Assignment 2

Team number: 14
Team members:

Name	**Student Nr.**	**Email**
Jop Zitman	2670863	j.zitman@student.vu.nl
Tiberiu Iancu	2659445	t.iancu@student.vu.nl
Sebastian Cristian Iozu (Chris)	2663383	s.iozu@student.vu.nl
Yingdi Xie	2669853	y7.xie@student.vu.nl

Summary changes from assignment 1

Author(s): Chris, Tibi

- "*This is good but does not completely describe how the game proceeds. An important aspect is that the column and row restrictions alternate.*"
  Location: **Introduction**
  Solution: Changed text to explain constraints better
  Example: "*[...] pick a cell in the same **row or column as the last chosen cell except for the first move.***" -> "*[...]* pick a cell in the same row, alternating with a column every time a cell is selected."

- "*Why "text file"? Try to avoid jargon and unnecessary information.* "
  Location: **Comparison with original version** -> **Differences** -> second paragraph
  Solution: These words were removed from Assignment 1
  Example: "*[...] specify a level **text file** while booting the game*" -> "*[...] specify a level while booting the game*"

- "*What is this "!"? Number of non-completed sequences, factorial?*"
  Location: In the formula used to calculate the score
  Solution: A reference was added to explain what the "***!***" means
  Example: "*completedSeq**!** * […]*" -> "*completedSeq!^{4} * […]*"

- "*not for children?* "
  Location: **Main audience** -> first paragraph
  Solution: Changed text to represent people of all ages
  Example: "*[...] it suits both young and old **adults***" -> "*[...] it suits children and adults*"

- "*No jargon*"
  Location: **Functional features** -> **F1**
  Solution: Changed text to remove jargon
  Example: "*Read level from text file [...] into **Java variables***" -> "*Read level from text file [...]* and store its contents"

- "*This is not a QR but a feature. (F1)* "

Location: **Quality requirements** -> **QR2**
Solution: Changed description
Example: "***New levels could be added to the game as text files, and loaded through F1.***" -> "*Adding multiple levels in the future should be easy.*"

- "*How do you intend to test the puzzles? This is not mentioned in any features.*"
  Location: (was) **Quality requirements** -> **QR3**
  Solution: Removed from quality requirements

- "*How do you do this? How does someone check whether this QR has been met or not?*"
  Location: (was) **Quality requirements** -> **QR4**
  Solution: Removed from quality requirements

- "*How do you do this? How does someone check whether this QR has been met or not?*"
  Location: (was) **Quality requirements** -> **QR5**
  Solution: Removed from quality requirements

- "*This is guaranteed by using Java.*"
  Location: (was) **Quality requirements** -> **QR6**
  Solution: Removed from quality requirements

- "*Doesn't make sense to include this.*"
  Location: (was) **Quality requirements** -> **QR8**
  Solution: Removed from quality requirements

- "*You cannot use Lombok, it interferes with the design choices you are supposed to make and motivate for this course*"
  Location: (was) **Java libraries**
  Solution: Removed from Java libraries

- Added QR3, QR4 and QR5.

- "[SEVERE] Missing feature, the ability to modify the buffer size"
  Location: Canvas Feedback
  Solution: Added a new feature in the game and the table (F13)

- Multiple problems with the score-calculation formula.
  Example: If the timer reaches 0, the score will always be 0, which is wrong.
  The previous feature (ability to modify buffer size) created a problem when calculating the score (increasing the buffer size would result in a higher score)
  Solution: Fixed formula

- JUnit testing framework doesn't work well with LibGDX.
  Solution: removed testing framework and fallback on manual testing.

Implemented features

ID	Short name	Description
F1	Load level	Read level from the text file (path given as an argument to the jar) and store its contents (i.e. the buffer size, matrix/grid values, and target sequences).

<table>
<tr><td>F2</td><td>GUI</td><td>Using libGDX, open a new window and draw a basic buffer, matrix/grid, target sequences and a timer to mimic the original game.</td></tr>
<tr><td>F3</td><td>Exit</td><td>The user can exit the game at any time. Any progress will be lost.</td></tr>
<tr><td>F4</td><td>Move selector</td><td>The user can move their current selector horizontally/vertically inside the matrix.</td></tr>
<tr><td>F5</td><td>Confirm selector</td><td>The user can confirm their current selector. The value of the selected cell is added to the buffer and the UI is updated.</td></tr>
<tr><td>F7</td><td>Undo select</td><td>The user can undo their last step. The last value is the buffer is removed and the UI is updated.</td></tr>
<tr><td>F8</td><td>Redo
select</td><td>The user can reverse their previous undo step. The undone selection is taken again and the UI is updated.</td></tr>
<tr><td>F9</td><td>Game end</td><td>The user's score is calculated and presented together with the corresponding buffer and sequences.</td></tr>
<tr><td>F10</td><td>Timer</td><td>The timer starts at a fixed time and only starts counting down when the user has made its first move. Once it has reached zero, the user can no longer make moves and the game ends (F9).</td></tr>
<tr><td>F11</td><td>Confirm buffer</td><td>The user can confirm their current buffer and stop the timer. The game ends (F9).</td></tr>
<tr><td>F12</td><td>Input handling</td><td>Only keyboard input is handled. The following keys are mapped:<table><tr><td><code>Escape</code></td><td>Exit game (F3)</td></tr><tr><td><code>Up</code>/<code>Down</code>/<code>Left</code>/<code>Right</code></td><td>Move selector (F4)</td></tr><tr><td><code>Spacebar</code></td><td>Confirm selector (F5)</td></tr><tr><td><code>Backspace</code></td><td>Undo select (F7)</td></tr><tr><td><code>R</code></td><td>Redo select (F8)</td></tr><tr><td><code>Enter</code></td><td>Confirm buffer (F11)</td></tr><tr><td><code>-</code></td><td>Decrease buffer length (F13)</td></tr><tr><td><code>=</code></td><td>Increase buffer length (F13)</td></tr></table></td></tr>
<tr><td>F13</td><td>Increase/
decrease buffer size</td><td>The user can increase or decrease the buffer size at the beginning of the game, but not during the game. This encourages the player to think ahead!</td></tr>
</table>

Used modeling tool: Lucidchart

Class diagram

Author(s): Chris, Yingdi, Tibi, Jop

software-design-vu/Class Diagram.png

Main

`Main` is used for initializing a new application window and changing it's default configuration, the breachGame object and the 'parsing' of the file path from the command argument.

Operations

- *main()* - the entrypoint for this application. It is referenced in the gradle manifest build attribute 'Main-Class'.

Associations

- **LwjglApplication** - Composition. In this project, `LwjglApplication` cannot exist without `Main`. No other classes should access the `LwjglApplication`.

LwjglApplication

`LwjglApplication` is a class included in the LibGDX library. It handles the creation and configuration of application windows.

BreachGame

`BreachGame` extends the `Game` class from LibGDX. It initializes the game level, the game state, and is responsible for setting the GameScreen.

Attributes

- *gameLevel: GameLevel* - used to initialize and access the information read from a file with a given path (chosen by the user)
- *gameState: GameState* - used to initialize the state of the game based on information extracted from `gameLevel`
- *currentScreen: Screen* - used to initialize main screen of the game based on `gameState`

Operations

- <<*create*>> - sets the screen to `currentScreen`, which displays all the GUI elements.

Associations

- ***GameLevel*** - Composition. The `BreachGame` must contain an instance of `GameLevel` to support the access of the game's setup. `GameLevel` should not exist independent of `BreachGame`.
- ***GameState*** - Composition. The `BreachGame` must contain an instance of `GameState` to handle the current game state. `GameState` should not exist independent of `BreachGame`.
- ***ScreenAdapter*** - Composition . `ScreenAdapter`, the super class of all classes in the `Screen` package is created by the BreachGame class and saved in *currentScreen*.

Game

`Game` is a class included in the LibGDX library. It implements internal (unused) methods and more specifically, the setScreen() function, which is used to switch the currently rendered screen.

GameState

`GameState` is the core of the game logic. It encapsulates the current state of the game and supports the transition between different game states as the player interacts with the system.

Attributes

- *buffer: Stack<Cell>* - represents the buffer where selected cells can be added or removed. It is initialized as an empty stack at the beginning of the game. The buffer is modified when the player confirms a selector or performs “undo”/“redo”. It is also used for score calculation.
- *gameLevel: GameLevel* - when the game is loaded from a text file, the file is parsed to `gameLevel` which contains information about the game’s setup (such as the matrix of cells and the sequences to be matched).
- *undoStack: Stack<Cell>* - holds the most recent confirmed cell(s) when the player performs “undo”. Cells are popped from this stack when the player performs “redo”.
- *selector: Cell* - represents the selected cell when the player moves around inside the matrix of cells.
- *timerLogic: TimerLogic* - an instance of the `TimerLogic` class to access the timer information.
- *maxOffset: Int* - maximum number of slots in the buffer that the player can add. Set to constant 2.
- *offsetBufferLength: Int* - current offset to the default buffer length set by `gameLevel`.
- *finalScore: Int* - the player’s final score calculated according to the formula defined in Assignment 1.

Operations

- *<<create>>* - it sets the game level to `GameLevel` and sets the selector to the default position.
- *undo()* - invoked when the player presses “Backspace”. It undoes the last confirmation. The last cell will be popped from `buffer` and pushed to `undoStack`. `Selector` will be set to the previous cell.
- *redo()* - invoked when the player presses “R”. It reverses the last “undo” step. The cell at the top of `undoStack` will be popped and pushed to `buffer`, which will be the new value of `Selector`.
- *move(dir: Direction)* - invoked when the player presses “Up”/“Down”/“Left”/“Right”. If the move is valid (checked by `isValidMove()`, see below), `Selector` will be set to the new cell depending on the move’s direction.
- *getCurrentBufferLength(): Int* - get the current buffer’s length and used to render `BufferGUI`.
- *increaseBufferLength()* - invoked when the player presses “Equals”. If the timer has not started yet and the current offset is smaller than `maxOffset`, then the buffer length is increased by 1.
- *decreaseBufferLength()* - invoked when the player presses “Minus”. If the timer has not started yet and the current offset is bigger than the minimum offset (i.e., negative `maxOffset`), then the buffer length is decreased by 1.
- *confirmSelector()* - invoked when the player presses “Enter”. It confirms the current `Selector` and adds it to `buffer` if the buffer is not full, and ignores the keypress otherwise. It also clears up the `undoStack`.
- *setScore(): Int* - used to calculate and set the player’s final score calculated when the game ends.
- *getScore()* - use to retrieve the player’s final score and used to render `ScoreGUI`.

- *isValidMove(dir: Direction): Boolean* - it checks whether a move is valid according to the game's constraints. A valid move is one which stays within the bound of the matrix and in the right direction. The direction must be switched between horizontal and vertical for each move, and the first move must be horizontal (i.e., left or right in the first row).
- *isSequenceCompleted(seq: List<String>): Boolean* - it checks whether a sequence is completed. A completed sequence is one which is matched by consecutive cells in the `buffer`.
- *isSequenceFailed(seq: List<String>): Boolean* - it checks whether a sequence cannot be completed. If the unmatched cells of a sequence are longer than the remaining free slots in `buffer`, it is failed.

Associations

- ***GameLevel*** - Shared. `GameState` holds a reference to an instance of `GameLevel` to access information about the game's setup. `GameLevel` can exist independent of `GameState`.
- ***TimerLogic*** - Composition. `GameState` holds a reference to an instance of `TimerLogic` to access the timer's information and start the timer. `TimerLogic` should not exist outside `GameState`.

GameLevel

`GameLevel` encapsulates all information about the game's setup. It is initialized when the game is loaded. By supporting the retrieval of relevant setup information, it helps maintain the consistency of the game's state.

Attributes

- *matrix: Map<Cell, String>* - represents the matrix where the player can navigate and select cells. It is initialized as a mapping from `Cell` to the alphanumeric letters residing in the cell.
- *solutions: List<List<String>>* - represents a list of sequences to be matched by the selected cells in the buffer. Each sequence is initialized as a list of strings.
- *bufferLength: Int* - represents the buffer's size, in other words, the maximum number of cells which the player can select during the game.
- *matrixSize: Int* - represents the width and height of the matrix in cells (width and height). It is used to draw the GUI and check whether a move is out of bound.

Operations

- *<<create>>* - it tries to load the game from a text file, scan through it and parse it as the attributes of `GameLevel`. It throws a "FileNotFoundException" in case that the file path is not valid.

TimerLogic

`TimerLogic` implements the logic of the timer which is used to both keep track of the time the user has left to finish the level and to render the timer text on screen.

Attributes

- *startTime: Long* - unix time in milliseconds, initialized when the timer is started.
- *totalTime: Long* - time in milliseconds until the timer ends.

Operations

- *<<create>>* - creates new TimerLogic object with proper totalTime (in ms)

- *start()* - start the timer.
- *hasStarted(): Boolean* - returns whether the timer has been started.
- *timeLeft(): Long* - returns the amount of seconds left until the timer ends.

ScreenAdapter

`ScreenAdapter` is a class included in the LibGDX library. It implements internal (unused) methods, including the commonly used *render()* and *show()* methods. These are overwritten by the GUIelements.

Associations

- ***GameState*** - Directed Association. `ScreenAdapter`, the super class of all classes in the `Screen` package, can access `GameState`, whereas `GameState` cannot access the fields of `ScreenAdapter`.
- ***GUIelement*** - Composition. Only screens are able to create `GUIelement` objects. Without the `ScreenAdapter`, `GUIelement` objects can't exist.
- ***GameScreen*** - Generalization. GameScreen extends ScreenAdapter so that all the necessary methods used by LibGDX are implemented.
- ***GameOverScreen*** - Generalization. GameOverScreen extends ScreenAdapter so that all the necessary methods used by LibGDX are implemented.

InputAdapter

`InputAdapter` is a class included in the LibGDX library. It helps implement the user input handling in the different screens.

GameScreen

`GameScreen` extends the `ScreenAdapter` class from libGDX. It contains the relevant information so as to support the rendering of the GUI when the player is playing the game.

Attributes

- *game: Game* - reference to `BreachGame` so that when the game ends, it can set the screen to `GameOverScreen` and update the GUI.
- *gameState: GameState* - reference to the current game state to retrieve necessary fields for GUI.
- *gridGUI: GridGUI* - an instance of `GridGUI` representing the GUI element for the matrix of cells. It is created during the initialization of `GameScreen`.
- *timerGUI: TimerGUI* - an instance of `TimerGUI`, the GUI element for the timer which keeps track of the remaining time during the name. It is created during the initialization of `GameScreen`.
- *bufferGUI: BufferGUI* - an instance of `BufferGUI` representing the GUI element for the buffer. It is created during the initialization of `GameScreen`.
- *sequenceGUI: SequenceGUI* - an instance of `SequenceGUI`, the GUI element for the sequences to complete. It is created during the initialization of `GameScreen`.

Operations

- *<<create>>* - it saves the `Game` and `GameState`, and it creates the GUI elements to display.
- *show()* - used to create instances of SpriteBatch and BitmapFont for all GUI elements at the beginning of the game and to initialize the InputProcessor used to receive user input.
- *render(delta: Float)* - used to render individual GUI elements.

GameOverScreen

`GameOverScreen`, which extends `ScreenAdapter`, encapsulates the relevant information to support the rendering of the GUI at the end of the game.

Attributes

- *game: Game* - reference to an instance of `BreachGame`.
- *gameState: GameState* - reference to the game's state to access information when the game ends.
- *scoreGUI: ScoreGUI* - an instance of `ScoreGUI`, the GUI element to display the player's score. It is created when `GameOverScreen` is initialized.
- *bufferGUI: BufferGUI* - an instance of `BufferGUI`, the GUI element representing the buffer. It shows the buffer when the game is over. It is created when `GameOverScreen` is initialized.
- *sequenceGUI: SequenceGUI* - an instance of `SequenceGUI`, the GUI element for the sequences. It is created when `GameOverScreen` is initialized. It displays the completed sequences at the end of the game.

Operations

- *<<create>>* - it saves the `Game` and `GameState`, and it creates the GUI elements to display.
- *show()* - used to create instances of SpriteBatch and BitmapFont for all GUI elements at the beginning of the game and to initialize the InputProcessor used to receive user input.
- *render(delta: Float)* - used to render individual GUI elements.

GUIElement

Attributes

- *batch: SpriteBatch* - object used to render elements on screen.
- *font: BitmapFont* - font object used to render text.
- *x: Float* - horizontal position of element on screen.
- *y: Float* - vertical position of element on screen.

Operations

- *<<create>>* - it sets the horizontal and vertical positions of the element.
- *render(gameState: GameState)* - draws the current frame of the game based on the GameState object. In this abstract class, the render function is empty, suggesting that the extending classes should override and implement it.

Associations

- ***GameState*** - Directed Association. `GUIElement` should access `GameState` to support the rendering of UI, whereas `GameState` cannot access the fields of `GUIElement`.
- ***GridGUI*** - Generalization. GridGUI depends on GUIElement to support the rendering.
- ***TimerGUI*** - Generalization. TimerGUI depends on GUIElement to support the rendering.
- ***ScoreGUI*** - Generalization. ScoreGUI depends on GUIElement to support the rendering.
- ***BufferGUI*** - Generalization. BufferGUI depends on GUIElement to support the rendering.
- ***SequenceGUI*** - Generalization. SequenceGUI depends on GUIElement to support the rendering.

Cell

Helper class that facilitates working with coordinates.

Attributes

- *x: Int* - represents a horizontal coordinate.
- *y: Int* - represents a vertical coordinate.

Operations

- *<<create>>* - creates a new Cell object and saves x and y.
- *applyDir(dir: Direction): Cell* - this function returns a new Cell object representing the outcome of moving in the given direction.
- *hashCode(): Int* - overridden function used internally by HashMap.
- *equals(givenCell: Object): Boolean* - overridden function that facilitates the comparison of different Cell objects.

Object diagram

Author(s): Tibi, Yingdi

[software-design-vu/Object Diagram.png]

The diagram captures a snapshot of the system at the start of level 1 of the Breach Game. In this model, the user chose to start at cell (0, 0) i.e. top left. Once the cell is selected, the timer is started and cell (0, 0) is added to the buffer stack.

- **:BreachGame:** This is the game object required by LibGDX and it gets created in `main()`. `currentScreen` is set to `gameScreen`, gameLevel is initialized to a new `GameLevel` object and `GameState` to a new `GameState` object.
- **gameLevel:** Contains the data as described by `01.txt` and is initialized before the game starts. `matrix` contains 25 elements, mapping each cell in the 5x5 grid (as `matrixSize` is 5) to a string value. `bufferLength` equals 3, meaning the user can select a maximum of 3 cells.
- **gameState:** The user just selected the top left cell, so buffer contains one element, namely cell (0, 0). The selector is still positioned at (0, 0) and can now only move vertically. `gameLevel` contains the information described above. `undoStack` is empty, as the user has yet to undo.

`offsetBufferLength` is still 0, as the user has yet to increase or decrease the buffer's length. `finalScore` will stay 0 until it is calculated when the game ends.

- **gameScreen:** This object contains a reference to a game object (from LibGDX) and a collection of GUI objects (namely for the grid, sequence, buffer and timer) that don't change position during the game.
- **timerLogic:** The timer is initialized with 1614614104420 (UNIX time in milliseconds) of when the user selected the first cell. The total time the user has to complete the level is 60000 milliseconds, as shown in `totalTime`.
- **GUI elements:** `sequenceGUI`, `gridGUI`, `bufferGUI` and `timerGUI` have fixed positions. All of them inherit a `SpriteBatch` and `Font` objects from `GUIElement`, that are initialized in each constructor individually with no arguments.

State machine diagrams

Author(s): Yingdi, Jop, Chris

[software-design-vu/State machine diagram.png]

In this section, we will describe 3 UML state machine diagrams for our game. The first diagram depicts the transition between different states for the `BreachGame` class, that is, the game's overall state machine. The second diagram illustrates the transition between states for the `SequenceGUI` class, which represents the internal behavior of the sequence GUI element when the player is playing the game. The third diagram shows the transition between states for the `GameLevel` class, which is responsible for parsing game files.

BreachGame

The state machine diagram above illustrates the transitions between different states of the `BreachGame` class. As shown in the class diagram, `BreachGame` encapsulates the whole game's state (i.e. current screens, `gameLevel`, and `gameState`), and thus the transitions of its states represent the dynamics when the player is playing the game. The general states are explained as follows:

- **Application window opened** - initially, the system tries to open a new application window. We assume that the program can always reach this state (relying on LibGDX).
- **Game file loaded** - next, the system tries to load the game from a file. If the game is correctly loaded, it transits to this state; otherwise, it goes to a terminating state. Once reaching this state, the game immediately transits to a new state (i.e., "Game running").
- **Game running** - a state where the player is playing the game and the user input is handled. During this state, the game screen is displayed. On entry of this state, the system enters a substrate where it is listening for a keypress by the player. Once a keypress is detected, the input is checked against the following guard conditions and the system transits to the corresponding states:
    - If the key pressed is "Spacebar", it transits to a decision state where it checks if the buffer is full. If the buffer is full, it does nothing and goes back to the listening state. If it is not full, the current selector value is added to the buffer.
    - If the key pressed is "R" or "Backspace", it examines whether the corresponding undo/redo stack is empty. If it is not empty, the game's state is restored (either undo or redo); otherwise, it simply returns to the listening state.

    - If the key pressed is “Up”/“Down”/“Left”/“Right”, it checks if the move is valid. If so, it moves the selector; elsewise, it does nothing and transits to the listening state.
    - If the key pressed is “Enter”, the current Game running state reaches its final state.
    - If any other key is pressed, the game’s state transits back to the listening state.
- **Game over** - after the Game running state reaches its final state, it transits to this state. Besides that, the exit from the “Game running” state can be triggered by two events, that is, either the timer is timeout or the player presses the “Escape” key. In this state, the score is displayed and the system waits for a keypress. As the player presses “Escape”, it exits the game and goes to the final terminating state.

The specific details of the actions taken by the system during the transitions are omitted for the sake of simplicity and understandability. These details are described in the Class diagram section as well as in the User Input Handling Sequence section.

SequenceGUI

The state machine diagram above depicts the transition of the internal states of the `SequenceGUI` class. This class is an example of a `GUIelement` class, which takes care of conditionally rendering to the screen. The transitions between its states reflect visual changes on the current screen. We chose to construct the state machine diagram for this class because it helps us to reason about the state transitions of other GUI classes which are similar to the sequence GUI.

The general states of `SequenceGUI` are described as follows:

- **UI at rest** - after the initial state where the sequence GUI element is created and initialized, it enters this state, during which the UI is waiting for re-renders, in which it triggers the exit event. Code-wise, this would be identical to the `render()` function getting called. On exit, the UI enters a decision state where different guard conditions are examined. If sequences are completed, the activity `change those sequences’ font color to green` is taken. If no sequences are completed, no activity is fired. The UI transits to the state `Completed Sequence UI up-to-date`.
- **Completed Sequence UI up-to-date** - the state where every correct sequence is highlighted green by the current values in the buffer. The UI immediately exits this state and continues to enter a decision state where it decides if any sequences have failed. If a sequence has failed, the activity `change those sequences' font color to red` is fired. If no sequences have failed, no activities are fired. The UI transits to the state `Failed Sequence UI updated`.
- **Failed Sequence UI up-to-date** - the state where every failed sequence is highlighted red by the current values in the buffer. The UI immediately exits this state and transits back to the UI-at-rest state.

During the states above, we use the fact that anything highlighted/drawn on the screen stays there (even though the corresponding state and activities have passed) until the screen is cleared at the start of each `render()` function of the current screen. Whenever the parent screen transits to its end state, this state machine immediately exits and terminates. Such transition is already described by the previous state machine diagram (`Game running` state and `do/show game screen`), and thus omitted here.

GameLevel

The state machine diagram above illustrates the transitions between different states of the `GameLevel` class. The error handling in this class is very simple: try to parse the file and throw exceptions on irregularities. The diagram clearly shows at which points during execution the loading of a file can fail.

Sequence diagrams

Author(s): Yingdi, Chris, Jop

In this section, we will describe 2 UML sequence diagrams for our game. The first diagram depicts the initialization and rendering of different game screens during the game. The second diagram illustrates the interaction between the player and our system while the keyboard input from the player is handled.

Game Rendering Sequence

software-design-vu/Sequence Diagram-Rendering.png

The sequence diagram above depicts the interaction between the different screens and the multiple GUI elements that are part of our system. This diagram covers the initialization of the 'GameScreen' and the 'GameOverScreen', which extend the `ScreenAdapter` class. Both of these screens actively use multiple `GUIelements` such as `SequenceGUI`, `BufferGUI`, `GridGUI, `TimerGUI, `ScoreGUI`.

In this diagram, we assume that the BreachGame object has just been made. We intentionally hide the interaction with the GameState class and user input (like how escape can terminate current execution), and obscure the conditions necessary to switch between the `ingame` and `on endscreen` such as to emphasize the rendering sequence. These conditions will be properly described in User Input Handling Sequence. The rendering objects used by the different GUI elements (such as SpriteBatch and BitmapFont) are also hidden to make the diagram more understandable (these objects also don't add more value to the diagram).

Below is a more detailed description of how the screens are rendered:

- GameScreen initialization and rendering:
    - The `breachGame` object invokes the `gameScreen` object, which in its constructor starts initializing the to be used GUI elements: `gridGUI`, `sequenceGUI`, `bufferGUI`, and `timerGUI`.
    - Once the `gameScreen` object has fully initialized, the `breachGame` object decides to hand over execution to the `gameScreen` object, by calling `setScreen(gameScreen)`.
    - LibGDX now starts the rendering sequence by calling `show()` on the current screen: `gameScreen`. The `gameScreen` object forwards this call to all the used GUI elements as `create()`. These calls are used for initializing rendering objects such as `SpriteBatch` and `BitmapFont` (omitted in this diagram).
    - LibGDX now starts to repeatedly (loop until not ingame) call `render()` on the current screen: `gameScreen`. This call is again forwarded to all of the used GUI elements. The GUI elements each internally use a `GameState` to draw on the screen (see the GameState's associations).
- The GameOverScreen initialization is very similar to the GameScreen initialization with the following exceptions:
    - Instead of the `breachGame` object, the `gameScreen` object initializes the next screen: `gameOverScreen`.
    - The `gameOverScreen` only initialized the following GUI elements: `sequenceGUI`, `bufferGUI`, and `scoreGUI`.
    - After setting the screen to `gameOverScreen`, the `gameScreen` is terminated by LibGDX.

    - The loop renders while the endscreen needs to be shown.

Interestingly, notice how the `gameOverScreen` is initialized by `gameScreen`. If centralized screen decision making is advised, this could be implemented in the BreachGame class.

User Input Handling Sequence

[software-design-vu/Sequence Diagram-User Input.png]

The sequence diagram above depicts the interaction between the player and 2 integral parts of our system (i.e., “libGDX” and the “GameState” class) following the initialization of the game. During the game, the user input handling sequence must run in a loop where the player sends a user input by pressing a key and the game handles the user input depending on which key the player pressed. This input-process-and-output loop must iterate at least once and should not end until the timer triggers a timeout or the player presses a key that terminates the game or advances to the next screen.

Specifically, once the game is correctly loaded and initialized, we assume that the relevant components of the game such as the `BreachGame` and `GameScreen` classes already exist. The class that handles the player’s input is `InputAdapter`, initialized by the `ScreenAdapter` class, both included in the libGDX library. Once it detects a keypress, the sequence of interactions takes place depending on the specific key. If the key pressed is a meta-command (i.e., “Escape”), the libGDX terminates the application; elsewise, it adapts the game’s state by invoking the corresponding operation in the `GameState` class where the internal state of `GameState` is modified. Then, the updated UI is presented to the player. In the case of an “Enter” keypress, the GameState is used to display the GameOverScreen. This sequence runs within a maximum amount of time set by the `TimerLogic` class.

Below is a more detailed description of handling the player’s input through the corresponding operations of the `GameSate` class:

- If the pressed key is “Up”, “Down”, “Left” or “Right”, the `move()` method of `GameState` is invoked, where in case of a valid move, the selector is advanced, and in case of an invalid move, the input is ignored.
- If the pressed key is “Spacebar”, the `confirmSelector()` method is invoked where the timer will start off if it has not started yet. Besides, if the buffer is not full, the selector is added to the buffer and the undoStack is emptied which effectively clears up the undo history. If the buffer is already full, the input is ignored.
- If the pressed key is “Backspace”', the `undo()` is called where if the buffer is not empty, the top element of the buffer is popped and pushed onto the undoStack. In case of an empty buffer, the input is ignored.
- If the pressed key is “R”', the `redo()` is called where if the undoStack is nonempty, its top element is popped and pushed onto the buffer. Thus, the game is reverted to the last undo-ed selection. In case of an empty undoStack, the input is ignored.
- If the pressed key is “Enter”, the game screen is set to `GameOverScreen`, which is displayed to the player.

A detailed explanation of the handling of the player’s meta-commands is as follows:

- If the pressed key is “Escape”, the `exit()` operation of the `Application` class in the LibGDX library is immediately invoked and the application is terminated.

The descriptions above assume that the player sends a meaningful input to our system which will then trigger the handling sequence. The `InputAdapter` class ensures that in case of other keypresses from the player, the user input will be ignored and the game's state will stay consistent across the timeline.

Implementation

Author(s): Tibi, Yingdi

Demo: https://youtu.be/U79jK943GF8

UML to Implementation Strategy

During the implementation of our system, we followed the "Agile Principle" where design, development and testing are essentially continuous activities. The system evolved iteratively and went through testing and modification at every stage of the development. Specifically, the class diagram is the starting point of our implementation where we translated every class in the diagram into a java class and the associations between classes were also realized in the code. The state machine diagrams served as a guide for us to implement the operations within each class which altered the state of the class in the way depicted in the diagram. The sequence diagrams helped us to reason about the interactions between different components of the system during the timeline of the game. Thus, the diagrams provided a prescriptive model for our system which describes its structure and behavior and also instructs us to build it up. The diagrams did not stand fixed during the development. In case that we detected some issues or discovered a better solution during the coding, the diagrams were adapted. Redundant attributes were removed, and new operations were added. It offers an opportunity for us to reason and check the correctness of the process again before the next iteration of the implementation.

The team split the workload and implemented the features according to the documentation of Assignment 1. Thus, each of us worked as a champion for some feature(s) and led the implementation of the feature(s). Specifically, three members first created the skeleton of the codebase. Then, "Todo" comments along with the name of the feature champion were added at the location where the actual implementation should take place. It helped us to divide the work in a clear and even manner, and when issues were encountered, they were discussed in the next team meeting.

Key Solutions for Implementation

- **Game Logic**: We implemented the game logic using two classes: `GameLevel` and `GameState`. `GameLevel` keeps track of level-specific information, namely the grid and its size, the sequences the user has to complete and the length of the buffer. This information is then used by `GameState` to perform moves according to user input. `GameState` keeps track of the current selector position, the cells the user has already selected and the undo's the user has made (in order to provide redo functionality; see below). Using `GameState`, all `ScreenAdapter` objects can properly render the GUI for the user.

- **Selector movement**: the selector is an instance of the class `Cell`. Moving the selector in a direction can be easily done through the function `applyDir` from the `Cell` class, which returns a new `Cell` object. There are some checks in place that prevent the user from moving the selector illegally: out-of-bounds checking and horizontal/vertical constraining (i.e. the user is only allowed to move the selector either on the same line or the same column). The latter check is performed using the current length of the buffer (i.e. if the buffer has even length, then the user can only move horizontally, otherwise vertically).

- **Undo/Redo**: the `GameState` object keeps track of the cells the user has selected in the `buffer` stack. Should the user want to undo a cell selection, then the buffer the last cell introduced in the buffer is popped and pushed into `undoStack`. If the user now wants to redo a series of selects, then the reverse operation happens: the top of `undoStack` is popped and then pushed in the `buffer` stack. Once the user performs a select, `undoStack` is cleared, since we can no longer perform a redo operation.
- **GUI**: The GUI is rendered from the `ScreenAdapter` `render()` function that `GameScreen` and `GameEndScreen` implement, which individually calls the `render()` function of each `GUIElement` that has to be rendered.
- **Timer**: when the timer is started, the `TimerLogic` object remembers the start time. Subsequent calls to the `timeLeft()` method can be satisfied using the formula `time_left = start_time + total_time - current_time`, where the current time is obtained through a call to `System.currentTimeMillis()`. `hasStarted()` can then be implemented by checking if `startTime` is greater than 0 (i.e. the `start()` method had been previously called).

Main Execution

The execution of the program starts in the class main.java.Main, where the `main()` method is called. It takes the level file name as argument and creates a `BreachGame` object using this filename.

Jar Location

The JAR file is outputted by Gradle in `/out/artifacts/software_design_vu_2020_jar` and is called `software-design-vu-2020.jar`.

Time logs

Along the way, we forgot to keep track of the time log. The latest version is shown below. If more details are really required, we can give you our Google Docs history and Git commit log.

Team number	14		
Member	**Activity**	**Week number**	**Hours**
Everyone	Group meeting with TA	3	0.5
Jop	Class Diagram	4	2
Jop	bde3a5f7aee290d828fa6efff98ed11beda9bbec	4	2
Tiberiu	Diagrams	4	6
Everyone	Group work on Discord	5	40
		TOTAL	50.5