Assignment 2

Team number: Cyberpunk-hacking-minigame Team 13
Team members

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*.

Implemented feature

ID	**Short name**	**Description**
F1	Commands	In the menu, the user may click the corresponding button to activate the following functions: - *set difficulty* - *start game* During the game: - the user may *select codes* from the code matrix by clicking the left mouse button - click “END” button to finish the game - click “MENU” button to back to the menu The player shall be able to close the game window at any time by clicking the “cross” on the top right of the application window.
F1B	Undo	**BONUS** During the game, the user may click the “UNDO” button to undo one latest move. The player can use UNDO once every 10 seconds. This is to prevent malicious scoring by keeping the process of undoing the last step or steps to complete a certain Daemon to obtain the rewards of SUCCEEDED multiple times. (**restrictions not implemented**)
F2	Puzzle	A randomly chosen puzzle read from TXT files which contain a buffer, a code matrix with matching daemons shall be displayed on the game window when the game starts. A new puzzle shall be loaded if the remaining time is not zero and every daemon has been rewarded.
F3	Difficulty	There shall be four difficulty levels for the user to choose from. Different levels shall be different in the buffer size, initial time limit, time rewarded and score rewarded.
F4	Basic play rules	Code selecting rules and Daemon completion rules shall respect the original game listed in the introduction section.
F5	Rewards	10 seconds and scores from 15 to 50 shall be rewarded for every successfully completed Daemon corresponding to different levels of difficulty. There shall be a penalty with a deduction of 5 seconds for each Daemon that fails to complete.
F6	Timer	There shall be a timer that starts to count down after the user picks the first code from the Code Matrix. The game shall be over if the remaining time becomes zero.

Used modelling tool: StarUML

Class diagram

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

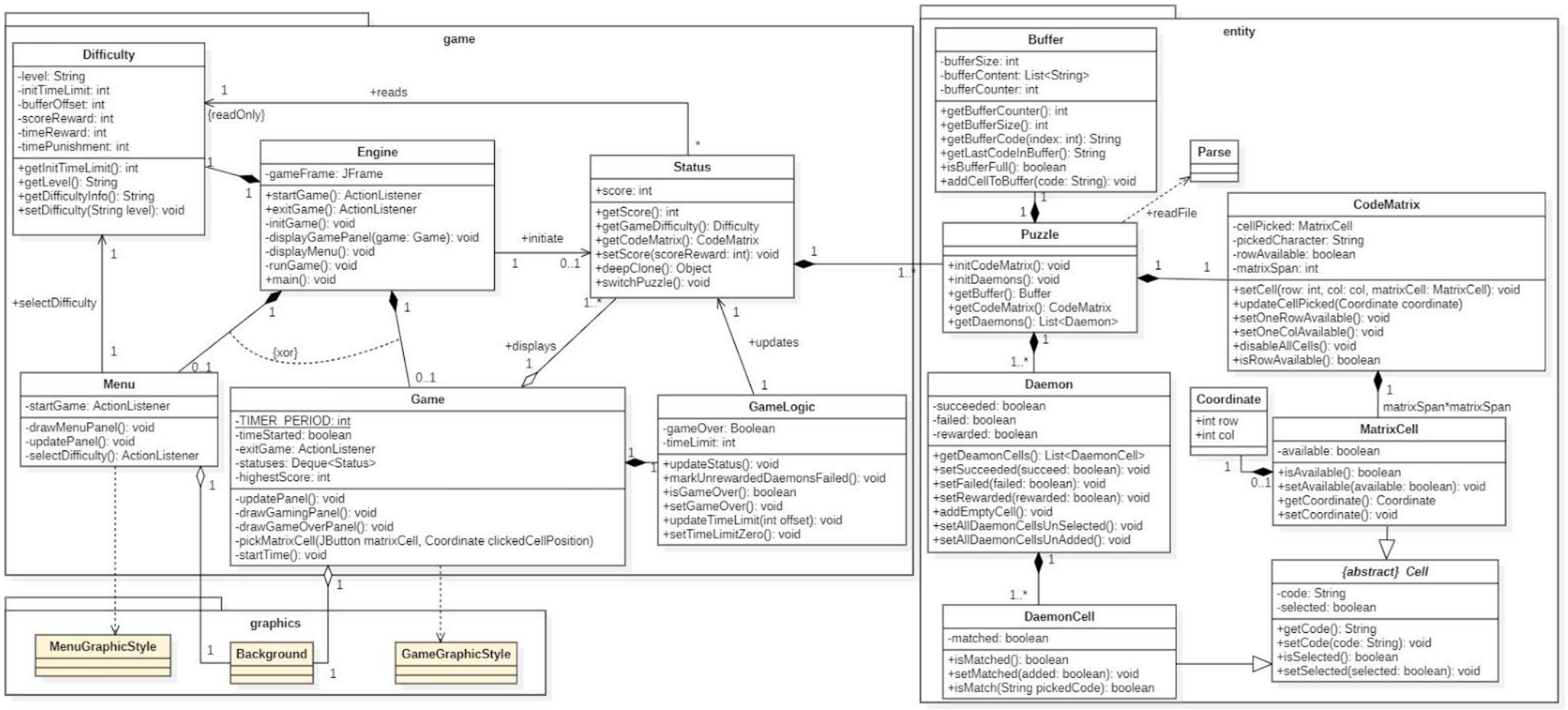
[**Figure 1]** below illustrates the class diagram model of the system. Each class in the model is elaborated as follows:

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 original 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.



[Figure 1 - Class Diagram (click to open original picture)]

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.
- setCellPicked(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().

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.

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.
- getCoordinate(): Coordinate – Return the coordinate of the matrix cell.
- setCoordinate(): void – Set the coordinate of the matrix cell.

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**.

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

- getBufferCounter(): int – Return the value of attribute bufferCounter.
- getBufferSize(): int – Return the value of attribute bufferSize.
- 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**.

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.

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.
- rewarded: boolean - A state of sequence that indicates the daemon is already marked with reward after the daemon is marked with SUCCESS or FAIL.

Operations

- setSucceeded(succeeded: boolean): void - Set the daemon to SUCCESS when the last cell of that daemon is marked with matched, thus the daemon is complete.
- setFailed(failed: boolean): void - Set the daemon to FAIL when the requirement of time or buffer space is not met.
- setRewarded(rewarded: boolean): void - Set rewarded is true when the daemon is marked with SUCCESS/ FAIL state and the time is awarded/ punished.

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.

DaemonCell

The **DaemonCell** class inherits from abstract class **Cell**, it contains the property of whether the cell is matched into the buffer and 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 marked with matched to the buffer and matched with the daemon's cell.
- setMatched(matched: boolean): void - This operation sets the state of the corresponding daemon cell.

Association:

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

<abstract> 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

- getCode():String / setCode():void - Get() operation returns the String of code and set() operation modifies the String.
- isSelected(): boolean / setSelected(selected: boolean): void - Return the boolean value of selected. / Set selected to the boolean value in parameter.

Puzzle

The **Puzzle** class has a subclass 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** saves the parsed puzzle data.

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 **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 **CodeMatrix**. Initiation includes set values and coordinates for each cell of the matrix.
- initDaemons(): void - This operation is called by the constructor of **Puzzle** as well. Similarly, it converts the data type of daemons from the list of array to the list of **Daemon**.
- getBuffer(): Buffer - All attributes can be read through get operations. Description of other get attribute operations is omitted.

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.

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”.

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

- getInitTimeLimit(): int - All attributes can be read through get operations. Description of other get attribute operations is omitted.
- 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 a set of private operations such as setDifficultyEasy() which will change the attributes to EASY 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.

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**.
- **Status** navigability - The **Engine** creates the *firstStatus* and passes it to the **Game**. Status does not exist depending on the Engine and 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**.

Attributes

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

Operations

- getScore() / getCodeMatrix() / getBuffer(), etc - Returns value or objects of the attributes in **Status**. **CodeMatrix**, **Buffer** and **Daemons** are not stored in the **Status** but in the **Puzzle**. These get() operations are called by the **Game** which displays their properties and **GameLogic** which updates their values. Different from the *score*, entities in the Puzzle are intentionally designed to be mutable through the get() operations which means changes done to objects get through **Status** are also impacting their original objects in the **Puzzle**. The reason for not encapsulate the **Puzzle** in **Status** is to update changes of entities in the Puzzle easily and quickly without copy all layers of entities such as **MatrixCell**s and **DaemonCell**s after every single action the player takes which may lead to an exploding number of objects.
- 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.
- setScore(): void - Modify the score. This is called in **GameLogic** when a **Daemon** is completed. (not implemented)
- 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

Holds the panel which displays every “snapshot” of **Status** and deals with user click events and Timer. It passes the current status to **GameLogic** and replaces the current status with the updated status through **GameLogic** and updates the panel to display the updated one.

Attributes

- TIMER_PERIOD: int - Static final variable. Determines the frequency of Timer runs.
- timeStarted: 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)

Operations

- drawGamingPanel(): void - Adds Time panel, CodeMatrix panel, Buffer panel, Daemons panel and Score panel on the game background panel. All the information displayed on these panels is retrieved from *currentStatus* except timeLimit which is stored in **GameLogic**. This is because operations such as undo() which reverses puzzle and score states should not reverse the time. Therefore time is stored separately.
- drawGameOverPanel(): void - This panel draws the same component panels as on the *gamingPanel* except CodeMatrix panel. Once the game is over, the CodeMatrix panel is closed and a Time Out panel is displayed to replace the CodeMatrix panel.
- startTime(): void - A Swing **Timer** is created in this operation. Once the **Timer** starts to run, the timeLimit is subtracted by 1 for every 1 second. The **Timer** stops if timeLimit equals zero and finishGame() is called to end the game.
- pickMatrixCell(JButton matrixCell, Coordinate clickedCellPosition): void - This operation creates a **MouseListener** event and binds the event to every **MatrixCell** in **CodeMatrix**. The timeStarted flag is checked first to implement the feature that the **Timer** only starts to count down if the player does click the **MatrixCell**. Then *currentStatus* is saved and copied status along with the clicked **MatrixCell** position (**Coordinate**) is passed to **GameLogic** to update the Puzzle correspondingly.

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 **Game** can create **Background**.
- **GameGraphicStyle** dependency - Similar to the relationship between **MenuGraphicStyle** and **Menu**. It provides functions for the **Game** to use to decorate the graphic components such as buttons, labels and panels inside the **Game**.
- **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.

GameLogic

Deals with status updating and rewards allocation. It takes status in from **Game** and returns a modified copy of status back to **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():void - The main interface function called by **Game** to update status. Inside this operation, *codeMatrix*, *buffer*, *daemons* in the status copied from **Game** is updated corresponds to the **MatrixCell** the player picked and rewards are allocated according to completion of each list of *daemon*.
- markUnrewardedDaemonsFailed():void - Set all unchecked daemons to failed equals true. This operation is called when gameOver sets to true. The remained unchecked daemons shall be set to FAILED.
- 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.

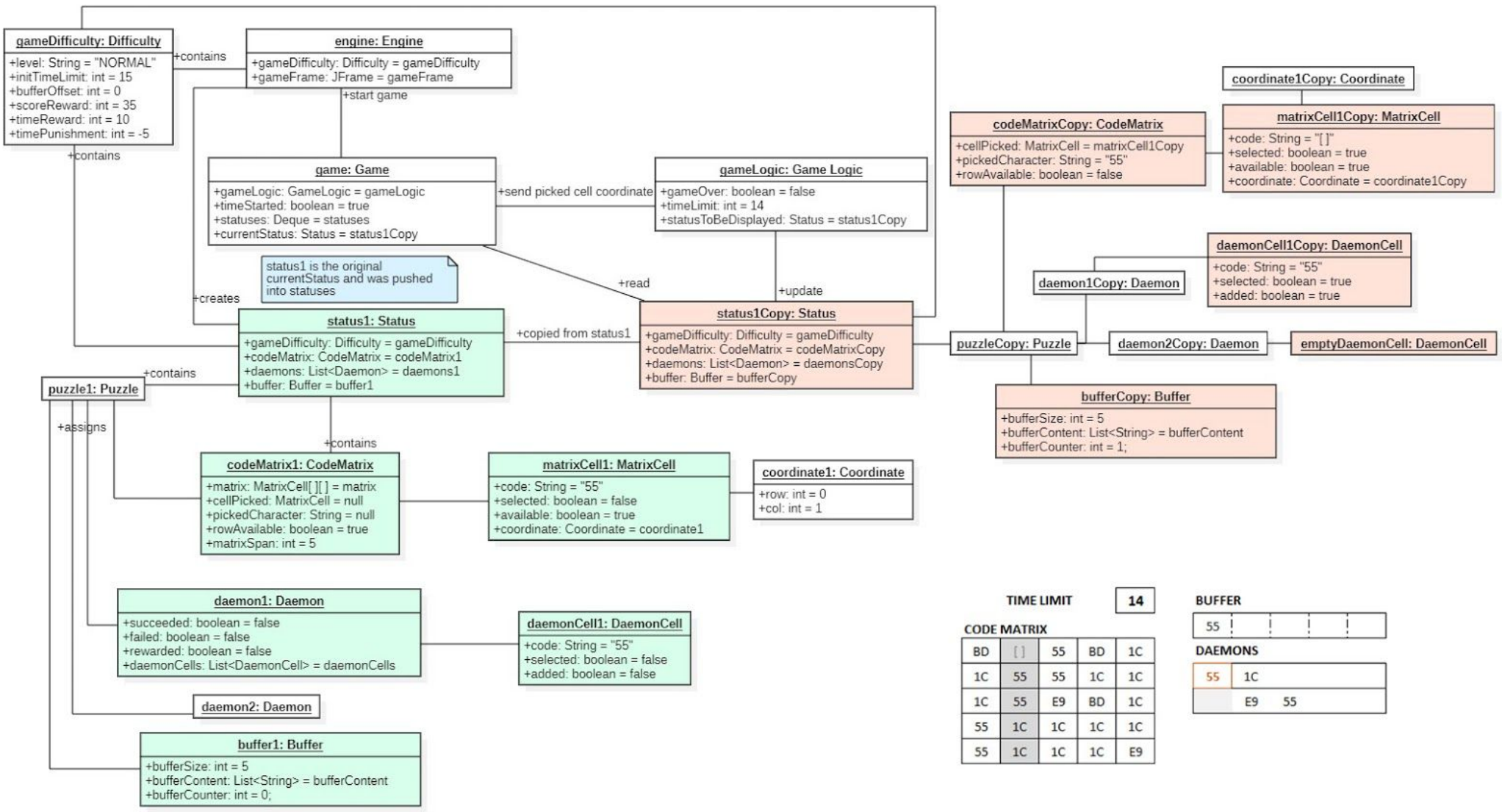
Graphics

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 2] below illustrates an object diagram model of the system.



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

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.

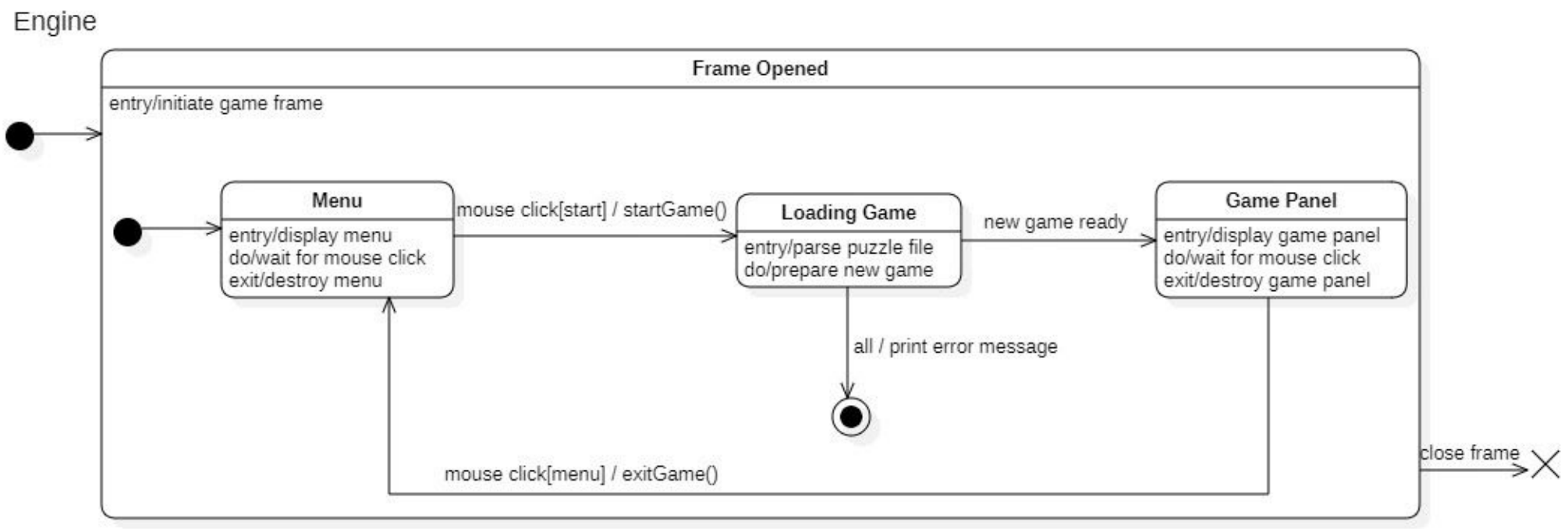
- *daemon1* vs. *daemon1Copy / daemon2* vs. *daemon2Copy* - The attributes did not change in these two daemon because none of them is marked as SUCCEEDED or FAILED. And since the cell in daemon1 matches buffer content, no alignment is needed in this daemon.

- *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

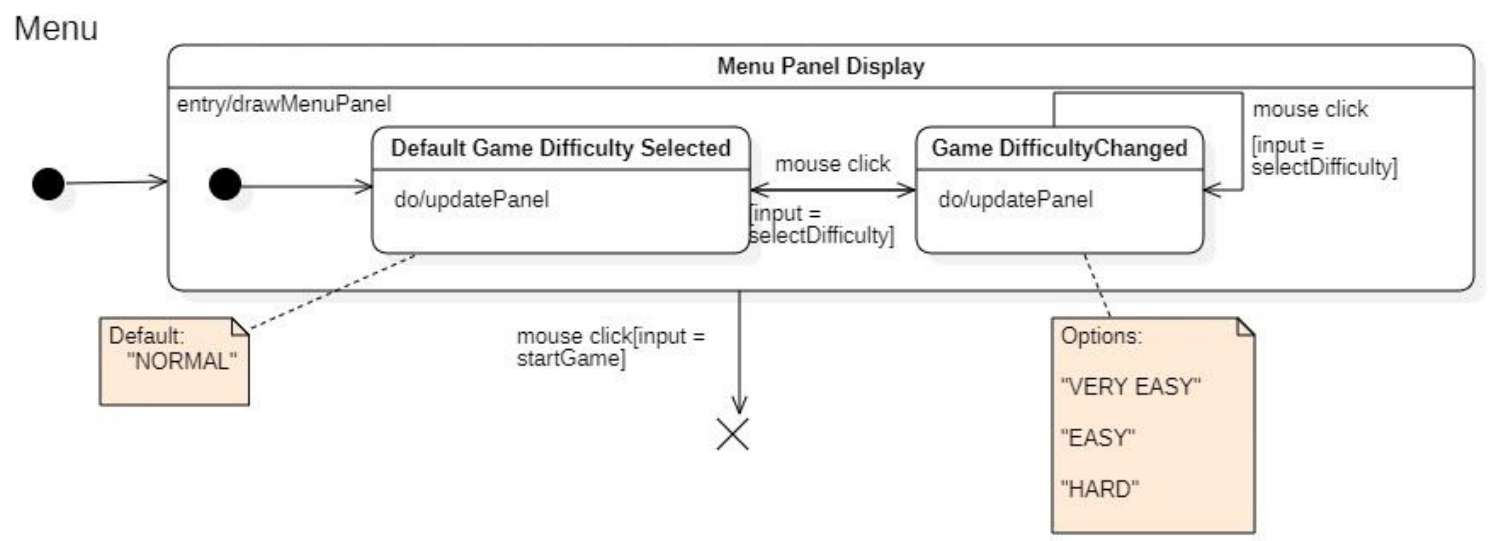


[Figure 3 - State Diagram: Engine (click here to open original picture)]

[Figure 3] 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 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. 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 4 - State Diagram: Menu (click here to open original picture)]

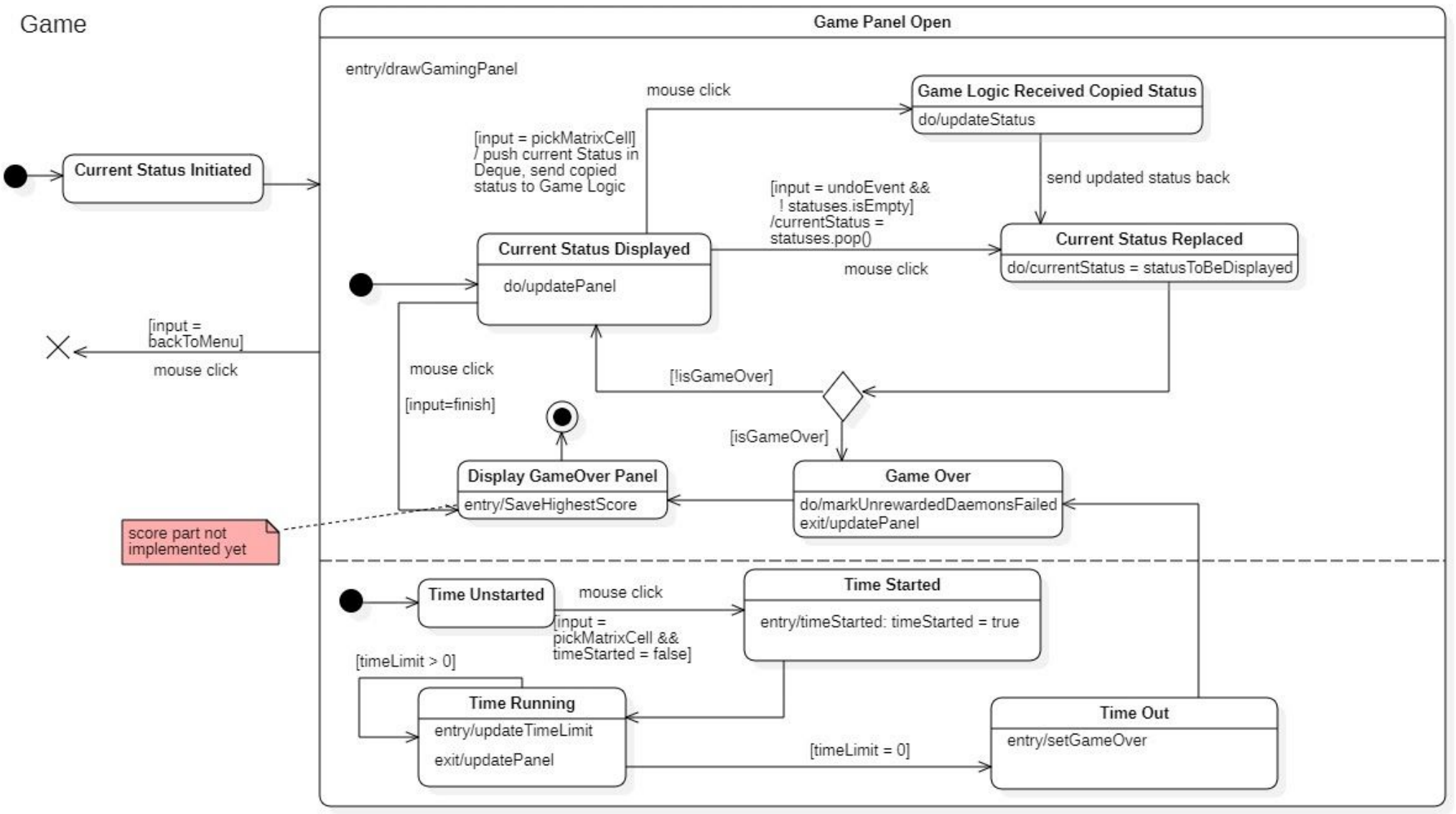
[Figure 4] on the left draws the state diagram of the **Menu**. The menu panel is displayed when **Menu** is created and terminated if *startGame* event is triggered (“START” button be clicked). Inside the panel displayed state, the gameDifficulty state can change between default level which is “NORMAL” to other levels.

[Figure 5] below is the state diagram of the **Game** class. The **Game** class controls the contents to be displayed to the user and decides when to start and when to stop one round of game. Once the initial status (*currentStatus*) is loaded, the game panel is opened and the information in the *currentStatus* is read and displayed on the graphic user interface. This is done by the drawGamingPanel() operation. The *currentStatus* is now displayed and the *Timer* is not created yet. These two concurrent substates are therefore waiting to be triggered by the player’s mouse click on a matrix cell 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**.

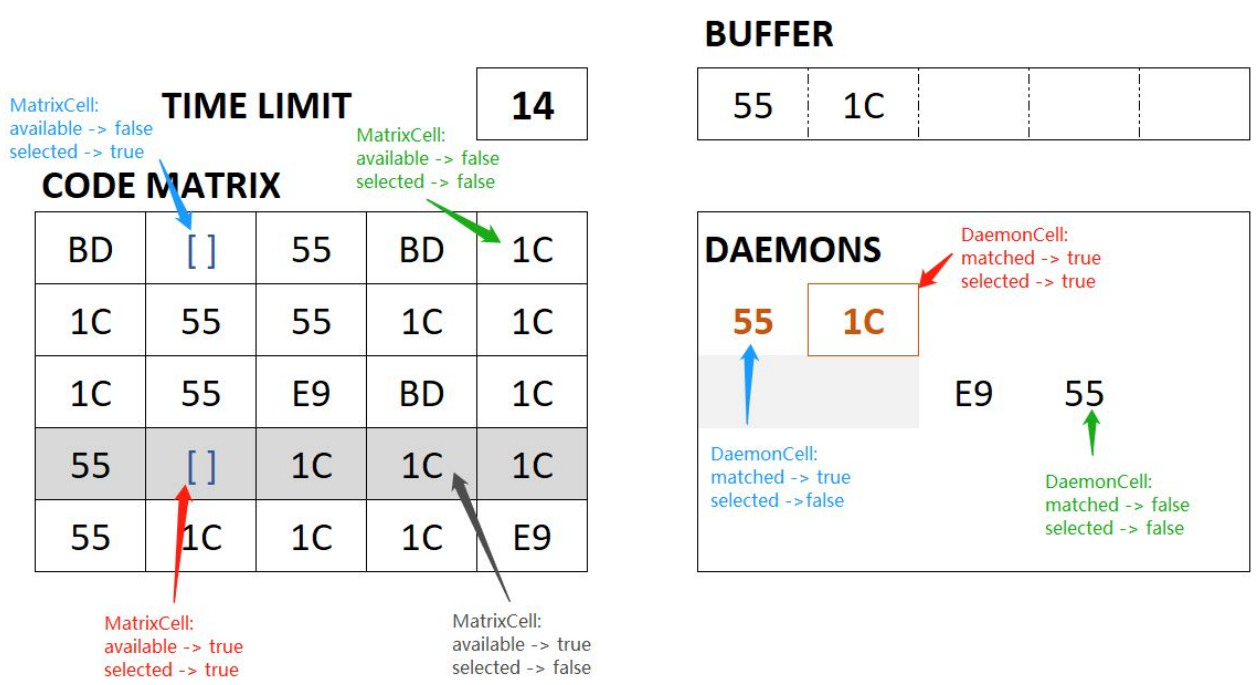
On the other side, the *currentStatus* is pushed in a Deque statuses and when a matrix cell is clicked. The copied *currentStatus* is sent to **GameLogic** to be updated. The updated version is then sent back to the **Game** and replaces the *currentStatus*. The replaced current status shall be displayed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

If the player clicks the “UNDO” button when the *currentStatus* is displayed, undo() is called and the last element in the status Deque is popped and replaces the *currentStatus*. Then the replaced *currentStatus* is displayed.



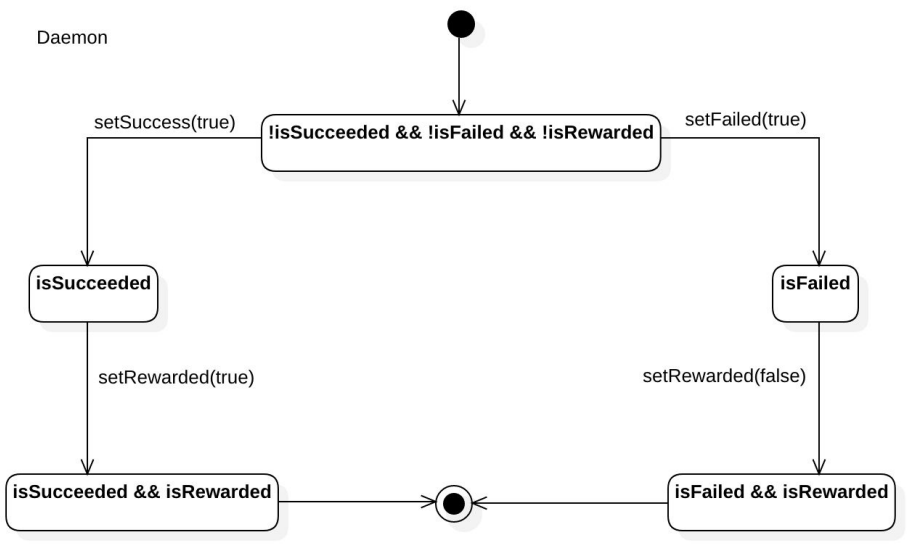
[Figure 5 - State Diagram: Game (click here to open original picture)]

State of Entities



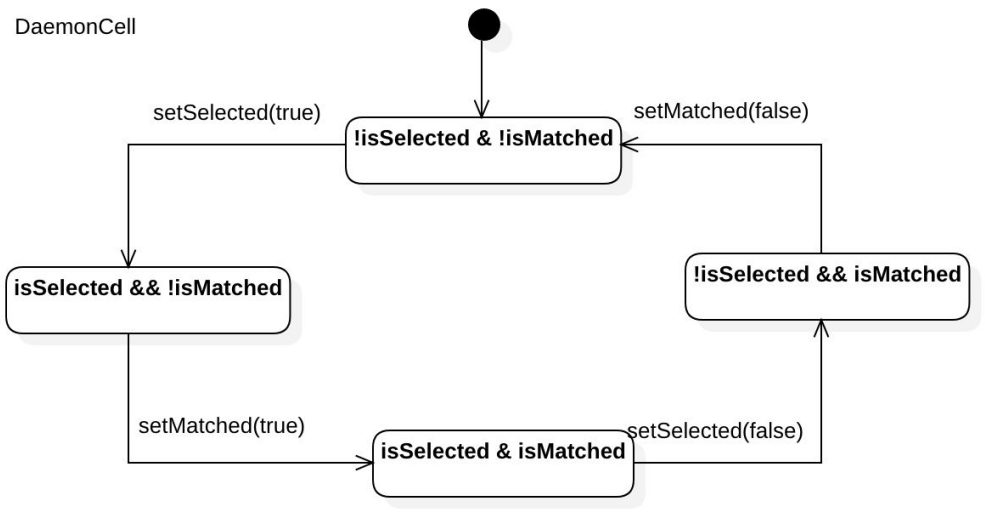
[Figure 6 - The performance of Entity States on Graphic User Interface (click to open original picture)]

[Figure 6] on the left provides an example that illustrates how the states of **Puzzle** entities affect their performance on the graphic user interface.



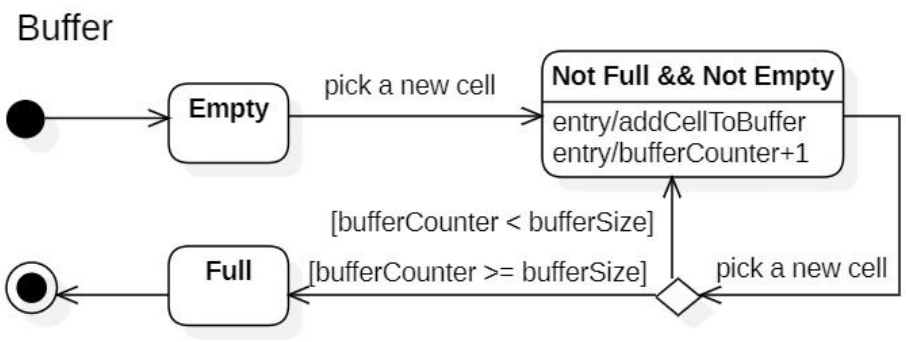
[Figure 7 - State Diagram: Daemon (click here to open original picture)]

[Figure 7] 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 Reward at the beginning. It will be marked with either SUCCESS or FAIL as the game runs and the conditions are met. Then the Reward state is set for each daemon when the time is awarded / punished respectively to SUCCESS / FAIL. Finally towards the final state and the daemon is completely marked.



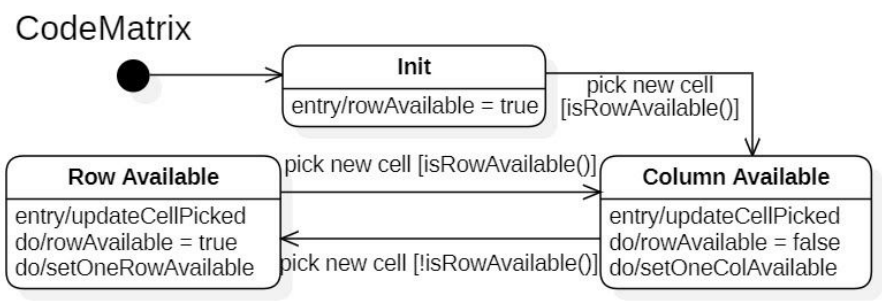
[Figure 8] is the state diagram of **DaemonCell**. It shows the states of a DaemonCell and the relation between states. At first, the daemon cell is not selected and not matched. The daemon cell becomes selected if there is a match between the user-picked character and the daemon cell. Then the daemon 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 8 - State Diagram : DaemonCell (click here to open original picture)]



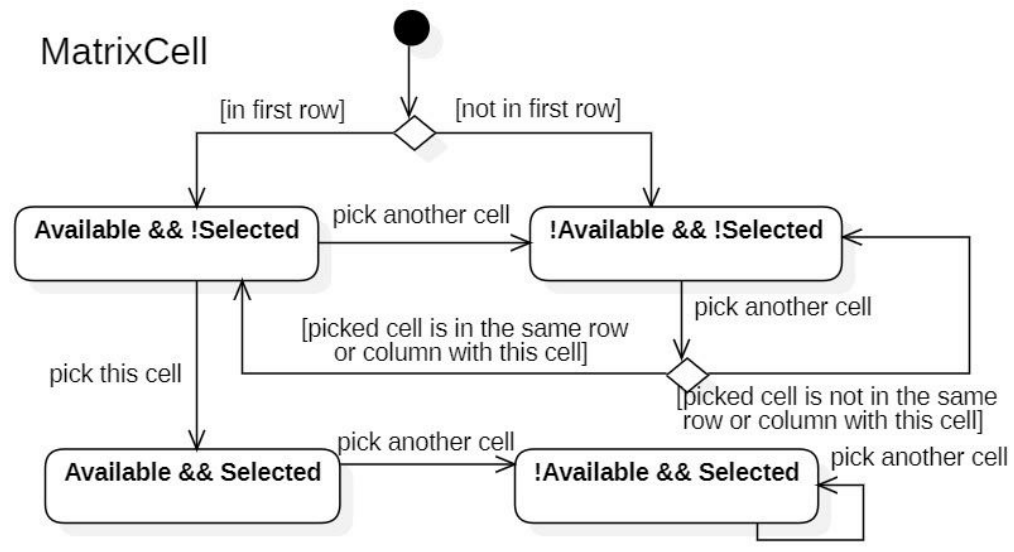
[Figure 9] on the left has modeled the possible states and transitions of class **Buffer**. Initially, the buffer is empty with no characters of cells inside. Once the player has picked a matrix cell, the event *pick a new cell* triggers the transition from state Empty to state Not Full && Not Empty, 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 9 - State Diagram : Buffer (click here to open original picture)]



[Figure 10] 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** during the process of game initialization. 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 10 - State Diagram: CodeMatrix (click here to open original picture)]

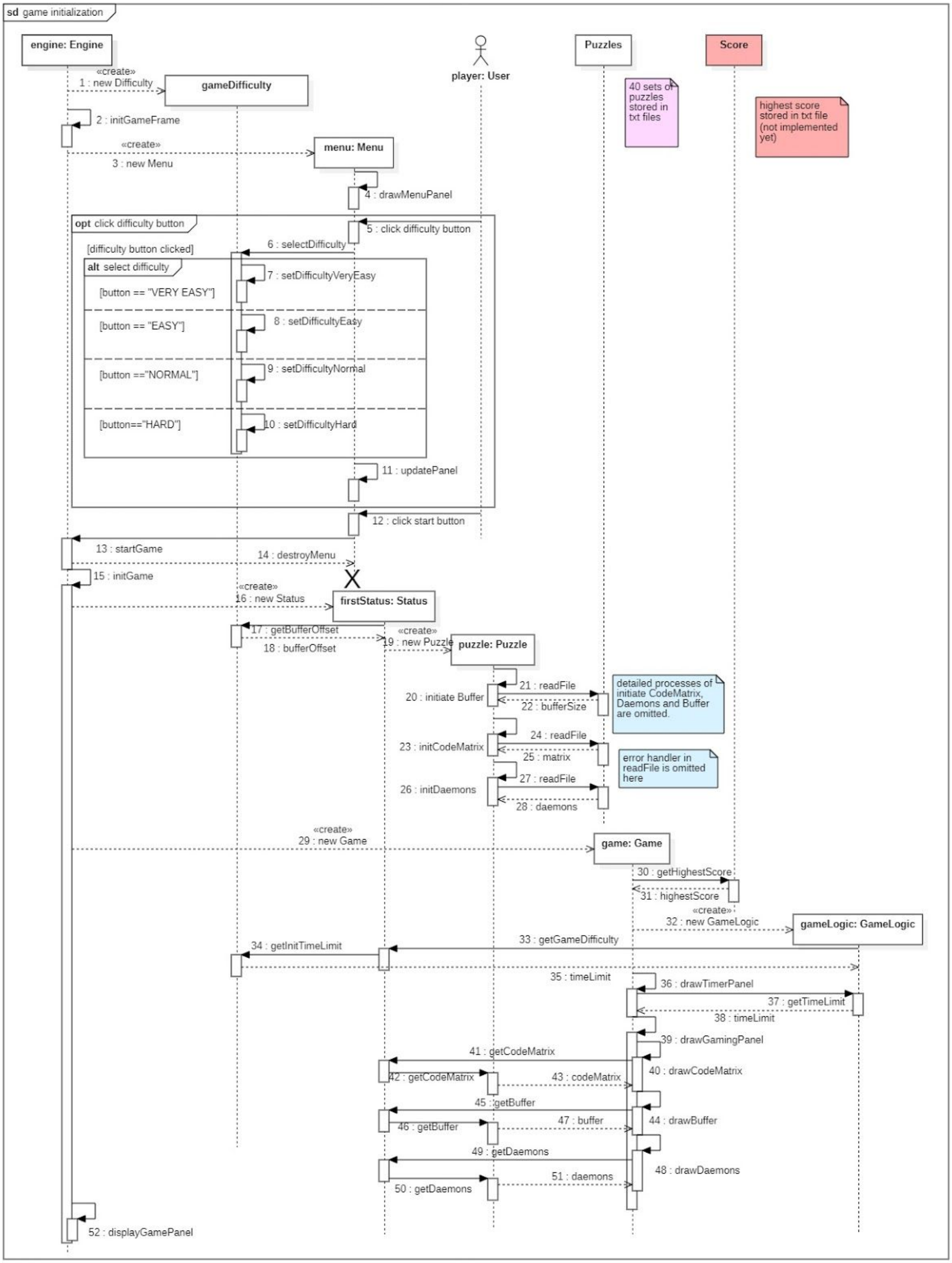


[Figure 11] on the left illustrates the possible states and transition relations of one matrix cell. A matrix cell can initially be in the first row or not, which leads to two different states, Available && !Selected and !Available && !Selected. If the cell currently in focus is in the state of Available 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 11 - State Diagram: MatrixCell (click here to open original picture)]

Sequence diagrams

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



[Figure 12 - Sequence Diagram: Game Initialization (click to open original picture)]

Sequence Diagram - Game Initialization:

[Figure 12] 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 firstly 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. Oppositely, 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 **Status** which contains parameters for the game will be created firstly. 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. After that, an instance of **Game** is created. The **Game** instance will first get the highest score from the score save file. Then it will create an instance of **GameLogic.** The **GameLogic** instance will get the time limit for the game by accessing the Status instance first. After the **Status** instance receives the request, it will ask the **Difficulty** instance to get the time limit. Finally, the Difficulty instance will respond to the **GameLogic** instance with the value of the time limit.

Eventually, the **Game** instance will start to draw the game. It will get the time limit from the **Gamelogic** instance and draw the timer Panel first. Then it will get puzzle data which are in types of **CodeMatrix**, **Buffer**, and **Daemon** by accessing the **Status** instance. The Status instance will then send requests to the **Puzzle** instance, and the **Puzzle** instance will send responses to the **Game** instance to draw the game panel respectively. Yet the initial game is constructed completely. The **Engine** instance will display the game panel on the game frame. All the initialization steps for starting a new game are finished.

Sequence Diagram - Player clicks on a Code Matrix Cell:

[Figure 13] The diagram below depicts the game running process that occurs when a player clicks on a cell in the matrix.

When the player clicks on a cell from the matrix, a mouse event is captured in the unique instance of the **Game** class, *game: Game*. If this is the first valid click by the user on the cells in the matrix, the attribute timeStarted of *game: Game* has its default value of false. In this case, *game: Game* calls its function startTime() to start the countdown of the game. After the handling of the game time, 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, after which *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 updateStatus, the function updateCodeMatrix is called with the position of the clicked cell as parameter to update the whole matrix. When function updateCodeMatrix is called, *gameLogic: GameLogic* firstly gets *codeMatrix: CodeMatrix* through *statusToBeDisplayed: Status*, 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 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.

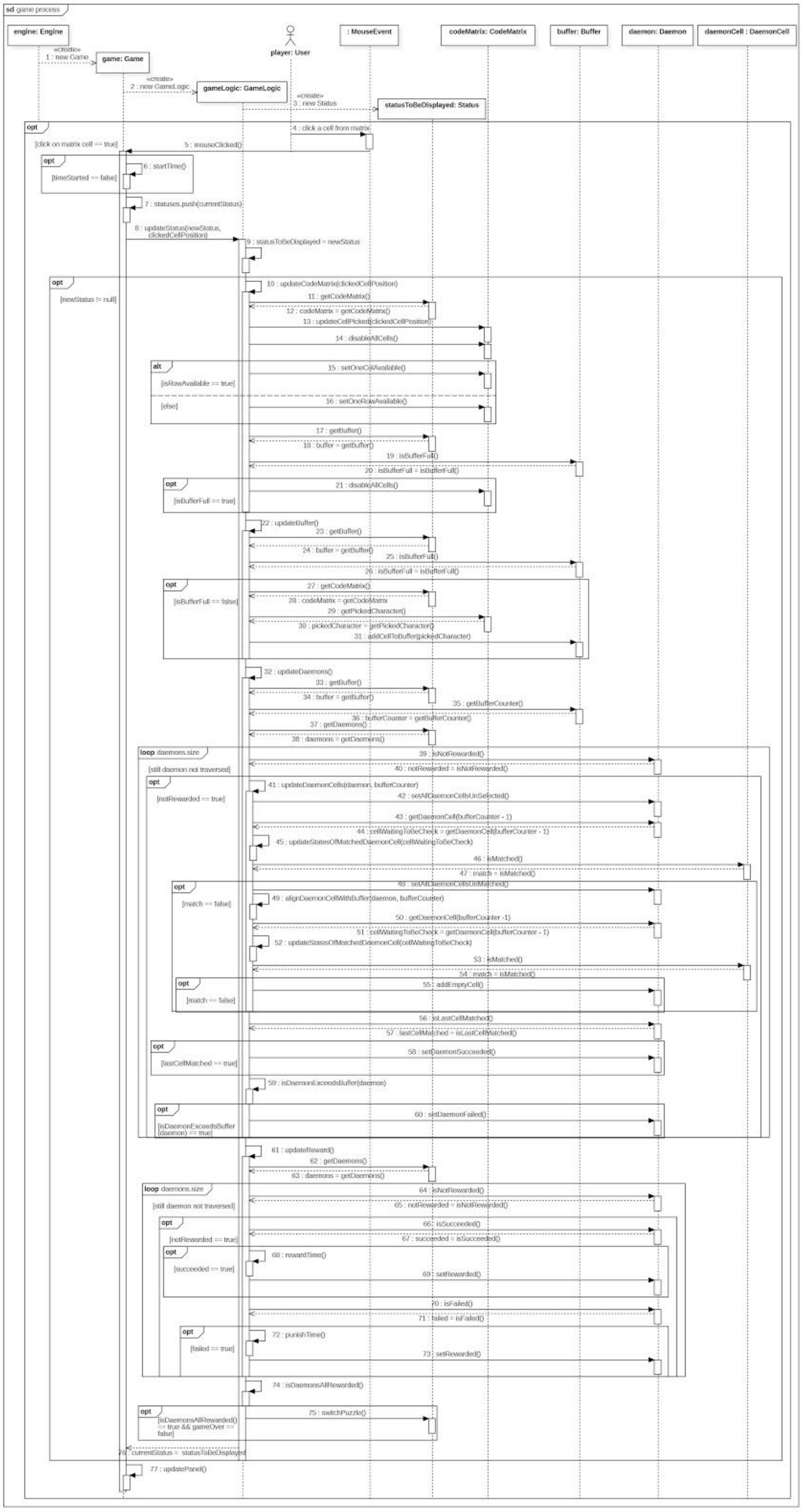
The function updateBuffer is called afterwards to update the status of the buffer. Initially, whether the buffer is full is determined. If buffer is not yet full, *codeMatrix: CodeMatrix* is accessed to get the character on the picked matrix cell by calling the function getPickedCharacter, after which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

Then, the function updateDaemon 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 rewarded 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 not, 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 afterwards to loop through all daemons and update the reward/ punish time based on the SUCCESS and FAIL, and also update the reward status of the daemon. If isDaemonsAllRewarded is true and not gameover, switchPuzzle is called to update new daemons. Thus, the statusToBeDisplayed from *statusToBeDisplayed: Status* is returned and becomes the current status. Then, updatePanel is called in *game:Game* to update the game panel based on the new status.



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

Implementation

Author(s): Yu Chen, Yudong Fan

From UML to JAVA

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 little 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

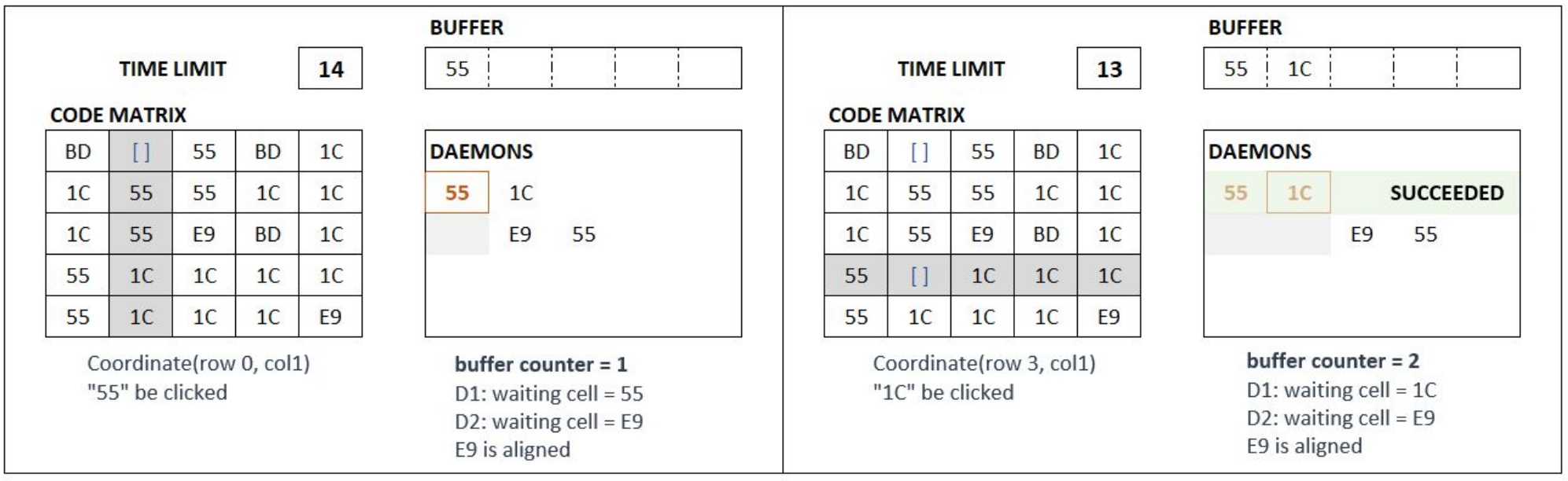
- **Command**: The most difficult part of this feature is how to fetch the code (character) of a code matrix cell the player clicked on and display the changes on **Puzzle** entities(e.g. code added in the buffer) after this click. We get the intuition from calculator applications which also have a bunch 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 sub-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 in 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 14]**) shows the waiting cell position and the process of alignment.



[Figure 14 - 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.

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/ijH5armY4IE

Time logs

1	**Team number**	cyberpunk-hacking-minigame13		
2				
3	**Member**	**Activity**	**Week number**	**Hours**
4	Group	Meeting	2	2
5	Yu Chen	Design UML	2	2
6	Yu Chen	Implementation	2	5
7	Berry Chen	Design UML	2	2
8	Berry Chen	Implementation	2	1
9	Yudong Fan	Implementation	2	6
10	Yudong Fan	Design UML	2	2
11	Xiaojun Ling	Design UML	2	2
12	Group	Meeting	3	5
13	Yu Chen	Design UML	3	6
14	Yu Chen	Implementation	3	10
15	Yudong Fan	Design UML	3	5
16	Yudong Fan	Implementation	3	2
17	Xiaojun Ling	Design UML	3	4
18	Xiaojun Ling	Implementation	3	4
19	Berry Chen	Daemons Implementation	3	5
20	Berry Chen	Design UML	3	4
21	Group	Meeting	4	6
22	Yu Chen	Design UML	4	4
23	Yu Chen	Implementation	4	15
24	Yudong Fan	Design UML	4	6
25	Yudong Fan	Implementation	4	1
26	Xiaojun Ling	Design UML	4	5
27	Xiaojun Ling	Implementation	4	3
28	Berry Chen	Daemons Implementation	4	5
29	Berry Chen	Design UML	4	5
30	Group	Meeting	5	6
31	Yu Chen	Write textual description	5	15
32	Xiaojun Ling	Write textual description	5	8
33	Berry Chen	Write textual description	5	7
34	Yudong Fan	Write textual description	5	10
35			TOTAL	163