Assignment 3

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 20 pages (excluding the contents above).

Summary of changes of Assignment 2

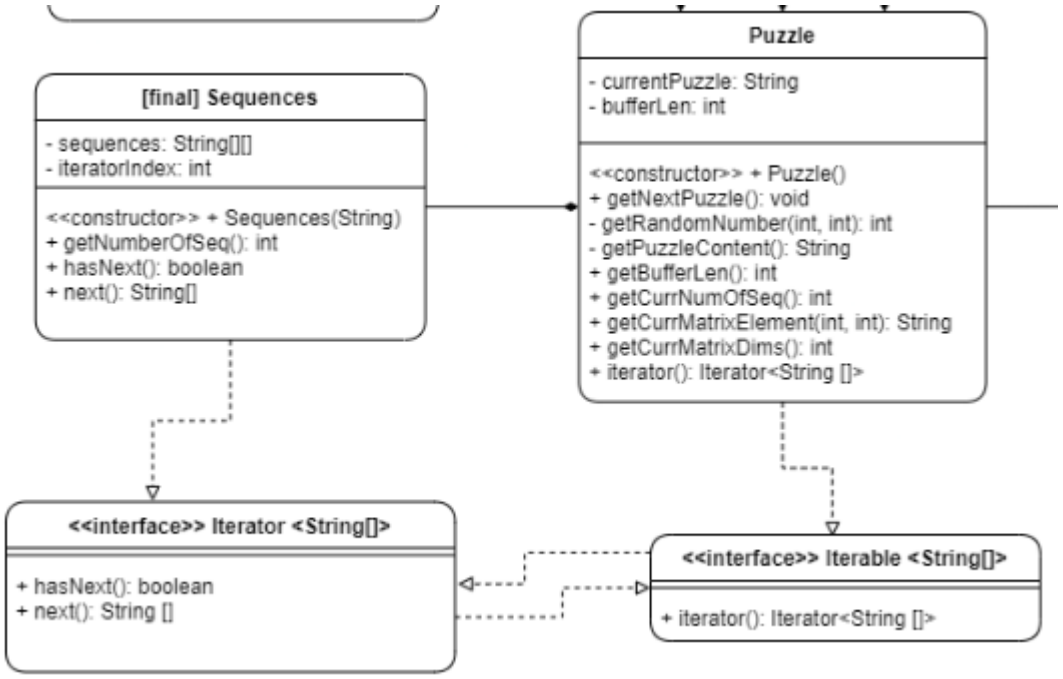
Author(s): Quinn Rutherford

Provide a bullet list summarizing all the changes you performed in Assignment 2 for addressing our feedback.

- Implemented GUI
- Implemented the refresh, undo and row highlight features
- Created ContainsQueue to ensure puzzles don't repeat for at least 5 games
- GUI created in GUIBuilder class
- Moved matrix and sequences class to be contained inside of puzzle
- Saved puzzle files inside of jar file (and retrieved them)
- Remove unused methods and cleaned up code
- Single timer class
- Merged timer display with game display
- Sequences class is now iterable.
- GameManager class now a singleton.
- Class diagram updated to contain all newly added classes and changes.
    - GUIBuilder class
    - ContainsQueue
    - TimerClass in GUIBuilder
    - Iterable and iterator interfaces
- Object diagram updated to contain all newly added classes and changes.
- State machine diagram of GameManager no longer contains matrix states and complies with the implementation of the refresh functionality.
- Updated Sequence diagrams and added additional details
- A more in depth discussion of design decisions.

Application of design patterns

Author(s): Quinn Rutherford, Nikita Sazhinov



	DP1
Design pattern	Iterator
Problem	The sequences had to be iterated over explicitly exposing their underlying representation. Previously every section which made use of the sequences had to implement how it would iterate over the sequences.
Solution	The Sequences class implements the Iterator interface which allows other classes to iterate over all of the sequences in the Sequences class without having to understand its internal methods. The client code can now use the next() function to retrieve the next sequence in the class, this also allows the foreach syntax to be used and abstract the inner workings of the class. The hasNext() checks if the iterator has gone through all of the values and returns true if there are move values to iterate through. It also restarts the iteratorIndex if hasNext() is false. The next() checks if there is a next value and if there is increments the iteratorIndex and returns the value, otherwise it returns false. To retrieve this iterator the Puzzle class implements Iterable<String []> so that other classes can call Puzzle.iterator() and the sequences iterator without worrying about its implementation or method name.
Intended use	The iterator is used in the gameOver class where each of the sequences is checked with the buffer. We were able to abstract the logic of iterating over

	the sequences from that class and place that responsibility in the Sequences class. Now the gameOver class has a for loop that has `'for (String[] seq : puzzle)'` to iterate over the sequences. The iterator is also used in the GUIBuilder class where the Labels to display the sequences are created. A similar for loop to the one displayed above is used to iterate through each of the sequences and display them.
Constraints	The client code would now have to implement an additional method if they wanted to retrieve only one sequence. We however no longer find this necessary as the client code is always used to loop over all of the sequences,

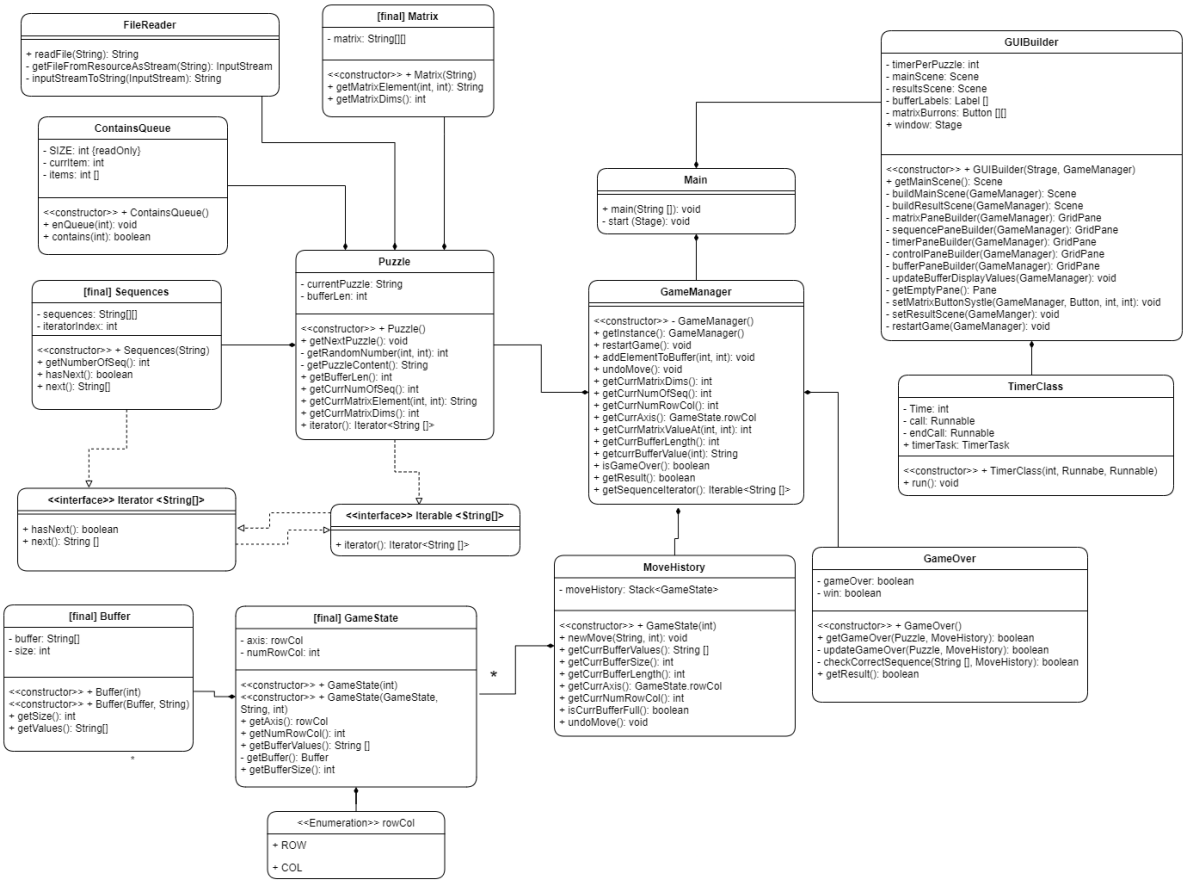
	DP2
Design pattern	Singleton
Problem	During runtime, it needs to be assured that only a single instance of the gameManager exists. Although previously only one instance of the gameManager existed, making it into a singleton will ensure that this property is retained during runtime.
Solution	The gameManager is made into a singleton class by making the constructor private and creating a getInstance method.
Intended use	The gameManager object will be a singleton object, an instance of which will be made in the main function and passed down to any other object requiring it.

Maximum number of pages for this section: 4

Used modeling tool: diagrams.net

Class diagram

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



[Link to higher quality image]

Overview: Our class structure is mainly focused around the GameManager class. This class has the responsibility of handling the interaction between 4 sections of our project. We decided to split the classes into this structure as we wanted to separate the responsibilities of the display, the puzzle, the users inputs, and the relationship between the puzzle and the users choice. The puzzle section has the responsibilities of retrieving the puzzle information, parsing, storing, and retrieving it. The fileReader class is incharge of retrieving puzzle information, the Matrix and Sequences of storing it, and the puzzle of retrieving it. The user input storage section which falls under the MoveHistory class is used to store the user interaction with the puzzle, it stores the current value of the buffer, number of row or column to be selected, and axis to be selected from, as well as the previous states to allow for our undo feature. The relationship between the puzzle and the user input storage sections is handled by the GameOver class, the main purpose of this is to be able to take input from the other two and determine the outcome of the game. We decided against managing these interactions in the GameManager class as it would have too many responsibilities and become too large. The final section is the display which has the purpose of displaying the puzzle information, taking in user input, and showing the user if they have won or lost.

GameManager: The GameManager class is responsible for running the logic of the game. This is done by managing the puzzle section, the user input storage, and checking the win conditions. As the GameManager is the glue which holds the whole project together, its main role is calling methods of the other classes to carry out the core functionality of the game. The methods addElementToBuffer() and undoMove(), interact with the user input storage section of the game to respond to user input from the display section. The gameManager mainly interacts with the puzzle section to retrieve information, as this information cannot be altered by the player unless setting up a new puzzle. The GameManger is used to check with the GameOver class to see the status of the game after every move.

MoveHistory: The MoveHistory class is responsible for storing the current state of the game as well as a history of game states. This is implemented by having a moveHistory attribute which is a Stack of the GameState class. We decided to use a stack to store the previous game states as it allows us to simply pop an element to return to the previous state. This class's operations are used mainly so the GameManager can create a new gamestate with new information or retrieve information about the current state. 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 creates a new GameState, saving the players move pushing it to the Stack.

GameState: This class's main purpose is to save the users previous and current information on the game. We needed to save each state to be able to return to the previous state to implement the undo feature. As the creation of a new one gameState is based on the values of the previous one, we decided to make a constructor which takes a previous gameState, a value to add to the new buffer, and the number of the next row or column, which then creates a new buffer and gets the correct values for the user input.

GameState.rowCol: We needed a way to determine if the user should pick the next value from a row or column. For this we decided it would be best to store this in an enumeration as it is simple to implement and work with. GameState.rowCol can be either ROW to represent row and COL to represent column.

The GameManger interacts with GameState through the MoveHistory class. GameState's operations are mainly used to update the next gameState or to pass its attributes to the MoveHistory class so it can be shared with the rest of the components of the game.

Buffer: The Buffer class is used to store the state of the buffer in a specific game state. When creating the buffer class the main considerations were, how should we store the buffer values and how can a value be added. For the first question we decided that when a new puzzle is set up the buffer should have an array of the length given by the puzzle as that is the maximum length it will ever be. For adding values to the buffer we only allow this through a constructor, which takes the previous buffer and a new value to add, this allows the class to be immutable so we can be sure that old gameStates which we save will not be altered if we return to them in the future. The Buffer(Buffer, String) constructor takes the old buffer values and adds the string value to it, creating a new Buffer for the next GameState. 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.

Puzzle: The Puzzle class retrieves and stores information on the current puzzle. This class selects a file number to send to FileReader, which returns the contents which are used to initialize the Matrix and Sequence classes. Puzzle implements QR4 by utilizing a containsQueue to make sure one of the last 5 puzzles is not repeated. Once the puzzle information is retrieved it is stored inside the immutable matrix and sequence classes, whose values will be changed if getNextPuzzle is called. The length of the buffer is retrievable for the creation of the initial buffer via a getter method as it needs to know the maximum buffer length.

ContainsQueue: To implement QR4 we needed a data structure which would remove the element which has been saved for the longest and also check if it contained a certain value. ContainsQueue, saves the current element queued in the spot of the last one entered, and provides a contains() method to check if a certain value has been saved.

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.

Matrix: The Matrix class contains the values of the matrix which 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. The matrix class allows for retrieval of a specific element in the matrix via the getMatrixElement(int x, int y), as well as retrieval of the dimensions of the matrix.

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 allows the user to retrieve the number of sequences and contains an iterator to allow the sequences to be iterated through. These functionalities are useful for the display and the comparison of each individual sequence to the buffer and for displaying the sequences.

GameOver: 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. This class is used so that the gameManager can simply ask it for the current state of the game at any point. If the game is over, getResult tells the GameManager if the game was won or lost.

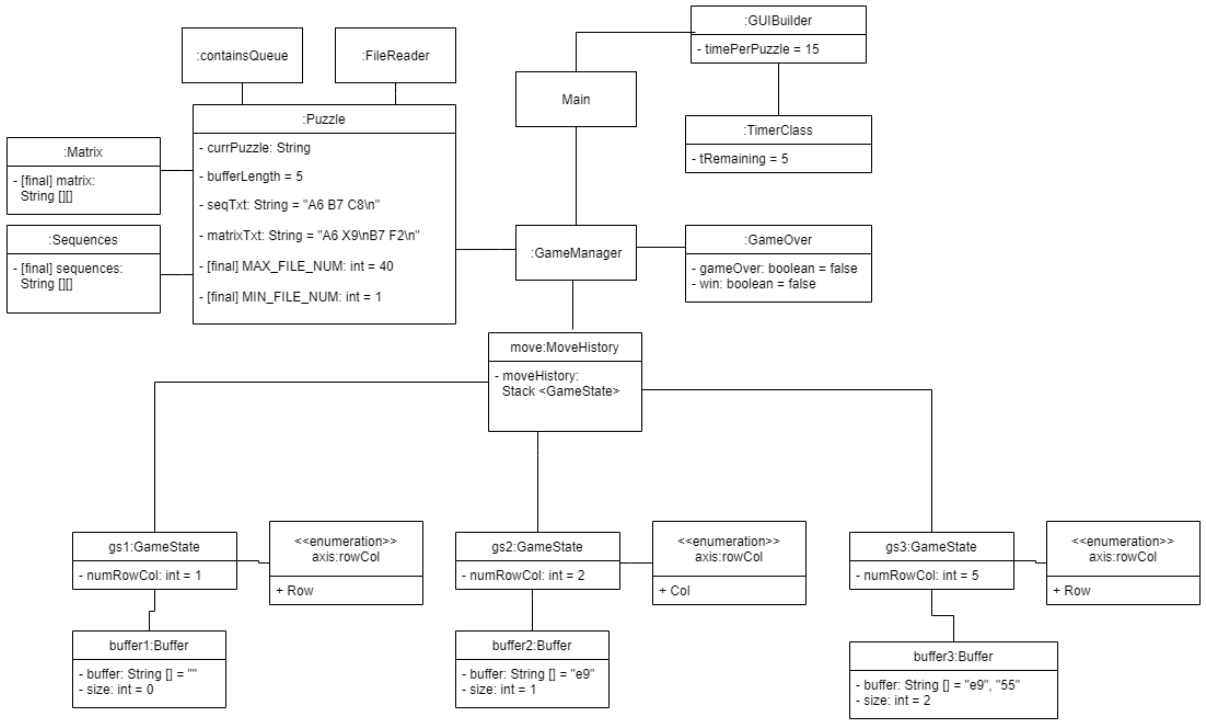
TimerClass: The TimerClass class is responsible for creation and running of the in-game countdown timer. The class is a runnable class which runs in a separate thread. The class has an attribute "time" which determines for how long the countdown timer runs for. The timer class behaviour is executed upon running an instance of the timer class. Having a separate timerClass allows you to define timed behaviour within the system without cluttering the rest of the system with the timer logic. An instance of a timer is initialized and passed two runnable functions, one to be executed every second, and one to be executed

upon time running out. This way classes other than the timer do not have to be “aware” of time passing in any way. Instance of the timer is contained within the GUIBuilder.

GUIBuilder: The GUIBuilder responsible for the creation of the GUI. Using fxml files was causing problems when creating a jar file of the project, so we opted for building all the elements of the GUI using JavaFX. The class creates all the elements of the GUI using an instance of the gameManager which contains all the information about the current game. Apart from building the elements the class also contains all the logic for the behaviour of these elements. This is for the most part done by calling the corresponding functions of the gameManager instance, for example the undo button calls gameManager.undo, then updates the values in the display of the buffer. The GUIBuilder generates a fresh GUI when the game is refreshed and creates a new window when the game is over. The GUIBuilder prevents from cluttering the main class with the GUI logic.

Object diagram

Author(s): Quinn Rutheford



Link to higher quality image

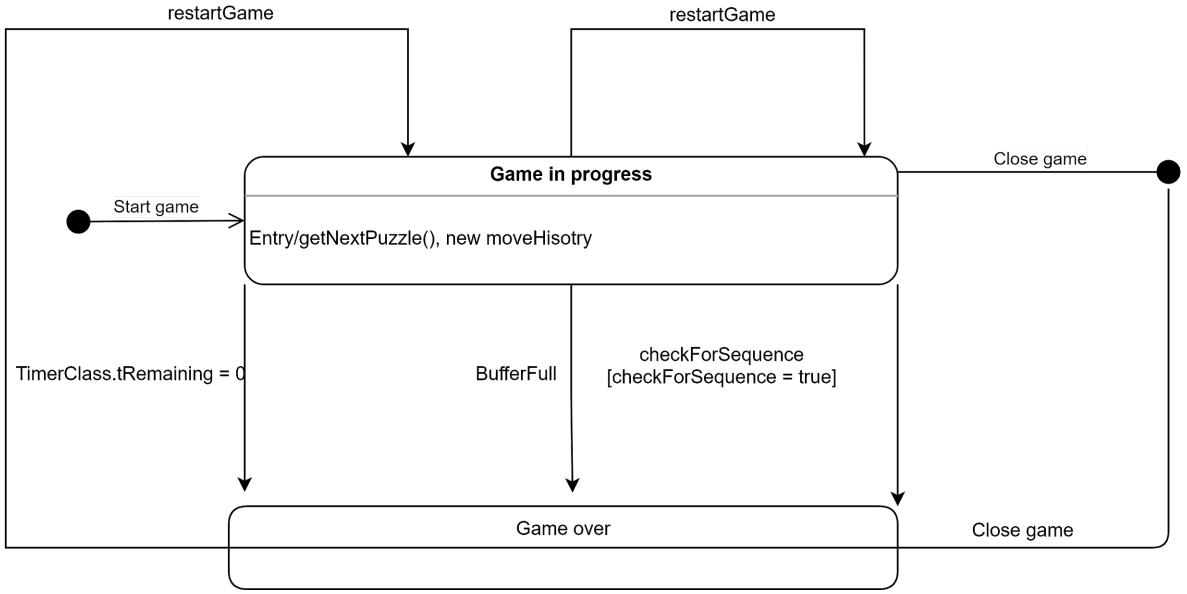
This object diagram shows the game once Puzzle has read a puzzle from fileReader, initialized the matrix and sequences with the values read (these are not shown as they are quite long). The length of the buffer has been passed to move so that it can initialize the buffer array to that length. This would then create gs1 which creates buffer 1 and initializes numRowCol to 1, and rowCol to Row. The GUIBuilder has already initialized the display and shown the matrix and the sequences. The user picked e9 as their first value and a new gameState gs2 was pushed onto the stack moveHistory in move. The rowCol in gs2 is Col that in gs1 as this value alternates between Row and Col. The user has also picked a third value from the second column (the previous numRowCol was 2). At this point in the game there are 3 separate GameStates as we have the initial state, the state with one value picked, and the current state. These previous gameStates are saved in a stack in move so that the user could go back a step at any point. At this point in time the game continues to run as GameOver.gameOver is false. The GameManager updates and checks this value everytime a new user choice is made. The time remaining (tRemaining) in the TimerClass is 5 and therefore the user has 5 second left to complete the puzzle or they will lose.

Note: The attributes of GUIBuilder are not shown as they are JavaFX classes.

State machine diagrams

Author(s): Nikita Sazhinov

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 in the “Game in Progress” state.

Game in Progress:
This state models the behaviour of a game in progress.
Upon entering the state, the gameManager executes getNextPuzzle() and creates a new moveHistory.
The restartGame method causes the gameManager to transition into the Game in Progress state.
This state transitions into the Game Over state, representing the state when a game is not in progress, meaning its won or lost. The following events cause the transition:

- TimerClass.tRemaining = 0: modelling the timer running out.
- BufferFull: modelling the user filling up the buffer
- checkForSequence[checkForSequence = true]: modeling the buffer containing one of the sequences.

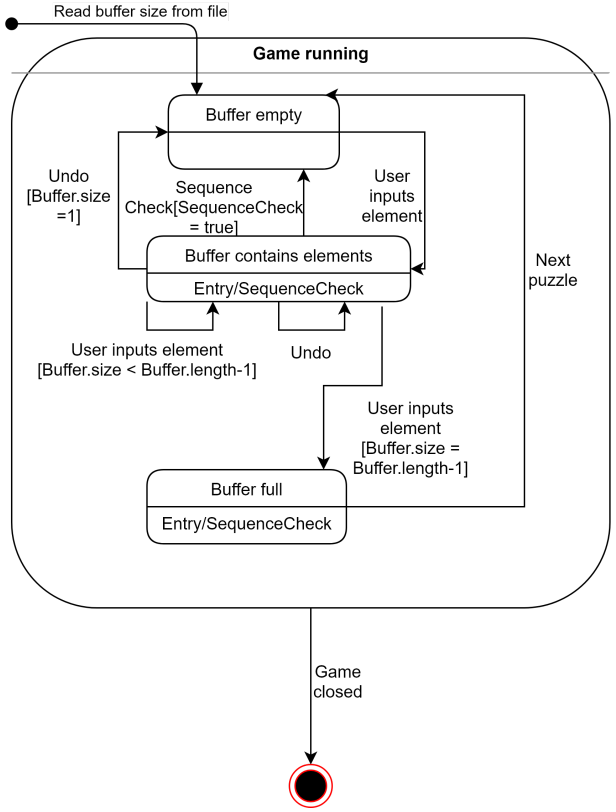
The state can be exited if the game is closed transitioning into a final state.

Game Over:
This state models the behaviour of the game being over, regardless of if the user wins or loses. This state can be reached by three transitions from the Game in Progress state:
From the game over state, the game can be closed transitioning into the final state or the game can be restarted when the restartGame event occurs, transitioning back into the game in progress state.

When the game is closed it ends.

This was designed in this way so the game may run indefinitely as long as the user chooses to continue onto the next puzzle, but also may be closed from any state simply by closing the game.

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

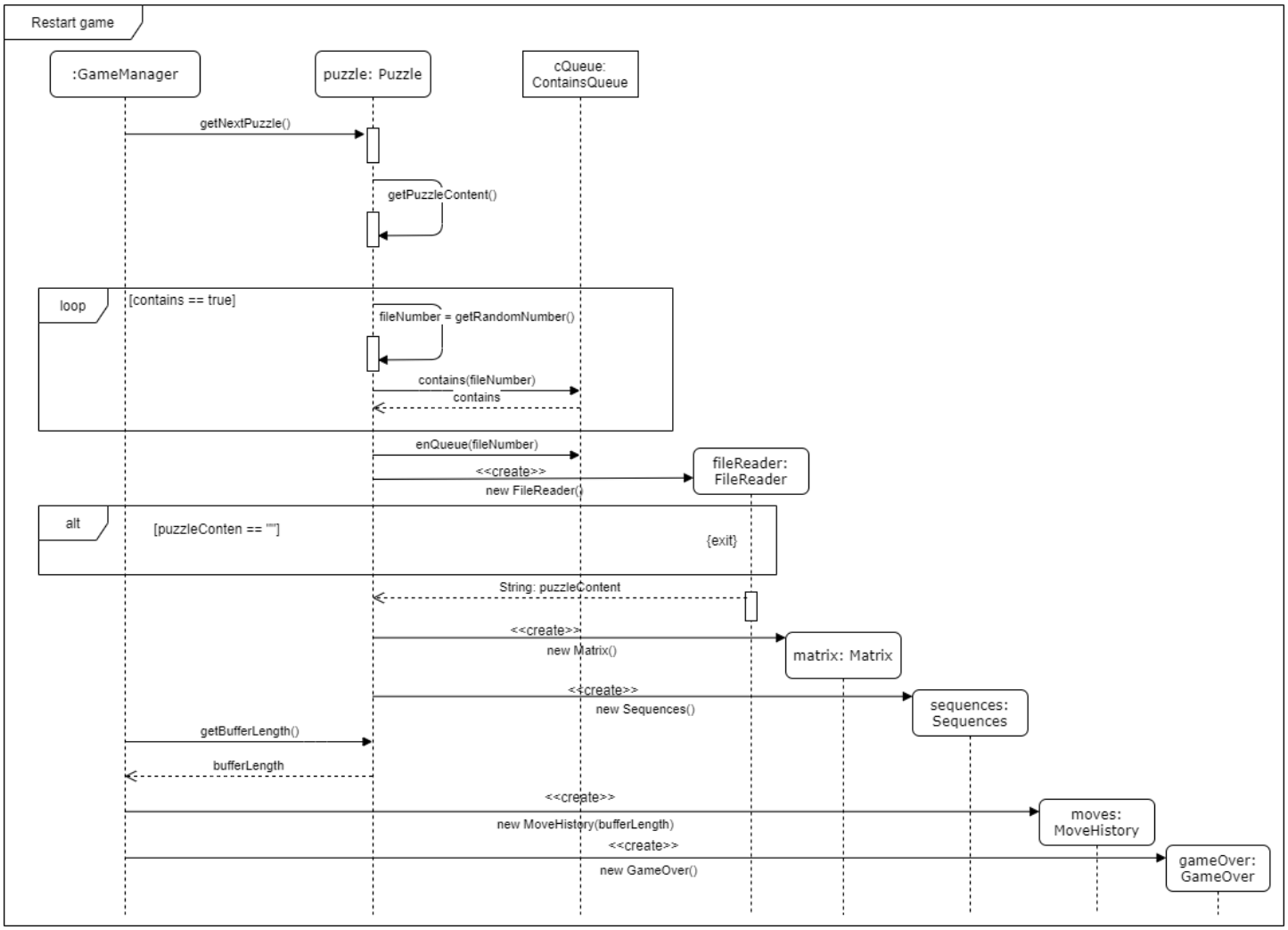
This state models the behaviour of the buffer when it is full. Upon entry to the state, the SequenceCheck event occurs, checking if the buffer contains any of the winning sequences. This state only has one outgoing transition, caused by the Next puzzle event occurring, which creates a new buffer, returning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

The buffer states are defined to be able to have the undo functionality, such that from any state of the buffer the user can undo his last move.

Sequence diagrams

Author(s):J. Antonio Fortanet C.

Sequence Diagram 1: Restart game



This sequence diagram represents the process of setting up the game, initiating the different classes within this class and reading the chosen file from the puzzle database.
This process is done every time the game is run for the first time and when the Refresh button is pressed to play the game again.

First, GameManager calls setupPuzzle to puzzle which handles the process of selecting, reading and parsing a text file from the ‘puzzles’ folder.

Then, GameManager calls getNextPuzzle to Puzzle. This method first randomly chooses a puzzle from the puzzle database by selecting a number from 1 to the amount of files in the puzzle folder, and then reading the file that is named after the chosen number. If the number for choosing the file is contained in the ContainsQueue, then a number is chosen again until this is not the case. This is done so a puzzle is not played in succession or after just a few

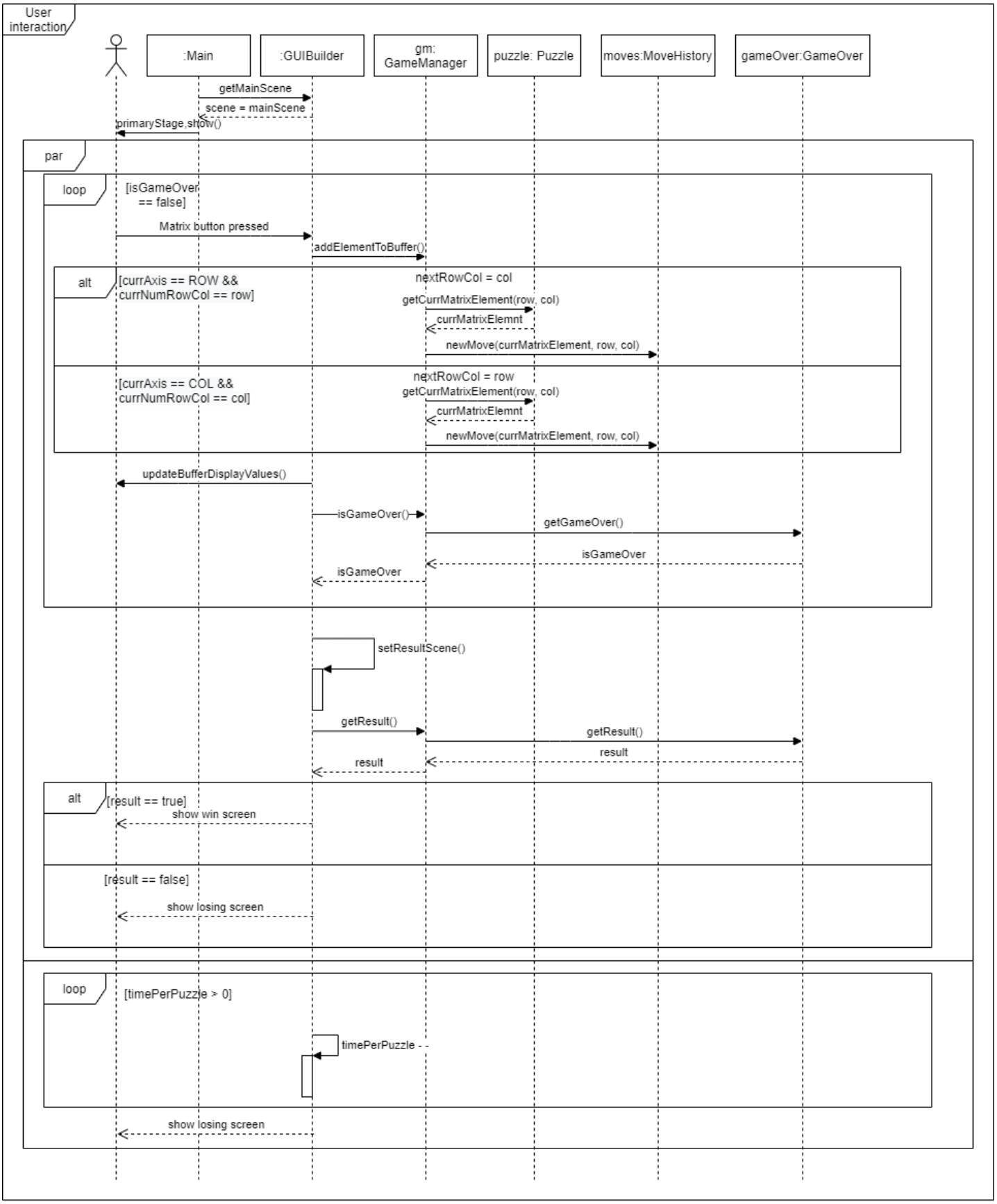
games. In our implementation we found that 5, as the least number of games to encounter the same puzzle, is an appropriate amount.
Then a FileReader is created to get the content of the chosen file. During this process the FileReader. In our implementation, we chose to read the file as an InputStream and then convert it to a String, so it can be used more easily, because with the use of a Scanner it would not be possible to create a .jar file out of the project. This part is omitted from the diagram because it is irrelevant to the implementation and is more of a technical trait.

If the reading the file is not successful, an error message is printed and the game exits after. Otherwise, the content of the file is returned to Puzzle where it is parsed and divided into its parts to be used to initialize Matrix and Sequence.

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 the internal working of the Puzzle class, and only the creation of Matrix and Sequence is shown.

Next, GameManager asks Puzzle for the buffer length to initialize MoveHistory. And lastly, MoveHistory and GameOver are created.

Sequence Diagram 2: User interaction with the game



This sequence diagram shows the interaction that the user will have with the game along its run. This is only to show the basic run of the game, so the refresh and undo functions are omitted, since they work under the same principles. At the beginning, the display is shown to the user by Main by using the GUI made by GUIBuilder according to the starting conditions of the game.

The inner workings of GUIBuilder are omitted and/or abstracted because, in this diagram, it is not relevant to show the logic that the running game is following, and would make the diagram needlessly complicated.

This sequence diagram after the previous part is divided in two parts that are surrounded by a par combined fragment to represent that they run in parallel in the code.

The first part represents how the user interacts with the game and how it changes according to said interaction.

This first part starts inside a loop, with the user choosing an element from the matrix and clicking on it. After, the action of the button is executed, which calls addElementToBuffer and if the current axis row/column matches the permitted matrix elements for the turn, the selected element is added to the buffer by calling newMove on MoveHistory.

Then, the display is updated according to the changes made by the user choice. Then GUIBuilder calls isGameOver on GameManager, and GameManager calls isGameOver on GameOver which returns whether the game has reached the conditions to end or not. This is designed this way to avoid deep calls from GUIBuilder to GameManager, and this also applies to other parts of the code. If isGameOver returns true then the loop ends and the program continues, else, the loop repeats until isGameOver returns true.

After the loop ends, GUIBuilder calls setResultScene on itself which calls getResult on GameManage,r and GameManager calls getResult to GameOver. If result returns as true, then the win screen is shown to the user, else, the losing screen is shown to the user.

The second part of the par combined fragment, there is a set time that runs in GUIBuilder and every second it goes down by one. If the time reaches zero the user is shown the losing screen.

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. 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 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. 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. For implementing the display we initially were going to contain the code inside the main class as this is the class JavaFX would run from, we quickly realized that the code for the display would be longer than initially thought. To handle building our display dynamically we created the GUIBuilder class which builds out each part in a separate function and then combines them to make it. Using each part of the GUIBuilder to build sections of our display helped us maintain the code organized and working. Implementing features such as the refresh button was quick as we had already built out all of the necessary functions and just had to call them. The implementation of making our sequences iterable helped remove iteration code from other sections of the project allowing for more simplicity. After all features were implemented we play-tested and fixed minor bugs, such as random array index out of bounds.

Key solutions

Storing Game-State and Buffer Values

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.

Checking of the Buffer/Sequences

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 design of the timer class has been changed multiple times throughout the development process. The class always was designed to keep track of the time passed during runtime and execute dynamic behaviour.

In the first version of the game, the timer was contained within the gameManager class and called a function passed as a parameter to it when the time ran out. This function contained the code for losing the game, since the game was played in the console that code was just printing ”Game Over” and exiting. A separate window that showed the time remaining opened because in the console view it would be hard to display the time remaining dynamically along with the actual game. This window was made using JavaFX animation features, dynamically changing the string of a label into the time remaining. Because the actual game timer was run in a separate thread the time for the GUI Timer was a separate variable, complicating and cluttering the design.

The final version of the timer class was modified to have two functions as attributes, one to be executed every second and one upon the time finishing. An instance of the timer class is now contained within the GUIBuilder class, this way the GUI does not have to utilize an animation but instead is dynamically modified by the functions called by the timer class. JavaFX does not allow for modifying its objects from a separate thread normally, so the functions had to be implemented as lambda functions run inside Platform.runLater method which allows for multithreading in JavaFX.

A new instance of the timer is created every time the game is refreshed, which at first caused a bug where the old timer would continue to run alongside the new one, decrementing the time remaining the amount of times the refresh button was pressed. This was fixed by making the timer a global variable within the GUIBuilder class, so this way the timer can be cancelled from within the refresh method.

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/assignment3.jar'.

Video Showcase

Time log

Member	Activity	Week number	Hours
Quinn Rutherford	Reviewing feeback	6	1
Quinn Rutherford	Working with JavaFX	6	3
Antonio Fortanet	Working with JavaFX	6	2
Nikita Sazhinov	Improve timer class	6	1
Quinn Rutherford	Resource files into Jar	6	3
Antonio Fortanet	Reviewing feedback	6	1
Nikita Sazhinov	Reviewing feedback	6	1
Antonio Fortanet	FileReader implementation	7	2
Antonio Fortanet	GUI buffer implementation	7	2
Antonio Fortanet	GUI undo implementation	7	2
Antonio Fortanet	Worked on result screen	7	1
Niatna Tesfaldet	Searching for information	7	3
Niatna Tesfaldet	Only possible to click another butto	7	3
Niatna Tesfaldet	Made a start on the highlight and th	7	2
Niatna Tesfaldet	Made sure that you could click on t	7	2
Niatna Tesfaldet	Made the sequences and buffer vis	7	2
Niatna Tesfaldet	Finished the restart button and step	8	4
Quinn Rutherford	Class diagram description	8	1
Quinn Rutherford	Code implementation	8	2
Nikita Sazhinov	Refactor timer	8	3
Nikita Sazhinov	Refactor GUI with current code	8	1
Nikita Sazhinov	Change document to match code	8	2
Antonio Fortanet	Refactor refresh function	8	1
Antonio Fortanet	Fixed buffer pane buider	8	1
Quinn Rutherford	Updating Class diagram	8	2
Nikita Sazhinov	Update state machines	8	1
Quinn Rutherford	Iterator Implementation	8	1
Quinn Rutherford	Object Diagram Update	8	1
Quinn Rutherford	Final review of document	8	1
Antonio Fortanet	Sequene Diagrams	8	2
Antonio Fortanet	Cleaning up Code	8	2
Antonio Fortanet	Finalizing report	8	1
Antonio Fortanet	Sequene Diagram Description	8	1
		TOTAL	58