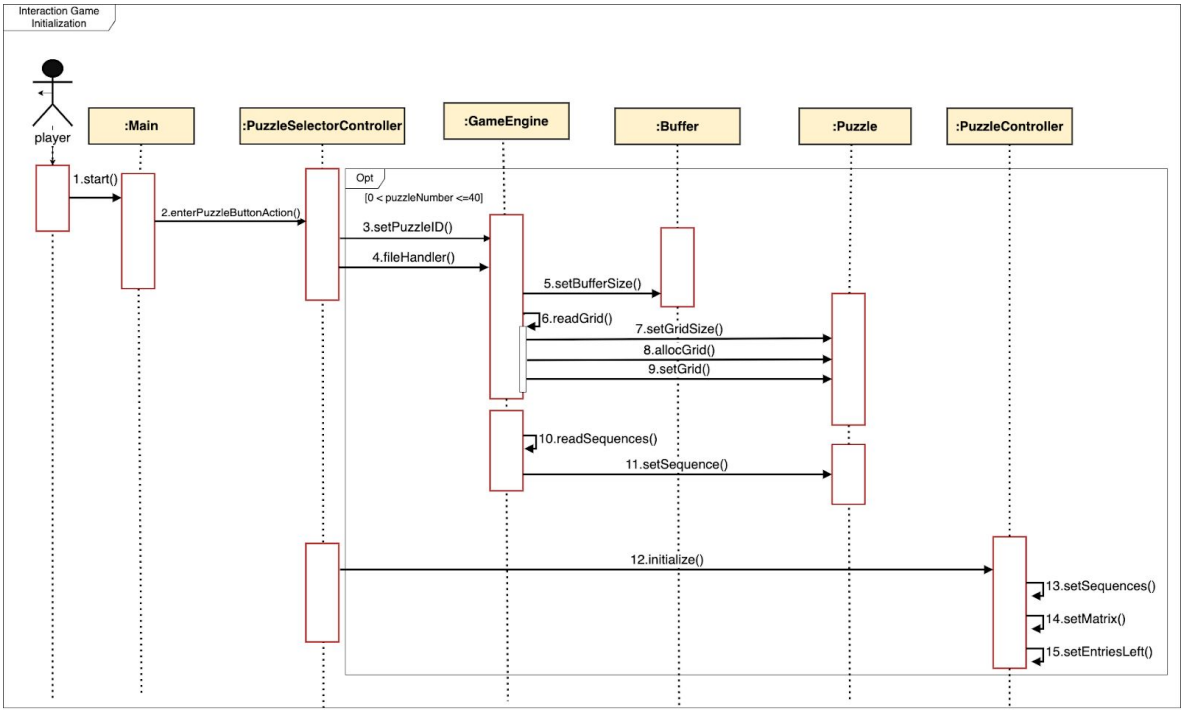
Sequence diagrams

Author(s): Xiaoxuan Lu

In this section, we will implement two sequence diagrams. The first sequence diagram is depicting the initialization of the game and the second sequence diagram is depicting the game execution while the player clicks. The sequence diagrams specify how messages and data are exchanged among objects.

Game Initialization Sequence

The sequence diagram of the game initialization is shown below. It covers the whole process from when the player opens the game up until they start playing the game. There are no return values, as we only have set methods during the process.



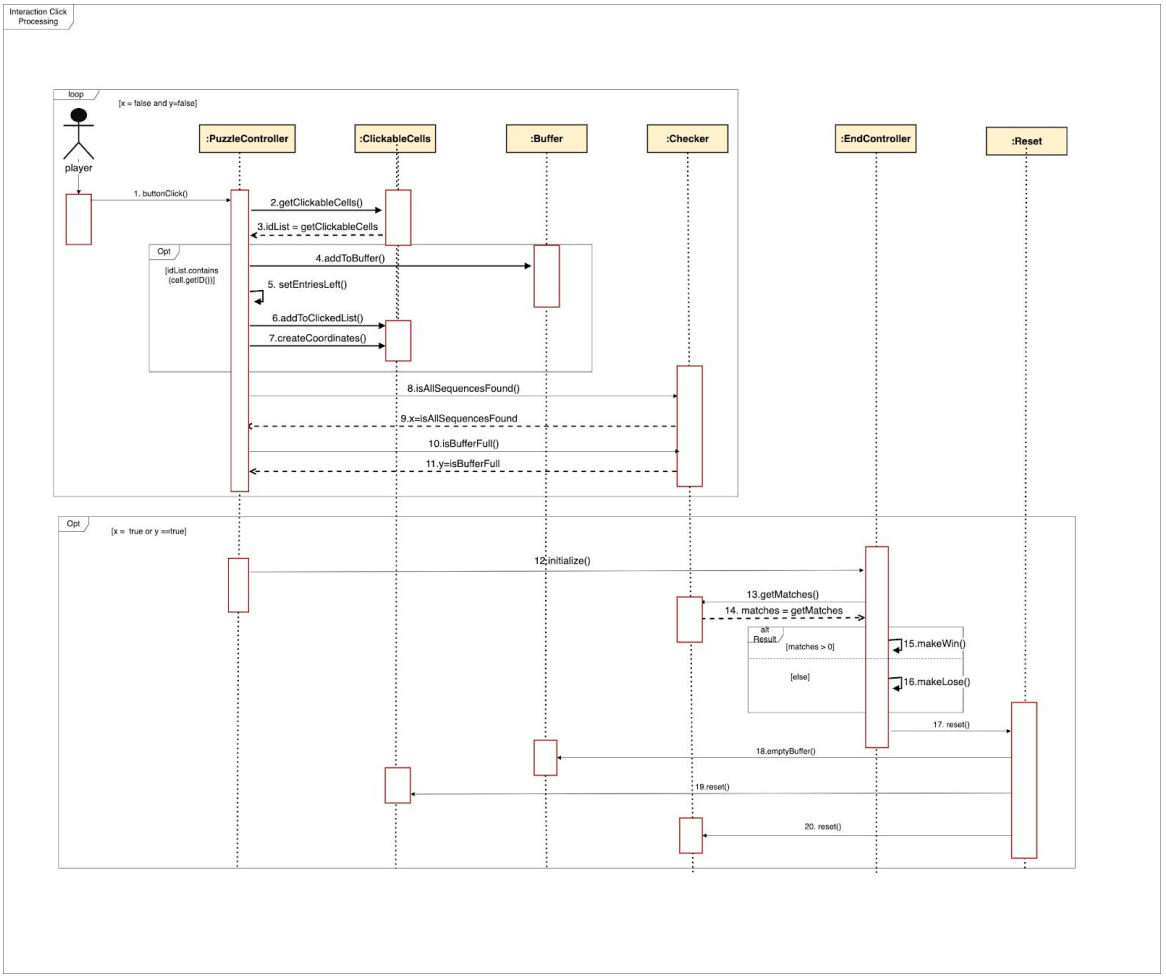
The game initialization process starts when the player opens the game. As the game opens, the main object is invoked by the player and the main object calls the *start*() function at the same time. **Main** is the main controller of the game. The object **Main**, when initialized, will invoke the **PuzzleSelectionController** by loading puzzleSelector.fxml. This is abstracted in the sequence diagram, because loading the .fxml file does not invoke any objects or methods. The initialization process occurs from this step, as explained next.

PuzzleSelectorController is invoked and starts asking the player to enter the number of puzzles he wants to choose by utilizing the *enterPuzzleButtonAction*() function. The method gets a number (*puzzleNumber*) from the player. The following sequences only occur when the *puzzleNumber* is not empty and the *puzzleNumber* is between 1 and 40(0<*puzzleNumber*<=40). This is because we only have 40 puzzle files named as 1 to 40. Thus, the following sequences are in an optional interaction. When the *puzzleNumber* meets the Guard, the *enterPuzzleButtonAction*() function in **PuzzleSelectorController** will utilize

the *setPuzzleID*() and *fileHandler*() functions of **GameEngine**. In *fileHandler*() method, **GameEngine** first invokes Buffer object to utilize the *setBufferSize*() function to set the length of the buffer in the game by reading the buffer size in the file. Then **GameEngine** calls self message - *readGrid*() function, which further calls *setGridSize*(), *allocGrid*(), and *setGrid*() functions of **Puzzle** object. These functions do not return anything, and they prepare the *grid* of the game by storing the *grid* in the file. After preparing the *grid*, the fileHandler will continue calling the self message - *readSequence*() function, which further calls the *setSequence*() function from the **Puzzle** object to prepare the sequence lists needed to upload in the game from the file. From 5 to 11, all the methods are called in the fileHandler. The fileHandler reads and initializes everything the game needs from the file chosen. When the interactions in the fileHandler finish, **PuzzleSelectorController** invokes the **PuzzleController** object by loading the puzzle.fxml which happens in the *enterPuzzleButtonAction*() method. This is abstracted in the sequence diagram, as loading the .fxml file does not invoke any objects or methods. As the **PuzzleController** object is invoked, the *initialize*() method will be automatically invoked. The *initialize*() function in the **PuzzleController** will call three self messages: *setSequence*(), *setMatrix* and *setEntriesLeft*(). The player can now play the game.

Click Processing Sequence

The second sequence diagram of the click processing is as follows.



The diagram follows the process involved when the player clicks the button among the *grid*. The player continues clicking a button until all sequences are found as long as the buffer is not full. Therefore, the clicking is inside a loop. I will explain the interactions inside the loop first. The loop starts when the player clicks a button in the *grid*. As the player clicks, the *buttonClick*() function of **PuzzleController** object will be invoked. The *buttonClick*() function further calls *getClick*() function of a **ClickableCells** object, and *getClick*() will return an ArrayList<String>(*idList*) to the **PuzzleController** object. The *idList* contains all clickable cells at this moment. The system will check if the button Clicked by the player is in the *idList*. Thus, there is an option interaction, and the guard for that is *idList.contains*(*cell.getID*()). When the button Clicked by the player is not in the *idList*, nothing will happen. However, if the guard meets, the *buttonClick*() method of the **PuzzleController** object will invoke *addToBuffer*() function from the **Buffer** object to update the buffer in the game. Then the **PuzzleController** object calls self message - *setEntriesLeft*() to update the entries left in the buffer. Finally, in the opt interaction, **PuzzleController** calls *addToClickedList*() and *createCoordinates*() from the **ClickableCells** object to update *idList*. Then, **PuzzleController** object invoked *isAllSequenceFound*() and *isBufferFull*() functions from **Checker** object, and **Checker** will return booleans x and y to **PuzzleController**. If either or y is true, the loop terminates, and the system checks the result of the game.

The interactions of the result in the game are in the optional interaction, because only when x = true or y = true (all sequences found by the player or the buffer is filled), the interactions will occur. This happens when the player finishes clicking, and the system will show the result to the player. In the opt, the **PuzzleController** invokes the **EndController** object by loading the end.fxml file. When the **EndController** is invoked, the *initialize*() method is automatically called. The *initialize*() method utilizes the *getMatches*() function from **Checker** to find the number of sequences found by the player and return the number back to the **EndController** as *matches*. There are two situations in the game, one is win and another is lose. If *matches* > 0, the player wins the game, otherwise, they lose the game. Thus, there is an alt interaction. When *matches* > 0, **EndController** calls self message - *makeWin*() to show the winning page to the player. If matches are not bigger than 0, the EndController calls *makeLose*() function to show the losing page to the player. After showing the result to the player, **EndController** calls *reset*() operation of the **Reset** object to clear all data in the game. The reset() method in the **Reset** calls *emptyBuffer*() function in the **Buffer**, *reset*() function in the **ClickableCells** and *reset*() function in the Checker to reset all values in the game. This is the whole process of clicking by the player, and in the sequence diagram we don't consider what can be chosen by the player when the game ends.

Implementation

Author(s): Emre Furkan Guduk, Yasir Uçkun

The Strategy:
The Agile Development Process was applied. First, a basic class diagram was created. In order to see what methods we will use, we depicted some attributes and functions within classes. After we completed our “sketch”, the implementation started.
With the implementation of all classes and methods without relations among each other, we had some ideas about the techniques we will use while creating the game. For instance, the file handling will be done by **GameEngine** and the data read will be sent to **Puzzle**. Basic state-machine diagrams were created and relationships between methods and attributes were added. After implementing the basic functionalities of the game, additions were made to make the game visually pleasing, such as a background and color changing *grid*. Then, we ran our code and tried to collect some feedback about how it could be improved.

Instead of working on parts one by one until they were perfected, it was decided to write outlines and improve them along the way. By using this method, it could be determined which parts were irrelevant and which were absolutely necessary in the final product. In the end, a game with the desired features was successfully created.

Key Solutions:
The first challenge was to find a way to know which .txt file should be loaded into the game board. To perform this, we asked for user input (an integer between 1-40 inclusive) and accordingly the corresponding .txt file was read. After reading the file, the variables *grid* and *sequence* under the **Puzzle** class and the *bufferSize* under the **Buffer** class were set.

Since we implemented our code on Scenebuilder, we only needed to create functionalities for the components on the screen. For instance, after the *grid* is loaded, a clicking action is allowed only for the buttons on the *grid*. We met this requirement by making the *grid* as the solid clickable part of the game. However, it led to another problem, namely that we had to find a way to keep track of whether the current move should be from a row or a column so that we can arrange a set of cells the player can actually interact with as the next move. In this case, every single entry on the *grid* was clickable, but not all of them were functional. This strategy helped us to build a set called *idList* under **ClickableCells** by using another variable under **ClickableCells** named *direction* (an integer alternating between 0 and 1). Initially, the direction is set to 0, meaning that all entries in the given **row** must be functional (any elements in that list can be passed to the buffer) and interactable entries have to be in the first row.

After this point, we encountered the problem that if we save an interactable buttons list with the values on the buttons, it may end up allowing the user to pick **any button on the grid** with a value matching an element from the list. For instance, if we have a list containing an element “e5” and if there is another “e5” in some other redundant row or column of the *grid*, the system would allow the user to pick that tile. In order to prevent such a situation, we created a function *createCoordinates*() under **ClickableCells**, to create a system where the clickable values are saved as coordinates rather than actual values. By using this coordinate system, we checked whether the clicked button’s coordinates are in the *idList* (coordinates of

all the clickable values in the *grid*). If a valid move was made, we changed the *direction* (if 1 set to 0, if 0 set to 1) and updated the *idList* by taking into consideration the *direction* and the clicked button's coordinates.

The location of the main Java class needed for executing our system in the code:

- "src/main/java/softwaredesign/Main.java"

The location of the Jar file for directly executing the system:

- "out/executable.jar

Execution of our system:

- https://youtu.be/WU3vpoQQGb4

Time logs

1	**Cyberpunk Group 12**	15		
2				
3	**Member**	**Activity**	**Week number**	**Hours**
4	X. Lu (Xiaoxuan)	discuss class diagram	3	2
5	E.F.Guduk(Emre)	discuss class diagram	3	2
6	Y.Y. Ma (Ying Ying)	discuss class diagram	3	2
7	Ö.F. Çakici (Omer Faruk)	discuss class diagram	3	2
8	X. Lu (Xiaoxuan)	draw the class diagram	4	6
9	E.F.Guduk(Emre)	draw the class diagram	4	6
10	Y.Y. Ma (Ying Ying)	draw the class diagram	4	6
11	Ö.F. Çakici (Omer Faruk)	draw the class diagram	4	6
12	A.Y.Uçkun(Yasir)	draw the class diagram	4	3
13	X. Lu (Xiaoxuan)	draw the sequence diagra	4	6
14	E.F.Guduk(Emre)	implement code	4	6
15	Y.Y. Ma (Ying Ying)	state diagram	4	6
16	Ö.F. Çakici (Omer Faruk)	object diagram	4	10
17	A.Y.Uçkun(Yasir)	implement code	4	10
18	X. Lu (Xiaoxuan)	debug the code/rewrite the	4	5
19	Y.Y. Ma (Ying Ying)	write explanation of state c	4	2
20	Y.Y. Ma (Ying Ying)	modify class diagram	4	1
21	A.Y.Uçkun(Yasir)	work on the code, and jar	5	5
22	E.F.Guduk(Emre)	work on the documentatio	5	5
23	X. Lu (Xiaoxuan)	work on the documentatio	5	5
24	Y.Y. Ma (Ying Ying)	work on the documentatio	5	5
25	Ö.F. Çakici (Omer Faruk)	work on the documentatio	5	5
26				
27				
28			**TOTAL**	106
29				