Assignment 3

Team number: CyberpunkHackingMinigame Team 13

Name	**Student Nr.**	**Email**
Yu Chen	2668777	yuna.chen@student.vu.nl
Berry Chen	2672919	z6.chen@student.vu.nl
Xiaojun Ling	2664286	x.ling@student.vu.nl
Yudong Fan	2662762	y.fan@student.vu.nl

Format: **The name of each class in bold**, the attributes, operations, and associations as underlined text, *objects are in italic*.

Used modelling tool: StarUML

FSummary of changes of Assignment 2

Author(s): Yu Chen

Revised Part:

Class Diagram:

- High-level descriptions and evaluations are added.
- Most of the getters and setters are removed from both the diagram and the textl description.
- About non-navigability in the Class Diagram: The modelling tool we use does not contain the cross ended lines for non-navigability. It only has the one arrow solid line to represent the direction.

State Diagram:

- In the Menu State, the guard condition of the transition from other game difficulty to the default difficulty is fixed.
- In the Game State, cross transition between orthogonal regions is fixed.

Sequence Diagram:

- Text descriptions are modified to avoid redundant explanation of the prescriptive parts in the diagram and try to be more evaluative.

Implementation:

- Grammar errors fixed.
- Several implementations are modified according to the code reviews. (e.g. setDifficulty() switch cases, coordinate pass-through, some redundant coding)

New Stuff:

Class Diagram:

- New Classes are introduced: **GameUI**, **PuzzleHandler**, **ScoreHandler**, **Color Factory, Command**
- Attributes in **Status** changed from codeMatrix, buffer, daemons to puzzle alone.
- Undo Cool Down system added in **Game** class.
- **GameLogic** no longer has direct access to the puzzle and its entities.
- In Daemon, the state rewarded is refactored to checked. The reason is explained in the Class Diagram section.

Object Diagram:

- *puzzleHandler* and *gameUI* objects are added.

State Diagram:

- A new orthogonal region which refers to the undo cool down system is added to the Game State.

Sequence Diagram:

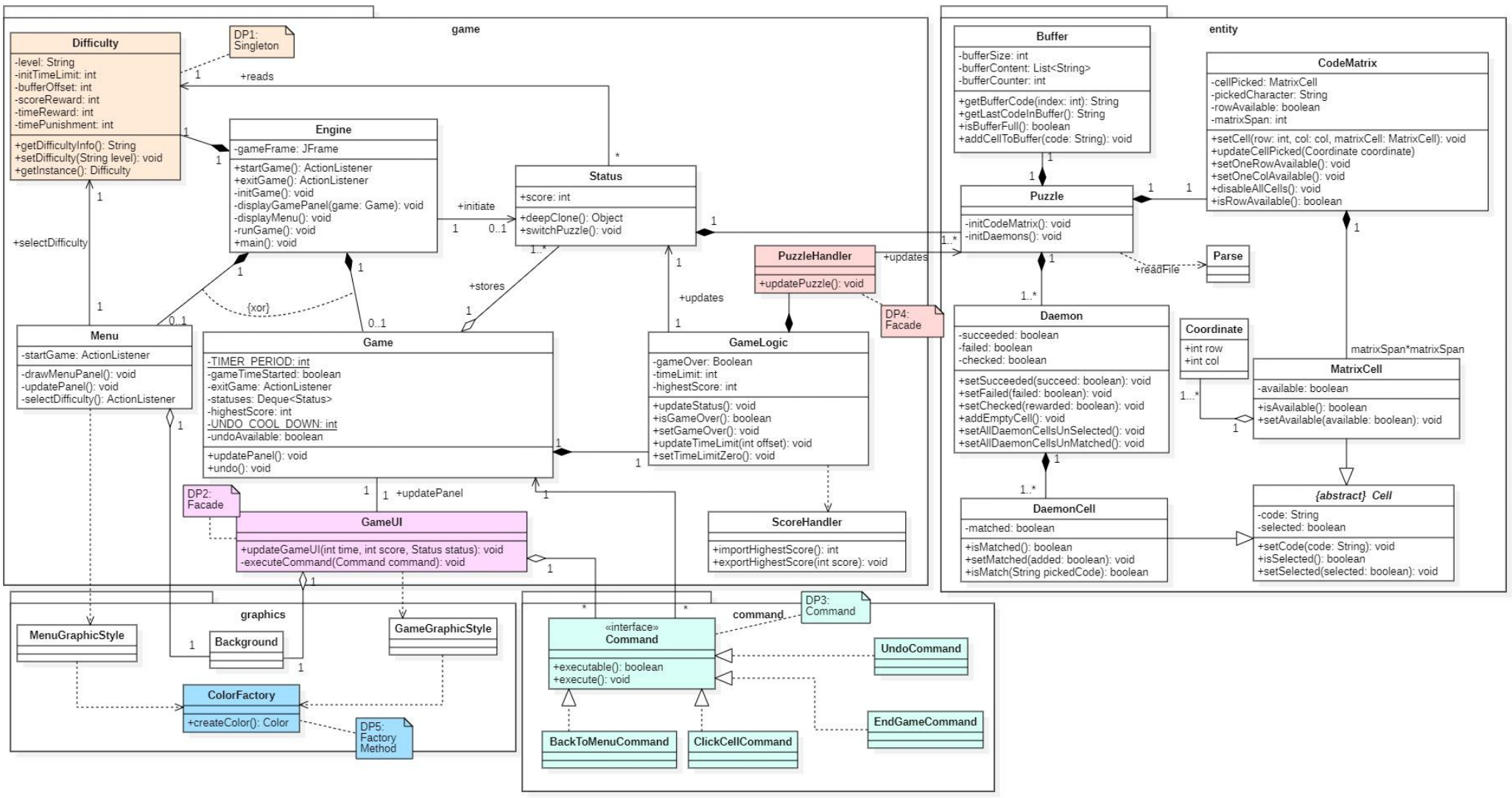
- Both of the sequence diagrams are remodelled because the communications between several classes are changed.

Implementation:

- The description of the Difficulty and Score features are added in the Key Solutions part.
- Undo feature added description about the cool down system.

Application of design patterns

Author(s): Yu Chen



[Figure 1 - Patterns (Click to open the original picture)]

	DP1
Design pattern	Singleton
Problem	In the system, we want to ensure that the **Difficulty** read by client classes such as **Status** and **GameLogic** is the same game*Difficulty* selected in the **Menu**. In this way, we can guarantee consistency with user choice and game performance in terms of difficulty level. If the unity of the **Difficulty** instance is not ensured, it could end up with the situation that the **GameLogic** calculates the timeLimit and score based on a different set of difficulty values against the difficulty chosen by the player.
Solution	Singleton is applied in **Difficulty** class which means once the **Difficulty** class is instantiated, other attempts of creating a new **Difficulty** object will lead to the reference of the existing object. Therefore, only one instance of **Difficulty** can exist in the execution of the code.
Intended use	The **Difficulty** instance is created inside the **Difficulty** class (a global variable for holding a reference to the single instance) and the constructor of **Difficulty** is kept private, so clients cannot create duplicate objects. A public accessor operator getInstance() is the only way to get the instance of **Difficulty** if needed. In the system, this operation is called by Engine. getInstance() returns the only object of **Difficulty** to the **Engine** and then this object is passed to the **Menu** for further operations.
Constraints	Since the singleton pattern is applied, we are compensating with the immutability of the **Difficulty** class. Clients who can access the instance of **Difficulty** also have the ability to change the level of difficulty through the public method setDifficulty(String level). Now, this is controlled by only calling the setter in the **Menu** class. If we want further security of the **Difficulty** class, a new package that only contains **Menu** and **Difficulty** can be created and the visibility of the setDifficulty(String level) method can be set to package only.
Additional remarks	An alternative pattern that can be implemented to solve the problem is the FLYWEIGHT pattern. Using this pattern we can have different levels of difficulty as distinct objects and their uniqueness is guaranteed. The biggest advantage is the objects are immutable. However, we need to create more Difficulty subclasses for the FLYWEIGHT pattern and have to change a lot in our original design.

	DP2
Design pattern	Facade
Problem	In the **Game** class, as panel objects and features increase, this class becomes more and more complex and needs to handle too many functionalities, such as draw the panels, store the statuses, choose which status to display, etc.
Solution	By applying the Facade pattern, we moved objects and methods related to create and display panels to the facade class and in the **Game** class, we can simply use the interface the facade class provides to draw the game panel. The complexity behind drawing the panel is hidden by the facade class. In this way, the functionality of the **Game** class is simplified and the **Game** class does not need to know the logic behind displaying the UI.
Intended use	A **GameUI** class is created that includes the updateGameUI() interface. In this class, all components that needed to draw the game UI are created and added to the background panel. The client **Game** class uses the interface by invoking the updateGameUI() method with parameters time, score and status.
Constraints	Apart from the information passed by parameters, the GameUI needs to know extra information from the **GameLogic**, such as isGameOver to handle which panel to display (gaming panel or game over panel). To solve this problem without exposing **GameLogic** to **GameUI**, intermediate operations to pass information from **GameLogic** to **GameUI** is needed in **Game**.

Additional remarks	Reference: *Engineering software products, Ian Sommerville*, chapter 8, page 239.

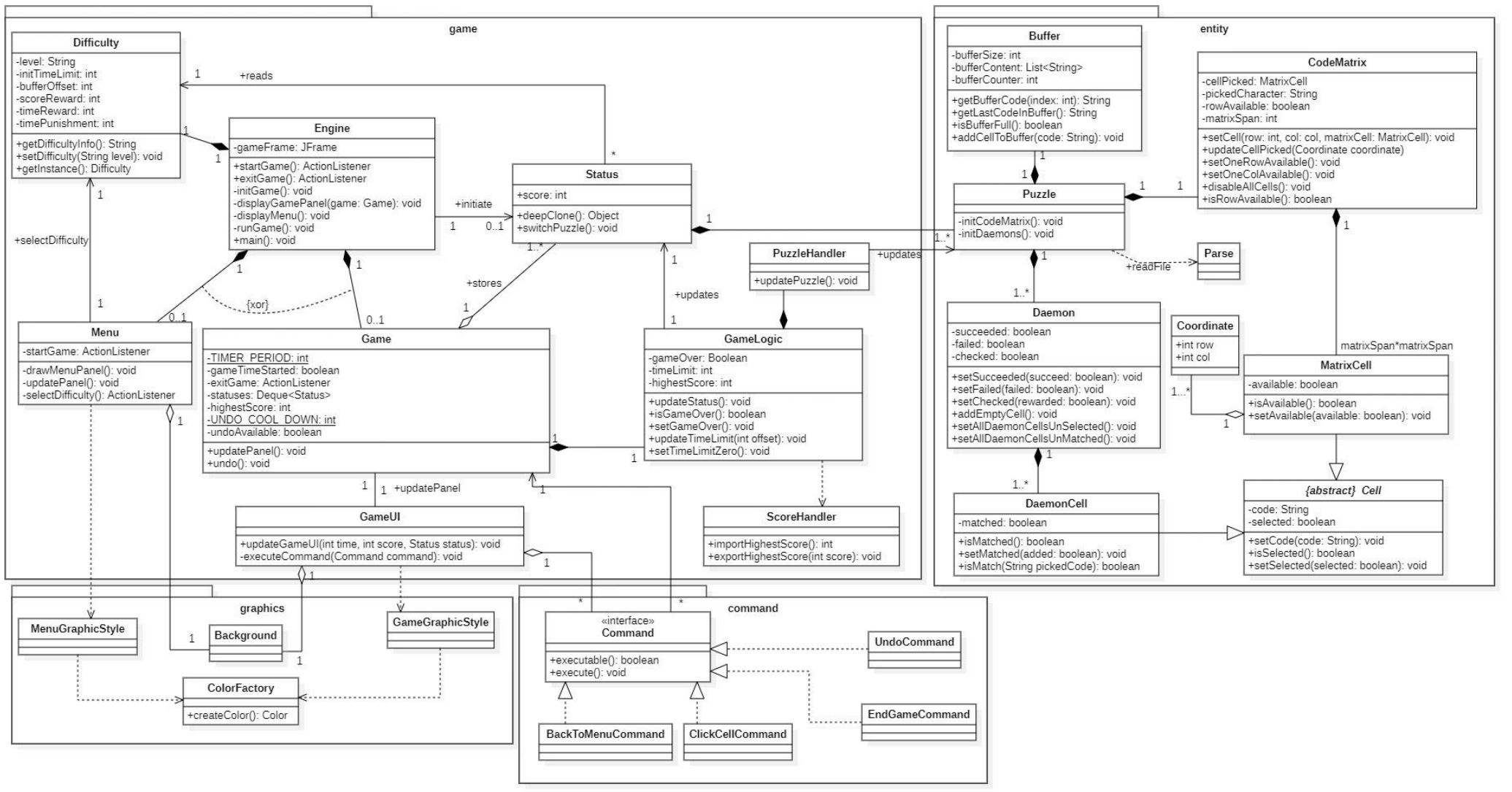
	DP3
Design pattern	Command
Problem	In the **GameUI** class, event listeners need to be added to corresponding buttons such as the click matrix cell event, back to menu event and undo event, etc. when creating the buttons. These events involve invoking functions in **Game** class and we do not want to pollute the purity of the **GameUI** class which means the **GameUI** class shall only need to draw the UI and add the events to related components without knowing the logic under these events.
Solution	Command pattern is implemented to encapsulate events and make them independent from the way of execution. A command interface that includes executable() and execute() abstract methods are defined and there are four concrete classes (**ClickCellCommand**, **UndoCommand**, **EndGameCommand**, **BackToMenuCommand**) that implement the command interface. These command objects act as storage that stores operations in **Game** and later on are executed in **GameUI**.
Intended use	The **GameUI** class contains a method executeCommand() which takes the command object as a parameter and calls its execute() method if its executable() operation returns a *true*. For each event listener (**ActionListener** and **MouseListener**) that is added to the buttons (e.g. undo button), it only needs to execute the method executeCommand() with corresponding command object as its parameter.
Constraints	Even though we are using **Command** to access **Game** operations and hide the underlying complexity, **GameUI** is free to invoke **Game** functionalities directly (because they need to be public to be used in **Command**). However, if we treat **GameUI** as a client, it can use the **Command**'s interface conveniently and if there are any changes in the event operations in **Game**, no changes in **GameUI** is needed. Thus the maintainability is improved by applying the **Command** pattern.
Additional remarks	Usually, the use of command patterns can help to track the history of commands and implement undo easily. However, this function is not suitable in our system, because the undo feature in this game is not based on the reverse of commands but the status of each entity.

	DP4
Design pattern	Facade
Problem	The **GameLogic** class updates entities such as **CodeMatrix** and **Buffer** through getters in **Status** which returns the result from the getters in **Puzzle**(e.g. getCodeMatrix(), getBuffer()). The message chain is too long and severely contradicts the Law of Demeter.
Solution	A facade class **PuzzleHandler** is introduced to connect **GameLogic** and entities in **Puzzle**. The facade class provides an interface for the GameLogic to call to update **Puzzle** so the **GameLogic** does not have direct access to the entities in **Puzzle** and does not need to know complexity under the interface. The introduction of **PuzzleHandler** reduces the overall complexity of the interactions between **GameLogic** and entities in **Puzzle**.
Intended use	When **GameLogic** tries to update the **Puzzle** of the *status*, it calls the **PuzzleHandler**'s method updatePuzzle() with the *status*'s *puzzle* and the *coordinate* of the matrix cell that is clicked by the player. The **PuzzleHandler** then updates **CodeMatrix**, **Buffer** and **Daemons** one by one according to the *coordinate*. The interface also tells the **GameLogic** if all **Daemons** are checked, so it then can notify the Status to switch a new Puzzle. Also, **GameLogic** can allocate rewards based on the counts of Succeeded and Failed **Daemons** provided by the **PuzzleHandler**.
Constraints	Although we shortened the message chain and decoupled **GameLogic** with **Puzzle** entities, the pattern does not prohibit **GameLogic** from accessing the functionalities of **Puzzle** and its entities directly. To avoid this, we added three interface methods isAllDaemonschecked(), markUncheckedDaemonsFailed() and countUncheckedDaemons(). These methods are irrelevant to the updatePuzzle() method but are needed for the **GameLogic** to allocate rewards and finish the game.
Additional remarks	None.

	DP5
Design pattern	Factory Method
Problem	In the **MenuGraphicStyle** and **GameGraphicStyle** interface, around 10 different colors are used to decorate the graphic user interface (GUI). The creation of **Color** objects which contain only rgb values are lack of readability and are difficult to debug. Also, there is an overlap between the colors used in the Menu and the Game panel.
Solution	A Factory method pattern is implemented to abstract the process of **Color** creation. The clients only need to use the interface provided by the **ColorFactory** to create the determined color. Furthermore, any new customized colors can be added easily into the **ColorFactory** for the **GameGraphicStyle** and **MenuGraphicStyle** to use.
Intended use	A **Color** is created in **MenuGraphicStyle** and **GameGraphicStyle** by call the createColor() function of **ColorFactory** with the name of the color in String type, such as createColor("theme"), the **ColorFactory** uses switch-case to match the String and return a new **Color** corresponding to the String's case.
Constraints	The String used to invoke the createColor() method may duplicate many times, for example, the "theme" and "subTheme" String are used more than 10 times.
Additional remarks	A **FontFactory** is created due to similar reasons. However, in our system, the style of all the **Font** used are the same and the only difference between the **Fonts** is the size. So the implementation of the **FontFactory** may look trivial but it actually helped to reduce the redundant code when creating new **Font** in **GameGraphicStyle** and **MenuGraphicStyle**.

Class diagram

Author(s): Yu Chen, Berry Chen, Xiaojun Ling, Yudong Fan



[Figure 2 - Class Diagram (click to open original picture)]

[**Figure 2]** above illustrates the class diagram model of the system. Each class in the model is elaborated as follows:

Package: entity

Classes that are entities of the **Puzzle** are located in this package.

CodeMatrix

CodeMatrix has a two-dimensional array as an attribute to store all the matrix cells, which is shown in the class diagram by the composition relationship between class **MatrixCell** and class **CodeMatrix**. All the attributes and operations regarding the available region, matrix as a whole and picked matrix cell are specified in this class.

Attributes

- cellPicked: MatrixCell – The matrix cell that was just picked by the user.
- pickedCharacter: String – The character of the matrix cell that was just picked by the user. This attribute is needed because the code in cellPicked is set to "[ ]" after **CodeMatrix** is updated.
- rowAvailable: boolean – The boolean value that indicates whether a column or a row of matrix cells are available in the matrix. If rowAvailable is true, a row is available, otherwise, a column should be available.
- matrixSpan: int – This number reflects how many rows or columns there are in the matrix.

Operations

- setCell (row: int, col: int, matrixCell: MatrixCell): void – This operation is called in **Puzzle** to initiate a CodeMatrix by placing the matrix cell into a two-dimensional array of matrix cells which represents the code matrix according to the row and column it is in.
- updateCellPicked(coordinate: Coordinate): void - It is called when an available matrix cell is picked by the user. If the operation is called with the parameter coordinate, the corresponding matrix cell is set as selected, and the attributes cellPicked and pickedCharacter are updated accordingly.
- setOneRowAvailable(): void – Changes the value of attribute rowAvailable to indicate the shape of the next available area and sets a row of matrix cells available.
- setOneColAvailable(): void – Same as above but a column of matrix cells are set available.
- disableAllCells(): void – All cells from the matrix are marked as unavailable. Used as a combo with setOneRowAvailable()/setColAvailable() to clear the availability of all matrix cells before the available area is changed.
- isRowAvailable(): boolean - Return the value of attribute rowAvailable to indicate the current available area in the code matrix.

Association

- **Puzzle** composition - The **Puzzle** initializes the object *codeMatrix: CodeMatrix*. Also, one instance of **Puzzle** corresponds to one instance of **CodeMatrix**, objects of **CodeMatrix** do not exist without the **Puzzle** instance.

Daemon

The **Daemon** class stores one or multiple daemons that correspond to the puzzle. It contains the states of daemons such as SUCCESS and Fail and also the operations relative to update the state of daemons. The reason to use separate states: succeeded, failed and checked instead of using an enum to package them into one daemonState attribute is because there is an overlap between these states. This is shown in the State Diagram section.

Attributes

- succeeded: boolean - A state of sequence that indicates the daemon is already marked with SUCCESS when the last cell in this daemon is marked with matched.
- failed: boolean - A state of sequence that indicates the daemon is already marked with FAIL when the time is up or there is not enough buffer space to match the daemon.
- checked: boolean - A state of sequence that indicates the daemon is already checked and counted by **PuzzleHandler** to be SUCCESS or FAIL. This attribute is used to avoid duplicated counting which leads to repeated rewards. The name of the state is refactor from rewarded to checked because in the design in Assignment 2, **GameLogic** directly allocated rewards and marked the daemon with rewarded state when traversing the Daemons. In Assignment 3, PuzzleHandler traverses the Daemon and counts the number of successes and fails, then the GameLogic starts to allocate the rewards, therefore, when the Daemon is checked, it has not been awarded yet. The name rewarded is not suitable in the new design.

Operations

- Setters: void - set the state of the Daemon to succeeded or failed and checked.

Association:

- **DaemonCell** composition: Dependency exists between **Daemon** and **DaemonCell** and one instance of **Daemon** consists of one or multiple **DaemonII**. If **Daemon** is deleted, **DaemonCell** no longer exists.

Buffer

The **Buffer** class stores all the buffer related attributes, the size of the buffer, the content in the buffer and an index to the available space in the buffer, respectively. It also defines the operations that manipulate these attributes or simply return the information of them.

Attributes

- bufferSize: int – Size of the buffer, indicating how many codes can enter the buffer.
- bufferContent: List<String> - A list of codes of the picked matrix cells sorted from left to right by the time of entry.
- bufferCounter: int – An index number that indicates the next free space in the buffer.

Operations

- getBufferCode(index: int): String – Return the code in the attribute bufferContent with the parameter as an index.
- getLastCodeInBuffer(): String – Return the last code to enter the buffer.
- isBufferFull(): boolean – Return true or false to determine whether the buffer is full.
- addCellToBuffer(code: String): void – Add the parameter code to the buffer and increase the buffer counter to point to the next free space in the buffer.

Association

- **Puzzle** composition – The **Puzzle** initializes the object *buffer: Buffer*. One instance of **Puzzle** corresponds to one instance of **Buffer,** and the existence of an instance of **Buffer** is dependent on **Puzzle**.

<abstract> Cell

An abstract class which stores the code String and the selected state. This class has two concrete inheritance classes: **CodeMatrixCell** and **DaemonCell**. This class is needed because its subclasses have the same code attribute and selected state but also extended the functionalities of **Cell**.

Attributes

- code: String - Stores String characters such as “55” and “E9”. The meaning of code is the same for both of its inheritance classes **MatrixCell** and **DaemonCell**.
- selected: boolean - The meaning of selected is different for **MatrixCell** and **DaemonCell**. In **MatrixCell**, a selected cell refers to a **MatrixCell** that has been selected by the player. Once the selected state of a **MatrixCell** is set to true, this state shall not be altered again. For **DaemonCell**, selected refers to the current **DaemonsCell** that is being selected by the player. For each **Daemon**, the number of selected **DaemonCell** it contains shall only be zero or one.

Operations

- setCode(String code): void - Although the code is determined by the constructor, there are cases that need to change the value of code. For example, when a MatrixCell is clicked by the player, the code of that cell shall be changed to “[ ]”.
- isSelected(): boolean / setSelected(selected: boolean): void - Return the boolean value of selected. / Set selected to the boolean value in parameter. The parameter is needed for setSelected() because the selected state can switch from false to true and true to false (in the case of **DaemonCell**).

MatrixCell

The **MatrixCell** class inherits from abstract class **Cell**, it stores the properties of the availability and the position of each cell in the matrix, also all the operations related to these properties are defined in this class.

Attributes

- available: boolean - A boolean value that indicates whether a cell can be selected by the user.

Operations

- isAvailable(): boolean – Return the value of attribute available.
- setAvailable(available: boolean): void – Set the value of attribute available to the same as the parameter.

Association

- **CodeMatrix** composition – **MatrixCell** is part of **CodeMatrix**, and one instance of **CodeMatrix** contains multiple(matrixSpan*matrixSpan) instances of **MatrixCell**. Only if **CodeMatrix** exists, related instances of **MatrixCell** exist.
- **Cell** generalization – **MatrixCell** inherits from abstract class **Cell**. **MatrixCell** as a subclass provides all the implementations for the operations and attributes of the parent class **Cell**.

DaemonCell

The **DaemonCell** class inherits from abstract class **Cell**, it contains the property of whether the user-selected cell is matched into the buffer's last character, and also matched with daemon cell. Also, the operations related to the property 'matched' are defined in this class.

Attributes

- matched: boolean - A state of the DaemonCell that indicates whether the daemon and buffer have the same code under the same index.

Operations

- isMatched(): boolean - Return true when the cell is already marked with matched.
- setMatched(matched: boolean): void - This operation sets the state of the corresponding daemon cell.
- isMatch(String pickedCode): boolean - Different from the operation 'isMatched()', this operation returns true when the user-selected code is the same as the pickedCode.

Association:

- **Cell** generalization - **DaemonCell** is a subclass of the abstract class **Cell** and it inherits the characteristics, associations and aggregations from **Cell**.

Puzzle

The **Puzzle** class has an inner class named **Parse**. Together they are competent for reading, parsing, and saving the puzzle data for other members to use. Specifically, the **Parse** reads and parses the puzzle source file, and the **Puzzle** initializes relevant entity classes and saves the parsed puzzle data respectively. By having all the relevant entity classes initialized in the **Puzzle**, other classes will only need to to access the Puzzle to get the entity data they need rather than visiting different lower level entity classes.

Attributes

- Buffer: buffer - store the parsed buffer.
- CodeMatrix: codeMatrix - store the parsed code matrix.
- List<Daemon> daemons - store the parsed daemons in a list.

Operations

- initCodeMatrix(): void - This operation is called by the constructor of the **Puzzle**. It traverses the raw code matrix which is in the form of a 2-dimensional array and initiates the new code matrix in the form of the **CodeMatrix**. Initiation includes set values and coordinates for each cell of the matrix.
- initDaemons(): void - This operation is called by the constructor of the **Puzzle** as well. It converts the raw daemon sequences which are in the form of List<String[]> to the new list of Daemon objects which are in the form of the **daemons**.

Association:

- **Buffer** composition - The **Puzzle** creates a new object *buffer: Buffer*, and this shall be the only **Buffer** instance that exists. No other class shall be able to create a **Buffer**.
- **CodeMatrix** composition - The **Puzzle** creates a new object *codeMatrix: Codematrix*, and this shall be the only **CodeMatrix** instance that exists. No other class shall be able to create a **CodeMatrix** .
- **Daemon** composition - The **Puzzle** creates a new object *cookedSeq: Daemon*, and this shall be the only **Daemon** instance that exists. No other class shall be able to create a **Daemon**.
- **Parse** dependency - The **Puzzle** and the **Parse** work together. The **Puzzle** needs to know about the **Parse** to use objects of the **Parse**. The **Parse** provides a set of utility functions that are capable of reading and parsing the puzzle source files. The **Puzzle** will use the utilities that the **Parse** offers.
- **PuzzleHandler** navigability - The **PuzzleHandler** updates the **Puzzle** every time when an action is taken. The **Puzzle** does not exist depending on the **PuzzleHandler** and has no access to the **PuzzleHandler**. This is a relatively weak one-way relationship.

Package: game

In the game package, classes related to the state, execution, display and logic of the game are included.

Difficulty

The **Difficulty** class stores properties such as the level name, initial time limit, etc. at different difficulty levels. There is a default level - "NORMAL". Clients can get the information such as bufferOffset and scoreReward through the getters and change the level of difficulty through the setter interface. There is only one instance of **Difficulty** existing in the whole implementation.

This class is needed to encapsulate the game difficulty settings which are used in **Puzzle** initialization and time and score control in **GameLogic**.
Attributes

- level: String - Name of the current difficulty.
- initTimeLimit: int - Initial time limit. This number is displayed on the game panel when a new game is generated.
- bufferOffset: int - Different buffer size is given at different difficulty. This is done by adding a buffer offset at the original buffer size provided by the set of puzzles files. The attribute bufferOffset stores the determined buffer offset at the current difficulty.
- scoreReward: int - Determine the score awarded to a complete Daemon.
- timeReward: int - Determine the time added to the time limit when a Daemon is complete.
- timePunishment: int - Determine the time subtracted from the time limit when a Daemon is marked as FAILED. Although this figure is the same among all the difficulties at this stage, this attribute is created for potential usage and better maintenance.

Operations

- getDifficultyInfo(): String - A string that describes the information of all attributes in Difficulty and is used to present information to the player at the Menu panel.
- setDifficulty(String level): void - Attributes can be set to values that correspond to the parameter level through matching operation and set attributes to parameter's level values. This means the player can only decide which difficulty to start with without changing the attributes to custom values.

Associations

- **Engine** composition - The **Engine** creates a new object *gameDifficulty: Difficulty* and this shall be the only **Difficulty** instance that exists. No other class shall be able to create a **Difficulty**.
- **Menu** navigability - The player shall be able to select the level of difficulty through buttons on the **Menu** panel. The **Menu** has access to the **Difficulty** but not the other way around. **Difficulty** does not have access to the **Menu**.
- **Status** navigability - **Status** acts as a bridge that can connect **Difficulty** with **GameLogic**. **GameLogic** reads **Difficulty**'s attribute values through **Status** and allocates rewards corresponding to a specific level of difficulty. This access is read-only which means no modification shall be done to **Difficulty** through **Status**.

Engine

This is where the **main** and application window is located, and where the **Menu**, **Game** and first Status are created. The player can switch between the **Menu** and the **Game** through the **Engine**. The whole application will be terminated, once the **Engine** is destroyed. This class acts as the heart of the system.

Attributes

- gameFrame: JFrame - JFrame imported from JAVA Swing library. This *gameFrame* is the carrier of the game and is the base of the graphic user interface. All graphic panels, for example, the menu panel and game panel and events such as mouse click and timer are added on the *gameFrame*. Once JFrame is instantiated, the game window will pop up on the Desktop.

Operations

- runGame(): void - This operation is called by main and creates a new **Engine**. It continuously loops to catch user events and process functions through the *engine* until *gameFrame* is closed by clicking the close button on the top-right of the game window.
- displayMenu(): void - Clear all the components added on the *gameFrame* and paint the *menuPanel* on the *gameFrame*.
- startGame(): ActionListener - This operation returns an **ActionListener** event which can be passed to the **Menu** and added to the *startButton*. Once the *startButton* is pressed, operation initGame() will be called. **ActionListener** is a type of event listener provided by JAVA Swing.
- initGame(): void - The first **Status** of a new **Game** is created. Once the *game* is initialized, displayGamePanel() will be called inside this operation.
- displayGamePanel(game: Game): void- Clear all the components on the *gameFrame* and paint the *gamePanel* according to the *currentStatus* of the *game*.
- exitGame(): ActionListener - The ActionListener returned by exitGame() is passed to the **Game** and added on the *menuButton* on the *gamePanel*. Inside this event, displayMenu() operation is called. In this way, the player may exit the game panel to the menu panel by clicking the menuButton.

Associations

- **Menu** composition - **Menu** is only created inside **Engine** and its graphic presentation which implemented by Swing **JPanel** is added on the *gameFrame* in **Engine**, which means if **Engine** is deleted, **Menu** can not exist alone.
- **Game** composition - Similar to Menu, the **Game** is also created inside the **Engine** and displayed on the *gameFrame*. There is an XOR relationship between **Menu** and **Game** which means they shall not exist simultaneously in the **Engine**. Once the START game button on the Menu is pressed, the **Menu** panel is destroyed and the **Game** object is instantiated.
- **Status** navigability - The **Engine** creates the *firstStatus* and passes it to the **Game**. Engine does not have the status as its attribute (no aggregation association) and status has no access to the Engine. This is a relatively weak one-way relationship.

Menu

The initial graphic interface, which is shown to the player after the application, is opened. In this class, a menu panel that contains several difficulty buttons and a start button is initiated and select difficulty events are created. The player can select difficulty and start the game with the selected difficulty through the menu.

Attributes

- startGame: ActionListener - Received from the **Engine** and is added to the *startButton* to switch to the game panel.

Operations

- drawMenuPanel(): void - A background panel with the game logo and other graphical features is created. There are four difficulty buttons and a start button on the panel. And the information of its corresponding difficulty level is also displayed in the text form.

- updatePanel(): void - Remove all components on the menu panel and repaint them. This operation is called when the selectDifficulty() event is triggered. In this way, displayed difficulty information can be updated quickly on the panel once the player selects a different difficulty level.
- selectDifficulty(): ActionListener - An event listener which clicks on the difficulty buttons and sets the *gameDifficulty* to the corresponding difficulty level. After *gameDifficulty* is changed, updatePanel() is called to refresh the menu panel.

Associations

- **MenuGraphicStyle** dependency - Graphic styles and layout are separated from their user interface components. The **MenuGraphicStyle** Class is not instantiated anywhere inside this model but imported by the **Menu**. This is a dependency relationship because static operations in **MenuGraphicStyle** are called by the **Menu** and work as utility functions that help to decorate those graphic components such as *startButton* and *difficultyButton*.
- **Background** shared aggregation - This panel is used as a background that has a background image and allows other graphical components to be displayed on top of it. Everything that does not change according to events and does not have any functions are drawn on the background image. The reason to use a background image to depict most of the graphics is to have a good looking GUI with a limited number of components that need to be created in JAVA.

Status

The Status keeps recording the current **Puzzle** and the *score*. It works as a bridge to connect the **Puzzle** to the **Game** and **GameLogic**.
The game state can be divided into three parts: puzzle, score, time . Puzzle and score are recorded by **Status** and puzzle is updated by **PuzzleHandler** and score is updated by **GameLogic**, time is recorded and modified by **GameLogic**. The reason that these three components are stored separately is for undo implementation. Once undo is called, puzzle and score are reversed but time shall not.

Attributes

- score: int - Stores current score the player achieves. (not implemented)

Operations

- deepClone(): Object - This function deep copies every single object in the Status including entities in the **Status**'s *puzzle*. Serializable interface is implemented by **Status**, **Puzzle** and its entities such as **CodeMatrix**. Objects in the **Status** are copied through serialization and deserialization. This copy method is most suitable for **Status** deep copy other than copying every object of an attribute until the base attribute is immutable because multiple layers of mutable objects are located under Puzzle, and serialization can avoid potential duplicated objects creation and copy.
- switchPuzzle(): void - Assign a new **Puzzle** to the attribute puzzle. This is called in **GameLogic** when all **Daemon**s in the current *puzzle* are rewarded.

Associations

- **Game** shared aggregation - Most of the *status* are created and saved in **Game** except the *firstStatus* which is created in the **Engine** and used to initiate a new **Game**. **Status** does not have access to the **Game** but **Game** can get every attribute inside the **Status**.
- **GameLogic** navigability - **GameLogic** updates attributes in the **Status**. No new **Status** is created in the **GameLogic** and **Status** does not have access to the **GameLogic**.
- **Puzzle** composition - **Puzzle**s are only created in **Status** and do not exist if **Status** is cancelled.

Game

The "canvas" to display every "snapshot" of **Status** and stores the status histories. It passes the current status to **GameLogic** and replaces the current status with the updated status through **GameLogic** and updates the panel by invoking the interface provided by **GameUI** which draws small panels, such as the code matrix panel and time panel on the "canvas". In Assignment 3, we reduced the complexity of **Game** by applying the Facade pattern.

Attributes

- TIMER_PERIOD: int - Static final variable. Determines the frequency of Timer runs.
- gameTimeStarted: boolean - A flag used to decide whether to start the Timer. The default value is false when a new Game is created. It will be set to true once the MatrixCell is clicked by the player.
- exitGame: ActionListener - An event listener takes from the Engine which calls displayMenu() operation in the **Engine**. The event is bound to the "MENU" button and is triggered when the button is clicked.
- statuses: Deque<Status> - A Deque which stores *currentStatus*. Since the *puzzle* in **Status** is mutable, after a **MatrixCell** in the code matrix is clicked by the player, *currentStatus* will be pushed into the Deque and deepClone() is called to make a copy of *currentStatus*. Only the copied version is sent to **GameLogic** to apply updates. In this way, the original *currentStatus* is saved for better maintenance and further implementation such as UNDO.
- highestScore: int - Stores the highest score the player achieves. The default value is retrieved from the txt file and it alters when the current score in **Status** exceeds the highestScore value. (not implemented)
- UNDO_COOL_DOWN:int - Static final variable. Determines the cool down time for the undo operation.
- undoAvailable: boolean - The default value is true. If the undo button is clicked, this variable switches to false. The value becomes true again once the undo cool down time goes to zero.

Operations

- updatePanel(): void - Passes the time and highest score from **GameLogic** and the *currentStatus* to **GameUI** to repaint the panel.
- undo():void - Reverse players last click on the matrix cell. The *currenStatus* is placed by the *status* popped from the status history - Deque statuses.

Association

- **GameLogic** composition - **GameLogic** is created in **Game** and is fully dependent on **Game**. It updates the status passed by **Game** and returns the updated version back to **Game**. If **Game** is destroyed, **GameLogic** can not exist either.
- **GameUI** association - **GameUI** is instantiated in **Game** and **Game** is an attribute of **GameUI**. There is a mutual communication between these two classes. **Game** provides information for **GameUI** to display and **GameUI** provides an interface for **Game** to use.

GameUI

The facade class which provides the interface to display the graphic user interface of the game panel. The reason for the absence of a **MenuUI** class is because the structure of the UI of **Menu** is rather simple and the functionalities of the **Menu** class is straightforward. On the contrary, the **Game** class became more and more difficult to understand and maintain as new features were added, such as the score panel and undo cools down control. Therefore, separating the set of drawing panel functions to another is necessary.

Operations

- updateGameUI(int time, int score, Status status): void - Repaint the panels based on the information provided by the parameters. Although **GameUI** can access some of the attributes and functions in the **Game** class, currentStatus in **Game** is kept private because in the design, **GameUI** only displays the given status and does not have the right to choose which status stored in **Game** to display. The **Game** controls what and when to update the game UI.

- executeCommand(Command command):void - Invoke the execute interface of a given command if it is executable. This method is used in event listeners that are added to certain buttons, such as UNDO, END and MENU buttons.

Association

- **Background** shared aggregation - Similar to the relationship between **Background** and **Menu**. **Background** is used as the background panel in the **Game**. It's a shared aggregation because both **Menu** and **GameUI** can create **Background**.
- **GameGraphicStyle** dependency - Similar to the relationship between **MenuGraphicStyle** and **Menu**. It provides functions for the **GameUI** to use to decorate the graphic components such as buttons, labels and panels inside the **GameUI**.

GameLogic

Deals with status updating and rewards allocation. It takes status in from **Game** and returns a modified copy of status back to **Game**. In assignment 3, the **GameLogic** first updates the puzzle part in the given status by calling the interface updatePuzzle() provided by **PuzzleHandler** and then reset its own attribute timeLimit and score in **Status** based on the number of Succeeded and Failed **Daemons** which is also given by **PuzzleHandler**.
GameLogic imports the highest score when a game is initiated and exports the highest score when the game finishes through the **ScoreHandler**.

The reason to add this class instead of putting everything in the **Game** class is to improve the maintainability of the system and reduce the complexity of the **Game**.

Attributes

- gameOver: boolean - The state which indicates if the game is over. The default value is false when a new *gameLogic* is generated inside the **Game**. When timeLimit counts to zero, gameOver shall be set to true.
- timeLimit: int - The time counter displayed on the Game's time panel. The initial figure is obtained from gameDifficulty. timeLimit is stored in **GameLogic** because it changes not only with the real time passing by but also increases or decreases with the reward and punishment according to the completion of **Daemon**.

Operations

- updateStatus(Status status, Coordinate clickedCellPosition):void / updateStatus(Status currentStatus):void - The main interface function called by **Game** to update status. Overloading is implemented here, because this interface can be used for different purposes. The method updateStatus() with two parameters applies the regular functionalities which update the puzzle, time and score. The method updateStatus() with only status parameter is called when undo is used. Since the currentStatus is reversed in **Game**, the status in **GameLogic** needs to be reversed as well. By implementing the overloading updateStatus() method, status in **GameLogic** is replaced by the reversed currentStatus in **Game**.

- isGameOver(): boolean / setGameOver():void : Returns the boolean value of gameOver. / Set gameOver to true.
- updateTimeLimit(int offset): void - Called in updateReward(). This operation can modify the value of timeLimit.
- setTimeLimitZero():void - Set timeLimit to zero. This operation is used to clear the timeLimit when the player presses the "END" button to finish a game.

Association

- **PuzzleHandler** composition - **PuzzleHandler** is instantiated only in **GameLogic** and is used to help **GameLogic** to update the current puzzle in current status. If **GameLogic** is destroyed, **PuzzleHandler** no longer exists.
- **ScoreHandler** dependency - This is a dependency relationship because **GameLogic** only uses the utility functions provided by **ScoreHandler** to import the saved highest score from a txt file and export the current highest score to the txt file.

PuzzleHandler

The facade class which helps **GameLogic** to avoid from accessing entities in **Puzzle** directly. It updates **CodeMatrix**, **Buffer** and **Daemons** sequentially based on the coordinate of the clicked MatrixCell and counts the number of Succeeded and Failed **Daemons** for **GameLogic** to allocate rewards.

ScoreHandler

The ScoreHandler class is used to read the highest score from the save file or write the highest score to the save file. The score will be encoded with Base64 encoder when writing. Conversely, the score will be parsed with Base64 decoder when reading.

Package: command

A set of command objects which are used to store functions in **Game** and can be executed in **GameUI**.
The reason that this pattern is only implemented for **Game** and not for the **Menu** is because there is no separate UI class similar to **GameUI** for the **Menu**, so this intermediate carrier is not needed.

The interface executable() ensures the functions of the buttons only work if the condition allows. For example, the UNDO button only undos when the cool down is finished and the Deque is not empty, the MENU button only switches the game panel to the menu panel if the game is over, in this way, the highest score is guaranteed to be saved.

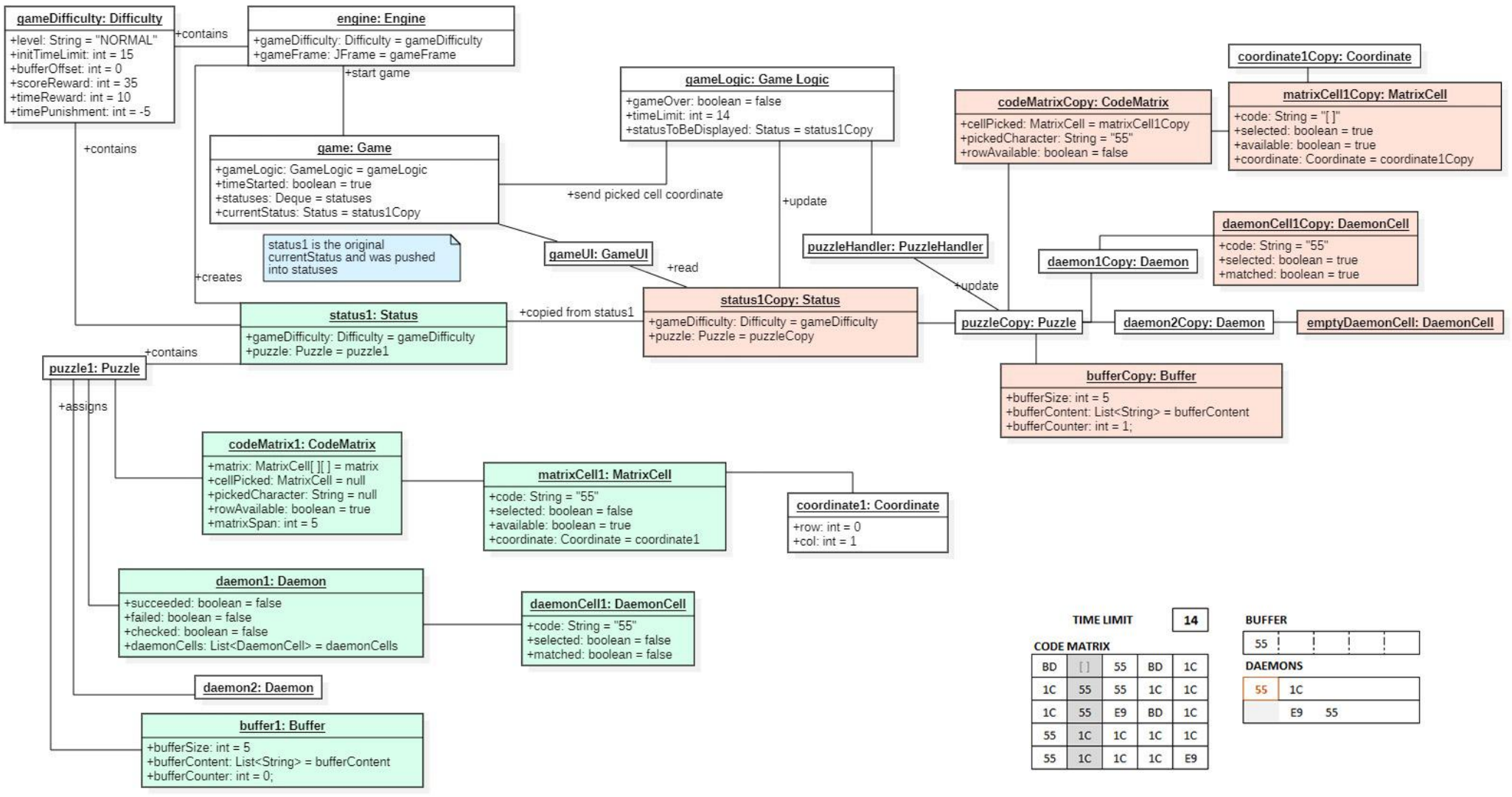
Package: graphics

Classes in graphics are colored in pink in this diagram. Only the associations of these classes are elaborated because these classes only serve the graphic styles of the system.

Object diagram

Author(s): Yu Chen

[Figure 3] below illustrates an object diagram model of the system.



[Figure 3 - Object Diagram (click here to open original picture)]

Differences in Assignment 3:

- The **Status** now does not have direct access to the entities of **Puzzle** like what it does in Assignment 2. The *puzzle* is stored in **Status** as a whole.
- *gameUI* object is added to connect between the *game* and *status1Copy* objects.
- *puzzleHandler* is added to help *gameLogic* to update *puzzleCopy*.

This is a “snapshot” of the scenario when a new “NORMAL” difficulty game is started with the first puzzle being puzzle no. 5 in puzzles txt files and the player clicked the *matrixCellButton* located on the first row and second column. The object diagram shows how the objects change after the first action taken by the player right after a new game started. A toy example of the graphic user interface on the left bottom corner visualizes the puzzle entities and is used to help better understand how the objects are altered.

When a *game’s* panel is generated, the **Menu** object in the *engine* is destroyed. The initial attribute timeStarted is false in game, the Deque statuses is empty and the currentStatus equals the *status1* which is created in the *engine*. After the player clicks a *matrixCellButton* in *codeMatrix1*, *status1* is pushed into Deque *statuses* and is copied and passed to *gameLogic*. Then *gameLogic* updates the *status1Copy* according to the *coordinate* of the clicked *matrixCell*. Objects colored with green indicate the original objects in *currentStatus* and objects colored with pink refer to the modified objects.

- *status1* vs. *status1Copy* - At this stage, the puzzle is not switched to a totally different one in *status1Copy* and the score is the same because none of the Daemons is completed or failed.

- *codeMatrix1* vs. *codeMatrixCopy* - Attribute cellPicked changed from null to the **MatrixCell** with coordinate(row: 0, col: 1) [*index in Coordinate row and column start from zero] which is the copy of *matrixCell1*. PickedCharacter changed from null to String value of “55”, which is the code of *matrixCell1*. Attribute rowAvailable switches from true to false.

- *matrixCell1* vs. *matrixCell1Copy* - Attribute code of the *matrixCell1* changed from “55” to “[ ]”. Its boolean state selected is switched from false to true, which means the cell is marked as selected. The available attribute is not changed because it sits on the second column of the code matrix which contains a list of available matrix cells. Notice that all matrixCells in this column have switched their available state to true and this is not indicated in the object diagram. And all matrixCells at the first row switched available to false except the picked one.

- *buffer1* vs. *bufferCopy* - The bufferSize did not change after this update. bufferCounter incremented by 1 and pickedCharacter which values “55” is added in the bufferContent.

- *daemonCell1* vs. *daemonCell1Copy* - This cell refers to the first cell in the first Daemon. It’s code value “55” matches the player picked matrixCell and the code in the bufferContent, so the selected state switched from false to true and also the added state.

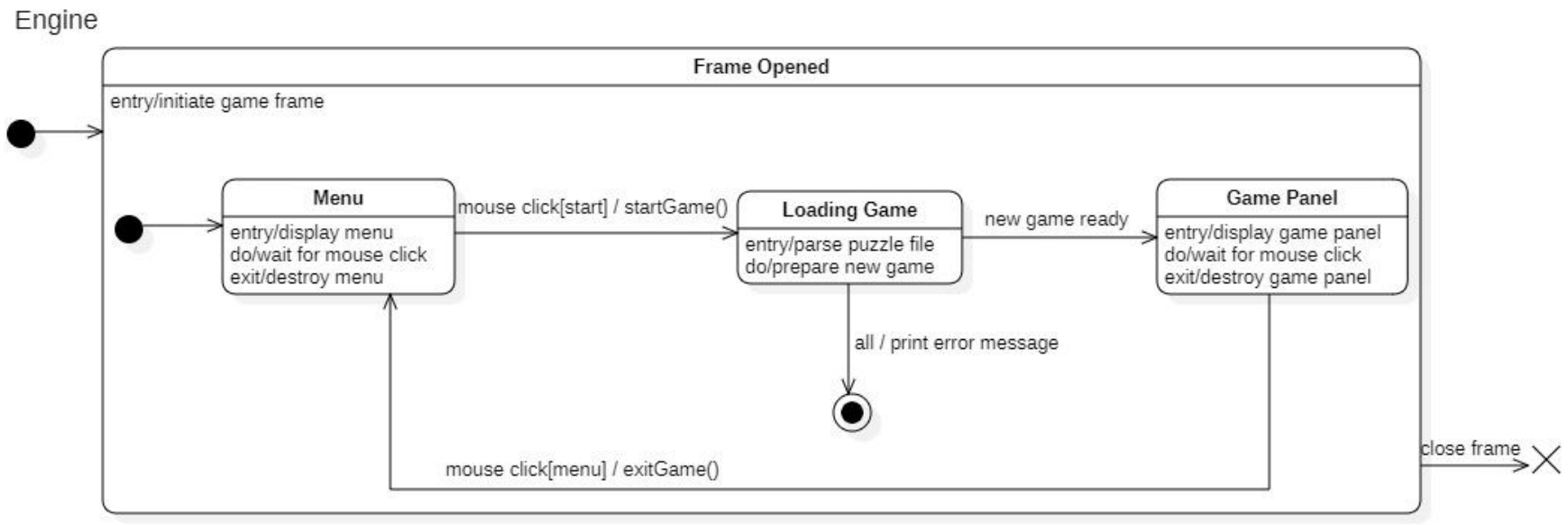
- *daemon1* vs. *daemon1Copy / daemon2* vs. *daemon2Copy* - The attributes did not change in these two daemon because none of them is marked as SUCCEEDED or FAILED. And since the cell in daemon1 matches buffer content, no alignment is needed in this daemon.

- *daemon2* vs. *daemon2Copy* - An emptyDaemonCell is added at the front of this daemon which is to align the first daemon cell which added state is false to the first empty position in the buffer.

The initial timeLimit of a “NORMAL” difficulty game is 15. This object diagram depicts the situation when the player made the first choice and one second passed by. So the time started and the timeLimit changed to 14 in *gameLogic*.

State machine diagrams

Author(s): Yu Chen, Yudong Fan, Xiaojun Ling, Berry Chen

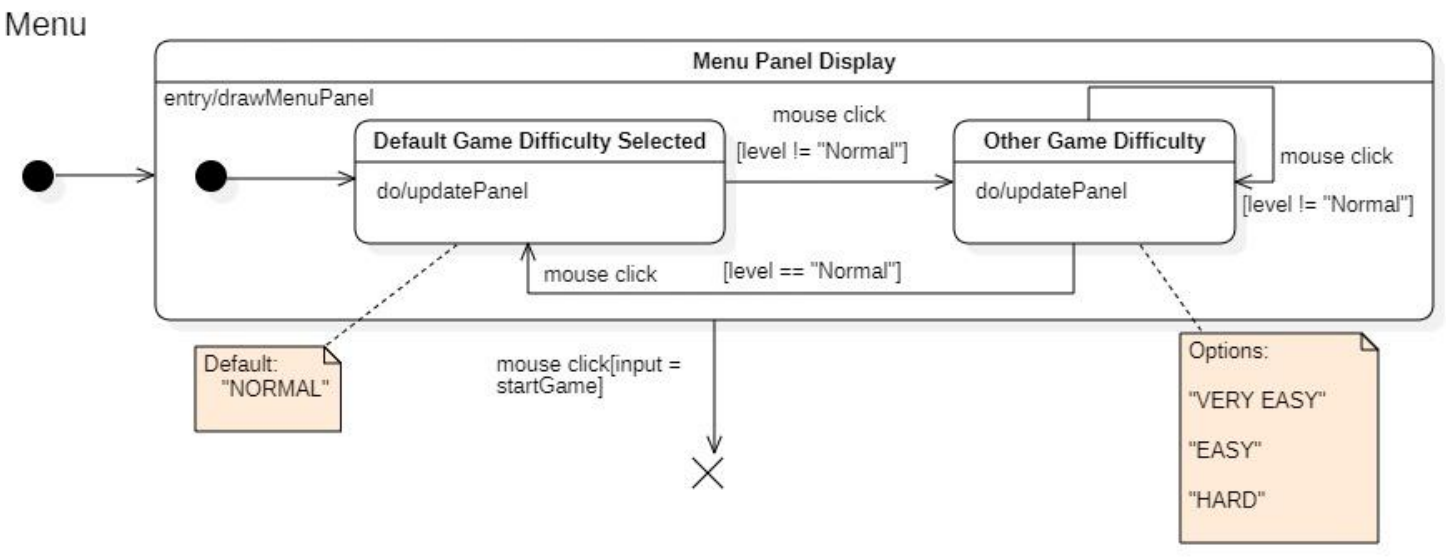


[Figure 4 - State Diagram: Engine (click here to open original picture)]

[Figure 4] above demonstrates the state diagram of the **Engine**. The **Engine** plays an important role in the entire system. It is an initializer of the game frame(GUI) which is the basement of this implementation, and it will prepare all the initialization steps using their default values for delivering a menu page and a game page. Moreover, it handles the interaction between the menu page and the game page.

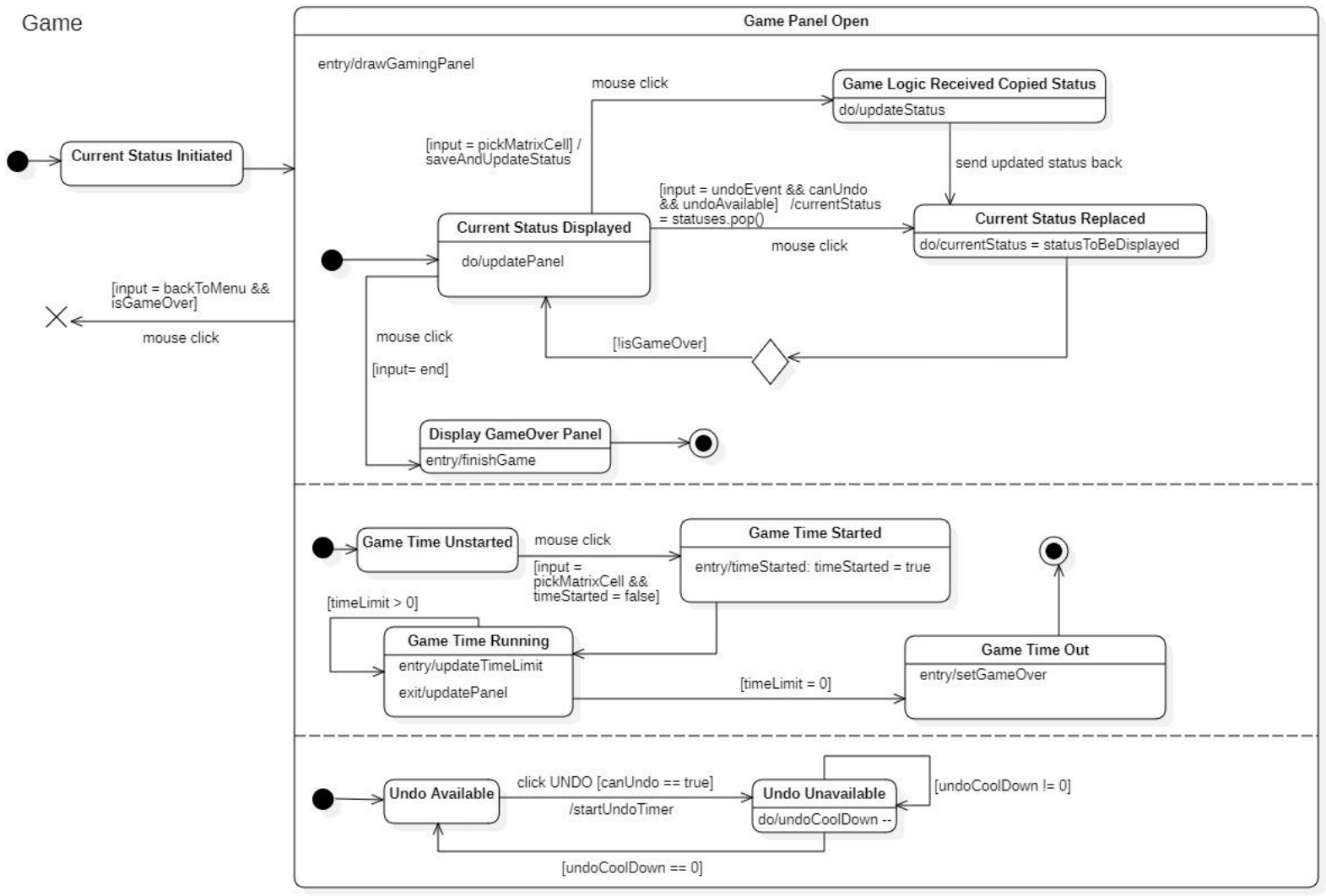
First of all, run the game, and **Engine** will be called. The *Frame Opened* state is entered. It is a composite state and supposed to be the out-most state which encloses all other states. This is because everything else in the system will be performed or delivered based on the game frame. Also, closing the frame at any point during the game will terminate the entire system regardless of any other states.

After the initialization of the game frame is done, *Menu* state will be entered. It will wait for specific user input for further actions. If a mouse click on the *start* button is listened, the function *startGame()* will be called and the *Loading Game* state is entered. If any error occurs in this state, an exception will be caught, and an error message will be delivered to the user. Then the process will stock at this point and be marked as finished. In this case, closing the frame to terminate the system is the only way out because we do not have reloading or regenerating features after the system is freezed. If the game is loaded successfully, the *Game Panel* state is entered. Similar to the *Menu* state, it will just wait for specific user input for further actions. Yet the initialization of the game is basically done. If a mouse click on the *menu* button is listened, the function *exitGame()* will be called and the *Menu* state is entered again.



[Figure 5] on the left draws the state diagram of the **Menu**. The menu panel is displayed when **Menu** is created and terminated if *startGame* event is triggered (“START” button be clicked). Inside the panel displayed state, the gameDifficulty state can change between default level which is “NORMAL” to other levels.

[Figure 5 - State Diagram: Menu (click here to open original picture)]



[Figure 6 - State Diagram: Game (click here to open original picture)]

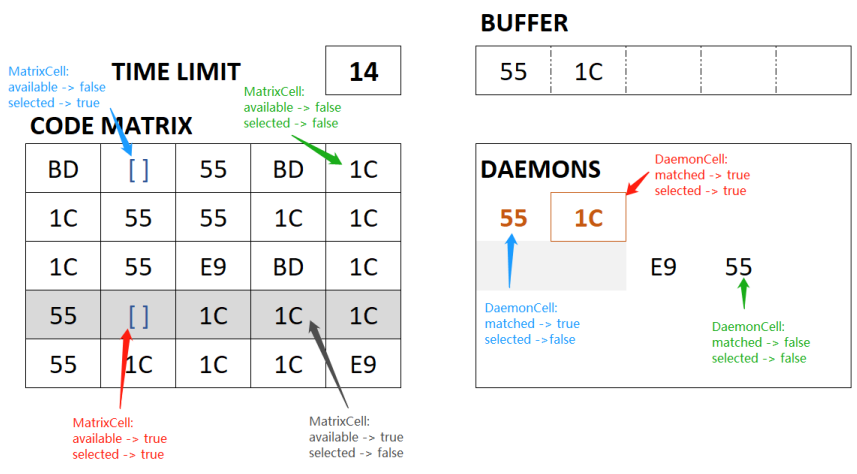
[Figure 6] above is the state diagram of the **Game** class. The **Game** class controls the status to be displayed to the user, decides when to start and when to stop one round of game and controls the cool down system of the undo feature. Once the initial status (*currentStatus*) is loaded, the game panel is opened and the information in the *currentStatus* is passed to GameUI which displays on the graphic user interface. This is done by the updateGameUI() operation. The *currentStatus* is now displayed, the *Game Timer* is not created yet and the undo button is available to be clicked. These three concurrent substates are therefore waiting to be triggered by the player's mouse click on a matrix cell button or the undo button.

The attribute timeStarted is set to be true and the *Timer* starts to run after the first click on the matrix cell. The timeLimit in **GameLogic** counts down until it goes to zero. The state changes from Time Running to Time Out. The gameOver attribute is set to true and the state turns to Game Over. All unchecked Daemons are marked as FAILED and the game over panel is displayed. This is also the finish state of *Timer* in the **Game**, the *Timer* shall not start again until a new **Game** is created in the **Engine**.

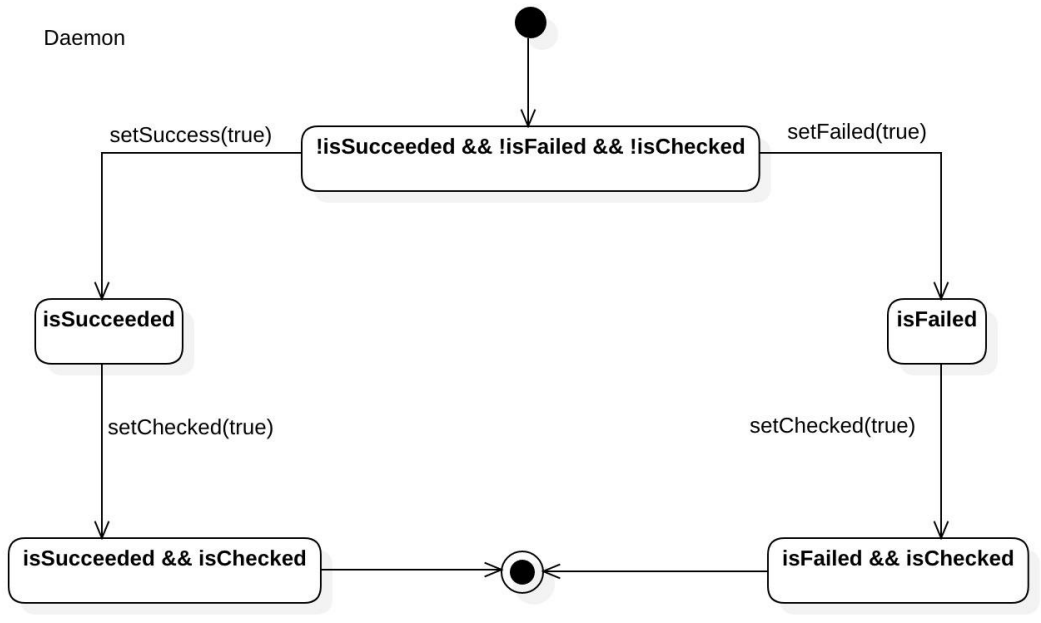
On the other side, the *currentStatus* is pushed in a Deque statuses and when a matrix cell is clicked. The copied *currentStatus* is sent to **GameLogic** to be updated. The updated version is then sent back to the **Game** and replaces the *currentStatus*. The replaced current status shall be displayed if the game is not over, otherwise the game over panel is displayed and the highest score is saved. (Score not implemented)

Notice that the Undo Available state only switches to unavailable if canUndo() returns true. This guarantees the chance to undo shall not be wasted if undo is not executable (e.g. the Deque of status is empty).

State of Entities



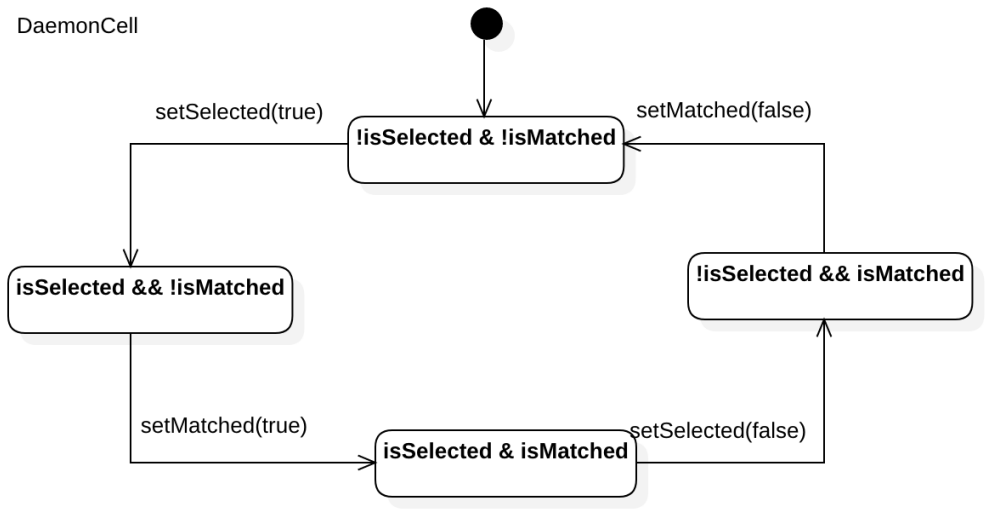
[Figure 7 - The performance of Entity States on Graphic User Interface (click to open original picture)]



[Figure 8 - State Diagram: Daemon (click here to open original picture)]

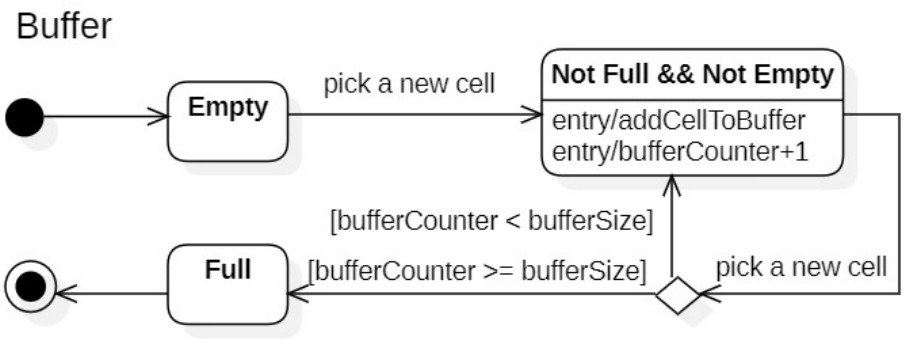
[Figure 7] on the above left provides an example that illustrates how the states of **Puzzle** entities affect their performance on the graphic user interface.

[Figure 8] is the state diagram of the **Daemon**, which indicates the several states of a daemon and their relationship. A Daemon is not marked Succeed, Fail, and Checked at the beginning. It will be marked with either SUCCESS or FAIL as the game runs and the conditions are met. Then the Checked state is set for each daemon when the PuzzleHandler adds the SUCCESS / FAIL daemon into its counts. Finally towards the final state and the daemon is completely marked.



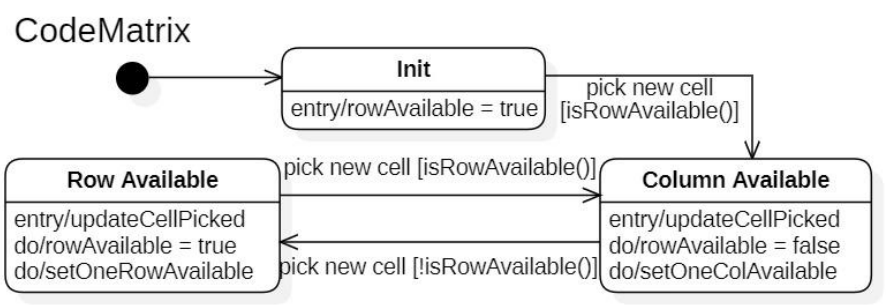
[Figure 9] is the state diagram of **DaemonCell**. It shows the states of a DaemonCell and the relation between states. At first, the daemon cell is not selected and not matched. The daemon cell becomes selected if there is a match between the user-picked character and the daemon cell. Then the daemon cell is marked as matched if the cell shares the same code with the buffer's code at the same index. The state of the daemon cell could also be unselected and then unmatched when there is not a match after the user picks the next character.

[Figure 9 - State Diagram : DaemonCell (click here to open original picture)]



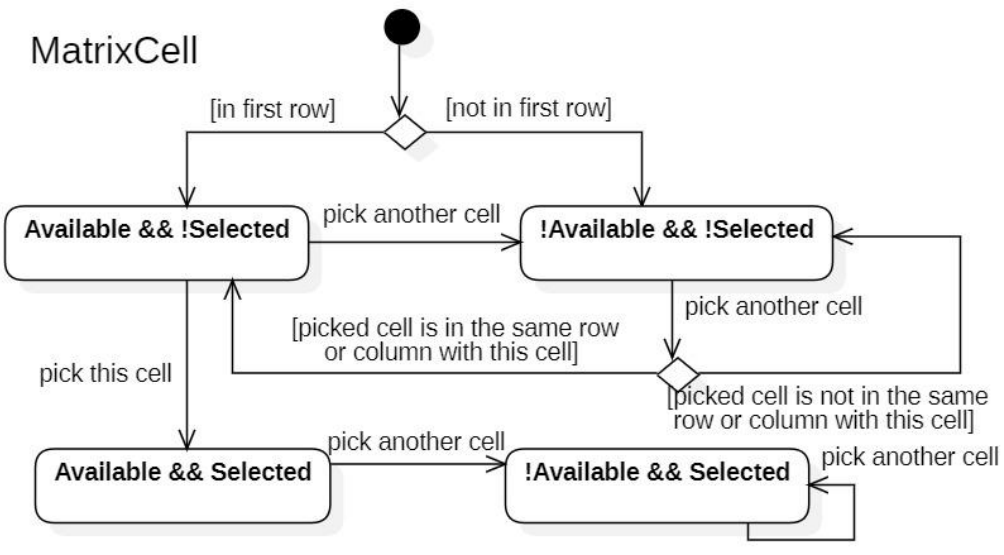
[Figure 10] on the left has modeled the possible states and transitions of class **Buffer**. Initially, the buffer is empty with no characters inside. Once the player has picked a matrix cell, the event *pick a new cell* triggers the transition from state Empty to state Not Full && Not Empty, and the buffer becomes neither full nor empty. Moreover, the two entry activities should be completed when the state is active. The activity addCellToBuffer adds the character on the picked matrix cell to the buffer, and the bufferCounter increases to point to the next free space in the buffer. After the next transition is triggered, a decision point decides whether the buffer becomes full to achieve the final state or enter the previous state and execute the activities again by determining whether bufferCounter is pointing outside the buffer.

[Figure 10 - State Diagram : Buffer (click here to open original picture)]



[Figure 11] on the left shows the transition relation between states of the whole code matrix and the behaviors exhibited in each state. Once an instance of class **CodeMatrix** is created, the first state it enters is the state init, which sets the default value of attribute rowAvailable as true indicating the first row of code matrix is originally available for the user to click. One thing needs to be noted is that the activity of setting the first row available is done in class **Puzzel** to initialize the code matrix. If the player picks a new matrix cell and currently a row is available in the matrix, the transition from state Init to state Column Available is triggered, the matrix becomes having a column of matrix cells available. The entry activity updateCellPicked is executed to update the status of the matrix cell just picked. The two do activities, altering rowAvailable to indicate current state and setting a column of matrix cells available are executed while the state is active. After that it is a back and forth between state Row Available and state Column Available in sequential order.

[Figure 11 - State Diagram: CodeMatrix (click here to open original picture)]



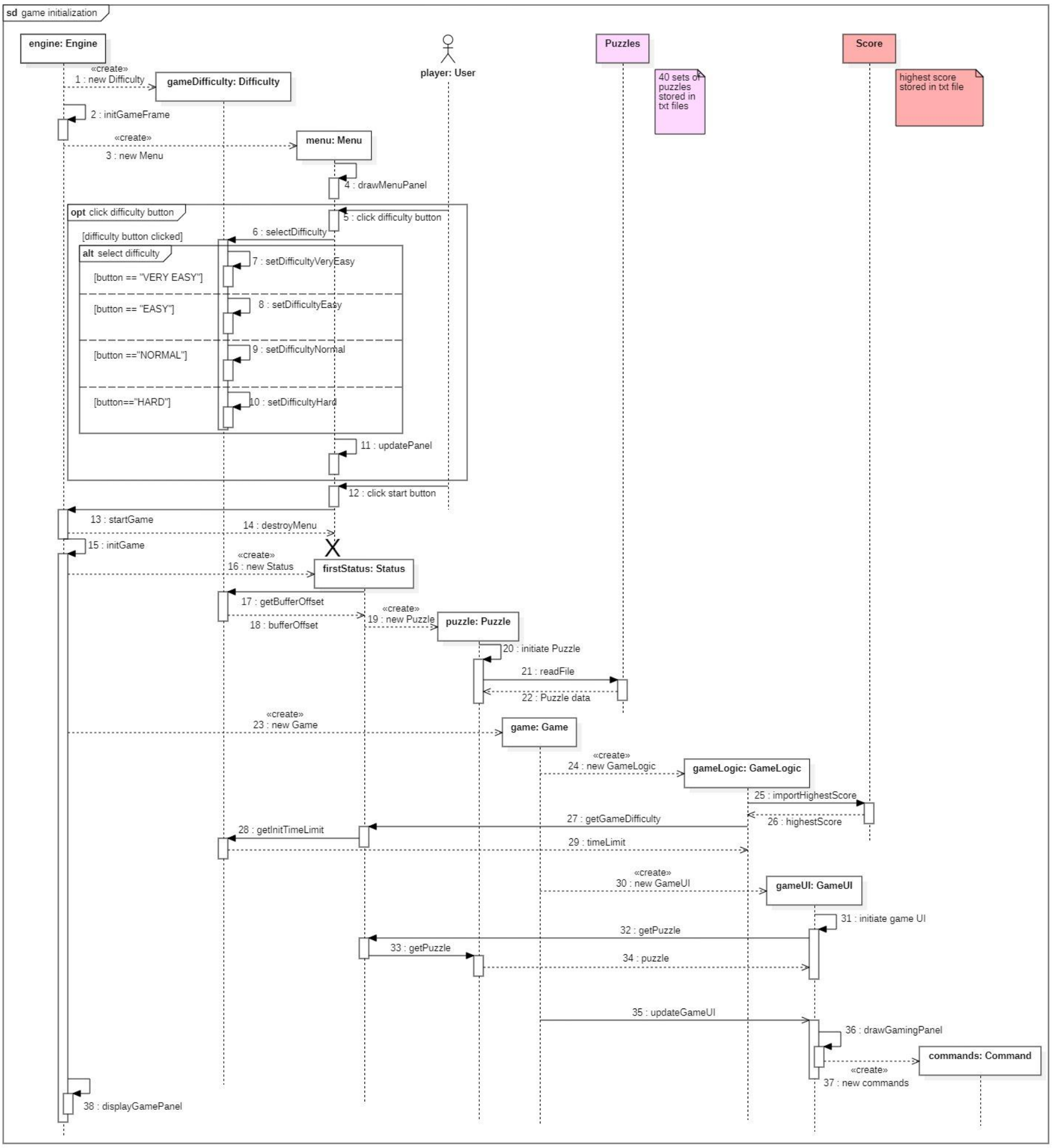
[Figure 12] on the left illustrates the possible states and transition relations of one matrix cell. The states of Available and !Available indicate whether the matrix cell is clickable, and the states of Selected and !Selected show if the matrix cell can be selected by the player. A matrix cell can initially be in the first row or not, which leads to two different states, Available && !Selected and !Available && !Selected. If the cell currently in focus is in the state of Available and is picked by the player, the transition from state !Selected to Selected is enabled. If a matrix cell other than the cell currently in focus is picked, the transitions between state Available and !Available and the self-transition of state !Available are possible. The reason is that the event *pick another cell* causes the changes of the shape of the available area in the matrix, since the currently focused cell is not picked, the changes only apply to the state related to availability.

[Figure 12 - State Diagram: MatrixCell (click here to open original picture)]

Sequence diagrams

Author(s): Yudong Fan, Xiaojun Ling, Berry Chen

Sequence Diagram - Game Initialization:



[Figure 13 - Sequence Diagram: Game Initialization (click to open original picture)]

[Figure 13] The diagram above exhibits all initialization steps of a new game when the system starts. Once the game is started, an instance of the **Engine** will be created. It is a starter of the system, and it should always take place first when the game starts to run. Everything else in the system will be delivered or invoked directly or indirectly later based on the instance of the **Engine**. Right after that, an instance of the **Difficulty** will be created. This instance itself, as an independent component to the system, contains game parameters under different difficulty settings and has a default value ‘NORMAL’ when it is newly created. Then the process will keep going and initialize the game frame(GUI). After the game frame is loaded, an instance of **Menu** will be created, it contains the ports for setting difficulty and starting the game. The default menu will be drawn using the default value of **Difficulty** which is ‘NORMAL’. Until now, a menu page with default difficulty setting is displayed on the game frame.

The first option emerges at this point. Users can choose different difficulties by clicking on the corresponding difficulty button. If a mouse clicks on any of the difficulty buttons, the system knows the difficulty setting is changed, and the instance of **Difficulty** will set its new value accordingly.Then the menu will be updated to show a new menu page with the new difficulty setting. Conversely, if there is no action for changing difficulty, nothing is going to happen, which means the difficulty will remain to be the default value. In both cases, difficulty is determined and users can click on the start button to start the game now. The start game request will be sent to the **Engine** instance. Then the **Engine** instance will destroy the **Menu** instance and start to initialize a new game using the determined difficulty.

An instance of the **Status** which contains parameters for the game will be created first. It will get the corresponding buffer offset from the **Difficulty** instance. Then it will create an instance of **Puzzle** which extracts puzzle data from the puzzle source file. Details in this process are omitted because they are rather unimportant and have been elaborated already in the class diagram. After that, an instance of the **Game** is created. Then the **Game** instance creates an instance of the **GameLogic** right away. After the **GameLogic** instance gets the highestScore and the timeLimit from the score file and the **Status** instance respectively, the **Game** instance will create an instance of the **GameUI**. The **GameUI** instance will interact with the **graphics** and the **Status** instance to initiate game UI.

In addition, the **graphics** is omitted here because it is only a matter of GUI styling and unimportant in the sequence diagram. Syncronys messages are needed in those interactions because the initialization of the game UI has to be done before the **Game** instance calling updateGameUI. One thing needs to be mentioned here is that, even though the **GameUI** instance has access to the **Game** instance, the updateGameUI still needs to be called from the **Game** instance rather than the **GameUI** instance because important attributes such as currentStatus are made private in the **Game** instance. The reason for doing that is to avoid access to important game states even when a class can access the **Game** instance. The **GameUI** will always need the **Game** instance to pass those private attributes as parameters to it when updating the game UI. This design ensures that the important game states remain safe after we separate out some functional methods to implement a certain design pattern. After updateGameUI is called, the **GameUI** instance will call the drawGamingPanel. The drawGamingPanel is going to invoke relevant command classes to create new instances of different command classes. Here we use the **Command** to represent all of those command classes in order to avoid redundancy of the diagram. In the diagram this process is simplified and partially omitted because the purpose of having the **Command** involved is only to demonstrate when and how the interaction between the **GameUI** and the **Command** is taking place. The actual processes of those interactions are rather trivial and repetitive. Yet the preparation of the initialization should be done. Then the **Engine** instance will display the loaded game panel to the user. It is time to start your game!

Sequence Diagram - Player clicks on a Code Matrix Cell:

[Figure 14] The diagram below depicts the game running process that occurs when a player clicks on a cell in the matrix.

In general, when the player clicks a cell, *game:Game* forwards the current status and the coordinate of the user-clicked cell to the *gameLogic:GameLogic*, and then *gameLogic:GameLogic* pass the puzzle of the current status to the *puzzleHandler:PuzzleHandler*. After the *puzzleHandler:PuzzleHandler* updates the puzzle, *gameLogic:GameLogic* updates the time and score according to the count of SUCCEEDED and FAILED from the *puzzleHandler:PuzzleHandler*. Then, status returns back to the *game:Game* and *gameUI:GameUI* will handle the display of new time and score. The description below provides a detailed explanation of this overview.

When the player clicks on a cell from the matrix, a mouse event is captured in the unique instance of the **GameUI** class, *gameUI: GameUI*. In this case, the execution in class **ClickCellCommand** is triggered. Firstly, *game: Game* calls its function triggerGameTimer() to start the countdown of the game. After the handling of the game time, operation saveAndUpdateStatus is called, in which the attribute statuses of *game: Game* stores the current status by calling the function statuses.push(currentStatus) in case there is a backtracking of game progress later, and *newStatus* is created as a deep copy of current status and the new changes are updated into it.

If the *newStatus* is not null, it will be used as a parameter to the function updateStatus in *gameLogic: GameLogic* along with the coordinates of the clicked cell, and the statuses of the matrix, the buffer, daemons and the reward are all updated in this function. A noteworthy point is that in class **GameLogic**, statusToBeDisplayed: Status is created as an attribute to indicate the status that is being updated and has not yet been displayed, which is identical to *newStatus*.
Inside the function updatePuzzel, the function updateCodeMatrix is called with the position of the clicked cell as parameter to update the whole matrix. When function updateCodeMatrix is called, *puzzleHandler: PuzzleHandler* firstly gets *codeMatrix: CodeMatrix* from *puzzle: Puzzle*, then it can access *codeMatrix: CodeMatrix* to call function updateCellPicked to set the matrix cell corresponding to coordinate parameter to picked. After the execution of updateCellPicked, all cells should be marked as unavailable as a preparatory step for updating the available area by calling the function diableAllCells. Furthermore, if the function isRowAvailable in CodeMatrix returns true meaning a row of matrix cells should be clickable, then function setOneRowAvailable is called in **CodeMatrix**, and the same story in the case of false. After this, *buffer: Buffer* is accessed through *puzzle: Puzzle* to decide whether the buffer is full, if it is the function disableAllCells should be called again meaning no more matrix cells can be picked.

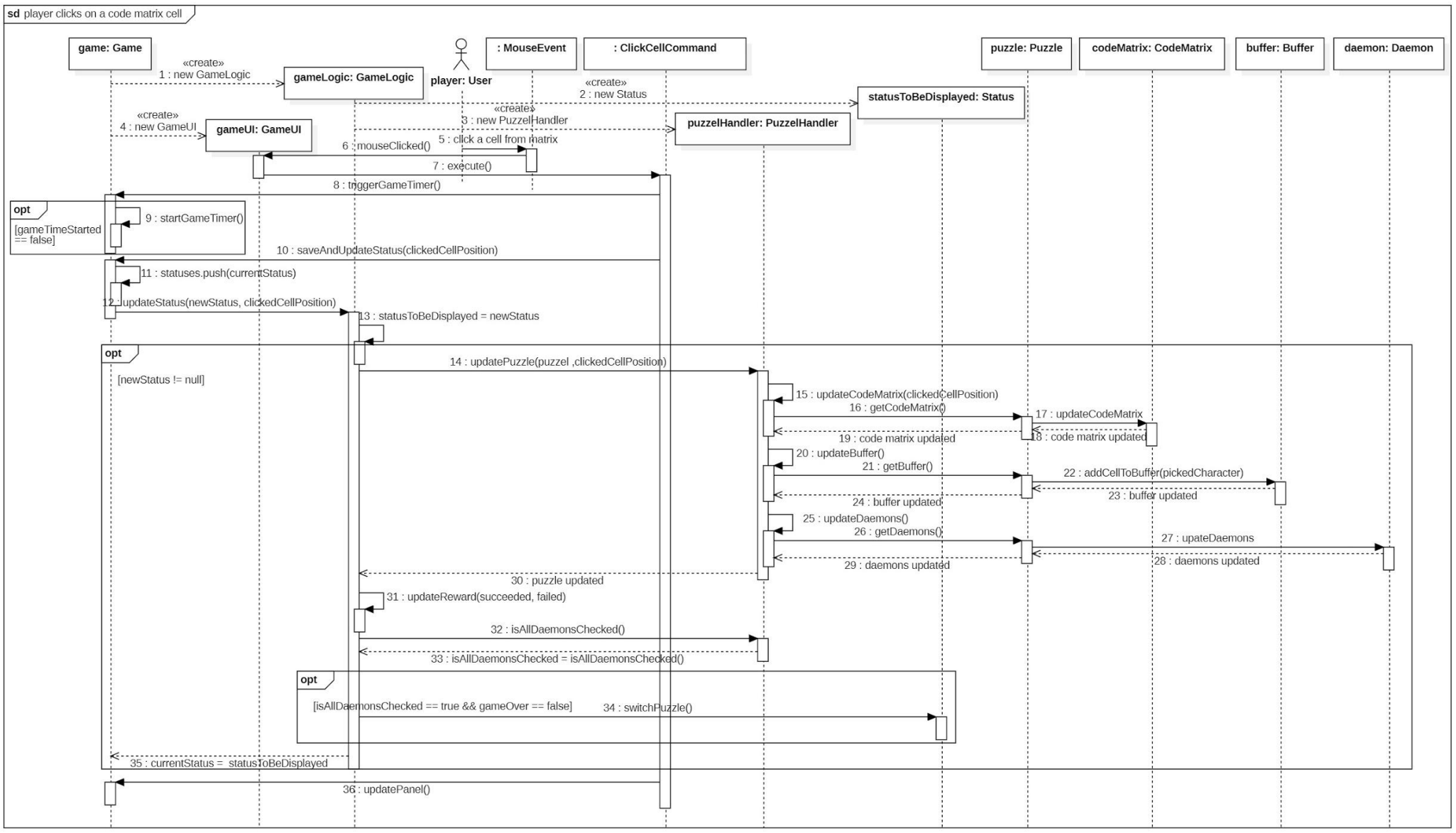
The function updateBuffer is called afterwards to update the status of the buffer. Initially, whether the buffer is full is determined. If the buffer is not yet full, *puzzle: Puzzle* is accessed to get the character on the picked matrix cell by calling the function getPickedCharacter in *codeMatrix: CodeMatrix*, after which the function addCellToBuffer is called in *codeMatrix: CodeMatrix* with that character as parameter to add the character to the buffer.

Then, the function updateDaemons is called, and a for loop is used to traverse all daemons in the game and update the status of them. If it is not checked for each daemon, which indicates this daemon is not completely marked, updateDaemonCells will check and update the cell's status based on the bufferCounter.

In the function updateDaemonCells, all cells are set to an unselected state at first. The cellWaitingToBeCheck is obtained from getDaemonCell in *daemon:Daemon*, which is the daemon cell at the buffercount -1 index position in the daemon. Then updateStateOfMatchedDaemonCell is called, and the daemon cell is matched with the user-picked character to setSelected and match with the last code in the buffer to setMatched. After the cell is checked and marked whether it is matched. If the cell is not marked matched, setAllDaemonCellUnMatched and alignDaemonCellWithBuffer are called. The align function aligns the daemon cell to the first empty space in the buffer, making the comparison of the next daemon cell easier. An addEmptyCell function is also called inside the alignment function, which is used to add an empty cell with an empty string to make the visualization more consistent and easier to understand. After that, similar steps of cellWaitingToBeCheck and updateStateOfMatchedDaemonCell are called again to check if there is a match of the daemon cell after alignment.

After updateDaemonCells, islastCellMatched is called to check whether the daemon meets the condition of SUCCESS. If the daemon's last cell is marked as matched, then the daemon is considered SUCCESS and the setDaemonSucceeded function is called. Otherwise, if isDaemonExceedsBuffer is true, which shows there is not enough space left in the buffer for comparing the rest of the daemon, the daemon is considered FAIL and the setDaemonFailed function is called.

The function updateRewards is called with two parameters, the number of SUCCESSED daemons and the number of FAILED daemons. Inside the function, a for loop is used to reward time and score based on the number of succeeded, then another for loop is used to punish time based on the number of failed. The reward time and score function are different based on the difficulty. Then, if all Daemons are checked, and the game is not over, a new puzzle is switched by calling switchPuzzle from *statusToBeDisplayed: Status*, and the new status becomes current status. Lastly, updatePanel is called from *game: Game*.



[Figure 14 - Sequence Diagram: Game Logic implementation (click to open original picture)]

Implementation

Author(s): Yu Chen, Yudong Fan

From UML to JAVA

(This part is the same as it in Assignment 2)

Before the class diagram is modelled, we first drew a very simple flowchart to list the key jobs of the initiating process of a basic game(without menu and difficulty level). For example, a game window should be created first and then a puzzle is loaded. And then a highly descriptive class diagram is created at this stage. In this diagram, relationships between essential classes such as **Engine**, **GamePanel**(which later renamed to **Game**), **GameLogic**, **CodeMatrix**, **Buffer** and **Daemon** are determined. Since our knowledge and experiences with Java and Swing(the Library we chose to implement user interface) is quite limited at that time and the characteristics of the game make it highly dependent on the user interface, we started carry some experiments on the graphic user interface(GUI to explore the features of Swing components such as those event listeners and to make sure the GUI can work as our expected. When this is done, we have implemented a game window that displays a 6*6 grid matrix and can catch mouse click events to add the clicked code character into a text box(the prototype of **CodeMatrix** and **Buffer** panel).

After we were confident about the features of key Swing utilities(e.g. **ActionListener**, **JButton**, etc.) and had an idea about how and where to use them, a more prescriptive version of class diagram was created. Class **Status** joined in and the hierarchy of Entities from **Puzzle** to **Cell** is finalized. To determine the attributes in **Puzzle** and its “children” (e.g. **MatrixCell** and **DaemonCell**), state diagrams of Entities are modelled. This state diagram needs to be finished at an early stage because states of **Cell**s(e.g. selected, matched) are highly related to the GUI and determine the technique we can implement the basic rules of the game (e.g. only one row/column of matrix cell can be selected).

We think the most successful point of our design is maintainability. When starting to implement the model in Java code, one person takes charge of building the frame of the game and applies a GUI without any styles. We tried to keep the classes deep and only leave one or two APIs for other classes to use such as load **Puzzle** from **Status** in **Game** and update **Status** in **GameLogic**. In this implementation, the GUI can already display an example code matrix (hard coded in **Puzzle**), add the clicked matrix cell code into the buffer’s first grid and shows the graphic effect of the matched and selected of a fixed daemon cell(hard coded in **GameLogic** without any logic behind). These hard-coded examples are used to check the implementation of GUI and help other group members to better understand the code. Then reading files and parsing them to **Puzzle** is done. The group members who did this implementation only need to work in **Puzzle** class and this also applies to the two members who implemented the basic rules of the game(they only need to write code in **GameLogic**). In this way, group members can work individually without conflict and their own implementation can be visually checked easily.

Because the size of **Buffer** is designed to be dynamic, after the basic version of the game is done with the implementation, we add the **Difficulty** feature without taking much effort.

Key Solutions

- **Difficulty:** Allowing the player to choose different difficulty levels means implementations of related parts needs to be dynamic. Our solution is to encapsulate a **Difficulty** object which is determined at run-time (set by user clicking on the different difficulty buttons) and pass it between clients which need the information of different levels to set up their initialization (e.g.buffer initialization) or dynamic allocation (reward system). A singleton pattern is added to ensure the uniqueness of the **Difficulty** object.

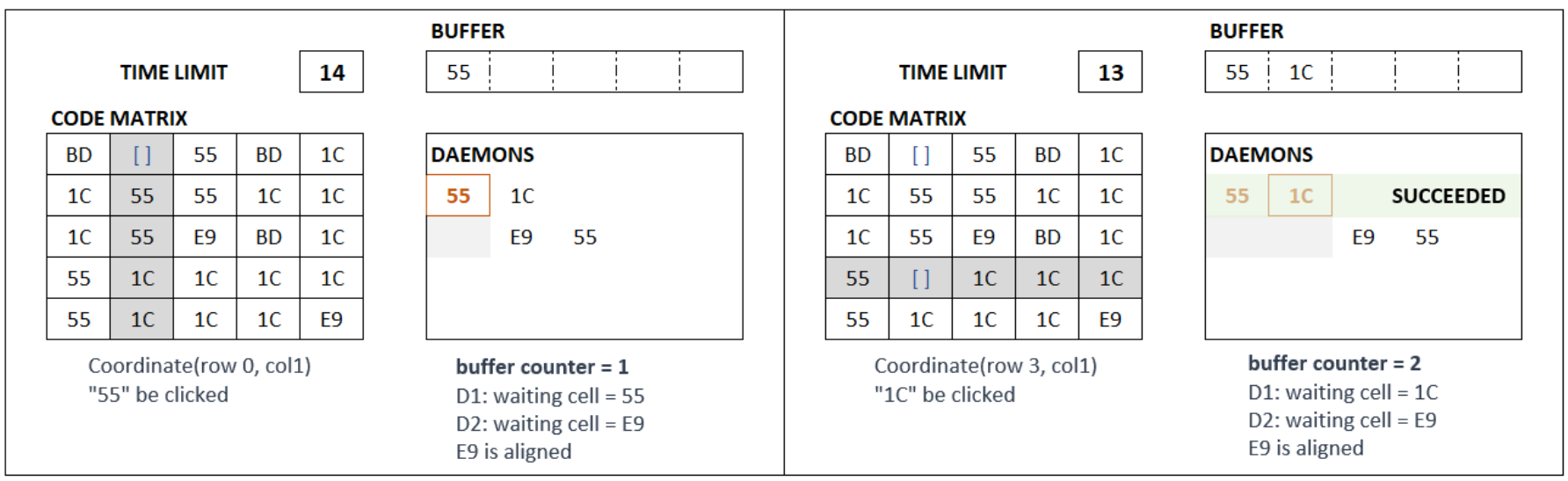
- **Command**: The most difficult part of this feature is how to fetch the code (character) of a code matrix cell the player clicked on and display the changes on **Puzzle** entities(e.g. code added in the buffer) after this click. We get the intuition from calculator applications which also have a bunch of strings and integers in a grid matrix and add the selected content on the display panel successively. After studying how these types of GUI works we decided to use a **JButton** for each code matrix cell and add a Mouse Click event which is all utilities from Swing on it. In our first version of implementation, **JButton** function getText() is used to get the code of a matrix cell and add it to the buffer. We changed this method later because more information is needed from a matrix cell rather than just the code and we want to avoid using GUI functions to get key information to make the implementation more maintainable. The second version is introduced in the **Basic Play Rules** solution.

  To display the changes, once the player makes a click on the matrix, the background panel will remove everything on the panel and repaint all the updated subpanels such as the code matrix panel and buffer panel. This technique is similar to the idea of changing frames in most of the games which depend on time such as *Snake* and *Tetris*.

- **Puzzle** : The **Puzzle** is an independent component to the system which is capable of reading the puzzle source files, parsing the readed puzzle data, and saving the parsed puzzle data for other members to use. It has one inner class named **Parse**, and it handles the reading and parsing of the puzzle data. Reading puzzle source files is done by using **java.util.Scanner** to create a Scanner object which contains the text information of the source file. Parsing the text information using method calls for Scanner such as hasNextLine() and nextInt(). Finally, the parsed puzzle data will be saved in the **Puzzle**. Everytime a new game is created, a random source file will be selected from the puzzle library, and the process of reading, parsing, and saving will take place step by step.

- **Basic Play Rules** : Code Matrix Cell available to be clicked - Apart from the code we need to know from the clicked matrix cell, the position of it is also essential in setting the next available row or column of matrix cells. Therefore we introduced the **Coordinate** class in each **MatrixCell** to record their coordinates when **CodeMatrix** is created in **Puzzle**. Once the matrix cell is clicked, it's coordinate is sent to **GameLogic** and the code of the cell is obtained from **CodeMatrix** based on the cell's coordinate instead of getText() from the button. A flag is used to decide whether to setRowAvailable or setColAvailable. Notice that only the **MatrixCell** with available = true and selected = false adds a Mouse Click event on it. So if the player tries to click an unavailable cell, nothing shall happen.

SUCCEEDED/FAILED – The matched state of each **DaemonCell**, the concept of waiting cell and alignment operation are keys to this solution. The waiting cell is the **DaemonCell** which is the cell waiting to be checked if its code matches the newly added code in the buffer. And it's index in a **Daemon** is the same as the (bufferCounter-1) before alignment. If the code of the waiting cell matches the code added in the buffer, the state of the cell is set to matched = true. If it does not match, the waiting cell switches to the first cell with code in the **Daemon** (not index 0), all cells are set matched = false and align operation is called to add empty cells at the front of the **Daemon** to make sure the index of waiting cell is still the same as (bufferCounter-1). The reason to use alignment and add empty cells is to serve the GUI. So visually, the player can see the waiting cell is always horizontally at the same line under the buffer's next empty grid. Also, the binding relationship between the waiting cell's index and the buffer counter simplifies the procedure to find the waiting cell. An alternative way to search for the waiting cell is to traverse the **Daemon** to find the first cell with matched == false. The complexity then changed from O(1) to O(n) and this is also a reason we did not choose this method. Given the matched state and alignment operation, to verify SUCCEEDED and FAILED **Daemon** becomes very simple. If the last **DaemonCell** is matched == true, then this **Daemon** can be set to SUCCEEDED. If the size of the Daemon exceeds the buffer size, then it's FAILED. A graphic example below (**[Figure 15]**) shows the waiting cell position and the process of alignment.



[Figure 15 - Alignment Example]

- **Timer** - **Timer** is a utility imported from Swing. We can control the frequency of applying operations in **Timer** when it starts to run and we can decide the condition to stop the **Timer**. In this game, the **Timer** is used to count down the time limit by one in every second and stops when the time limit hits zero.

- **UNDO** - Statuses are saved in a Deque, therefore UNDO is implemented by popping the Deque as long as the game is not over and the Deque is not empty and replace the currentStatus which information is displayed to the player. To make sure the original *status* is not "polluted", we implemented serialization and deserialization methods to deep copy the original status and only apply modifications on the copied status.

  If the player can undo freely then there is a bug that the player can keep repeating the action of clicking the *matrixCell* to complete a *daemon* and undo and then click that cell again. In this way the player can get the rewards quickly and easily and achieve a high score without much effort. To prevent this, we add a cool down time to the undo function so that the player can only make a second undo action in at least 8 seconds. This is done by adding a new **Timer** to the game panel. The **Timer** starts to count down when the undo button is pressed and the flag to indicate the availability of undo is set to unavailable. Once the Timer counts to zero, the flag becomes available again and the cool down time is reset to 8.

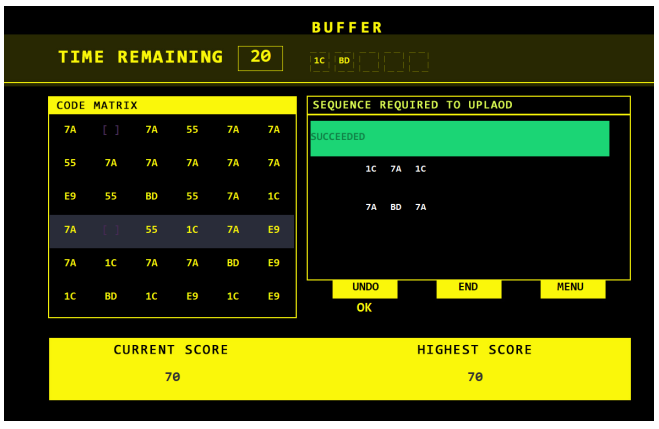
- **Score** - **ScoreHandler** uses **java.util.Scanne**r to read the save file just like the **Parse** do. For the encryption purpose, **java.util.Base64** is introduced, and it can be used to invoke both the base64 encoder and decoder. The port between the **ScoreHandler** and the system is in the **GameLogic**. When a new game is initialized, the **GameLogic** will invoke the **ScoreHandler** to read and decode the current highest score from the save file. When a game is over, the **GameLogic** will fetch the score of this game, and compare it with the record in the system. If the new score turns out to be the new highest score, the **GameLogic** will invoke the **ScoreHandler** to encode and write the new highest score to the save file.

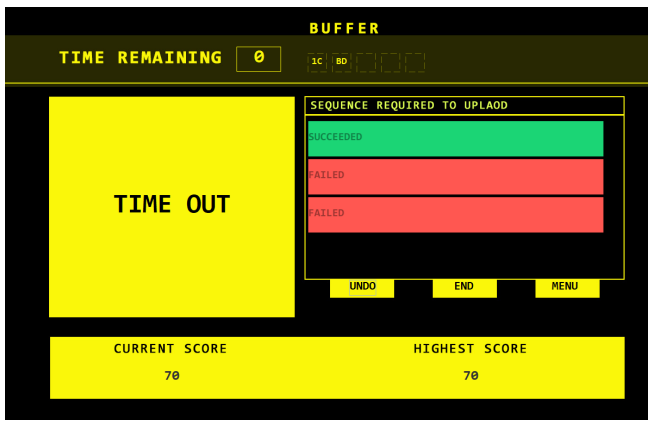
- Thinking about the **ScoreHandler**, the structure of reading from the save file seemed overlapped a lot with the reading in the **Parse** as they both use **java.util.Scanner** and read only from normal text files. Because of that, our first intention from the perspective of software design is to apply template design patterns. By doing that, the common structure of the reading part should be factored out and used by both the **Parse** and the **ScoreHandler**. This design shall potentially increase the maintainability as it will reduce code redundancy. Moreover, if any feature which requires the same reading technique is going to be added in the future, this design should make the thing easier because it can simply reuse the common structure and therefore the readability and the extensibility will be improved. This should be theoretically feasible. However, when we tried to implement it,things did not work as we thought. Given the previous version of our design, we made the **Parse** an inner class of the **Puzzle** as it contains utility functions which are static functions. We made the **ScoreHandler** in the exact same way. The functions in the ScoreHandler are static and can be used directly without creating an object of the **ScoreHandler**. Nevertheless, static functions cannot be adopted by the template because a template will pass down an abstract function to its lower classes. This abstract function needs to be specified for distinguished behaviors after the common operations in the template is done. We never knew that abstract functions and static functions are totally incompatible until we actually tried to do that. Changing the overall structure of the **Parse** and the **ScoreHandler** to normal java class rather than utility class is possible. However, it will cost too much effort and we risk having unexpected bugs because if we do that, we are going to change the code in the **GameLogic** and **Status** as well. Eventually, after we measured the trade off, we chose to maintain the old style. The static function has its own advantage in our system. It can reduce the unnecessary creation of objects, and therefore potentially reduce the complexity of the system by reducing the interactions between objects. Undeniable we might lose the benefits for using the template design pattern, but we think the old design suits better in our system at this point.

Location of Main
The main class to run the system is in the **Engine** in game package. (src/main/java/game/Engine.java).

Location of Jar
The Fat Jar file is located at : out/artifacts/software-design-vu-2020_jar/software-design-vu-2020.jar
（resources file need to be placed in the same directory with jar file）

Demo Video
https://youtu.be/b-5DFa3TB6U





Time logs

Team number	Cyberpunk-13

Member	Activity	Week number	Hours
Group	Group Meeting	6	1
Yu Chen	Design Patterns	6	3
Yudong Fan	Revise Diagram	6	2
Yu Chen	Implement Patterns	7	3
Yudong Fan	Implement Score	7	4
Xiaojun Ling	Revise assignment2	7	1
Berry Chen	Revise assignment2	7	1
Yu Chen	Modify Codes	7	2
Yu Chen	Revise Assignment2	7	3
Group	Group Meeting	7	2
Yu Chen	Modify Codes	8	1
Yu Chen	Revise UML Models	8	3
Yu Chen	Write Assign3 Doc	8	10
Yudong Fan	Write Assign3 Doc	8	6
Yudong Fan	Revise Assignment2	8	1
Berry Chen	Write Assign3 Doc	8	3
Berry Chen	Revise UML Models	8	4
Xiaojun Ling	Revise Assignment2	8	3
Xiaojun Ling	Revise UML Models	8	5
	TOTAL		58