Assignment 3

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

Format:
All class names are in bold form, all variables, attributes, methods are in italic form and all associations are underlined and in italic form. All the constant values are typed in all capitals (e.g. PUZZLE_LOCATION)

Summary of changes of Assignment 2

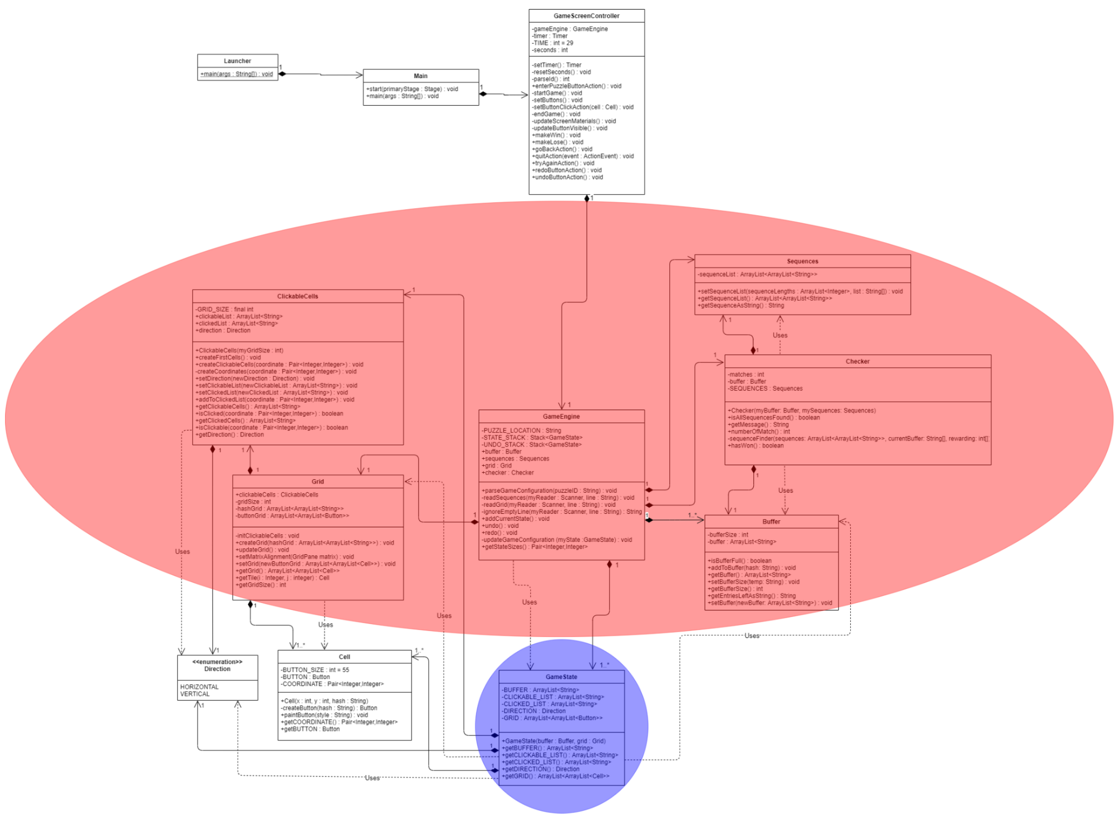
Author(s): All

- In the implementation of the code, many redundant parts have been removed. Duplicated parts of the code have been encapsulated with functions in order to be used in different methods. Return values are not the direct object themselves anymore, but the copy of an original object. Some labels and classes have been renamed in order to increase readability and some functions have been reduced in size in order to keep single functionality per function.
- New comments have been added to increase understandability and an enumerator has been created in order to take over the *direction's* responsibility.
- A brand new class named Cell has been created in order to fix the primitive coordinate system problem. The Cell class is used to avoid any primitive obsessions and to obtain a higher quality design system.
- In this version of the system, the implementation of the system is based on object-oriented programming structure instead of static classes. Besides, game initialization and game termination processes are handled in an object-oriented friendly design structure. When the game starts, initial components are created as objects and those objects are deleted as soon as the game is terminated.

- The state machine diagrams were modified to represent states of specific classes instead of states of the entire application. This means that the diagram for the overall state was modified to represent the PuzzleSelectorController class and the diagram for the game over state was modified to represent the EndController class. In the first diagram, the initial 'Idle','Puzzle chosen', 'Buffer full' and 'All sequences found' states were removed. A missing [else] guard was added. A new transition was added from 'Cell chosen' to the diamond symbol, with the guard [timer==0). The composite 'Game ended' state was removed and then added to the second diagram. In the composite state, the initial state was replaced by a diamond symbol.
- "There is no execution bar in an actor's lifeline.", Milica (TA)
  Delete all execution bars from the player in the sequence diagram. For game initialization, there is only one enter puzzle move by the player.
- Improve the format and style in the sequence diagram that all execution bars starting from the same line as the operation arrow and write the parameters inside the brackets for the operation. Moreover, the self call
- The upper margin of the loop segment is placed below the object's rectangle representations and the actor.
- IsAllSequencesFound() and isBufferFull() operations are put inside the opt segmentation. Delete the opt segmentation when the clicking by the player stops, because the game achieves the end.

Application of design patterns

Author(s): Emre Furkan Guduk, Ahmet Yasir Uckun



Command is marked with a blue circle. Singleton is marked with a pink ellipse.

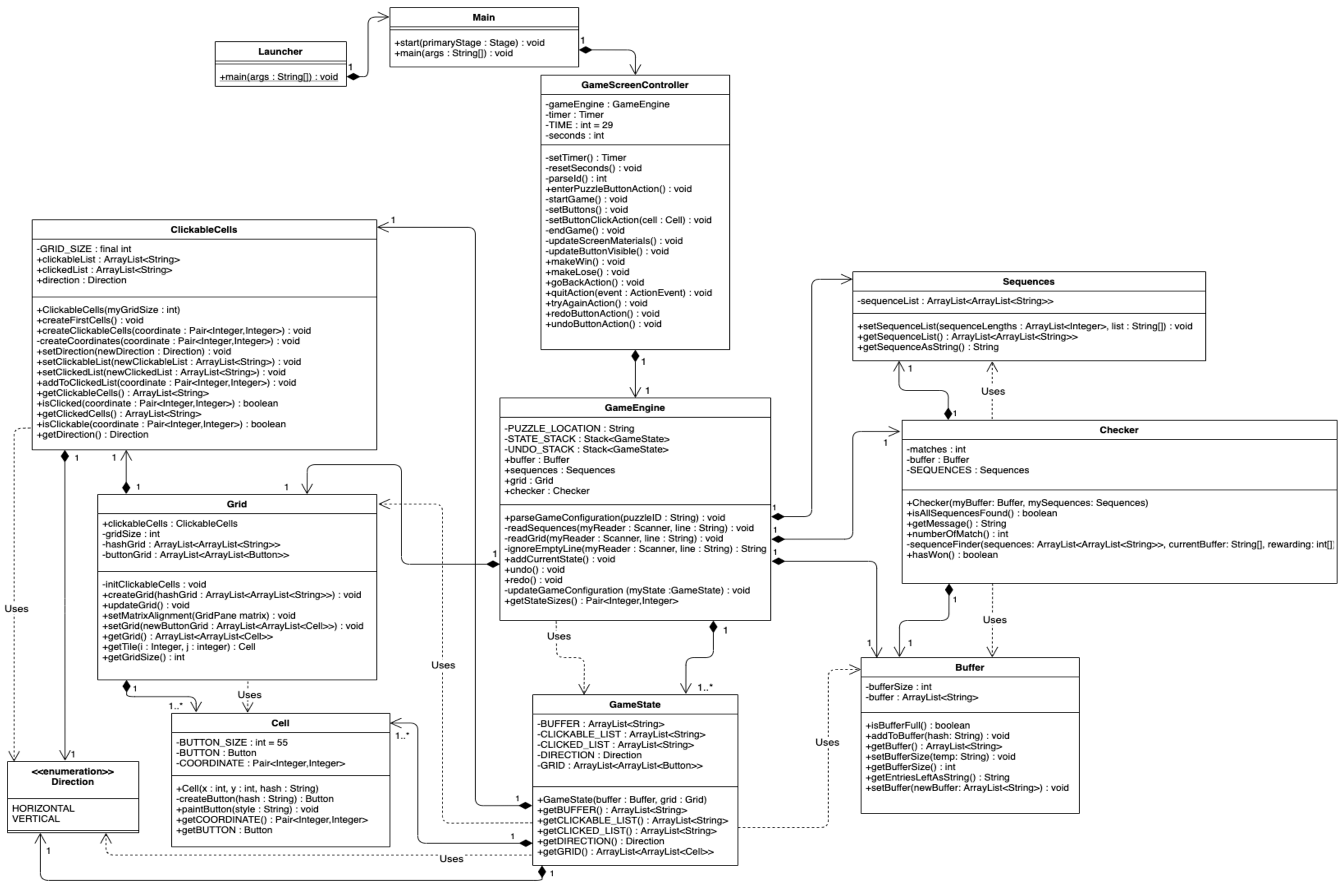
	DP1
Design pattern	Singleton
Problem	Having more than one principal component during the runtime might cause a variety of problems.
Solution	The Singleton Design Pattern is applied in order to ensure that there exists exactly one instance of each class. This helped us to prevent having more than one buffer, sequence, grid and checker object. So, no external interference which was able to modify the game’s content has occurred during the runtime.
Intended use	For the intention of the use of Singleton, here are a couple of examples: 1. When the *checker* object checks the number of matches found in the *buffer*, if there are more than one *buffer*, then the *matches* under **Checker** class will not give the correct number of found sequences. 2. Same story from a different aspect can be taken into consideration here. If there exists more than one **Sequence** object and if those objects contain different sequences, then the *matches* will not show the correct number of found sequences. 3. In case there were more than one *grid* in the game, it would cause confusion for both the player and the designer to know where to interact with and where to expect an input from. Since the grid is an interactable part of the game, there mustn’t be more than one instance of the *grid*. 4. If there were more than one **Checker** object, then the **GameEngine** would have two identical objects with the same intention. In such a case, it would be redundant to have the copy of an object. Because the **Checker** object is used in order to check if there are any matches in the *buffer*. It is clear that the number of objects must be at least one and at most one in all the cases above. So, this problem can be fixed very well with the Singleton Design Pattern.
Constraints	Since Singleton is already so close to the designed system of ours, there are no further constraints in this matter.
Additional remarks	In some cases, Singleton could not be used anymore (because of the number of object limits per class). That’s why we needed to find another design pattern.

	DP2
Design pattern	Command
Problem	A move must be placed and performed but it also should be saved for later use. Otherwise, complexity and the cost of the design for later use of the previous states of the game will increase.
Solution	The Command Design Pattern is applied to save each of the interaction

	requests taken by the user. When the player wants to make a move, a GameState object is generated and the previous request is saved inside of the newly created object.
Intended use	There are a few reasons for the usage of Command Design Pattern. Those are: 1. Reducing the complexity of undo button actions for later use. As a briefing, if the player wants to switch back to one of the previous stages of the game, it should not be too laborious. 2. Reducing the complexity of redo button actions for later use. When the user wants to perform a redo action, it shouldn't be too laborious. 3. Reducing the cost for redo/undo actions. Basically, when the user performs an undo/redo action, it should use as few sources as possible. 4. Since the interactions are saved as an object in Command Design Pattern, it's easy to observe what has been previously done at any time during the runtime. Also, this system can be used to train ML models how to play the game.
Constraints	In Command Design Pattern, an operation must be saved as an object. In our game, all the interactions are saved as an object. So, Command Design Pattern fits perfectly to our system.

Class diagram

Author(s): Emre Furkan Guduk, Ahmet Yasir Uckun


Launcher
+main(args : String[]) : void
Main
+start(primaryStage : Stage) : void
+main(args : String[]) : void
GameScreenController
-gameEngine : GameEngine
-timer : Timer
-TIME : int = 29
-seconds : int
-setTimer() : Timer
-resetSeconds() : void
-parseId() : int
+enterPuzzleButtonAction() : void
-startGame() : void
-setButtons() : void
-setButtonClickAction(cell : Cell) : void
-endGame() : void
-updateScreenMaterials() : void
-updateButtonVisible() : void
+makeWin() : void
+makeLose() : void
+goBackAction() : void
+quitAction(event : ActionEvent) : void
+tryAgainAction() : void
+redoButtonAction() : void
+undoButtonAction() : void
ClickableCells
-GRID_SIZE : final int
+clickableList : ArrayList<String>
+clickedList : ArrayList<String>
+direction : Direction
+ClickableCells(myGridSize : int)
+createFirstCells() : void
+createClickableCells(coordinate : Pair<Integer,Integer>) : void
-createCoordinates(coordinate : Pair<Integer,Integer>) : void
+setDirection(newDirection : Direction) : void
+setClickableList(newClickableList : ArrayList<String>) : void
+setClickedList(newClickedList : ArrayList<String>) : void
+addToClickedList(coordinate : Pair<Integer,Integer>) : void
+getClickableCells() : ArrayList<String>
+isClicked(coordinate : Pair<Integer,Integer>) : boolean
+getClickedCells() : ArrayList<String>
+isClickable(coordinate : Pair<Integer,Integer>) : boolean
+getDirection() : Direction
Sequences
-sequenceList : ArrayList<ArrayList<String>>
+setSequenceList(sequenceLengths : ArrayList<Integer>, list : String[]) : void
+getSequenceList() : ArrayList<ArrayList<String>>
+getSequenceAsString() : String
GameEngine
-PUZZLE_LOCATION : String
-STATE_STACK : Stack<GameState>
-UNDO_STACK : Stack<GameState>
+buffer : Buffer
+sequences : Sequences
+grid : Grid
+checker : Checker
+parseGameConfiguration(puzzleID : String) : void
-readSequences(myReader : Scanner, line : String) : void
-readGrid(myReader : Scanner, line : String) : void
-ignoreEmptyLine(myReader : Scanner, line : String) : String
+addCurrentState() : void
+undo() : void
+redo() : void
-updateGameConfiguration (myState :GameState) : void
+getStateSizes() : Pair<Integer,Integer>
Checker
-matches : int
-buffer : Buffer
-SEQUENCES : Sequences
+Checker(myBuffer: Buffer, mySequences: Sequences)
+isAllSequencesFound() : boolean
+getMessage() : String
+numberOfMatch() : int
-sequenceFinder(sequences: ArrayList<ArrayList<String>>, currentBuffer: String[], rewarding: int[])
+hasWon() : boolean
Grid
+clickableCells : ClickableCells
-gridSize : int
-hashGrid : ArrayList<ArrayList<String>>
-buttonGrid : ArrayList<ArrayList<Button>>
-initClickableCells : void
+createGrid(hashGrid : ArrayList<ArrayList<String>>) : void
+updateGrid() : void
+setMatrixAlignment(GridPane matrix) : void
+setGrid(newButtonGrid : ArrayList<ArrayList<Cell>>) : void
+getGrid() : ArrayList<ArrayList<Cell>>
+getTile(i : Integer, j : integer) : Cell
+getGridSize() : int
Uses
Uses
Uses
Uses
Uses
Uses
Uses
Uses
Buffer
-bufferSize : int
-buffer : ArrayList<String>
+isBufferFull() : boolean
+addToBuffer(hash: String) : void
+getBuffer() : ArrayList<String>
+setBufferSize(temp: String) : void
+getBufferSize() : int
+getEntriesLeftAsString() : String
+setBuffer(newBuffer: ArrayList<String>) : void
Cell
-BUTTON_SIZE : int = 55
-BUTTON : Button
-COORDINATE : Pair<Integer,Integer>
+Cell(x : int, y : int, hash : String)
-createButton(hash : String) : Button
+paintButton(style : String) : void
+getCOORDINATE() : Pair<Integer,Integer>
+getBUTTON : Button
GameState
-BUFFER : ArrayList<String>
-CLICKABLE_LIST : ArrayList<String>
-CLICKED_LIST : ArrayList<String>
-DIRECTION : Direction
-GRID : ArrayList<ArrayList<Button>>
+GameState(buffer : Buffer, grid : Grid)
+getBUFFER() : ArrayList<String>
+getCLICKABLE_LIST() : ArrayList<String>
+getCLICKED_LIST() : ArrayList<String>
+getDIRECTION() : Direction
+getGRID() : ArrayList<ArrayList<Cell>>
<<enumeration>>
Direction
HORIZONTAL
VERTICAL
1
1..*

Buffer

Buffer is the class that holds the buffer information as ArrayList<String>. It is created by the **GameEngine** class.

Fields:
- **bufferSize : int -** This integer holds the information of the maximum capacity of the buffer.
- **buffer : ArrayList<String> -** This container holds the string information. It is updated after every click made in the matrix.

Operations:
- **isBufferFull() : boolean -** This function is used to detect if the buffer is full. It helps to call the endGame() in **GameScreenController.**
- **addToBuffer(hash : String) : void -** This function adds a string value to the buffer. It is called in the setButtonClickAction() in **GameScreenController** whenever a button is pressed in the grid.
- **getEntriesLeftAsString() : String -** Returns a string to show the user how many more entries he/she has.
- **setBuffer(ArrayList<String> newBuffer) : void -** Sets the buffer list to the input. This method is used for the undo and redo methods in **GameEngine**.

Cell

This class' main purpose is to have coordinate info and button objects in one place. Since the button can not have a coordinate info inside of it, this class does that job for the rest of the code.

Fields:
- **BUTTON_SIZE : int = 55 -** This is for easier code managing, changing this value will be easier than looking for it in the file.
- **BUTTON : Button -** This is the button object **Cell** holds. It is reached by **GameScreenController** later to make it functional.
- **COORDINATE : Pair<Integer, Integer> -** This is a container for the coordinate variable of the **Cell**. Since the button object can now have such a container, we implemented the **Cell** class to hold everything in one object.

Operations:
- **Cell(x : int, y : int, hash : String) -** This is the constructor method. It has the arguments for the coordinates and the hash string for the button text.
- **createButton() : Button -** This is a helper method for creating a button with the given hash in the constructor. It returns a button with the desired text and size.
- **paintButton(style : String) : void -** This method is used for updating the grid. It takes the style string as input. It basically changes the color of the button to red, green or null.

Checker

This class is focused on win/lose methods. After every move and at the end of the game, this class does the calculation of the score and returns the information.

Fields:

- **matches : int -** This holds the number of sequences that match with the hashes in the buffer.
- **BUFFER: Buffer -** This is assigned to reach the information of the buffer object.
- **SEQUENCES : Sequences -** This is assigned to reach the information of the sequences object.

Operations:

- **Checker(myBuffer : Buffer, mySequences : Sequences) -** This is the constructor method. It is called in the **GameEngine**. Arguments are for being able to reach the information in the buffer and sequences objects.
- **isAllSequencesFound() : boolean -** This method is called after every move. It helps to call the endGame() in **GameScreenController** at any point where all sequences are found.
- **getEndMessage() : String -** This method returns the end screen message. If the game is won, the message will also contain the number of sequences found.
- **numberOfMatch() : int -** This method finds the number of sequences that match with the current buffer hashes.
- **sequenceFinder(sequences : ArrayList<ArrayList<String>>, currentBuffer : String[], rewarding : int[]) -** This is a helper method to find the sequences in the current buffer. It ignores the repetition of the sequences in the buffer.
- **hasWon : boolean -** Returns true if there are any matches between buffer and sequences.

Associations:

- Has a composition to **Buffer** class. Also, **Checker** uses (*dependency*) the *getBuffer*() method under the **Buffer** class.
- Has a composition to **Sequences** class. Also, **Checker** uses (*dependency*) the *getSequenceList*() method under **Sequences** class.

ClickableCells

This class does the calculation for which cells are going to be clickable in the next move. The calculation is done with the new coordinates after every move.

Fields:

- **GRID_SIZE : int -** This field holds the matrix size to do numerical calculations.
- **clickableList : ArrayList<Pair<Integer,Integer>> -** This is a list that holds the coordinates of the clickable cells. It is initialized in the construction of the class, and calculated again after every move.
- **clickedList : ArrayList<Pair<Integer,Integer>> -** This list holds the coordinates of the clicked cells.
- **direction : Direction = Direction.HORIZONTAL -** Global enumerator that helps to calculate the direction of the next cells which will be clickable.

Operations:

- **ClickableCells(myGridSize: int) -** This is the constructor method. It has the grid size argument. This is the reason why it is called under the grid instead of **GameEngine**
- **createFirstCells() : void -** Creates the first coordinates which are the first row cells of the grid.

- **createClickableCells(coordinate : ArrayList<Pair<Integer,Integer>>) : void -** Creates the clickable cell coordinates with the argument and the global enumerator.
- **createCoordinates(coordinate : ArrayList<Pair<Integer,Integer>>) : void -** This method helps to create coordinates.
- **addToClickedList(coordinate : ArrayList<Pair<Integer,Integer>>) : void -** This is called in the setButtonClickAction() method. It adds the current coordinate to the *clickedList*. So the same button will not be pressed twice.
- **isClicked(coordinate : ArrayList<Pair<Integer,Integer>>) : boolean -** This method checks if a cell is clicked
- **isClickable(coordinate : ArrayList<Pair<Integer,Integer>>) : boolean -** This method checks if a cell is clickable.

Associations:

- Has a composition with an enumerator named **Direction. Direction** helps define the current clickableList (either Horizontal line or Vertical Line).

GameEngine

GameEngine is the backbone of our system. It creates and instantiates the principal components of the game such as **Grid**, **Buffer**, **Sequences**, **Checker**. Besides, It takes care of all the internal operations regarding the instantiated components and communication between different parts.

Fields:

- **PUZZLE_LOCATION : String = “puzzles/” -** This is for easier editing in code.
- **STATE_STACK : Stack<GameState> -** This stack holds the **GameState** objects for every move.
- **UNDO_STACK : Stack<GameState> -** This stack holds the **GameState** objects whenever the undo button is pressed. With the help of this container, the redo button can recall the **GameState** objects.
- **buffer : Buffer -** This is the buffer class used in the whole game.
- **sequences : Sequences -** This is the sequences class used in the whole game.
- **grid : Grid -** This is the grid class used in the whole game.
- **checker : Checker -** This is the checker class used in the whole game.

Operations:

- **parseGameConfiguration(puzzleID : String) : void -** This is the first method called in the start of the game. This method manages the information in the file with the helper methods.
- **readSequences(myReader : Scanner, line : String) : void -** Reads the sequences from the file and fills the containers in the sequences.
- **readGrid(myReader : Scanner, line : String) : void -** Reads the grid from the file and fills the containers in the grid.
- **ignoreEmptyLine(myReader : Scanner, line : String) : String -** This is a helper method to ignore empty lines in the file.
- **addCurrentState() : void -** Adds the current state to the *STATE_STACK* after every move.
- **undo() : void -** Calls the previous **GameState** and updates the containers.
- **redo() : void -** Calls the last undone **GameState** and updates the containers.

- **updateGameConfiguration (myState :GameState) : void -** This is a helper method to update the game configuration. It basically updates the current *buffer*, *grid*, *clicakableList* and *clickedList* by using its argument.

Associations:
Since the game is based on fundamental components which are Buffer, Sequences, Checker, Grid and GameState, GameEngine cannot exist without those primary classes.

- Has a composition with **Sequences** class. **GameEngine** generates a **Sequences** object where it separates the work related to each of the correct sequences held in *sequenceList* that must be found in the **Grid**.
- Has a composition with **Buffer** class. **GameEngine** creates a **Buffer** object in order to divide the input wise operations regarding the *buffer* content.
- Has a composition with **Checker** class. **GameEngine** generates a **Checker** object by taking into consideration **Buffer** and **Checker** where it controls if any sequence is found.
- Has a composition with **Grid** class. **GameEngine** generates the **Grid** object and takes care of the operations related to the interactable *grid* under this object. All the visual changes are handled with this object.
- Has a composition with **GameState** class. Whenever the player wants to make a move, the **GameEngine** creates a **GameState** object and saves it inside of the *STATE_STACK* with the relevant information regarding the current game components(buffer, grid etc.). Also, if the user wants to undo/redo a move, then **GameEngine** uses (*dependency*) the **GameState** objects in order to update the current principal components(*buffer, grid etc.*).

GameScreenController

GameScreenController is one of the major classes in the system. It provides the connection between the graphical user interface and the code. It also generates a **GameEngine** object in order to separate the code work(how the files are read etc.) from the visualization work(color of the tiles etc.).

Fields:

- **gameEngine : GameEngine -** This is the **GameEngine** object that will be used in the whole game process. It will be reset to a new one after play again buttons are used.
- **timer : Timer -** This is the timer object for the countdown feature of the game.
- **TIME : int = 29 -** This is the time constant for the timer function. It is a weird number due to the delay of the timer. It will be one second longer than this value in the game screen.
- **seconds : int -** This number is positioned here because the **Timer** object accepts variables from outside of itself.

Operations:

- **setTimer() : Timer -** This function sets the timer to 30 seconds. Whenever time is up, the endGame function is called.
- **resetSeconds() : void -** This is a simple helper method to reset the seconds after a restart.
- **enterPuzzleButtonAction() : void -** This is the first ever executed action. Whenever the player presses enter in the puzzle selector screen, this function is called.

- **startGame() : void -** This is the one of the most important methods in this class. It initializes the game. By creating a new **GameEngine** object and calling *parseGameConfiguration*(), it starts the game and switches the scene to puzzle scene.
- **setButtons() : void -** This function creates cells and makes them functional with the help of *setButtonClickAction*() method. Then it fills the gridpane with the buttons of those cells.
- **setButtonClickAction(cell : Cell) : void -** This is the helper method for functionality of the buttons on the grid.
- **endGame() : void -** This method is called in 3 situations. When the buffer is full, when all sequences are found and when time is up. It switches scene to end scene.
- **updateScreenMaterials() : void -** This is a helper method to update screen materials. It is called after every change done in the game (Clicking a tile in grid, undo, redo).
- **updateButtonVisible() : void -** This sets the button visibility for undo and redo. Whenever there are moves for undo or redo, it makes the relative button visible.
- **makeWin() : void -** This is the helper method for the end scene. It is called when the player finds at least one sequence.
- **makeLose() : void -** This is the helper method for the end scene. It is called when the player could not find any sequences.
- **goBackAction() : void -** This is the button action for the “Try another puzzle” button in the end scene.
- **quitAction(event : ActionEvent) : void -** This is the button action for the “Quit” button in the end scene. It closes the window.
- **tryAgainAction() : void -** This is the button action for the “Try the same puzzle” button in the end scene. It restarts the game without asking for a puzzle number.
- **redoButtonAction() : void -** This button calls the *redo* function in **GameEngine**. It recalls the last move that is undone.
- **undoButtonAction() : void -** This button calls the undo function in **GameEngine**. It undoes a move.

Associations:
The game starts with the **GameScreenController** class and **GameScreenController** generates a **GameEngine** object which takes care of the rest of the internal operations (generating basic objects such as *grid*, *buffer*, *checker*, *sequences* etc.). That’s why there is a strict relationship between **GameScreenController** and **GameEngine**.

- Has a composition for **GameEngine** class. In the beginning of the game it creates a **GameEngine** object and initializes its materials. By using this object, **GameScreenController** reaches all classes which belong to **GameEngine** such as **Buffer**, **Sequences**, **Grid** etc.

GameState

GameState is a fundamental class for redo and undo operations. As a briefing, when the player interacts with the grid –attempting to make a move–, a GameState object is created and all of the necessary game components are saved inside of the newly created object. By using this system, it’s easy to go back and forth between the game states.
Fields:

- **BUFFER : ArrayList<String> -** Saves the content of the current buffer in the ArrayList form.
- **CLICKABLE_LIST : ArrayList<String> -** Saves the content of the clickableList under the ClickableCells class in the ArrayList form.
- **CLICKED_LIST : ArrayList<String> -** Saves the content of the clickedList under the ClickableCells class in the ArrayList form.
- **DIRECTION : Direction -** Saves the direction of the state(Horizontal/Vertical).
- **GRID : ArrayList<ArrayList<Button>> -** Saves the content of the grid in the form of 2-dimensional ArrayList.

Operations:

- **GameState(buffer : Buffer, grid : Grid) -** Constructor of the **GameState**. It initializes the newly created object with the given *buffer* and *grid* objects and assigns the *BUFFER, CLICKABLE_LIST, CLICKED_LIST, DIRECTION* and *GRID* with the corresponding values.

Associations:

GameState has a direct relationship with **Cell** and **Direction** (enumerator). That's why it cannot exist without **Cell** and **Direction**.

- Has a composition with **Cell**. The **GameState** object creates a *grid* with **Cell** objects in it. It also uses (dependency) these **Cell** objects.
- Has a composition with the enumerator named **Direction**. When the **GameEngine** wants to make a move, it initializes a **GameState** object and the **GameState** object creates a variable in **Direction** type in order to save the current *direction*.
- **GameState** also uses **Grid** in order to store the current *grid* inside of a 2-dimensional ArrayList named *GRID*. Basically, **GameState** saves the current *grid*'s content inside of the *GRID*. So, whenever a move is undone, the *GRID* variable under the specified **GameState** object will be used in order to update the current *grid*'s content.
- Uses (dependency) the **Buffer** class in order to save the content of the current *buffer* inside of an ArrayList named *BUFFER*.

Grid

This class encapsulates all the operations and variables regarding the Grid component in the game.

Fields:

- **clickableCells : ClickableCells -** An object which contains the relevant information about the currently available cells, previously used cells and current direction of the game which is used to decide how the next **clickableCells** list is constructed. It is constructed here and reached via **Grid** class. Because the clickableCells need the *gridSize* information, it is decided to be called here.
- **gridSize : int -** Contains the size of the Grid.
- **grid : ArrayList<ArrayList<Cell>> -** A 2-dimensional matrix where each of the tile on the grid is saved as a Cell object. This feature can be seen as the container of the grid where content of the grid is saved.

Operations:

- **initClickableCells : void -** Initializes the clickableCells. It's invoked when the game starts and it initially fills up the clickableCells with the first row of the grid.

- **createGrid(hashGrid : ArrayList<ArrayList<String>>) : void -** This method creates a grid with cells as its children and calls initializeClickableCells.
- **updateGrid() : void -** This method updates the style information of the button in Cells
- **getTile(i : Integer, j : integer) : Cell -** Returns the cell in the clicked coordinate on the grid.

Associations:

- Has a composition with **Cell**. The Grid class creates **Cell** objects and stores them in the *grid*. *updateGrid*() and *getTile*() methods use (*dependency*) these **Cell** objects in order to modify the content of the *grid*.
- Has a composition with **ClickableCells**. Since initializing **ClickableCells** requires *gridSize* variable, the object is created inside of the **Grid** class.

Sequences

Sequences class takes care of the operations related to correct sequence forms read from the source file (given txt file). In general, it's used with getter and setter functions in order to modify or access the content of the sequences.

Fields:

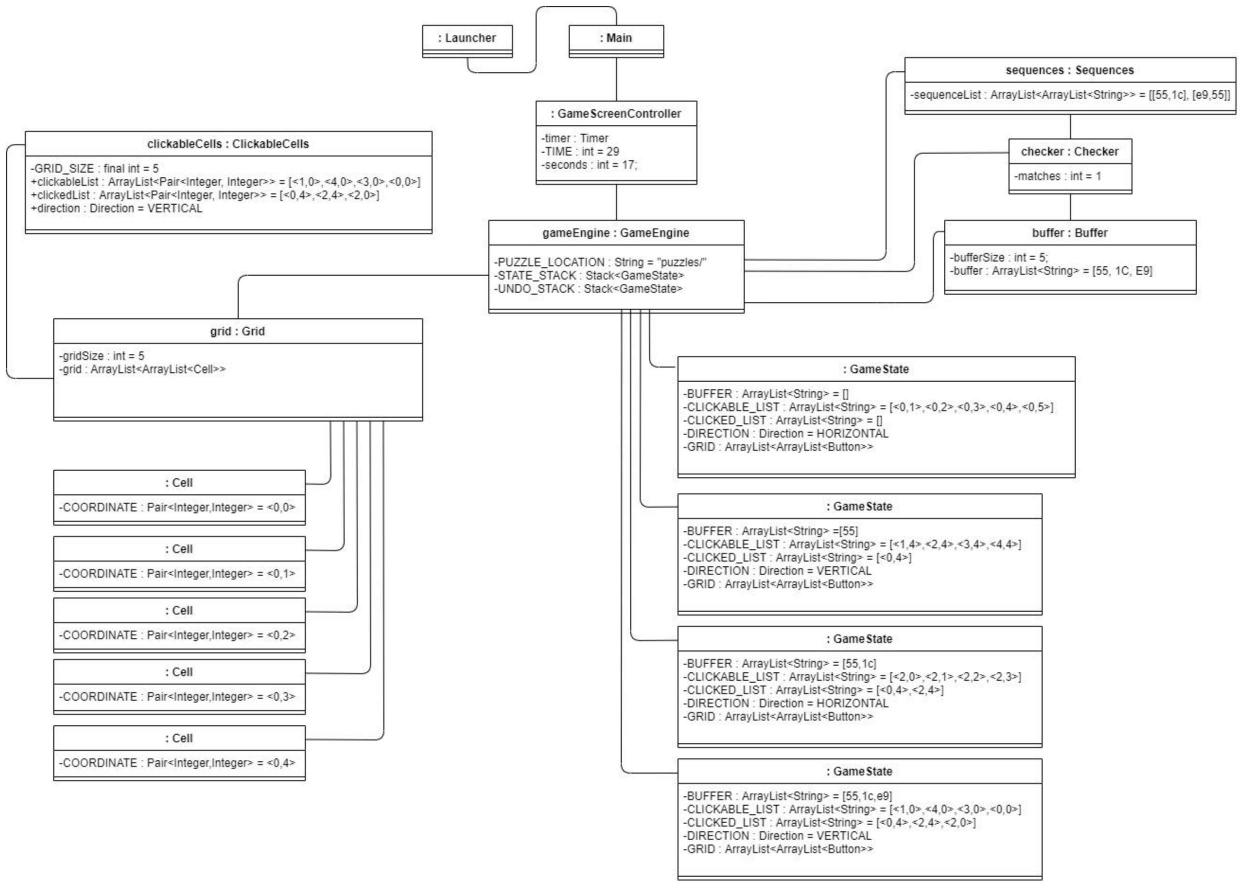
- **sequenceList : ArrayList<ArrayList<String>> -** This is the essential variable where all the correct sequences are saved.

Operations:

- **setSequenceList(sequenceLengths : ArrayList<Integer>, list : String[]) : void -** Fill up the private *sequenceList* variable with the correct entries read from the txt file.
- **getSequenceAsString() : String -** Fetch the content of the *sequenceList* in order to print on the screen.

Object diagram

Author(s): Omer Faruk Cakici



: Launcher
: Main
: GameScreenController
-timer : Timer
-TIME : int = 29
-seconds : int = 17;
sequences : Sequences
-sequenceList : ArrayList<ArrayList<String>> = [[55,1c], [e9,55]]
checker : Checker
-matches : int = 1
buffer : Buffer
-bufferSize : int = 5;
-buffer : ArrayList<String> = [55, 1C, E9]
clickableCells : ClickableCells
-GRID_SIZE : final int = 5
+clickableList : ArrayList<Pair<Integer, Integer>> = [<1,0>,<4,0>,<3,0>,<0,0>]
+clickedList : ArrayList<Pair<Integer, Integer>> = [<0,4>,<2,4>,<2,0>]
+direction : Direction = VERTICAL
gameEngine : GameEngine
-PUZZLE_LOCATION : String = "puzzles/"
-STATE_STACK : Stack<GameState>
-UNDO_STACK : Stack<GameState>
grid : Grid
-gridSize : int = 5
-grid : ArrayList<ArrayList<Cell>>
: Cell
-COORDINATE : Pair<Integer,Integer> = <0,0>
: Cell
-COORDINATE : Pair<Integer,Integer> = <0,1>
: Cell
-COORDINATE : Pair<Integer,Integer> = <0,2>
: Cell
-COORDINATE : Pair<Integer,Integer> = <0,3>
: Cell
-COORDINATE : Pair<Integer,Integer> = <0,4>
: GameState
-BUFFER : ArrayList<String> = []
-CLICKABLE_LIST : ArrayList<String> = [<0,1>,<0,2>,<0,3>,<0,4>,<0,5>]
-CLICKED_LIST : ArrayList<String> = []
-DIRECTION : Direction = HORIZONTAL
-GRID : ArrayList<ArrayList<Button>>
: GameState
-BUFFER : ArrayList<String> =[55]
-CLICKABLE_LIST : ArrayList<String> = [<1,4>,<2,4>,<3,4>,<4,4>]
-CLICKED_LIST : ArrayList<String> = [<0,4>]
-DIRECTION : Direction = VERTICAL
-GRID : ArrayList<ArrayList<Button>>
: GameState
-BUFFER : ArrayList<String> = [55,1c]
-CLICKABLE_LIST : ArrayList<String> = [<2,0>,<2,1>,<2,2>,<2,3>]
-CLICKED_LIST : ArrayList<String> = [<0,4>,<2,4>]
-DIRECTION : Direction = HORIZONTAL
-GRID : ArrayList<ArrayList<Button>>
: GameState
-BUFFER : ArrayList<String> = [55,1c,e9]
-CLICKABLE_LIST : ArrayList<String> = [<1,0>,<4,0>,<3,0>,<0,0>]
-CLICKED_LIST : ArrayList<String> = [<0,4>,<2,4>,<2,0>]
-DIRECTION : Direction = VERTICAL
-GRID : ArrayList<ArrayList<Button>>

This object diagram represents the snapshot of the system after the user clicked three cells respectively: [<0,4>,<2,4>,<2,0>] as you can see from *clickableCells* object in **ClickableCells** class. Moreover it has been 12 seconds since the game started as you can see from fields in the **GameScreenController** class (Time - seconds).

:Launcher - This class calls the Main class and does not have a field.
:Main - Main class does not have any fields, it simply calls GameScreen.fxml and that scene builder file creates a controller object named **GameScreenController**.
:GameScreenContoller - It has three fields. Creates a **GameEngine** and **Timer** object. The **GameEngine** object will be used with all the basic operations regarding the **Buffer**, **Sequences**, **Checker** and **Grid**.
gameEngine : GameEngine - It has three fields. *STATE_STACK* field contains every move of GameState objects. *UNDO_STACK* field stores GameState objects once the undo button is clicked. *PUZZLE_LOCATION* : String = “puzzles/” it gives a solution for effortlessly changing the code.

:GameState - In this snapshot of this system we explicitly showed 4 different **GameState** objects. Firstly the *BUFFER* and *CLICKED_LIST* were empty. After each move, they get more elements and after three moves, they have three items in their list. Initially, *DIRECTION* was HORIZONTAL and each move changed the value of *DIRECTION* back and forth between HORIZONTAL and VERTICAL. *CLICKABLE_LIST* changed during every step of this process.

clickableCells : ClickableCells - *GRID_SIZE* is 5. At this moment of the game, three cells were clicked and added to the *clickedList*: [<0,4>,<2,4>,<2,0>]. *clickableList* consists of the elements of the same column, excluding the cells that have been clicked from the start of the game, particularly: [<1,0>,<4,0>,<3,0>,<0,0>]. *Direction* in this snapshot of the system is VERTICAL.

:Grid - Current size of the *grid* is five. Each tile of the *grid* is saved as a **Cell** object.

:Cell - Normally, we have 25 (5x5) cell objects, but in this diagram only five cells are explicitly shown: [<0,0> ,<0,1>, <0,2>, <0,3>, <0,4>]

:buffer : Buffer - In this moment of the game, buffer has three items because the user clicked three cells respectively: [55, 1C, E9]. **Buffer** will hold them until all the entries of the **Buffer** are filled.

:checker : Checker - *matches* is equal to 1. This means that a correct sequence is found in the **Buffer** object. When it finds a match, it increases the *matches* by one.

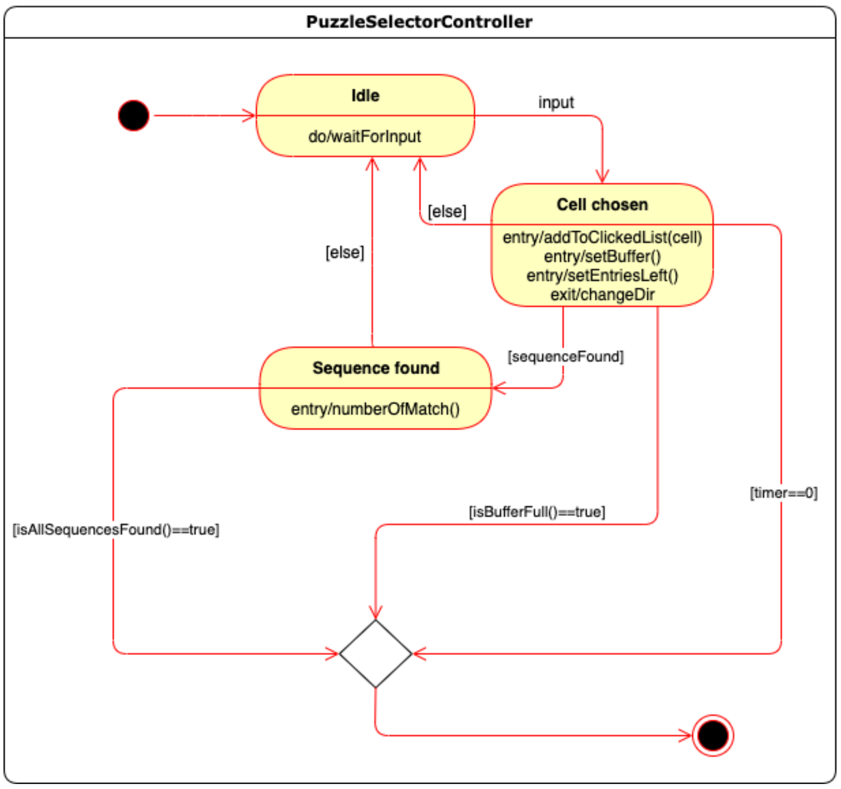
:sequences : Sequences - *sequenceList* field stores all the correct sequences. At this snapshot it holds two items inside of *sequenceList*.

State machine diagrams

Author(s): Ying Ying Ma

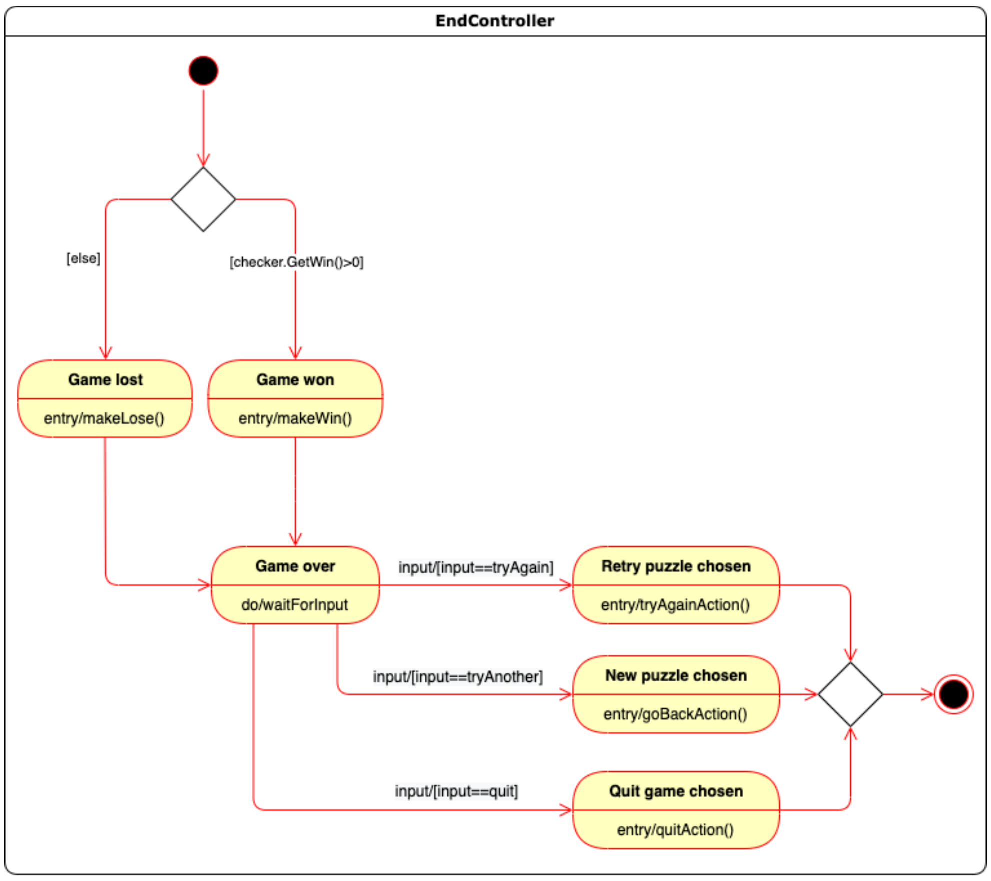
In this section, two state machine diagrams will be presented to describe the states of the system.

State machine diagram of PuzzleSelectorController class:



This state machine diagram is a description of the PuzzleSelectorController class. Initially, the state machine is in the ‘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 [timer==0] is true, a transition from ‘Cell chosen’ to a diamond symbol takes place. Secondly, if the guard condition [isBufferFull()==true] is true, there is another transition to the diamond symbol. Lastly, if the guard condition [sequenceFound] is true, a transition from ‘Cell chosen’ to ‘Sequence found’ takes place. Here, the activity ‘numberOfMatch()’ is executed. From ‘Sequence found’, a transition to the diamond symbol occurs if the guard condition [isAllSequencesFound()==true] is true. If this condition is false, a transition from ‘Sequence found’ back to ‘Idle’ occurs. If none of the three 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 the diamond symbol, the system transitions to the end of the state machine.

State machine diagram of EndController class:

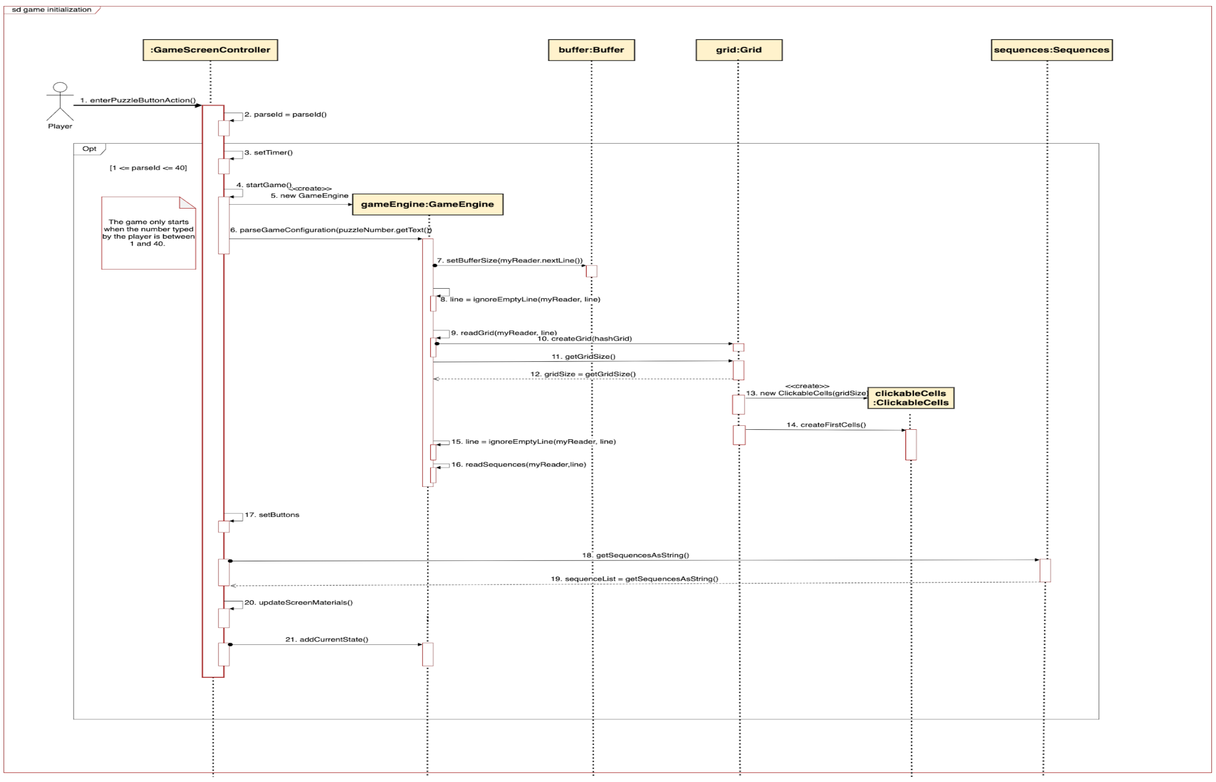


This state machine diagram is a description of the EndController class. Initially, the state machine transitions to a diamond symbol, where two transitions are possible. A transition to 'Game won' occurs if the guard condition [checker.GetWin()>0] is true. In 'Game won', the entry activity 'makeWin()' 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 the 'Game over' state, 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 present two sequence diagrams. The links of the two sequence diagrams are added to the pictures below. 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.


sd game initialization
:GameScreenController
buffer:Buffer
grid:Grid
sequences:Sequences
Player
1. enterPuzzleButtonAction()
2. parseId = parseId()
Opt
[1 <= parseId <= 40]
3. setTimer()
4. startGame()
<<create>>
5. new GameEngine
gameEngine:GameEngine
The game only starts when the number typed by the player is between 1 and 40.
6. parseGameConfiguration(puzzleNumber.getText())
7. setBufferSize(myReader.nextLine())
8. line = ignoreEmptyLine(myReader, line)
9. readGrid(myReader, line)
10. createGrid(hashGrid)
11. getGridSize()
12. gridSize = getGridSize()
<<create>>
13. new ClickableCells(gridSize)
clickableCells :ClickableCells
14. createFirstCells()
15. line = ignoreEmptyLine(myReader, line)
16. readSequences(myReader,line)
17. setButtons
18. getSequencesAsString()
19. sequenceList = getSequencesAsString()
20. updateScreenMaterials()
21. addCurrentState()

The game initialization process starts when the player clicks the enter button in the GameScreen interface. The method *enterPuzzleButtonAction()* of the **GameScreenController** is called at the same time.

Then *parseId()* method converts the input string *puzzleNumber* to int *parseId*. The following sequences only occur when the *parseId* is not empty and the *parseId* is between 1 and 40(1<=*parseId*<=40). This is because we only have 40 puzzle files named as 1 to 40. Thus, the following sequences are in an optional interaction.

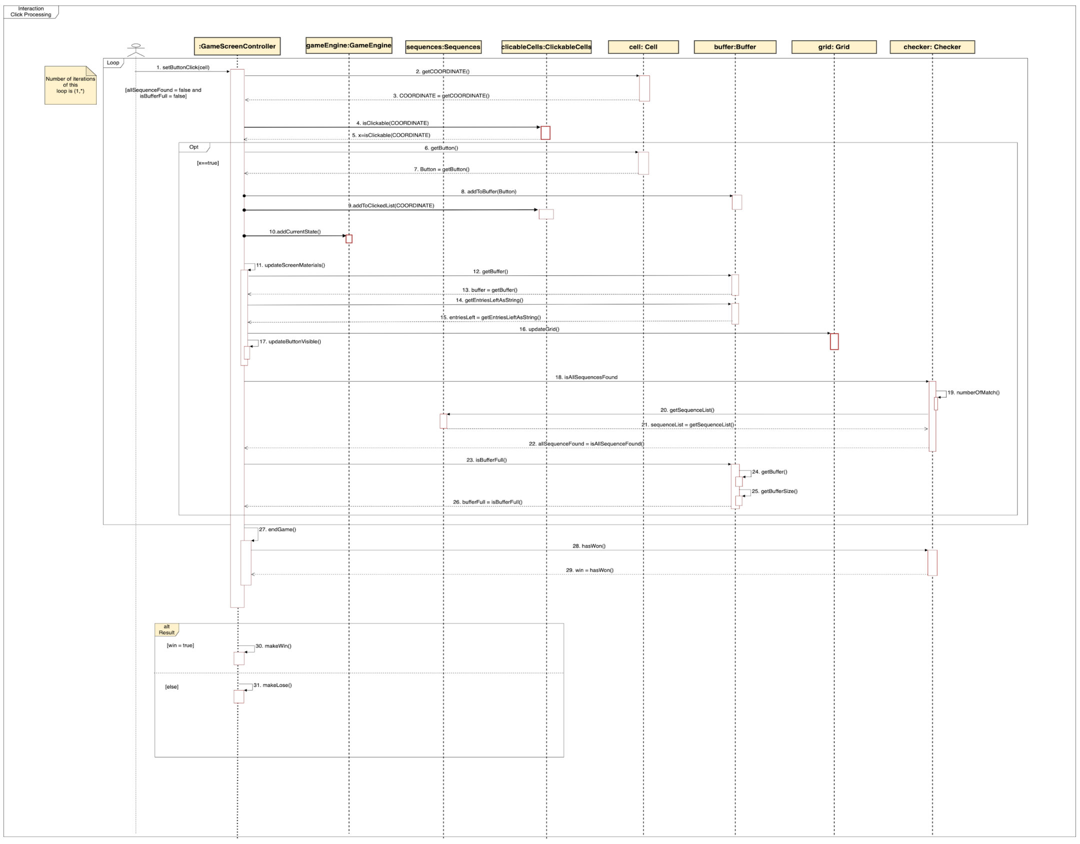
When the *parseId* meets the Guard, the **GameScreenController** will call self message - setTimer(). The *seTimer()* operation sets a timer on the screen, and the timer is a countdown from “seconds”. After that, the **GameScreenController** will call self message again - *startGame()* operation.

In *startGame()* method, **GameScreenController** first invokes a new **GameEngine** object. Then it utilizes the *parseGameConfiguration()* operation in the **GameEngine** to reads the file and fills the containers: Buffer, ClickableCells, Sequences, Grid. The *parseGameConfiguration()* method further utilizes *setBufferSize*eperation() from **buffer** object to read the buffer size. Then the method calls self massage -*ignoreEmptyLine()* to ignore empty lines because there are empty lines that are useless in the puzzle txt file. Next, it calls *readGrid()* to read the grid from the puzzle file. The *readGrid()* invokes the *createGrid()* from the **grid** object. After reading the grid from the puzzle file, *parseGameConfiguration()* initializes clickable coordinates as the first row of the matrix by creating a new **ClickableCells** to utilize the *createFirstCells()* method. The method ignores empty lines again by calling self message -*ignoreEmptyLine().* Finally, the *parseGameConfiguration()* calls self message - *readSequences()* to read sequences from the txt file. The *parseGameConfiguration()* reads the buffer size, grid, and sequences from the puzzle file, and initiates the clickable coordinates for the player when the game begins.

When the *parseGameConfiguration()* operation finishes, the *startGame()* method calls self message - *setButtons()* to set buttons on the game screen. Then it utilizes the *getSequenceAsString()* method from the **sequences** object in the **gameEngine** object to show the sequence list on the screen. It calls self message - *updateScreenMaterials()* to update the material on the screen. The *updateScreenMaterials()* method is not completely shown in the sequence diagram, because it's not really important in the game initialization interaction and it will be shown in the clicking processing sequence diagram. Finally, the *startGame()* invokes the *addCurrentState()* from the **gameEngine** object to add the current to the state stack. The detail of the method is not shown either, because this is not the main part of the game initialization. The method is mainly used for redo and undo. 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 the game ends. 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 *setButtonClick(cell)* function of the **GameScreenController** object will be invoked. The *setButtonClick()* method further calls *getCOORDINATE()* operation from the **cell** object and returns the *COORDINATE* to the **GameScreenController**. Then the **GameScreenControlle**r determines if the *COORDINATE* is clickable by invoking the *isClickable()* operation from the **cell** object. *IsClickable()* method returns a boolean *x* to the **GameScreenController**. Only when the *x* is true, the following sequences will occur, which means that only when the button clicked by the player is clickable, the game will continue. Therefore, the following interactions in the loop are inside the opt interaction, and the guard is [x==true]. If the *x* equals true, the button clicked by the player is clickable, then the **GameScreenController** will call *getButton()* method from the **cell** object. The Button returned by the cell **object** will be put into the buffer by invoking the *addToBuffer(Button)* method from the **buffe**r object.

The **GameScreenController** object then add the *COORDINATE* into the *clickedList* by utilizing the *addClickedList(COORDINATE)* of the **clickableCells** object. Then the **GameScreenController** calls *addCurrentState()* from the **gameEngine** object to create the current state and adds it to the state stack.

For each valid clicking move from the player, the **GameScreenController** will update all information in the game by invoking *updateScreenMaterials()*, *updateGrid()*, and *updateButtonVisible()* operations. Once all information is updated, the **GameScreenController** object will check if all sequences are found or the buffer is full by calling *isAllSequencesFound()* from the checker object and *isBufferFull()* from the **buffer** object. The **checker** object will return a boolean variable *allSequencesFound* and the **buffer** object will return a boolean variable *bufferFull* to the **GameScreenController**. If both of them are false, the player will continue clicking in the grid.

However, if one of them is true, the loop interaction ends. The **GameScreenController** will call self message - *endGame()* to check the result of the game. The *endGame()* method calls *hasWon()* method from the **checker** object, and **checker** returns a boolean variable *win* to the **GameScreenController** object. If the *win* is true, the player wins the game, the **GameScreenController** will call *makeWin()* method. Otherwise, the player loses the game, the **GameScreenController** will call *makeLose()* method. The final two interactions are in the alt interaction, because there are two situations in the result.

This is the whole process of clicking by the player. In summary, the player continues clicking the button on the grid until the game ends. Everytime, the player clicks on a valid button, the system will update all information in the game and determine if the game achieves the end. If not, the player continues clicking the button. If so, the system will check whether the player wins or not by seeing how many sequences found by the player.

Implementation

Author(s): Emre Furkan Guduk, Ahmet Yasir Uckun

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. The first version of our design had some issues and those issues have been pointed out by feedback. In this

version, both minor and major issues are fixed regarding the design techniques, functional adjustments, structural patterns and maintainability of the project.

In this version, instead of depending on static classes, we switched to a more object friendly design structure. Besides, some classes have been reduced by size and responsibility assigned to them, and some of them entirely removed. As a result, this change led to new classes. Namely, **Puzzle**, **PuzzleController**, **PuzzleSelectorController**, **Reset** and **EndController** were removed and it ended up with new classes named **Grid**, **Sequences**, **Cell** and **GameScreenController**.

Change of the design structure ended up with very intense debugging sessions. However, this version eventually led us to apply **Singleton** and **Command** design patterns.

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:
One of the main problems after the change in the implementation was the scene switching between different occurrences. For instance, the initial game screen should be shown for the very first time in order to get input from the user. After taking an input in the correct form, the actual game screen where the **Grid**, **Buffer**, **Timer**, **Sequences** are placed should be shown. Since the system is built on Scene Builder and JavaFX, there must be a button in order to switch between scenes and not every triggering event was caused by a button click action in our model(e.g. the game should be terminated if the *timer* hits 0). That's why we concatenated all of the fxml files and created a single file where everything could be accessed at any time. This solution also helped us to pass objects into the built in JavaFX functions rather than static classes and it also led us to control the number of the objects at a runtime thus, it allowed us to apply some design patterns.

In addition, in the first version of the system, coordinates in the **Grid** were being tackled by a primitive system which was assigning coordinates to each of the tiles on the **Grid**. In this version, a more generalized solution technique was applied. As a briefing, a class named **Cell** has been created and in the beginning of the game, the grid was filled with **Cell** objects which are compatible with the Scene Builder **Button** objects. This solution helped us to combine the SceneBuilder **Button** objects with our **Cell** classes and allowed us to avoid unnecessary calculations for coordinate values. As a result, a dynamic intractable table(**Grid**) made of **Cell** objects was obtained.

As the last solution technique, object-oriented programming design was applied to tackle the initialization and the end of the game. As a briefing, the game is initialized with a controlled number of objects, and when the termination or new game is necessary, those objects are either deleted or replaced with new objects. In the previous version of our system, we were using static classes and tackling this problem with a class **Reset** which was resetting all the classes with initial values. In this version, **Reset** class is removed and instead, dynamic object creation/ destruction is used.

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:

- [CyberPunk 2077 Hacking Minigame | Assignment 3 - YouTube]

Time logs

Member	Activity	Week number	Hours
X. Lu (Xiaoxuan)	discuss diagram	7	2
E.F.Guduk(Emre)	discuss diagram	7	2
Y.Y. Ma (Ying Ying)	discuss diagram	7	2
Ö.F. Çakici (Omer Faruk)	discuss diagram	7	2
A.Y.Uçkun(Yasir)	discuss diagram	7	2
X. Lu (Xiaoxuan)	implement diagram	8	4
E.F.Guduk(Emre)	implement code	8	30
Y.Y. Ma (Ying Ying)	implement diagram	8	4
Ö.F. Çakici (Omer Faruk)	implement diagram	8	4
A.Y.Uçkun(Yasir)	implement code	8	30
X. Lu (Xiaoxuan)	documentation	9	4
E.F.Guduk(Emre)	documentation	9	4
Y.Y. Ma (Ying Ying)	documentation	9	4
Ö.F. Çakici (Omer Faruk)	documentation	9	4
A.Y.Uçkun(Yasir)	documentation	9	4
		TOTAL	102