Assignment 2

Team number: 3
Team members:

Name	Student Nr.	Email
Nikita Sazhinov	2631755	n.sazhinov@student@vu.nl
Quinn Rutherford	2670152	q.d.rutherford@student.vu.nl
J. Antonio Fortanet C.	2671921	j.a.fortanetcapetillo@student.vu.nl
Niatna Tesfaldet	2707692	n.h.tesfaldet@student.vu.nl

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

Implemented feature

ID	Short name	Description
F1	Read puzzle	The system shall **read puzzle information** from .txt files and **store the information** in the matrix and buffer.
F3	Countdown Timer	The system shall have a **countdown timer** which stops the minigame when it runs out and **display this value** at all times.
F5	Code Check	The system shall **check the buffer** to see if one of the sequences is completed. If the sequence is completed, then the game ends and the user wins.

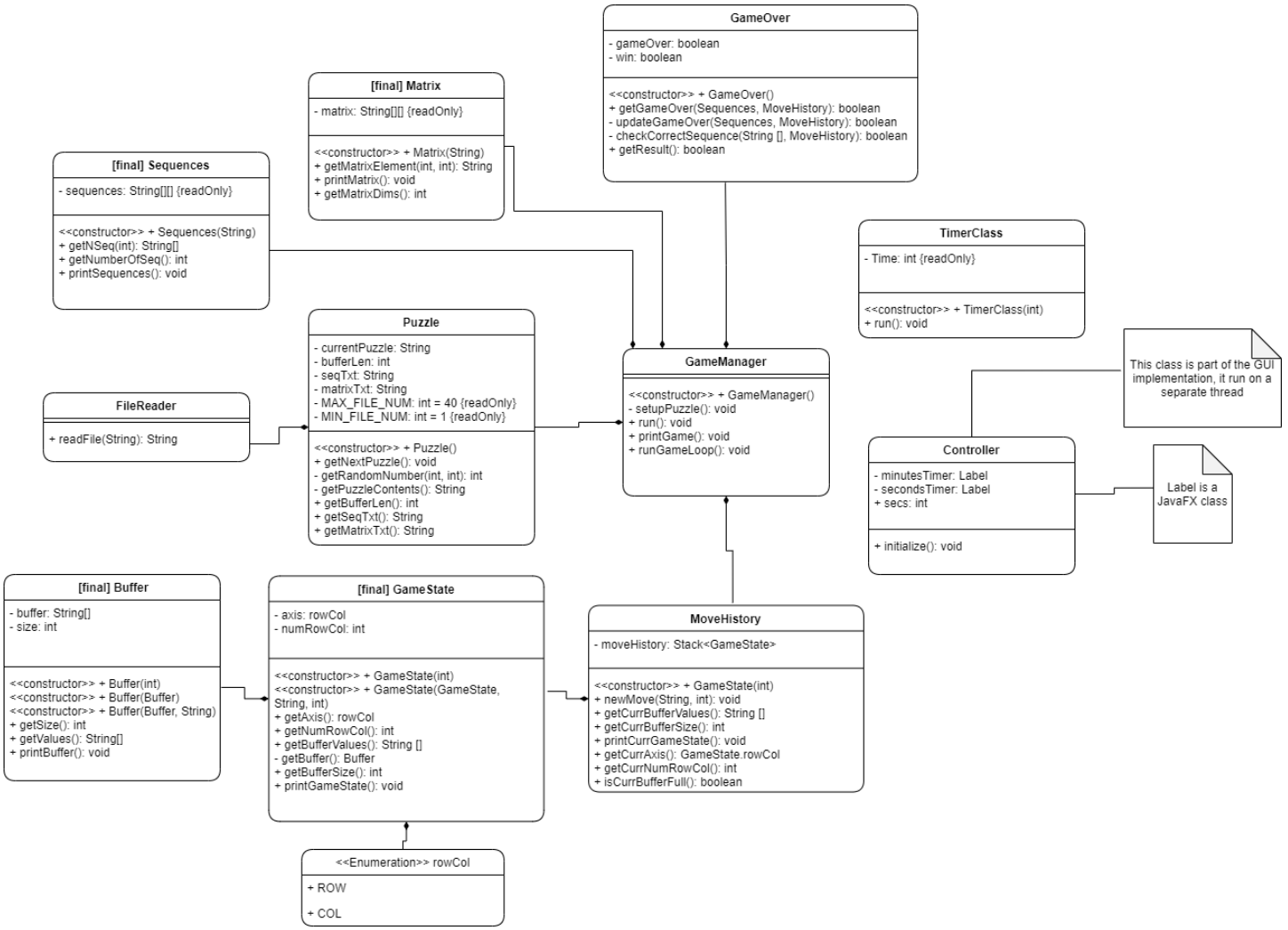
Used modeling tool: diagrams.net

Link to game demo.

** Note: We were unable to get the puzzle files inside the fat jar file, therefore the game must be run inside of Intellij.*

Class diagram

Author(s): Quinn Rutherford, J. Antonio Fortanet



Link to higher quality image

GameManager: The GameManager class is responsible for running the game loop which contains the logic of the game. This is done by managing three core parts of the functionality of the game: reading and parsing information from the text files which contain the puzzles information, saving the game information including user input, and the checking for the win and loss conditions.

- Reading and parsing: This part is managed mainly by the Puzzle class. In this class, a file is picked at random from a folder and then is read using the FileReader class. The information is then separated into its different components and saved to be used to initialize the buffer, matrix, and sequences.
- Game information: This contains three parts, the matrix class, the sequences class and the moves class. The matrix and sequences are only used to save information which is displayed to the user and compared with user input. These two classes are

final and not changed during runtime (after initialization). The MoveHistory class is used to save the moves made by the user during the run of the game. The purpose of this is so there can be a function to undo a move during the game in case the user gets stuck in a puzzle.

- Conditions for ending game: This functionality is managed by the GameOver class. The GameOver class compares the buffer and the sequences' values and determines if a game over state has been reached and is used to determine when the game should end.

The main operations present in this class are setupPuzzle(), gameLoop() and run(). The first is used to call the Puzzle attribute to pick a puzzle and then ask for the puzzle information so the other attributes can be initialized. The gameLoop() method is used as the core game loop which keeps the game running until it ends. It gets the moves made by the user and updates the moveHistory. Lastly, the run() is used to be executed in a thread so GameManager can run simultaneously with the TimerClass class.

MoveHistory: The MoveHistory class is incharge of storing the current state of the game and a history of game states. This functionality is implemented by having a moveHistory attribute which is a Stack of the GameState class. The main purpose of this is to store the previous values to allow the user to undo previous moves.
Its operations are used mainly so the GameManager class can retrieve information about the current state of the game and can communicate it to the other parts of the game. This class fully encapsulates the gameState class so that all interaction related to the game states are done through the methods in the MoveHistory class to allow for information hiding. The moveHistory class allows other classes to get information about the current game state and buffer without having to access them directly. This is done through methods such as getCurrBufferValues() or getCurrAxis. The most notable operation that this class performs is the newMove method (which is called by GameManager) which creates a new GameState, depending on the move made by the user, and pushes it to the Stack. The only place this class is present is as an attribute to GameManager.

GameState: The GameState class is used to save previous and current game states. This state changes every time the user selects a new value. Instead of modifying a single GameState instance, these are made immutable and new ones are made to be added to the stack in the MoveHistory class.
The GameState class is only contained in the MoveHisotry class in the moveHistory (Stack) attribute. The GameManger interacts with GameState through the MoveHistory class.
The attributes contained in this class are a Buffer, an int and a rowCol. The Buffer attribute contains the state of the buffer in the given state of the game. The rowCol contains row or column which dictates if the user must select a row or column from the matrix in the next move. The int is used to store which row or column specifically is to be used by the user for the next move.

GameState.rowCol: This is an enumeration which is used by the attribute rowCol and is either ROW to represent row and COL to represent column.

GameState's operations are mainly used to pass its attributes to the MoveHistory class so it can be shared with the rest of the components of the game. GameStates most important operations are getBufferValues(), which returns a String array with the buffer values of said

GameState, and getBufferSize(), which returns an integer with the number of values which have been input into the buffer.

Buffer: The Buffer class is only contained in the GameState class and is used to store the state of the buffer in a specific game state. Each GameState class has exactly one Buffer. The Buffer class has attributes buffer (String[]) and size (int). The buffer (attribute, not class) is used to store the values in the buffer. The buffer is initially empty, but each new Buffer which is created every time a new GameState is created has one additional value. The size attribute of the buffer represents the amount of spaces in the buffer that are filled.
The operations contained in this class are used to parse information, initialize attributes, and information to the GameState class.
The Buffer(Buffer, String) constructor, takes the old buffer (attribute) and adds the string value to it, creating a new Buffer for the next GameState.
The getValues() operation, returns a String array with all of the values in the buffer.

Matrix: The Matrix class contains the values that can be chosen by the user during the game. Depending on the location of the element this is placed accordingly in the matrix: String[][] attribute. An element's value can be obtained by knowing the x and y coordinates in which they are saved in the matrix.
The Matrix class is only present as an attribute in the GameManager class.
The functions of the Matrix class are:

- getMatrixElement() -> returns the value of a matrix element given its x and y coordinates.
- getMatrixDims() -> returns the amount of rows/columns that a matrix has (num of columns = num of row because all matrices are square).
- printMatrix() -> prints the values of the matrix in a user friendly fashion.

Sequences: The Sequences class is used to store the sequences that the user should complete in the buffer. The sequences can vary in number and size depending on the puzzle.
This class is only present as an attribute in the GameManager class.
The functions of the Sequences class are:

- getNSeq() -> returns a specific sequence as a String[] given its index.
- getNumberOfSeq() -> returns the number (integer) of sequences that the user is able to complete in the current puzzle.
- printSequences() -> (void) prints the sequences in a user friendly fashion.

Puzzle: The Puzzle class is incharge of randomly choosing the puzzle and parsing the content of the chosen text file. Also the Puzzle class saves the content of the text file separated into its parts to return to the matrix sequence to initialize them. It also saves the amount of values allowed in the buffer. As attributes, there is a String array with three values. This array saves the content of the text file, which has sequence text and the matrix text, and an int to save the size. There are two constant int that represent the lowest and highest number of files to be chosen from the puzzle folder.
The getMatrixTxt, getSeqTxt and getBufferLen methods contained in this class are used to pass the values of the attributes and initiate the Matrix, Sequence and MoveHistory classes respectively. The getRandomNumber chooses a random number in the span of MAX_FILE_NUM and MIN_FILE_NUM (inclusive). The method getNextPuzzle reads the

content of the chosen text file(with the FileReader). The getNextPuzzle operation gets the file, reads it, separates it into the 3 parts, and saves them in the attributes to be returned by the GameManager.
This class is only contained in GameManager as an attribute.

FileReader: This class is responsible for reading the text files containing the puzzle information. The fileReader class has a readFile method which returns a string containing the components needed to create a fill the matrix, fill the sequences, and get the length of the buffer.

GameOver: The GameOver class is responsible for evaluating if the game is over and if the game was won or lost. An instance of this class is contained within the gameManager class. This class has two boolean attributes, whose values are evaluated within the class methods:

- gameOver is initialized as false is evaluated as true if the game is over.
- Win initialized as false is evaluated as true if the user solves the puzzle correctly.

The class updates the attribute values using its operations.

- CheckForCorrectSequence() -> This method takes as arguments a string containing the sequence and a moveHistory object (which is used to get the buffer values) and returns a boolean on if the buffer contains the sequence.
- updateGameOver() -> This method checks for all sequences that could solve the puzzle using the checkForCorrectSequence() method. It returns true if any are found and evaluates the win attribute to true.
- getGameOver() -> evaluates the gameOver attribute using the methods above.

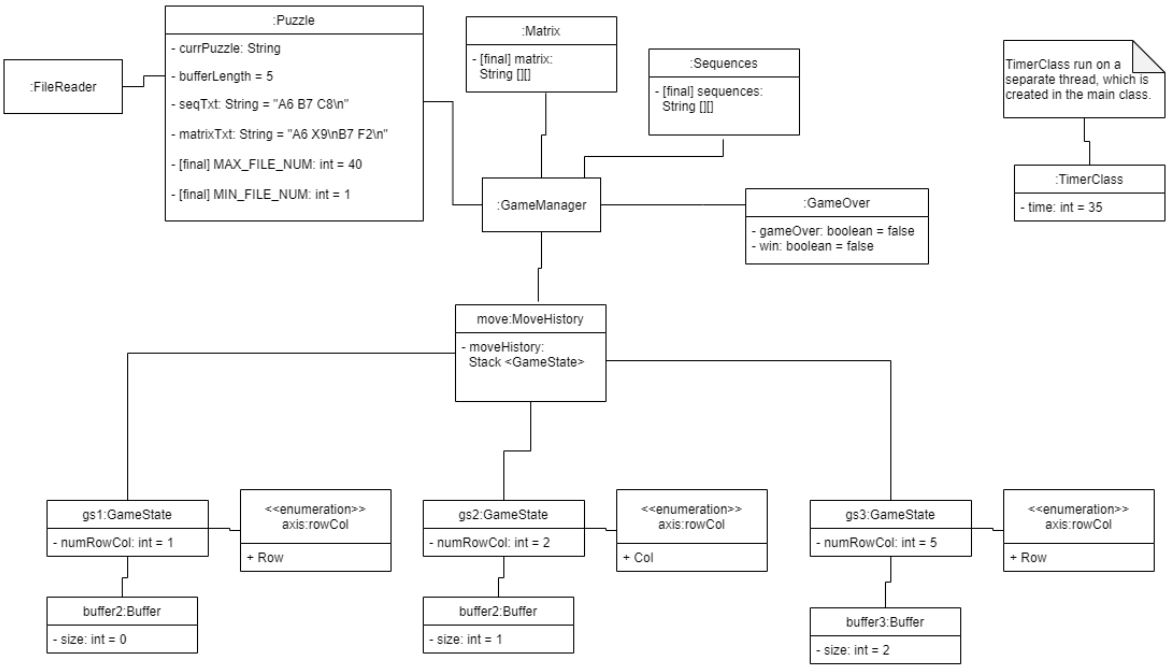
TimerClass: The TimerClass class is responsible for creation and running of the in-game countdown timer. The class is a runnable class which runs in a separate thread simultaneously with the GameManager class. The class has an integer attribute “time” which determines for how long the countdown timer runs for. The timer class behaviour is implemented inside the overridden run() method, which is executed upon starting the TimerClass thread. Inside the run function, an instance of the anonymous class TimerTask is created, which contains within itself the logic of the task to be repeatedly executed (decrement the time remaining).

Controller: The Controller class is responsible solely for the logic of the GUI. It is linked to the fxml file containing the structure of the GUI. It has attributes that each correspond with GUI components from the JavaFX library which have programmable logic in our GUI. At the moment the Controller class only contains logic for the timer GUI, which is implemented in the “initialize()” method which is called on startup.

Object diagram

Author(s): Quinn Rutheford

This chapter contains the description of a "snapshot" of the status of your system during its execution. This chapter is composed of a UML object diagram of your system, together with a textual description of its key elements.

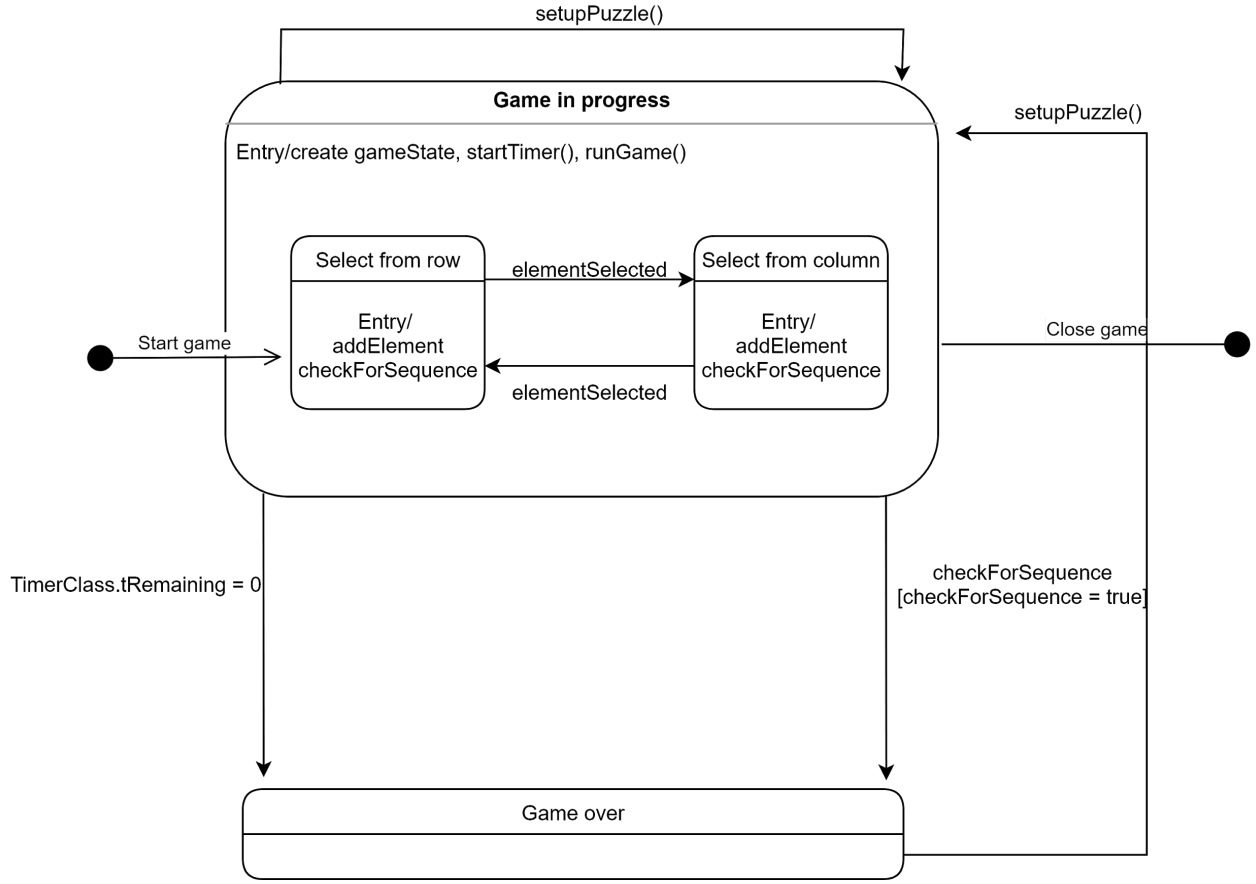


[Link to higher quality image]

At this point in the execution the GameManager class has created a Puzzle, which created a FileReader which reads the contents from a file. The puzzle class has saved these values and they have been passed to create a new Sequences and Matrix class which have been initialized (their values are not shown to save space). The GameManager has also created a GameOver class where gameOver = false, as the game is still running. The TimerClass has been created in Main and is running on a separate thread. MoveHistory has a Stack of GameState gs1, gs2, and gs3 (gs3 the top, and gs1 the bottom), which represent the previous and current states of the game. The GameStates enumeration rowCol alternates between ROW and COL for each GameState in the MoveHistory Stack.

State machine diagrams

Game Manager and Matrix state machine



This state machine diagram models the internal behaviour of the GameManager class showing the abstract states it may be in throughout runtime.

The game starts when the user selects a puzzle element, and enters into the “Game in Progress” state.

Game in Progress:
This state models the behaviour of a game in progress.
Upon entering the state, the following events are executed:

- startTimer() starts the internal game timer that countdowns the time to solve the puzzle
- runGame() commences all the internal logic of the puzzle

Within the Game in Progress state, two substates are defined to model the behaviour of the game related to the matrix. Each time a user selects a matrix element, the game transitions between the two inner states: “Select from row” and “Select from column”. These states transition between one and the other when the “ElementSelected" event occurs. This transition represents the user clicking on a puzzle element. Each state has two internal events:

- addElement: Represents the selection of an element by the user to be added to the buffer.

- checkForSequence: Checks if the current buffer contains any of the puzzle sequences. If the checkForSequence event returns true, meaning that a correct sequence is in the buffer, the game transitions into the “Game Over” state.

Additionally, the Game In Progress state can enter again, restarting the game with thesetUpPuzzle() transition, which gets a new puzzle reinitiate.

Game Over:

This state models the behaviour of the game being over, regardless of if the user wins or loses. Upon entering this state, the “GameOver()” functionality is executed, notifying the user that the game is over. This state can be reached by two transitions from the Game in Progress state:

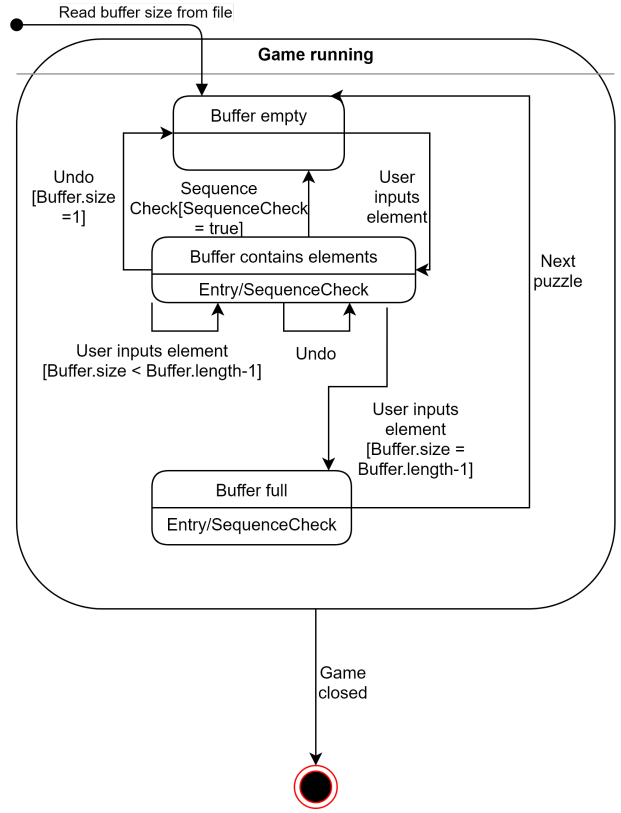
- checkForSequence [true]: As described above, when the checkForSequence event is fired, if evaluated to true, the game transitions to game over state, as the user has solved the puzzle.
- TimerClass.tRemaining = 0: This event models the countdown timer runs out, meaning that the user failed to solve the puzzle.

The game over state is reachable from the Game In Progress state through a single transition:

- setupPuzzle(): This transition happens when the user chooses to proceed to the next puzzle.

When the game is closed it ends.

Buffer states diagram



This state machine diagram models behaviour of the buffer class and its states during runtime.
The initial state where the game is not running transitions into the “Game Running” state.

Game Running
The state models a running game, it is reached from the initial state with the “Read buffer size from file” event which transitions directly into one of the substates of “Game Running '' representing the buffer class states.
Buffer Empty
This state represents the initial state of the buffer class during runtime after its creation. It contains no elements at this point. This state can only transition to the “Buffer contains elements state” when the “User inputs element” event occurs, which models the user selecting a puzzle element.
Buffer contains elements
This state represents the state of the buffer when it contains elements but is not full. This state has the following outgoing transitions:

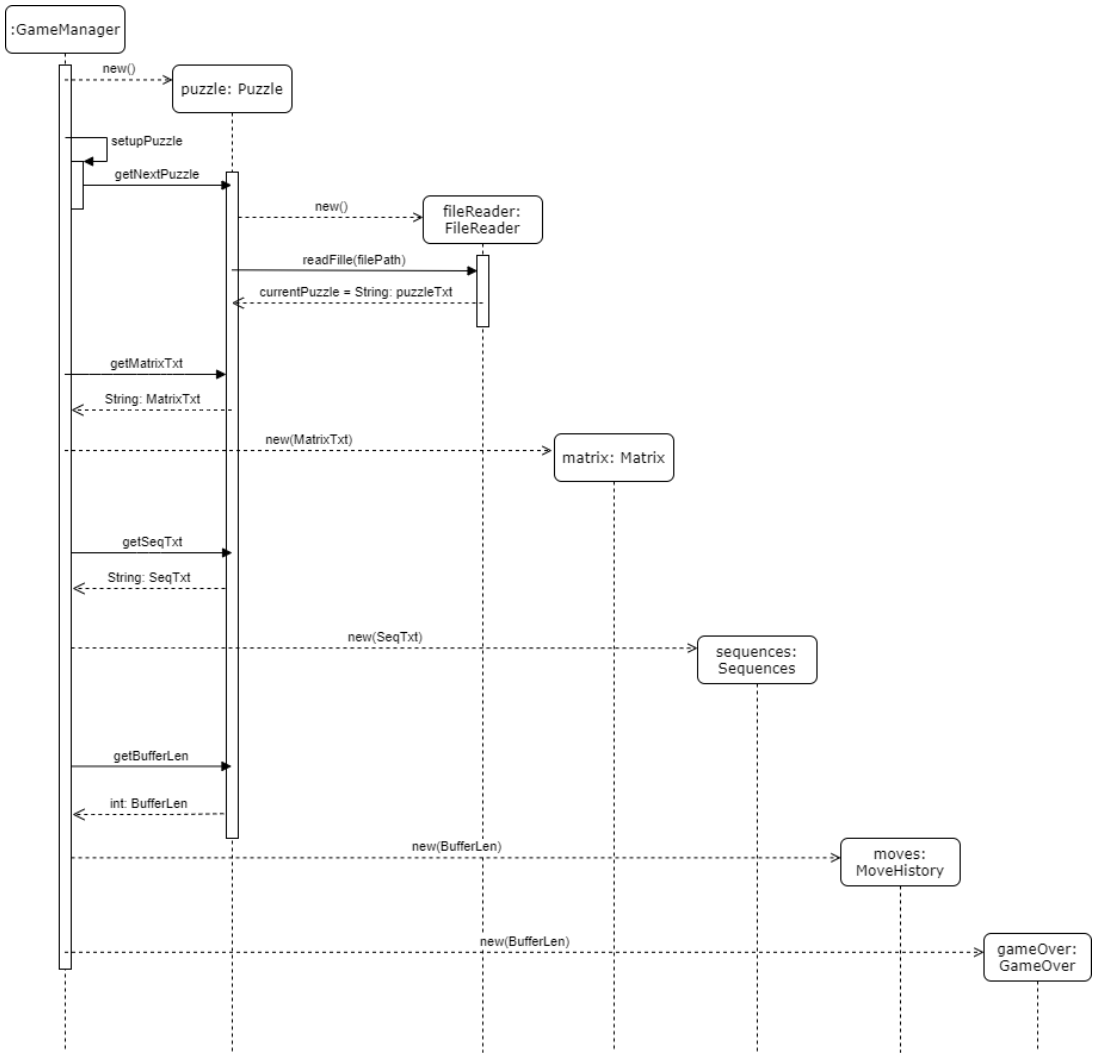
- Undo [Buffer.size = 1]: this transition is the event of the user “undoing” his last move, causing the buffer to go back to the “Buffer Empty” state.
- Undo: This transition models removing one element from the buffer.
- Sequence Check[SequenceCheck = true]: This transition executes when the buffer is checked for containing the sequence and it returns true, meaning that the user has solved the puzzle, creating a new empty buffer.
- User inputs element [Buffer.size < Buffer.length-1]: This transition models the user selecting a puzzle element but it does not make the buffer full, in that case the transition goes back to the “Buffer contains elements”.
- User inputs element [Buffer.size = Buffer.length-1]: this transition models when the user inputs the “last” element into the buffer, causing it to go into the “Buffer Full” state.

Buffer Full
This state models the behaviour of the buffer when it is full. Upon entry to the state, the SequenceCheck event occurs, checking if the buffer contains any of the winning sequences. This state only has one outgoing transition, caused by the Next puzzle event occurring, which creates a new buffer, returning it to the “Empty Buffer” state.

Sequence diagrams

Author(s):J. Antonio Fortanet C., Nikita Sazhinov and Niatna Tesfaldet

Sequence Diagram 1: Game setup (GameManager constructor)

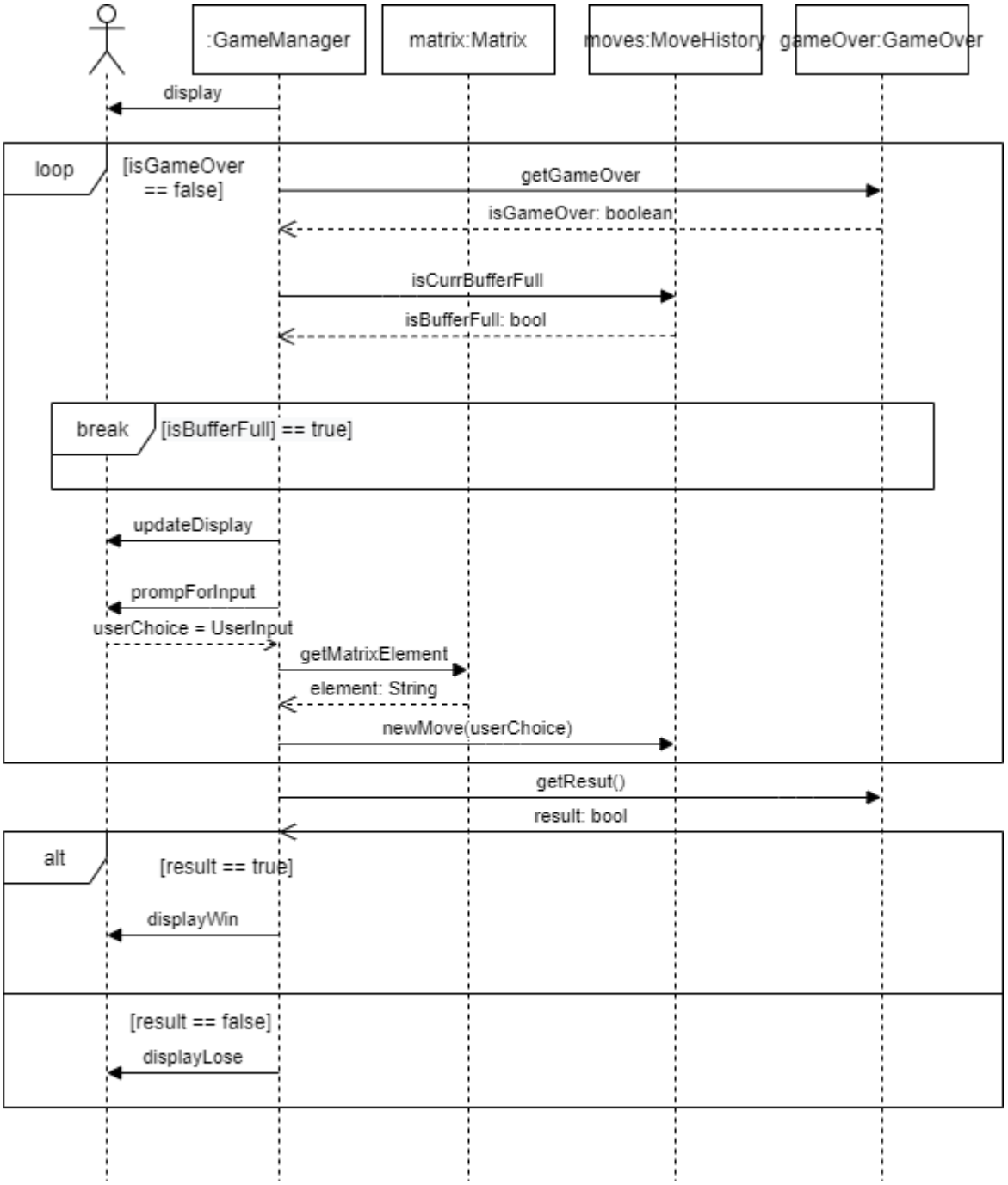


This sequence diagram represents the process of setting up the game, initiating the different classes in the project.

First, GameManager initializes the puzzle attribute with a new Puzzle class and GameManager calls setupPuzzle which handles the process of selecting, reading and parsing a text file from the ‘puzzles’ folder.

Then, GameManager calls Puzzle.getNextPuzzle. This method randomly chooses a puzzle from the library and creates a FileReader to get the content of said file. During this process the FileReader creates a Scanner and reads the content of a file in the ‘puzzles’ folder. This specific part was not represented in the sequence diagram because we considered that it was out of scope for the process we want to represent, since it represents an external class. The absence of this part gives more abstraction to the diagram and skips over unnecessary technical details.

After getting the content of the text files, the information is separated into the 3 parts of the puzzle text and saved in the matrixTxt, seqTxt and bufferLen attributes in Puzzle.
Then, the Puzzle.getMatrixTxt is used to create a new Matrix, Puzzle.getSeqTxt() is called and a new Sequence is created with it. Then getBufferLen() returns an int which is passed to move history, which internally creates a gameState, which creates a new Buffer. The creation of the gameState and Buffer is not shown as we show this process in sequence diagram 2. Finally, an instance of GameOver is created.



Sequence Diagram 2: User interaction with the game

This diagram has abstracted most of the internal workings of the classes so that it is more understandable to the reader.

This sequence diagram shows the interaction that the user will have with the game. At the beginning the display is shown to the user according to the starting conditions of the game.

At the start of the gameLoop GameManager checks with gameOver if the game is over, if this is true then the game ends, else the game continues inside the loop.

Then the GameManager checks with moves: MoveHistory if the buffer in use is full. If this is the case, then the game ends, if the current buffer is not full then the game continues.

The display is updated according to the current state of the game and the user is prompted to make a move (choose an element from the matrix). When the user makes a move this is saved in GameManger and passed to Matrix with getMatrixElement to get the value of the chosen element. Then, the value of the element returns to GameManager and is used to call newMove to MoveHistory so the move made is saved in the move history of the game.

After this the loop checks the condition again. If the loop breaks then the GameManager to get the result of the game with getResult. If the function returns true then the WIN display is shown, else the LOSE display is shown.

Implementation

Author(s): J. Antonio Fornanet, Nikita Sazhinov, Quinn Rutherford

Strategy used to Implement UML models into code

The first thing we did was create a very abstract class diagram, without many attributes or operations, and planned out the role each class would have. We then created these initial classes in IntelliJ to create a skeleton for our project. Then in smaller groups we planned out the implementation of a few classes which were related to each other keeping in mind the general structure of the entire project. For example we planned out initial attributes and operations of the Buffer class and the GameState class as they work closely together. Once we had a general plan for each part of the system we began coding with the Buffer and GameState classes. We often returned to the class diagram to add new operations for sections we had overlooked. We then continued the implementation of the Puzzle, Matrix, and Sequence classes which are all closely linked. We then continued with implementing the timer class. Once we had the majority of the individual components working on their own, we tested their functionalities individually in the Test class. Initially we were not going to have a GameOver or a MoveHistory class and just store this information in the main class, but we realized that this class was quickly getting too chaotic and we lacked information hiding. We returned to the class diagram to add the classes. Once all of the components were ready we implemented the core game loop in the Main class, and debugged.

Key solutions

<u>Storing Game-State and Buffer Values</u>

The main factor we considered when implementing the gameState, is that we wanted to be able to add an undo feature. Given this we quickly realized that we should save previous gameStates and should be able to retrieve them in reverse sequential order. We decided that the best way to implement this would be to have a stack of GameState, each one would contain the axis from which a value needs to be selected (row/column), the current row or column, and a Buffer. The number could be simply stored in an int and the row or column in an enumeration. We created a separate class so that we could perform some operations on the buffer. Since the Buffer was inside the gameState class we wanted to hide its implementation to allow for more information hiding. We then realized that we could allow for even more information hiding if we encapsulated the stack of GameState objects inside a moveHistory class which made internal calls to the GameState, which made internal call to the Buffer, never having to reveal the internal workings of the buffer, but allowing for all necessary operations and information retrieval.

<u>Checking of the Buffer/Sequences</u>

We initially contemplated placing the checks for the correct buffer values inside of the Buffer class or the Sequence class, but decided against that as they were simply used to store information about the game and adding a check values operation would be out of the scope we had defined for them. We determined that we should create an additional class GameOver which would contain the information about if the game was over and to check for it too. We send the GameOver class the Sequences object and the moveHistory object (which contains operations to retrieve Buffer values). The GameOver class has a function which loops over each of the sequences in Sequences and test is any of the is contained

within the Buffer, we did this by combining the buffer (String []) into a single String and the sequence (String []) into another String and used the .contains() method to check if the buffer had the values of sequence. If the buffer contained the values of one of the sequences then gameOver (boolean) was set to true and returned in the getGameOver() method.

Timer

The timer class implementation required some planning as the class has to run as an independent background task while also interacting with the rest of the system, notifying it when the time is up. As a result an instance of the timer class runs in a separate thread. The logic of the timer itself is implemented as an anonymous class contained in the timerClass. The timerClass is a runnable object that overrides the run() method. The GUI displays a timer independent from the timer class, which is implemented instead using the Javafx animation functionality. This timer serves solely as a visual representation of the system timer, as having the GUI timer interacting with the game logic could be unsafe.

Location of the main Java class

The main Java class needed for launching the execution of our systems source code is '/src/java/launcher.java'. The launcher class only calls the functions Main.main(args). This launcher class is necessary as our main class extends Application to be able to run a JavaFX GUI.

Location of the Jar file

The Jar file needed to directly execute the program is located in '/out/assignment2.jar'.

Video Showcase

https://www.youtube.com/watch?v=6wOmHTXR-Z8

This video shows the current implementation of the minigame, with the core gameplay running in the terminal and a GUI to display the timer. It shows the user selecting two correct values and completing a sequence and the printing of the "YOU WIN!!!" message.

Time logs

	Member	Activity	Week number	Hours
4	Quinn Rutherford	UML Diagram Planning	3	1.5
5	Quinn Rutherford	Meeting with team members	3	1
6	Antonio Fortanet	Meeting with team members	3	1
7	Nikita Sazhinov	Meeting with team members	3	1
8	Niatna Tesfaldet	Meeting with team members	3	1
9	Nikita Sazhinov	State diagram	3	1
10	Quinn Rutherford	Start Class Diagram	3	1
11	Antonio Fortanet	Updating class diagram	4	1
12	Quinn Rutherford	Creating Buffer and GameState classes	4	1
13	Antonio Fortanet	Creating Buffer and GameState classes	4	1
14	Quinn Rutherford	Creating Matrix, Sequences and GameMa	4	1.5
15	Antonio Fortanet	Creating Matrix, Sequences and GameMa	4	1.5
16	Nikita Sazhinov	Create timer class	4	2
17	Quinn Rutherford	Editing Class Diagram	4	1.5
18	Niatna Tesfaldet	State machine diagram	4	2
19	Antonio Fortanet	Updating classes to diagram and code	5	1
20	Nikita Sazhinov	State diagram	5	1
21	Nikita Sazhinov	Work on GUI and Timer	5	1
22	Quinn Rutherford	Group Meeting/Team coordination	5	1.5
23	Antonio Fortanet	Group Meeting/Team coordination	5	1.5
24	Nikita Sazhinov	Group Meeting/Team coordination	5	1.5
25	Antonio Fortanet	Work on sequence diagram	5	1
26	Antonio Fortanet	Implementation	5	3
27	Quinn Rutherford	Implementation	5	2.5
28	Nikita Sazhinov	Implementation	5	1
29	Niatna Tesfaldet	The display of the matrix	5	2.5
30	Quinn Rutherford	Object diagram	5	1.5
31	Quinn Rutherford	Implementation of game	5	3
32	Antonio Fortanet	Working on Sequence diagram	5	1
33	Antonio Fortanet	Game impremention	5	4
34	Niatna Tesfaldet	Sequence Diagram	5	3
35	Quinn Rutherford	Class Diagram	5	1
36	Antonio Fortanet	Class Diagram	5	1
37	Antonio Fortanet	Sequence Diagram	5	1
38	Nikita Sazhinov	Work on document	5	4
39	Antonio Fortanet	Work on document	5	4
40	Quinn Rutherford	Project Finalization	5	5
41			TOTAL	65