Huu Trinh Kha Le - 1519529
Minh Thuan Nguyen - 1520008
Duy Phuc Ho - 1524048

# Make ’N’ Break

(Winter Semester 2023-2024)

Course Name: OOP/Java
Instructor’s Name: Doina Logofatu

February 9, 2024

Springer Nature

# Declarations of Authorship

We, group UNBREAKABLES, hereby certify that the project report we are submitting is entirely our own original work except where otherwise indicated. We did not submit this work anywhere else before. We are aware of the University's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Signed:

1. Huu Trinh Kha Le
2. Minh Thuan Nguyen
3. Duy Phuc Ho

Date:

09.02.2024

# Contents

# Chapter 1
# Team Work

## 1.1 Team's member

- **Huu Trinh Kha Le**
  - Create menu page
  - Link menu with game play
  - Design poster (90%)
  - Write report (33%)
- **Minh Thuan Nguyen**
  - Gameplay
  - Design Logic game
  - Design poster(10%)
  - Write report (33%)
- **Duy Phuc Ho**
  - Gameplay
  - Countdown timer, Score
  - Ending scene
  - Write report (33%)

## 1.2 Communication and Collaboration

Regular team meetings via Google Meet and Discord. Furthermore, Messenger is also used for communication between group members.

## 1.3 Decision making

All decisions are approved by the group and noted in a google doc.

## 1.4 Technology

- **Project Management Tools:** Google doc, Google sheet for task tracking
- **IDE:** Android Studio, IntelliJ IDEA
- **References:** Youtube, Tetris Game and LibGDX library
- **Collaboration:** Github
- **Communication tools:** Google meet, Discord and Messenger

## 1.5 Ideas

Make 'N' Break is a challenging but interesting game, especially for ages to enjoy. Some elements were designed on Canva, some were searched on Google. The LibGDX library is used to provide a smoother game experience. Operations are used from the mouse and keyboard. Make 'N' Break promises to be an attractive game for children and teenagers that helps increase reflexes.
Initially we have 3 approach to implement this game in Java using LibGDX:

- We can code every single block of the grid using x and y coordinates. This way we need to detect where the mouse is in the event of a click and change the block accordingly. This would be the most straight forward approach to this game.
- Create a grid of button object which has its own methods. By using built-in libraries of LibGDX framework, we could work around this idea and create some sort of game out of the available properties. This way we can utilize the tools of our chosen framework.
- Implement a drag and drop where the pieces is the object that can be put in an defined area. With the provided methods and libraries in LibGDX, this could be an interesting take on this game and even closer to the original game. But it would take a lot of work to implement some kind of playable area since there are a lot of possibilities or situations that could happen that may cause a error.

Finally, we decided to create a table of button object which has its own methods.

# Chapter 2
# Introduction

## 2.1 General topic

This research project delves into the fundamentals and advanced techniques of 2D game development using the Java programming language. It covers essential aspects such as graphics rendering, user input handling, and game mechanics, providing a comprehensive understanding of the intricacies and challenges involved. Both theoretical concepts and practical implementation are explored, offering developers a well-rounded learning experience.
The focus of this report is on the design and development journey of Make 'N' Break, a 2D game implemented using the LibGDX library for Java. Throughout the investigation, we examine the distinctive attributes of Make 'N' Break, showcasing the fusion of scientific design principles and innovative ideas. By exploring the decision-making process, technology choices, and creative considerations behind the game, we gain valuable insights into the thought and effort invested in its development.
The study of Make 'N' Break provides a practical example that demonstrates how the Java programming language and the LibGDX library can be used to create engaging and interactive 2D games. By analyzing the design choices and development strategies employed in Make 'N' Break, developers may acquire knowledge and experience that can be applied to their own game projects. This research project may serves as a valuable resource for aspiring game developers, offering insights, lessons learned, and a solid foundation for creating compelling games using Java and the LibGDX library.

## 2.2 Definitions

Make ’N’ Break is a tile-based puzzle game that challenges players to construct various structures using a set of predefined blocks. The objective of the game is to arrange the given blocks in a specific pattern within a given time limit. The game provides players with a grid-like playing area, where they can manipulate and arrange the blocks strategically to achieve the desired structure.

## 2.3 Classification

Make ’N’ Break falls under the category of puzzle games and specifically belongs to the tile-based puzzle genre. The game requires logical thinking, spatial awareness, and problem-solving skills to successfully complete each level. It offers a captivating and intellectually stimulating experience, making it popular among puzzle enthusiasts of all ages.

## 2.4 Game content

**Make ’N’ Break is an exciting but challenging for people from eight years old.**
**Components:**
8 colored bricks, a 7x6 grid, timer, building cards and score calculation
**Game objective:**
Players will have 90 seconds to build as many buildings as possible and score the most points. Sample structures will be available on the cards and players must build their structures similar to the cards.

## 2.5 Gameplay

After click ”Let’s build” button, players will be moved to gameplay screen. Timer starts to countdown from 90 second. They look the sample card carefully, and build structures like the sample card.
To build the structures, players need to choose color from eight colored bricks. Then, they start to build structure on the 7x6 grid. On the grid, they click left to place the brick horizontally and click right to place vertically. Finally, the player presses the spacebar on the keyboard to check the results.
**If correct:** 10 points will be added and the new sample card will appear.
**If wrong:** the grid is cleared and players need to build again.
**The end of the game:** The game end after 90 second and players will know their total score on the end screen.

## 2.6 Example

### 2.6.1 Menu screen

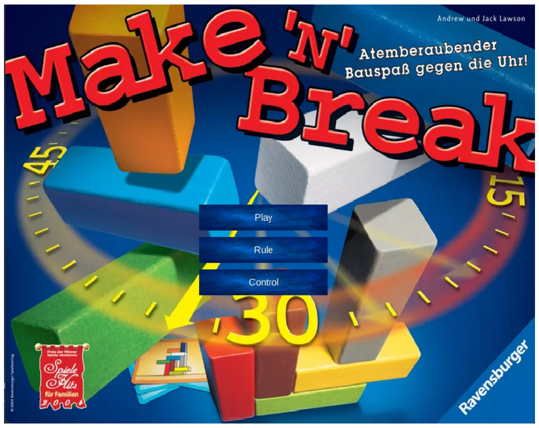


**Fig. 2.1** Menu page

### 2.6.2 Control screen

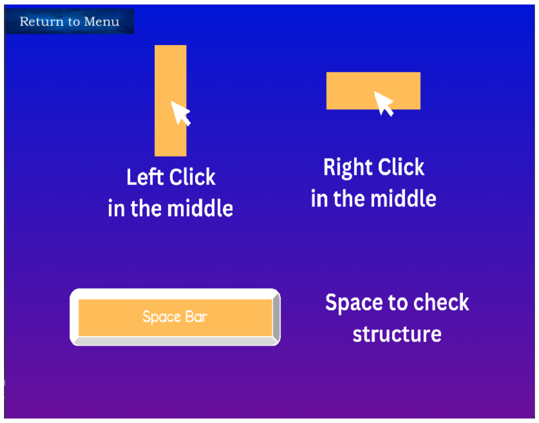
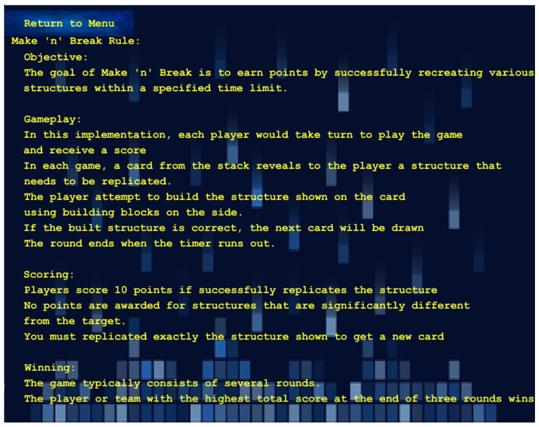


**Fig. 2.2** Control page

### 2.6.3 Rule screen



```
Make 'n' Break Rule:
  Objective:
  The goal of Make 'n' Break is to earn points by successfully recreating various
  structures within a specified time limit.

  Gameplay:
  In this implementation, each player would take turn to play the game
  and receive a score
  In each game, a card from the stack reveals to the player a structure that
  needs to be replicated.
  The player attempt to build the structure shown on the card
  using building blocks on the side.
  If the built structure is correct, the next card will be drawn
  The round ends when the timer runs out.

  Scoring:
  Players score 10 points if successfully replicates the structure
  No points are awarded for structures that are significantly different
  from the target.
  You must replicated exactly the structure shown to get a new card

  Winning:
  The game typically consists of several rounds.
  The player or team with the highest total score at the end of three rounds wins
```

**Fig. 2.3** Rule page

### 2.6.4 Ready screen

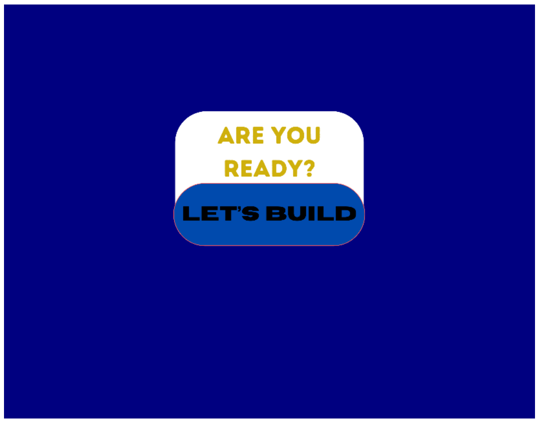


**Fig. 2.4** Ready page

### 2.6.5 Gameplay screen

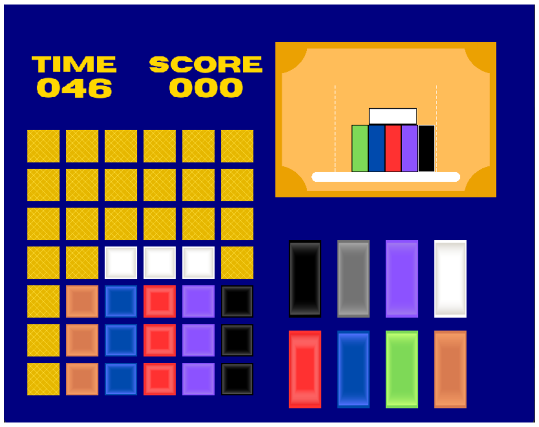


**Fig. 2.5** Gameplay page

### 2.6.6 End screen

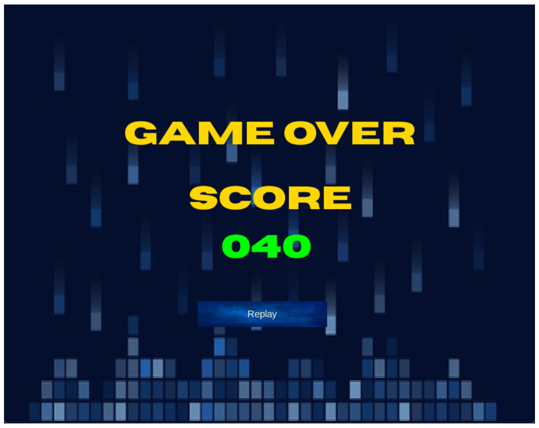


**Fig. 2.6** End page

# Chapter 3
# Problem Description

In the process of developing, we have encounter a number of unforeseen obstacles:

- During the development process, we encountered several challenges that required us to find alternative solutions and seek assistance from various resources. One initial difficulty arose when we realized that our team members were using different IDEs (integrated development environments). Specifically, Kha's Android Studio encountered issues that prevented its use, prompting him to switch to IntelliJ as an alternative.
- Additionally, as this was our first encounter with the LibGDX framework, we faced difficulties in finding comprehensive and easily understandable reference materials and videos online. Many of the available resources lacked clarity. Fortunately, we came across a helpful video tutorial on YouTube that demonstrated the creation of a Mario game using the LibGDX framework. This tutorial provided valuable insights and knowledge about working with the framework.
- Another challenge we encountered was related to coding errors that significantly impacted system performance by consuming excessive RAM. These errors caused the computer to lag, hindering our ability to perform tasks in parallel. To resolve this issue, we turned to stackoverflow.com, a popular website for developers, to find solutions and guidance from the programming community.
- Furthermore, we faced a minor issue regarding the selection of suitable images for our game. Many of the images we found online were of low resolution, resulting in a blurred background within the game. To address this problem, we utilized Canva, a graphic design tool, to create and customize certain elements of the game.
- This is a first time we have come across a framework such as LibGDX so it took us a great amount of time to understand and progress through with the project.
- To make a playable area for all the pieces is a hazard since every ideas seems out of reach. Whether is was too hard, unrealistic, not module-able or even straight up too complicated. Beside cycle between ideas, interpret it into code poses another big complication.

# Chapter 4
# Related Work

In the development of Make 'N' Break, several existing works and projects have been explored, providing insights and inspiration for the design and implementation of various components. The related work can be categorized into the following areas:

## 4.1 Game Development Frameworks and Libraries

Numerous game development frameworks and libraries have contributed to the foundation of Make 'N' Break. LibGDX is the chosen platform which has provided essential tools and functionalities for graphics rendering, user input processing, and overall game development architecture.

## 4.2 Educational Game Design

Make 'N' Break draws inspiration from the field of educational game design. Works such as Tetris, Chess have explored the integration of gameplay mechanics with educational content, influencing the approach taken to engage and educate players effectively.

## 4.3 Puzzle and Quiz Game Genres

The puzzle and quiz game genres have been influential in shaping the gameplay mechanics and challenges present in Make 'N' Break. Games like Blocks have provided valuable insights into creating engaging and thought-provoking game scenarios.

## 4.4 User Interface and Experience Design

The user interface and experience design of Make ’N’ Break have been informed by studies in interactive design and user experience. Works such as ”Don’t Make Me Think” by Steve Krug have guided decisions related to menu design, feedback mechanisms, and overall player engagement.

## 4.5 Color Theory and Aesthetics in Games

Exploration of color theory and aesthetics in game design has played a crucial role in determining the visual aspects of Make ’N’ Break. Studies such as Overview of Color Theory have influenced decisions related to color schemes, visual appeal, and the overall atmosphere of the game.

## 4.6 Previous Projects in Similar Domains

Projects with similarities to Make ’N’ Break have been examined to identify successful strategies and potential pitfalls. Chess and Tetris have provided valuable lessons and ideas in areas such as game mechanics, level design, and player engagement.

# Chapter 5
# Proposed Approaches

## 5.1 Input/Output Format

In this game, player interaction is facilitated through three primary input methods: the space bar, right-click, and left-click.

- **Space bar** performs an important role as a means for evaluating the player's work inside the grid. When you press the space bar, the game transforms the current playing grid into an array containing the color information for each square on the grid. This array is then compared to the specified answer, allowing the game to determine the accuracy of the player's selections. The space bar assumes the role of a virtual judge, holding the authority to evaluate and decide the correctness of the player's actions within the grid. And base on the judgement, the player can progress to the next card or reattempt the current card until correctness is achieved.
- **Left Click and Right Click** are the primary controls which dictate the player's interactions with given pieces. The left-click function serves as the mechanism to place the provided pieces in a vertical orientation within the grid. On the contrary, the right-click function is used to horizontally "color" the grid. This distinction in control mechanisms adds a layer of strategy to the gameplay, allowing players to make decisions on both piece placement and grid coloring. The use of the mouse provides a straightforward and accessible means for players to interact with the game environment, allowing them to effortlessly grasp and manipulate the given pieces. Despite the initial simplicity, the game adopts a "easy to grasp, hard to master" approach, it will be beneficial for both the players who just pick up the game and those who spend hours master the control.

The game heavily relies on graphic to provide a seamless and immersive experience for players. Therefore, the visual output on the screen should accurately reflect the player's input interactions. When a player clicks on a block using either the left or right mouse button, the corresponding block should change color according to the player's choice, or the move should be denied if it attempts to modify a block at the edge of the grid. If the left-click option is chosen, the game must promptly update the color of the selected block, as well as the blocks above and below it, in response to the player's input. Respective the same to the right click. This visual feedback should be displayed on the screen quickly and responsively to ensure smooth gameplay. Additionally, verifying the player's work is crucial for maintaining the integrity of the game. Upon pressing the Space bar, the game should promptly load and compare the current state of the grid with the correct answer. If the player's input matches the solution, appropriate sound feedback should be provided to indicate success. If the player's input is incorrect, a sound indicator should alert the player, indicating the discrepancy between their input and the expected outcome. This feedback mechanism enhances player engagement and fosters a sense of accomplishment when successfully completing a puzzle.

## 5.2 Why LibGDX Library?

LibGDX is an open-source game development framework written in Java. It provides a comprehensive set of tools and libraries for developing cross-platform games, supporting desktop, Android, iOS, and HTML5 platforms. LibGDX follows a modular design philosophy, enabling developers to structure their games in a clean and organized manner. It promotes the use of composition over inheritance, encouraging developers to build functionality by combining smaller, reusable components.

In the scope of this project, we have identified LibGDX as the perfect library for learning and exploration. LibGDX's deep-rooted approach in Object-Oriented Programming (OOP) aligns with the knowledge we've accquired in this module. The framework strongly advocates OOP principles, making it an ideal environment for us to put theoretical concepts learned in lectures into practical application. Through its support for encapsulation, inheritance, polymorphism, abstraction, and composition, LibGDX empowers us to structure our game with modularity and organization. The event-driven programming model for user input and game events further reinforces OOP principles, allowing us to create a well-designed and maintainable codebase.

In spite of its cross-platform capability, we've only used this library mainly to develop game which targeted PC player.

## 5.3 Algorithms

### 5.3.1 Basic Structure of LibGDX

LibGDX offer a wide range of highly customizable interfaces to create objects and screens. There are some key concept that required understanding before dive into the Algorithm. Each of these concepts plays a vital role in the design, implementation, and optimization of games using the LibGDX framework.

- `Screen`: LibGDX follows a screen-based approach for managing different states or screens within a game. Each screen typically corresponds to a specific game state, such as the main menu, gameplay, options menu, etc. `Screen` in LibGDX is defined by the Screen interface. To create a screen, developers implement this interface and define the methods such as `show()`, `render()`, `resize()`, `pause()`, `resume()`, and `hide()`. One of the most important method in `Screen` is `render() - render(float delta)`: Called continuously to update and render the screen. The delta parameter represents the time elapsed since the last frame.
- `Stage`: The `Stage` class in LibGDX represents a 2D scene that contains actors. It serves as a container for managing and rendering visual elements, handling input events, and applying transformations. The Stage class manages a hierarchical structure of visual elements known as the scene graph. This structure organizes actors (visual entities) in a tree-like hierarchy, enabling efficient rendering and manipulation of objects within the scene. The Stage handles rendering of its actors by set up the drawing process.
- `Actor`: Actors serve as the building blocks of the scene graph and can represent a wide range of visual elements such as sprites, images, text, buttons, and more. Actors are responsible for rendering visual content on the screen. They encapsulate information such as position, size, rotation, scale, color, ... Actors are typically added to a Stage, which manages their rendering, input handling, and lifecycle. Some noticeable Actor classes with which we utilized in this project are: `Table`, `Button` and `ImageButton`
- `Input Handling`: Input events generated by keyboards and mice can trigger actions or events within the game, such as player movement, character interaction, or menu selection. By implementing the `InputProcessor` interface, developers can define custom logic to handle specific input events.

### 5.3.2 Algorithms in Pseudocode

#### 5.3.2.1 Game Screen

```
# 1. Load in music and sound effects
    # 1a. Create music with the mp3 file
        music = new_music("background_music.mp3")
        music.setLooping(true)
        music.setVolume(0.7)
        play_music(music)
    # 1b. Create sound effects for correct, false, and victory.
        correct_sound = new_sound_effect("correct_sound.mp3")
        false_sound = new_sound_effect("false_sound.mp3")
        victory_sound = new_sound_effect("victory_sound.mp3")

# 2. Create the playing grid
    # 2a. Initialize Stage object
```

```
        stage = new Stage()
    # 2b. Initialize Grid object
        rows = 7
        columns = 6
        customButtons[][] = create_empty_custom_buttons_array(
            rows, columns)
        FOR each row (i) FROM 1 TO rows
            FOR each column (j) FROM 1 TO columns
                customButtonsArray[row][column] = new
                    CustomButton(this_grid,i*rows+j)
                grid.add_custom_button(customButton, dimensions
                    ="50px*50px", padding="5px")
            END FOR
        END FOR
    # 2c. Set Grid position in Stage
        grid.setPosition(x,y)
        stage.add(grid)

# 3. Create the Piece color selector.
    # 3a. Define an array of color options in the game.
        colorOptionsArray = {red, blue, green, ...}
    # 3b. Create for each element in Color array a ColorSelector
        object that corresponds to each color.
    # 3c. Set ColorSelector object's position in the stage to
        create a 2*4 array.
    # 3d. Set ColorSelector object's size to 50px*120px.
    # 3e. Add that object to the stage.
    FOR each color in colorOptionsArray
        colorSelector = new ColorSelector(color, texture)
        colorSelector.setPosition(x,y)
        colorSelector.setSize(50px,120px)
        stage.add(ColorSelector)

# 4. Initialize a Quiz object
    # 4a. Create an array filled with the ids of quizzes.
        quizzes[] = {1, 2, 3, 4, 5, ...}
    # 4b. Shuffle that array.

# 5. Setup Input Processor (Input handles) for the screen
    # 5a. Create an Input processor for check button
    # 5b. Create a Multiplexer for setting multiple input
        processors
    # 5c. Add input processors to Multiplexer
    # 5d. Set the game input to Multiplexer.

# 6. Create a scoreboard and timer
    # 6a. Create a Table and set its position.
    # 6b. Initialize timer variables.
    # 6c. Define labels such as timeLabel, countdownLabel,
        scoreLabel and add them to the table.
    # 6d. Add the table to the stage.

# 7. Game loop:
    WHILE game_not_over:
        # Update game logic
            IF change == true:
                IF img != null:
                    dispose_texture(img)
                img = assign_new_card()
            END IF
            # Update the timer
            IF one_second_has_passed:
                IF world_timer > 0:
                    world_timer--
                ELSE:
                    game_over()
            END IF
            # Update the labels

        # Render a clear color background
        draw_clear_color_background()

        # Draw the card
        draw_card(img)

        # Draw the stage and its components
        draw_stage(stage)
```

#### 5.3.2.2 Checking Algorithm

```
#Algorithm to check if player's selection is correct:
# Iterate through each button in the grid
FOR each button IN currentGrid:
    # Add the color of the button to the currentGrid array
    currentGrid.add(button.color)

# Read the answer from the corresponding txt file
ans = read_file("quiz" + quizNumber + ".txt")

# Compare the currentGrid array with the answer array
IF currentGrid.equals(ans):
    # The player has answered correctly
    PRINT "Correct!"
ELSE:
    # The player has answered incorrectly
    PRINT "Incorrect!"
```

#### 5.3.2.3 Placing a piece

```
#Algorithm to color the grid after a click:
# If it's a right-click
IF right_click:
    # Check if the clicked button is not at the edge of the grid
    IF position_of_clicked_button_not_at_edge:
        # Get the color of the adjacent buttons
        rightColor = getColor(row, col + 1)
        leftColor = getColor(row, col - 1)

        # Check if the colors match the button's color
        IF rightColor == leftColor == buttonColor:
            # Check if the colors match the specific condition (e
                .g., YELLOW)
            IF rightColor == YELLOW:
                # Set the color of the three buttons to
                    GlobalState color
                setColor(row, col, GlobalState.selectedColor)
                setColor(row, col + 1, GlobalState.selectedColor)
                setColor(row, col - 1, GlobalState.selectedColor)
            ELSE:
                # Reset the color of the three buttons to YELLOW
                setColor(row, col, YELLOW)
                setColor(row, col + 1, YELLOW)
                setColor(row, col - 1, YELLOW)
        ELSE:
            # Invalid Move
            PRINT "Invalid Move"
    ELSE:
        # Invalid Move because it is at the edge
        PRINT "Invalid Move"
ENDIF

# This algorithm is the same for left-click with the
    corresponding top and bottom buttons
```

### 5.3.3 Color picker

```
#Algorithm for choosing a color: Create a Class that store the
    information on the color picking
# Event handler for color selection button click
colorSelector.addClick{
    # Set the global variable selectedColor to the color of the
        clicked button
    GlobalState.selectedColor = button.color
}
```

# Chapter 6
# Implementation Details

## 6.1 Tools

### 6.1.1 LibGDX

LibGDX is a powerful open-source game development framework that provides developers with powerful tools and libraries to build cross-platform games and interactive applications This allows developers to write code once with their creations Windows, macOS, Linux, Android, iOS, . a broad range of other web browsers and platforms can be used.

One of the key features of LibGDX is its focus on performance and performance. It harnesses the power of hardware acceleration and uses a low-level API to ensure that gameplay is smooth and functional even on less feature-rich devices this makes it an ideal choice for games that need to run well on meetings.

LibGDX offers a wide range of features and modules for graphics rendering, input handling, audio playback, file loading, and more. It provides a simple and intuitive API that makes it relatively easy for developers to create and manage complex game objects, scenes, and assets. In addition, LibGDX supports popular programming languages such as Java and Kotlin, making it accessible to a wide range of developers.

Another notable feature of LibGDX is its dynamic and supportive community. Developers can access extensive documentation, tutorials, and sample projects that help them get started quickly. The community also actively contributes to the development of LibGDX, providing bug fixes, enhancements, and other features.

### 6.1.2 Android Studio

Android Studio is an official integrated development environment (IDE) provided by Google for application development.

Android Studio provides an intuitive and user-friendly interface that facilitates smooth navigation and access to tools and features. It includes a code editor with intelligent code completion and syntax highlighting.

Android Studio uses the Gradle build system, which allows developers to define and manage dependencies, build configurations, and create custom projects. It simplifies the process of building, testing and packaging applications.

Android Studio has a powerful layout editor that allows developers to visually create their app's user interface. It supports drag-and-drop functionality and provides real-time previews, making it easy to create and modify settings.

Android Studio offers an emulator that allows developers to test their applications on virtual Android machines with different systems. It also supports the integration of physical devices for testing and maintenance.

Android Studio provides powerful debugging and profiling tools that help developers identify and fix problems in their applications. It allows step-by-step debugging, breakpoints, and performance analysis to optimize app performance.

## 6.2 Make 'N' Break

### 6.2.1 Main Menu Screen

```
package Screens;
```

The Main Menu Screen locates in Screens package.

```
import com.badlogic.gdx.Game;
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.audio.Music;
import com.badlogic.gdx.audio.Sound;
import com.badlogic.gdx.ScreenAdapter;
import com.badlogic.gdx.graphics.GL20;
import com.badlogic.gdx.graphics.OrthographicCamera;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import com.badlogic.gdx.graphics.g2d.TextureRegion;
import com.badlogic.gdx.scenes.scene2d.InputEvent;
import com.badlogic.gdx.scenes.scene2d.Stage;
import com.badlogic.gdx.scenes.scene2d.ui.TextButton;
import com.badlogic.gdx.scenes.scene2d.utils.ClickListener;
import com.badlogic.gdx.scenes.scene2d.utils.TextureRegionDrawable;
import com.badlogic.gdx.utils.viewport.FitViewport;
import com.badlogic.gdx.utils.viewport.ScreenViewport;
import com.badlogic.gdx.utils.viewport.Viewport;
import com.makeandbreak.game.MakeAndBreak;
import java.io.FileNotFoundException;
```

These lines import various classes and packages needed for the code.
They include game-related libraries from LibGDX, such as graphics, audio, and UI components.

```
public class MainMenuScreen extends ScreenAdapter {
```

This line begins the definition of the **MainMenuScreen** class, which extends **ScreenAdapter** from LibGDX.

```
private final MakeAndBreak game;
private Stage stage;
private BitmapFont font;
private OrthographicCamera gamecam;
private Viewport gameport;
private Music music;
private Sound clicksound;
private Texture img = new Texture(Gdx.files.internal("anhgame.png"));
```

These lines declare class variables:
**game** is a reference to the main game instance.
**stage** is a UI stage for handling buttons and other UI elements.
**font** is a bitmap font for rendering text.
**gamecam** is an orthographic camera.
**gameport** is a viewport for managing the screen's dimensions.
**music** is a background music instance.
**clicksound** is a sound effect instance.
**img** is a texture for the background.

```
public MainMenuScreen(MakeAndBreak game) {
this.game = game;
gamecam = new OrthographicCamera();
gameport = new FitViewport(1520, 1200, gamecam);
}
```

This is the constructor for MainMenuScreen. It initializes the game reference, sets up the camera, and creates a viewport.

```
@Override
public void show(){
    ...
    ...
}
```

This method is called when the screen becomes the current screen. It contains the setup code for the main menu.

```
music = Gdx.audio.newMusic(Gdx.files.internal("mainmenu_msc.mp3"));
music.setLooping(true);
music.setVolume(0.5f);
music.play();
```

This initializes the background music.
It loads the music file, sets it to loop, adjusts the volume, and starts playing it.

```
clicksound = Gdx.audio.newSound(Gdx.files.internal("clicksound.mp3"));
```

This initializes the click sound effect.

```
font = new BitmapFont();
stage = new Stage(new ScreenViewport());
Gdx.input.setInputProcessor(stage);
```

These lines initialize the bitmap font, create a new stage for UI elements, and set the input processor to the stage.

```
TextButton playButton = createButton("Play", 300,
  Gdx.graphics.getHeight() - 350);
TextButton controlButton = createButton("Control", 300, game.HEIGHT - 450);
TextButton ruleButton = createButton("Rule", 300,
  Gdx.graphics.getHeight() - 400);
```

These lines create three **TextButton** instances using a helper method **createButton**.
The buttons are used for switching to Play Screen, Control Screen, and Rule Screen.

```
private TextButton createButton(String text, float x, float y) {
    TextButton.TextButtonStyle style =
      new TextButton.TextButtonStyle();
    style.font = font;

    TextButton button = new TextButton(text, style);
    button.setPosition(x, y);
    style.up = new TextureRegionDrawable(new TextureRegion(new
      Texture(Gdx.files.internal("blueBackground.jpg"))));
    button.setWidth(200); // Set your desired width
    button.setHeight(40); // Set your desired height

    style.fontColor = null;

    return button;
}
```

This is the helper method **createButton** for generating a **TextButton** with specified position, text font, size and background.

```
playButton.addListener(new ClickListener() {
    @Override
    public void clicked(InputEvent event, float x, float y) {
        music.stop();
        clicksound.play();
        game.setScreen(new BufferScreen((MakeAndBreak) game));
    }
});
```

This adds a click listener to the **playButton**.

When clicked, it stops the music, plays the click sound, and sets the game screen to **BufferScreen**.

When the **ruleButton** is clicked, it stops the music, plays the click sound and sets the game screen to **RuleScreen**.

When the **controlButton** is clicked, it stops the music, plays the click sound and sets the game screen to **ControlScreen**.

```
stage.addActor(playButton);
stage.addActor(ruleButton);
stage.addActor(controlButton);
```

These lines add the buttons to the stage.

```
@Override
public void render(float delta) {
    ...
    ...
}
```

This method is called continuously to update and render the screen.

```
Gdx.gl.glClearColor(0, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
```

These lines clear the screen with a black color.

```
game.batch.begin();
game.batch.draw(img, -0, -0, Gdx.graphics.getWidth(),
  Gdx.graphics.getHeight());
game.batch.end();
```

This draws the background texture using the game batch.

```
stage.act(Math.min(Gdx.graphics.getDeltaTime(), 1 / 30f));
stage.draw();
```

These lines update and draw the stage.

```
@Override
public void dispose() {
    game.batch.dispose();
    img.dispose();
    font.dispose();
    stage.dispose();
}
```

This method is called when the screen is no longer needed. It disposes of resources to free up memory.

```
@Override
public void resize(int width, int height) {
    gameport.update(width, height);
}
```

This method is called when the screen is resized, updating the viewport dimensions.

### 6.2.2 Rule Screen

When the **Rule** button is clicked, the screen switches to Rule Screen.

This screen uses same packages as Main Menu Screen.

```
public class RuleScreen extends ScreenAdapter {
```

This line begins the definition of the **RuleScreen** class, which extends **ScreenAdapter** from LibGDX.

```
private final MakeAndBreak game;
private BitmapFont font;
private OrthographicCamera camera;
private Texture backgroundTexture;
private Stage stage;
private Music music;
private Sound clicksound;
```

These lines declare class variables:
**game** is a reference to the main game instance.
**font** is a bitmap font for rendering text.
**camera** is an orthographic camera.
**backgroundTexture** is a texture for the background.
**stage** is a UI stage for handling buttons and other UI elements.
**music** is a background music instance.
**clicksound** is a sound effect instance.

```
public RuleScreen(MakeAndBreak game) {
    this.game = game;
}
```

This is the constructor for **RuleScreen**. It initializes the **game** reference.

```
@Override
public void show() {
     ...
     ,,,
}
```

This method is called when the screen becomes the current screen. It contains the setup code for the rule screen.

```
music = Gdx.audio.newMusic(Gdx.files.internal("mainmenu_msc.mp3"));
music.setLooping(true);
music.setVolume(0.5f);
music.play();

clicksound = Gdx.audio.newSound(Gdx.files.internal("clicksound.mp3"));
```

These lines initialize the background music and click sound effect, similar to what was done in the **MainMenuScreen**.

```
game.batch = new SpriteBatch();
font = new BitmapFont(Gdx.files.internal("ruleFont1.fnt"));
camera = new OrthographicCamera();
camera.setToOrtho(false, Gdx.graphics.getWidth(),
  Gdx.graphics.getHeight());

backgroundTexture = new Texture("endscreen_img.png");
```

These lines initialize the sprite batch, bitmap font, camera, and load the background texture.

Here we import a new text font using fnt file.

```
stage = new Stage(new ScreenViewport());
Gdx.input.setInputProcessor(stage);
```

These lines create a new stage for UI elements and set the input processor to the stage.

```
TextButton returnButton = createButton("Return to Menu", 0, 600);
```

This line creates a **TextButton** labeled "Return to Menu" at position (0, 600) using a helper method **createButton**.

```
returnButton.addListener(new ClickListener() {
    @Override
    public void clicked(com.badlogic.gdx.scenes.scene2d.InputEvent event,
      float x, float y) {
        clicksound.play();
        music.stop();
        game.setScreen(new MainMenuScreen(game));
    }
});
```

This adds a click listener to the **returnButton**. When clicked, it plays the click sound, stops the music, and sets the game screen to a new **MainMenuScreen**.

```
stage.addActor(returnButton);
```

This adds the return button to the stage.

```
@Override
public void render(float delta) {
     ...
     ...
}
```

This method is called continuously to update and render the screen.

```
Gdx.gl.glClearColor(0, 0, 0, 1);
Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
```

These lines clear the screen with a black color.

```
game.batch.setProjectionMatrix(camera.combined);

game.batch.begin();
game.batch.draw(backgroundTexture, 0, 0,
  Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
```

These lines set the projection matrix for the sprite batch and draw the background texture.

```
font.draw(game.batch, "Make ’N’ Break Rule:", 10,
  Gdx.graphics.getHeight() - 50);
font.draw(game.batch, "Objective:\n" +
    "The goal of Make ’N’ Break is to earn points by successfully recreating
    various \nstructures within a specified time limit.\n" +
    // ... (more rules text)
    "The game typically consists of several rounds.\n" +
    "The player or team with the highest total score at the end of
      three rounds wins.", 30, Gdx.graphics.getHeight() - 75);
```

These lines use the font to draw text on the screen, providing information about the rules of the game.

```
game.batch.end();

stage.act(Math.min(Gdx.graphics.getDeltaTime(), 1 / 30f));
stage.draw();
```

These lines end the sprite batch, update the stage, and draw the UI elements.

```
@Override
public void dispose() {
    font.dispose();
    backgroundTexture.dispose();
    stage.dispose();
}
```

This method is called when the screen is no longer needed. It disposes of resources to free up memory.

```
private TextButton createButton(String text, float x, float y) {
    ...
    ...
}
```

This is the helper method **createButton** for generating a **TextButton** with specified text and position.

### 6.2.3 Control Screen

When the **Control** button is clicked, the screen switches to Control Screen.

This screen uses same packages as Main Menu Screen.

```
public class ControlScreen implements Screen {
```

This line begins the definition of the **ControlScreen** class, which implements the LibGDX **Screen** interface.

```
private final MakeAndBreak game;
private BitmapFont font;

private OrthographicCamera gamecam;
private Viewport gameport;
private Stage stage;
private Music music;
private Sound clicksound;
private final Texture background =
  new Texture(Gdx.files.internal("control.png"));
```

These lines declare class variables:
**game** is a reference to the main game instance.
**font** is a bitmap font for rendering text.
**gameport** is a viewport for managing the screen’s dimensions.
**gamecam** is an orthographic camera.
**background** is a texture for the background.
**stage** is a UI stage for handling buttons and other UI elements.
**music** is a background music instance.
**clicksound** is a sound effect instance.

```
public ControlScreen(MakeAndBreak game) {
    this.game = game;
}
```

This is the constructor for **ControlScreen**. It initializes the **game** reference.

```
@Override
public void show() {
    ...
    ...
}
```

This method is called when the screen becomes the current screen. It contains the setup code for the control screen.

```
music = Gdx.audio.newMusic(Gdx.files.internal("mainmenu_msc.mp3"));
music.setLooping(true);
music.setVolume(0.5f);
music.play();

clicksound = Gdx.audio.newSound(Gdx.files.internal("clicksound.mp3"));

gamecam = new OrthographicCamera();
gameport = new FitViewport(game.WIDTH, game.HEIGHT, gamecam);
gamecam.setToOrtho(false, game.WIDTH, game.HEIGHT);

font = new BitmapFont(Gdx.files.internal("ruleFont.fnt"));
stage = new Stage(gameport);
Gdx.input.setInputProcessor(stage);
TextButton returnButton = createButton("Return to Menu", 0, 600);
returnButton.addListener(new ClickListener() {
    @Override
    public void clicked(com.badlogic.gdx.scenes.scene2d.InputEvent event,
      float x, float y) {
        music.stop();
        clicksound.play();
        game.setScreen(new MainMenuScreen(game));
    }
});

// Add the return button to the stage
stage.addActor(returnButton);
```

These lines initialize the background music, sound effect, camera, viewport, font, and stage. It also creates a return button and sets up a click listener to return to the main menu when the button is clicked.

```
@Override
public void render(float delta) {
    game.batch.setProjectionMatrix(gamecam.combined);
    game.batch.begin();

    game.batch.draw(background, 0, 0, Gdx.graphics.getWidth(),
      Gdx.graphics.getHeight());

    game.batch.end();

    // Draw the stage (UI elements)
    stage.act(Math.min(Gdx.graphics.getDeltaTime(), 1 / 30f));
    stage.draw();
}
```

This method is called continuously to update and render the screen. It draws the background texture and UI elements.

```
@Override
public void resize(int width, int height) {

}

@Override
public void pause() {

}

@Override
public void resume() {

}

@Override
public void hide() {

}

@Override
public void dispose() {
    font.dispose();
    background.dispose();  // Dispose of the background texture
    stage.dispose();
}
```

These methods are part of the **Screen** interface but are not used in this implementation. The **dispose** method is used to free up resources when the screen is no longer needed.

```
private TextButton createButton(String text, float x, float y) {
    ...
    ...
}
```

This is the helper method **createButton** for generating a **TextButton** with specified text and position.

### 6.2.4 Buffer Screen

When the **Play** button is clicked, the screen switches to Buffer Screen which gives player time to get ready for the game.

```
public class BufferScreen implements Screen {
```

This line begins the definition of the **BufferScreen** class, which implements the LibGDX **Screen** interface.

```
private final MakeAndBreak game;
private OrthographicCamera gamecam;
private Viewport gameport;
private Stage stage;
```

These lines declare class variables:
**game** is a reference to the main game instance.
**gameport** is a viewport for managing the screen's dimensions.
**gamecam** is an orthographic camera.
**stage** is a UI stage for handling buttons and other UI elements.

```
public BufferScreen(MakeAndBreak game){
    this.game = game;
    gamecam = new OrthographicCamera();
    gameport = new FitViewport(game.WIDTH, game.HEIGHT, gamecam);
    gamecam.setToOrtho(false, game.WIDTH, game.HEIGHT);
    gamecam.translate(-game.WIDTH/2,-game.HEIGHT/2);
    this.stage = new Stage(gameport, game.batch);
}
```

This is the constructor for **BufferScreen**. It initializes the **game** reference, camera, viewport, and stage. It also sets the camera position to the center and translates it to create a buffer effect.

```
@Override
public void show() {
    TextureAtlas atlas =
      new TextureAtlas(Gdx.files.internal("ready.atlas"));
    Skin skin = new Skin(Gdx.files.internal("ready.json"),atlas);
    Gdx.input.setInputProcessor(stage);
```

These lines load a texture atlas (**ready.atlas**) and a skin (**ready.json**) to manage UI elements. The input processor is set to the stage to handle input events.

```
ImageButton.ImageButtonStyle letsbuildStyle =
  new ImageButton.ImageButtonStyle();
letsbuildStyle = skin.get("letsbuild", ImageButton.ImageButtonStyle.class);

ImageButton.ImageButtonStyle readyStyle =
  new ImageButton.ImageButtonStyle();
readyStyle = skin.get("ready", ImageButton.ImageButtonStyle.class);

ImageButton readyButton = new ImageButton(readyStyle);
ImageButton letsbuildButton = new ImageButton(letsbuildStyle);
letsbuildButton.setSize(300,102);
readyButton.setSize(99*3,65*3);
readyButton.setPosition(game.WIDTH / 2 - readyButton.getWidth() / 2,
  game.HEIGHT / 2 - readyButton.getHeight() / 2 +
  letsbuildButton.getHeight()/2 + 10);
letsbuildButton.setPosition(game.WIDTH / 2 - letsbuildButton.getWidth() / 2,
  game.HEIGHT / 2 - letsbuildButton.getHeight() / 2);
letsbuildButton.addListener(new ClickListener() {
    @Override
    public void clicked(InputEvent event, float x, float y) {
        // Change the screen to ClassicScreen when the button is clicked
        try {
            game.setScreen(new ClassicScreen(game));
        } catch (FileNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
});

stage.addActor(readyButton);
stage.addActor(letsbuildButton);
```

These lines create two **ImageButton** instances (**readyButton** and **letsbuildButton**) using styles from the skin. They are positioned and sized accordingly.
A click listener is added to the **letsbuildButton** to change the screen to **ClassicScreen** when clicked.
Both buttons are added to the stage.

```
@Override
public void render(float delta) {
    Gdx.gl.glClearColor(0/255f, 0/255f, 128/255f, 100/255f);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    stage.setViewport(gameport);
    stage.draw();
}
```

This method is called continuously to update and render the screen. It sets the clear color and clears the screen. The stage is drawn with the current viewport.

```
@Override
public void resize(int width, int height) {

}

@Override
public void pause() {

}

@Override
public void resume() {

}

@Override
public void hide() {

}

@Override
```

```
public void dispose() {
    stage.dispose();
}
```

These methods are part of the **Screen** interface but are not used in this implementation.
The **dispose** method is used to free up resources when the screen is no longer needed.

### 6.2.5 Classic Screen

When the **Let's Build** button is clicked, the screen switches to Classic Screen (a.k.a Play Screen).

```
import Classes.ColorSelector;
import Classes.CustomButton;
import Classes.Grid;
import Classes.Quiz;
```

These lines import classes that are needed for the code.

```
public class ClassicScreen extends ApplicationAdapter implements Screen,
  InputProcessor {
```

This line declares a class named **ClassicScreen** that extends **ApplicationAdapter** and implements the **Screen** and **InputProcessor** interfaces. This class represents the main gameplay screen.

```
final MakeAndBreak game;
```

This line declares a final variable **game** of type **MakeAndBreak**, which is likely the main game class that holds the game's core functionality.

```
private OrthographicCamera gamecam;
private Viewport gameport;
private Stage stage;
private Texture img;
private Quiz quiz = new Quiz();
private boolean change = true;

private BitmapFont font;
private float timeCount; // Timer variable
private static Integer score;
private int worldTimer; // Initial timer value in seconds

private Music music;
private Sound cor_sound,fal_sound,vic_sound;
```

These lines declare class variables:
**game** is a reference to the main game instance.
**gameport** is a viewport for managing the screen's dimensions.
**gamecam** is an orthographic camera.
**stage** is a UI stage for handling buttons and other UI elements.
**music** is a background music instance.
**clicksound** is a sound effect instance.
**timeCount** is a timer variable.
**font** is a bitmap font for rendering text.
**score** is a score variable.
**worldTimer** is timer value in seconds.
**img** is a image for the background.

```
private Label countdownLabel;
private Label timeLabel;
private Label scoreLabel;
private Label scoreCountLabel;
```

These lines declare variables for managing the game's timer, score, and various labels for displaying information on the screen.

```
public ClassicScreen(MakeAndBreak game) throws FileNotFoundException {
```

This is the constructor for the **ClassicScreen** class, which takes a **MakeAndBreak** game instance as a parameter. It throws a **FileNotFoundException**.

```
//Gamecam
this.game = game;
gamecam = new OrthographicCamera();
gameport = new FitViewport(game.WIDTH , game.HEIGHT, gamecam);
gamecam.setToOrtho(false, game.WIDTH, game.HEIGHT);
gamecam.translate(-game.WIDTH/2,-game.HEIGHT/2);
```

These lines initialize the game camera (**OrthographicCamera**) and viewport (**FitViewport**).
The camera is set to orthographic projection, and its position is translated to the center of the screen.

```
//music
music = Gdx.audio.newMusic(Gdx.files.internal("playscreen_msc.mp3"));
music.setLooping(true);
music.setVolume(0.8f);
music.play();
```

This section initializes background music (**Music**) for the gameplay screen. The music is set to loop and play at a specific volume.

```
//sound
cor_sound=Gdx.audio.newSound(Gdx.files.internal("correct_sound.mp3"));
fal_sound=Gdx.audio.newSound(Gdx.files.internal("false_sound.mp3"));
vic_sound=Gdx.audio.newSound(Gdx.files.internal("victory.mp3"));
```

These lines initialize various sound effects (**Sound**) for correct answers, incorrect answers conditions.
The victory sound plays when time is up and the screen switches to Game Over Screen.

```
//Init Stage
stage = new Stage();
Grid grid = new Grid();
grid.setPosition(-200,-75);
stage.addActor(grid);
```

Here, a new **Stage** is created, and a **Grid** is added to it. The **Grid** likely represents the main gameplay grid where the game elements will be displayed.

```
//Color Selector
Color[] colors = {Color.RED, Color.BLUE, Color.GREEN,
  Color.ORANGE,Color.BLACK, Color.GRAY, Color.PURPLE, Color.WHITE};

for (int i = 0; i < colors.length; i++) {
    String colorName = getColorName(colors[i]);
    ColorSelector colorSelector = new ColorSelector(colors[i],
      colorName + ".png");
    // Adjust these values as needed
    colorSelector.setPosition(30 + i%4*75, -300 + i/4*140);
    colorSelector.setSize(50, 120);
    stage.addActor(colorSelector);
}
```

This section creates a set of color selectors using a loop. Each color selector is positioned based on the loop index and has a specific size.

```
 //Quiz
quiz.returnQuiz();
```

This line initializes the **Quiz** class and calls the **returnQuiz** method.
It sets up the initial quiz for the gameplay.

```
//Handle Input
InputProcessor inputProcessor = new InputAdapter(){
    @Override
    public boolean keyDown(int keycode) {
        if (keycode == Input.Keys.SPACE){
            try {
                if(grid.checkMatrix(quiz.getQuizFiles()[1])){
                    System.out.println("Correct");
                    cor_sound.play(0.7f);
                    quiz.currentQuiz = quiz.returnQuiz();
                    change = true;
                    addScore(10);
                } else {
                    System.out.println("TryAgain");
                    fal_sound.play();
                }
            } catch (FileNotFoundException e) {
                throw new RuntimeException(e);
            }
            for (int i = 0; i < grid.getRows(); i++) {
                for (int j = 0; j < grid.getColumns(); j++) {
                    CustomButton button = grid.getButton(i, j);
                    button.setColor(Color.YELLOW);
                }
            }
            return true;
        }
        return false;
    }
};

    InputMultiplexer Multiplexer = new InputMultiplexer();
    Multiplexer.addProcessor(stage);
    Multiplexer.addProcessor(inputProcessor);
    Multiplexer.addProcessor(this);
    Gdx.input.setInputProcessor(Multiplexer);
```

The player clicks the **CustomButton** to choose the color then places it to the **Grid** in order to form the correct building as in the quiz.
After that the player presses **Space** button to check.
Compare to the building in the quiz, if it is right, the system prints **Correct** to the console, the **corsound** plays, **returnQuiz()** activates, the Quiz changes and 10 points are added.

If it is wrong, the system prints **Try Again** to the console, the **falsound** plays.

An **InputMultiplexer** is used to combine multiple input processors, including the stage and this class, allowing them to handle input events.

```
//Quiz
game.batch = new SpriteBatch();
```

A new **SpriteBatch** is created, presumably for rendering graphics.

```
//Timer and score
//define a table used to organize hud’s labels
Table table = new Table();
table.setPosition(0,-400);
//Top-Align table
table.top();
//make the table fill the entire stage
table.setFillParent(true);
```

This section sets up a UI table to organize labels for displaying time and score on the screen.

```
// Initialize timer variables
worldTimer = 90; // Initial time in seconds
timeCount = 0;
score = 0;
```

The timer and score variables are initialized.

```
//load font
font=new BitmapFont(Gdx.files.internal("horizon.fnt"));
```

An internal font is loaded for rendering text.

```
//define our labels using the String,
//and a Label style consisting of a font and color
timeLabel = new Label("TIME", new Label.LabelStyle(font, Color.GOLD));
timeLabel.setFontScale(0.7f);

countdownLabel = new Label(String.format("%03d", worldTimer),
  new Label.LabelStyle(font, Color.GOLD));
countdownLabel.setFontScale(0.85f);

scoreLabel = new Label("SCORE", new Label.LabelStyle(font, Color.GOLD));
scoreLabel.setFontScale(0.7f);

scoreCountLabel=new Label(String.format("%03d", score),
  new Label.LabelStyle(font, Color.GOLD));
scoreCountLabel.setFontScale(0.85f);

//add our labels to our table, padding the top
table.add(timeLabel).expandX().padRight(50);
table.add(scoreLabel).expandX().padRight(1200);

table.row();//add a second row to our table

table.add(countdownLabel).expandX().padRight(50);
table.add(scoreCountLabel).expandX().padRight(1200);

//add our table to the stage
stage.addActor(table);
```

This part creates labels (**timeLabel, countdownLabel, scoreLabel, scoreCount-Label**) to display time and score information on the game screen.

The labels are styled with a specified font and color, and their font scale is adjusted.

These labels are then added to a table (**table**) that is positioned on the stage.

```
@Override
public void show() {
}
```

The **show()** method is overridden but left empty. It is typically used to initialize resources when the screen is displayed.

```
private String getColorName(Color color) {
    if (Color.RED.equals(color)) {
        return "red";
    } else if (Color.BLUE.equals(color)) {
        return "blue";
    } else if (Color.GREEN.equals(color)) {
        return "green";
    } else if (Color.ORANGE.equals(color)) {
        return "orange";
    } else if (Color.BLACK.equals(color)) {
        return "black";
    } else if (Color.GRAY.equals(color)) {
        return "gray";
    } else if (Color.PURPLE.equals(color)) {
        return "purple";
    } else if (Color.WHITE.equals(color)) {
        return "white";
    } else {
        return "unknown";
    }
}
```

This method maps a **Color** object to a corresponding color name.

```
@Override
public void resize(int width, int height) {
    gameport.update(width, height);
    gamecam.update();
}
```

The **resize** method is called when the screen is resized, and it updates the **gameport** and **gamecam** accordingly.

```
@Override
public void render(float delta) {
    update(delta);

    //main screen color ( Navy blue )
    Gdx.gl.glClearColor(0/255f, 0/255f, 128/255f, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    //Cards
    game.batch.begin();
    if (change) {
        // Dispose of the old texture
        if (img != null) {
            img.dispose();
        }

        // Load the new texture
        String[] quizFiles = quiz.getQuizFiles();
        img = new Texture(quizFiles[0]);
        change = false;
    }

    game.batch.draw(img, 420, 350, 456/4*3, 320/4*3);
    game.batch.end();

    //Grid
    stage.setViewport(gameport);
    stage.draw();
}
```

The **render** method is where the main game loop is implemented. It clears the screen, draws the cards based on the current quiz image, and updates the stage.

```
private void update(float delta) {
    timeCount += delta;
    if(timeCount >= 1){
        if (worldTimer > 0) {
            worldTimer--;
        } else {
            music.stop();
            vic_sound.play();
            gameOver();
        }
        countdownLabel.setText(String.format("%03d", worldTimer));
        timeCount = 0;
    }
}
```

The **update** method is responsible for updating game-related parameters, such as the countdown timer.

```
public void addScore(int value){
    score += value;
    scoreCountLabel.setText(String.format("%03d", score));
}
```

The **addScore** method increments the player’s score and updates the corresponding label.

```
private void gameOver() {
    music.stop();
    game.setScreen(new GameOverScreen(game,score));
}
```

The **gameOver** method stops the music, sets the game screen to the **GameOver-Screen**, and passes the final score to it.

```
@Override
public void create() {
    img = new Texture(quiz.getQuizFiles()[0]);
    super.create();
}
```

The **create** method initializes the texture for the current quiz image.

```
@Override
public void pause() {

}

@Override
public void resume() {

}

@Override
public void hide() {

}

@Override
public void dispose() {
    img.dispose();
    font.dispose();
}

@Override
public boolean keyDown(int keycode) {
    return false;
}

@Override
public boolean keyUp(int keycode) {
    return false;
}

@Override
public boolean keyTyped(char character) {
    return false;
}

@Override
public boolean touchDown(int screenX, int screenY,
  int pointer, int button) {
    return false;
}

@Override
public boolean touchUp(int screenX, int screenY,
  int pointer, int button) {
    return false;
}

@Override
public boolean touchCancelled(int screenX, int screenY,
  int pointer, int button) {
    return false;
}

@Override
public boolean touchDragged(int screenX, int screenY, int pointer) {
    return false;
}

@Override
public boolean mouseMoved(int screenX, int screenY) {
    return false;
```

```
}

@Override
public boolean scrolled(float amountX, float amountY) {
    return false;
}
```

The remaining methods are overrides or implementations of various application lifecycle, input handling, and disposal methods. These include methods like **pause, resume, hide, dispose, keyDown, keyUp**, etc. They handle different aspects of the application flow and user input.

**Grid Class**

```
package Classes;

import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.files.FileHandle;
import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.scenes.scene2d.ui.Table;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
```

Import statements for the necessary classes and libraries.

```
public class Grid extends Table {
private CustomButton[][] buttons;
public final int COLNUM = 6;
public final int ROWNUM = 7;
```

The Grid class extends Table and declares private variables: **buttons** (a 2D array of **ustomButton** objects), **COLNUM** (number of columns in the grid), and **ROWNUM** (number of rows in the grid).

```
public CustomButton getButton(int row, int col) {
    if (row >= 0 && row < buttons.length && col >= 0 &&
      col < buttons[row].length) {
        return buttons[row][col];
    } else {
        throw new IllegalArgumentException("Invalid row or column");
    }
}
```

**getButton** method returns the **CustomButton** at the specified row and column.

```
public Grid(){
    super();
    // Initialize the array
    buttons = new CustomButton[ROWNUM][COLNUM];

    // Create a 6x6 grid of buttons
    for (int i = 0; i < ROWNUM; i++) {
        for (int j = 0; j < COLNUM; j++) {
            buttons[i][j] = new CustomButton(this,i*ROWNUM+j);
            add(buttons[i][j]).width(50).height(50).pad(5);
        }
        row();
    }
}
```

Constructor initializes the **buttons** array and creates a 6x7 grid of **CustomButton** objects, adding them to the **Table**.

```
public boolean checkMatrix(String filepath) throws
  FileNotFoundException {
    ArrayList<String> currentGrid = new ArrayList<String>();
    ArrayList<String> answer = null;
    try {
        answer = Text2Array(filepath);
    } catch (IOException e) {
        e.printStackTrace();
        return false;//Return false or handle the exception as needed
    }
    for (CustomButton[] row : buttons){
        for (CustomButton button : row){
            currentGrid.add(button.selectedColor.toString());
        }
    }

    return currentGrid.equals(answer);
}
```

The **checkMatrix** method reads a text file and checks if the current state of the grid matches the expected answer.

It calls **Text2Array** to convert the text file into an array.

```
public ArrayList<String> Text2Array(String filepath) throws
  FileNotFoundException {
    ArrayList<String> lines = new ArrayList<String>();
    FileHandle file = Gdx.files.internal(filepath);
    String text = file.readString();
    BufferedReader reader = new BufferedReader(file.reader());

    String line;
    try {
        while ((line = reader.readLine()) != null) {
            lines.add(line);
        }
        reader.close();
    } catch (IOException e) {
        e.printStackTrace();
    }

    return lines;
}
}
```

The **Text2Array** method reads the contents of a text file and returns them as an **ArrayList** of strings. It handles file reading and exceptions.

**ColorSelector Class**

```
import com.badlogic.gdx.Gdx;
import com.badlogic.gdx.graphics.Color;
import com.badlogic.gdx.graphics.Texture;
import com.badlogic.gdx.graphics.g2d.TextureRegion;
import com.badlogic.gdx.scenes.scene2d.InputEvent;
import com.badlogic.gdx.scenes.scene2d.ui.Button;
import com.badlogic.gdx.scenes.scene2d.ui.ImageButton;
import com.badlogic.gdx.scenes.scene2d.utils.ClickListener;
import com.badlogic.gdx.scenes.scene2d.utils.TextureRegionDrawable;
```

The necessary imports for the classes and libraries used in this code.

```
public class ColorSelector extends ImageButton {
private Color color;
```

```
public ColorSelector(Color color, String textureFilePath) {
    super(createButtonStyle(textureFilePath));
    this.color = color;
    this.addListener(new ClickListener(){
        @Override
        public void clicked(InputEvent event, float x, float y) {
            GlobalState.selectedColor = color;
//          System.out.println(textureFilePath);
        }
    });
}
```

The **ColorSelector** class extends **ImageButton**. It takes a **Color** and a file path to a texture as parameters during instantiation.

**super(createButtonStyle(textureFilePath))**: The superclass (**ImageButton**) is initialized with a specific button style created by the **createButtonStyle** method.

**this.color = color;**: The **ColorSelector** object is associated with a specific color.

**this.addListener(new ClickListener() ... );**: A click listener is added to the button. When the button is clicked, the **clicked** method is executed.

**GlobalState.selectedColor = color;**: The clicked color is assigned to a global state variable **selectedColor**. This suggests that this button is part of a color selection mechanism in the game.

**// System.out.println(textureFilePath);**: This line is commented out but suggests that the file path to the texture can be printed for debugging purposes.

```
private static ImageButtonStyle createButtonStyle(
    String textureFilePath) {
    //Texture to draw
    Texture texture =  new Texture(Gdx.files.internal(textureFilePath));
    TextureRegion textureRegion = new TextureRegion(texture);
    TextureRegionDrawable drawable = new
      TextureRegionDrawable(textureRegion);

    ImageButtonStyle buttonStyle = new ImageButtonStyle();
    buttonStyle.up = drawable;
    return buttonStyle;
}
```

The **createButtonStyle** method takes a file path to a texture and creates an **ImageButtonStyle** for the button.

**Texture texture = new Texture(Gdx.files.internal(textureFilePath));**: It loads the texture from the specified file path.

**TextureRegion textureRegion = new TextureRegion(texture);**: It creates a **TextureRegion** from the texture.

**TextureRegionDrawable drawable = new TextureRegionDrawable(textureRegion);**: It creates a **TextureRegionDrawable** from the **TextureRegion**.

**ImageButtonStyle buttonStyle = new ImageButtonStyle();**: It creates an **ImageButtonStyle**.

**buttonStyle.up = drawable;**: It sets the drawable as the "up" state of the button.

**return buttonStyle;**: It returns the created **ImageButtonStyle**.

**CustomButton Class**

```
public class CustomButton extends ImageButton {
    private boolean clicked;
    private int id;
    private Grid grid;
    public Color selectedColor = Color.YELLOW;
    private static final ImageButtonStyle buttonStyle;
    static {
    // Default Texture
    Texture texture =  new Texture(Gdx.files.internal("yellow.png"));
    TextureRegion textureRegion = new TextureRegion(texture);
    buttonStyle = new ImageButtonStyle();
    buttonStyle.up =  new TextureRegionDrawable(textureRegion);
}
```

The **CustomButton** class extends **ImageButton** and has some private fields like **clicked, id, grid, and selectedColor**. It also has a **buttonStyle** field, which is a static **ImageButtonStyle** initialized with a default texture ("yellow.png").

```
    public CustomButton(ImageButtonStyle style, Grid table, int id) {
    super(style);
}
```

A constructor that takes a specific **ImageButtonStyle**, a **Grid**, and an ID as parameters during instantiation. However, it doesn't seem to be used in the provided code.

```
public CustomButton(Grid grid, int id){
    super(buttonStyle);
    this.id = id;
    this.grid = grid;
    this.addListener(new ClickListener(){
        @Override
        public boolean touchDown(InputEvent event, float x, float y,
          int pointer, int button) {
            boolean wasChecked = !isChecked();
            Color wasColor = getSelectedColor();
            int row = id / grid.ROWNUM;
            int col = id % grid.ROWNUM;

            if (button == Input.Buttons.RIGHT){
            //Handle right-click logic for changing colors horizontally
            } else if (button == Input.Buttons.LEFT){
            //Handle left-click logic for changing colors vertically
            }
            return false;
        }
    });
}
```

Another constructor that takes a **Grid** and an ID as parameters. It initializes the button with the default **buttonStyle** and adds a **ClickListener** to handle touch events. The click listener checks whether the right or left mouse button was clicked and handles changing colors horizontally or vertically accordingly.

```
    public void setColor(Color color) {
    ImageButtonStyle style = getButtonStyle(color);
    this.setStyle(style);
}
```

A method to set the color of the button. It updates the button's style based on the specified color.

```
private ImageButtonStyle getButtonStyle(Color color) {
    ImageButtonStyle style = new ImageButtonStyle();

    // Set the image of the button based on the color
    String imagePath;
    if (color == Color.RED) {
        imagePath = "red.png";
    } else if (color == Color.GREEN) {
        imagePath = "green.png";
    } else if (color == Color.BLUE) {
        imagePath = "blue.png";
    } else if (color == Color.ORANGE) {
        imagePath = "orange.png";
    } else if (color == Color.BLACK) {
        imagePath = "black.png";
    } else if (color == Color.PURPLE) {
        imagePath = "purple.png";
    } else if (color == Color.GRAY) {
        imagePath = "gray.png";
    } else if (color == Color.WHITE) {
        imagePath = "white.png";
    } else {
        imagePath = "yellow.png";
    }

    Texture texture = new Texture(Gdx.files.internal(imagePath));
    TextureRegion textureRegion = new TextureRegion(texture);

    style.up = new TextureRegionDrawable(textureRegion);
    this.selectedColor = color;
    return style;
}
```

A method that generates an **ImageButtonStyle** based on the provided color. It maps the color to a specific image path and creates a **TextureRegionDrawable** for the button style.

```
@Override
public boolean isPressed() {
    return super.isPressed();
}

public boolean isChecked() {
    return super.isChecked();
}

public Color getSelectedColor() {
    return this.selectedColor;
}
```

Methods to check whether the button is pressed, isChecked, and to get the selected color.

**GlobalState Class**

```
public class GlobalState {
public static Color selectedColor = Color.YELLOW;
```

The **GlobalState** class has a public static field named **selectedColor**, initialized with the color **Color.YELLOW**. This field is marked as **public static**, indicating that it can be accessed directly using the class name (**GlobalState.selectedColor**) without the need for an instance of the class.

**Quiz Class**

```
package Classes;

import static com.badlogic.gdx.math.MathUtils.random;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
```

Import statements for necessary classes and utilities from LibGDX, including static imports for the **random** method from **MathUtils**.

```
public class Quiz {
private final int NUMQUIZ = 21;
private List<Integer> quizzes;
public int currentQuiz;
```

Class declaration for the **Quiz** class. It contains a constant **NUMQUIZ** representing the total number of **quizzes**, a list of integers quizzes to store the available quizzes, and a public integer **currentQuiz** representing the current quiz.

```
public Quiz(){
    quizzes = new ArrayList<>();
    for (int i = 1; i <= NUMQUIZ; i++) {
        quizzes.add(i);
    }
    // Shuffle the list to ensure randomness
    Collections.shuffle(quizzes);
}
```

Constructor **Quiz()**: Initializes the **quizzes** list with integers from 1 to **NUMQUIZ**, representing the available quizzes. It then shuffles the list to ensure randomness in quiz selection.

```
public int returnQuiz(){
    if (quizzes.isEmpty()) {
        throw new IllegalStateException("No more quizzes left");
    }
    int newQuiz;
    do {
        newQuiz = quizzes.remove(random.nextInt(quizzes.size()));
    } while (newQuiz == currentQuiz);
    currentQuiz = newQuiz;
    return newQuiz;
}
```

Method **returnQuiz()**: Retrieves the next quiz randomly from the available quizzes. It throws an exception if there are no more quizzes left. The method ensures that the newly selected quiz is different from the current quiz to avoid repetition.

```
public String[] getQuizFiles() {
    String imgFile = "cards/" + this.currentQuiz + ".png";
    String txtFile = "ans/" + this.currentQuiz + ".txt";
    return new String[] {imgFile, txtFile};
}
```

Method **getQuizFiles()**: Returns an array of two strings representing the file paths for the image and text files associated with the current quiz.

### 6.2.6 Game Over Screen

When the timer becomes 0, the screen switches to Game Over Screen,

```
public class GameOverScreen extends ScreenAdapter {
```

Declares the **GameOverScreen** class, which extends **ScreenAdapter**.

```
private final MakeAndBreak game;
private Stage stage;
private Texture img =
  new Texture(Gdx.files.internal("endscreen_img.png"));
private BitmapFont font, font1;
private int score;
private Label scoreCountLabel;
private Music music;
private Sound clicksound;
```

Declares member variables including the game instance, a Stage for UI elements, a background image (**endscreen**), fonts, score, a label for displaying the score, and objects for handling music and sound effects.

```
public GameOverScreen(MakeAndBreak game, int score) {
    this.game = game;
    this.score = score;
}
```

Constructor for the **GameOverScreen** class, initializes the game instance and score.

```
@Override
public void show() {
```

Overrides the **show** method from **ScreenAdapter**, which is called when the screen becomes the current screen.

```
//music
music = Gdx.audio.newMusic(Gdx.files.internal("mainmenu_msc.mp3"));
music.setLooping(true);
music.setVolume(0.5f);
music.play();

//sound
clicksound=Gdx.audio.newSound(Gdx.files.internal("clicksound.mp3"));
```

Set up background music and sound effect for button clicks.

```
game.batch = new SpriteBatch();
```

Initializes the SpriteBatch for rendering graphics.

```
font = new BitmapFont(Gdx.files.internal("horizon.fnt"));
font1 = new BitmapFont();
```

Initializes two BitmapFont objects for rendering text.

```
stage = new Stage(new ScreenViewport());
Gdx.input.setInputProcessor(stage);
```

Creates a new Stage and sets it as the input processor, allowing the stage to handle input events.

```
// Create title label
Label.LabelStyle labelStyle = new Label.LabelStyle(font, Color.GOLD);
Label titleLabel = new Label("Game Over", labelStyle);
titleLabel.setPosition(Gdx.graphics.getWidth() / 2
  - titleLabel.getWidth() / 2, Gdx.graphics.getHeight() / 2 + 100);
```

Creates a label with a gold-colored style displaying ”Game Over” and positions it at the center of the screen.

```
Label scoreLabel = new Label("Score", labelStyle);
scoreLabel.setPosition(Gdx.graphics.getWidth() / 2
  - titleLabel.getWidth() / 2 + 100, Gdx.graphics.getHeight() / 2);
```

Creates a label for displaying ”Score” and positions it.

```
scoreCountLabel = new Label(String.format("%03d", score),
  new Label.LabelStyle(font, Color.GREEN));
scoreCountLabel.setPosition(Gdx.graphics.getWidth() / 2
  - titleLabel.getWidth() / 2 + 150,Gdx.graphics.getHeight() / 2 - 75);
```

Creates a label for displaying the score and positions it.

```
stage.addActor(titleLabel);
stage.addActor(scoreLabel);
stage.addActor(scoreCountLabel);
```

Adds the labels to the stage.

```
// Create buttons
TextButton replayButton = createButton("Replay", 300,
  Gdx.graphics.getHeight() - 500);
```

Creates a replay button using the **createButton** method.

```
replayButton.addListener(new ClickListener() {
    @Override
    public void clicked(InputEvent event, float x, float y) {
        clicksound.play();
        music.stop();
        game.setScreen(new MainMenuScreen(game));
    }
});
```

Adds a click listener to the replay button that plays a sound, stops the music, and sets the screen to the main menu.

```
stage.addActor(replayButton);
```

Adds the replay button to the stage.

```
private TextButton createButton(String text, float x, float y) {
    TextButton.TextButtonStyle style =
      new TextButton.TextButtonStyle();
    style.font = font1;

    TextButton button = new TextButton(text, style);
    button.setPosition(x, y);
    style.up = new TextureRegionDrawable(new TextureRegion(new Texture(
      Gdx.files.internal("blueBackground.jpg"))));
    button.setWidth(200); // Set your desired width
    button.setHeight(40); // Set your desired height
    style.fontColor = null;

    return button;
}
```

Defines a method **createButton** that generates a **TextButton** with specified text, position, and style. It uses a blue background texture and sets width and height.

```
@Override
public void render(float delta) {
    Gdx.gl.glClearColor(0, 0, 0, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    game.batch.begin();

    // Draw the background
    game.batch.draw(img, 0, 0, Gdx.graphics.getWidth(),
      Gdx.graphics.getHeight());

    game.batch.end();

    stage.act(Math.min(Gdx.graphics.getDeltaTime(), 1 / 30f));
    stage.draw();
}
```

Overrides the **render** method, which is called continuously to render the screen. Clears the screen, draws the background image, and renders the stage.

```
@Override
public void dispose() {
    game.batch.dispose();
    font.dispose();
    stage.dispose();
}
```

Overrides the dispose method, which is called when the screen is no longer needed. Disposes of resources like the sprite batch, fonts, and stage.

### 6.2.7 MakeAndBreak Class

```
public class MakeAndBreak extends Game {
```

Declares the **MakeAndBreak** class, which extends the LibGDX **Game** class. This class represents the main game application.

```
public SpriteBatch batch;
public ShapeRenderer shape;
```

Declares public instances of **SpriteBatch** and **ShapeRenderer**. These will be used for rendering graphics and shapes, respectively.

```
public int WIDTH;
public int HEIGHT;
```

Declares public variables to store the width and height of the game window.

```
@Override
public void create() {
    batch = new SpriteBatch();
    shape = new ShapeRenderer();
    WIDTH = Gdx.graphics.getWidth();
    HEIGHT = Gdx.graphics.getHeight();

    setScreen(new MainMenuScreen(this));
}
```

Overrides the **create** method from the **Game** class. This method is called when the game is first created. It initializes the **SpriteBatch** and **ShapeRenderer**, gets the width and height of the screen, and sets the initial screen to the **MainMenuScreen**.

```
@Override
public void render() {
    super.render();
}
```

Overrides the **render** method from the **Game** class. This method is continuously called to update and render the game. It calls the **render** method of the current screen.

```
@Override
public void dispose() {
    // Clean up resources when the game is disposed
}
```

Overrides the **dispose** method from the **Game** class. This method is called when the application is closed, and it should be used to dispose of resources such as textures, sounds, or other assets.

### 6.2.8 DesktopLauncher Class

This code is the desktop launcher class for a LibGDX game. It sets up the configuration for the desktop version of the game and launches it using the LWJGL3 (Lightweight Java Game Library 3) backend.

```
import com.badlogic.gdx.Graphics;
import com.badlogic.gdx.backends.lwjgl3.Lwjgl3Application;
import com.badlogic.gdx.backends.lwjgl3.Lwjgl3ApplicationConfiguration;
```

Imports necessary classes from LibGDX and LWJGL3. **Graphics** is used to access display mode information, and **Lwjgl3Application** and **Lwjgl3ApplicationConfiguration** are part of the LWJGL3 backend.

```
public class DesktopLauncher {
```

Declares the **DesktopLauncher** class.

```
public static void main(String[] arg) {
    Lwjgl3ApplicationConfiguration config =
      new Lwjgl3ApplicationConfiguration();
    Graphics.DisplayMode dm =
      Lwjgl3ApplicationConfiguration.getDisplayMode();
    config.setWindowedMode(822, 648);
    config.setForegroundFPS(60);
    config.setTitle("MakeAndBreak");
    config.setResizable(false);
    new Lwjgl3Application(new MakeAndBreak(), config);
}
```

The main method is the entry point for the desktop application. It does the following:
Creates a new instance of **Lwjgl3ApplicationConfiguration** named **config**.

Retrieves the current display mode using **Lwjgl3ApplicationConfiguration.getDisplayMode()** and stores it in the variable **dm**. Note that this line is redundant as it doesn't seem to be used later in the code.

Sets various configuration parameters for the game window using methods on the **config** object. For example, **config.setWindowedMode(822, 648)** sets the windowed mode size to 822x648 pixels. Creates a new instance of **Lwjgl3Application** with an instance of the **MakeAndBreak** game class and the configuration. This effectively starts the game on the desktop platform.

## 6.3 UML Diagrams

### 6.3.1 Activity diagram

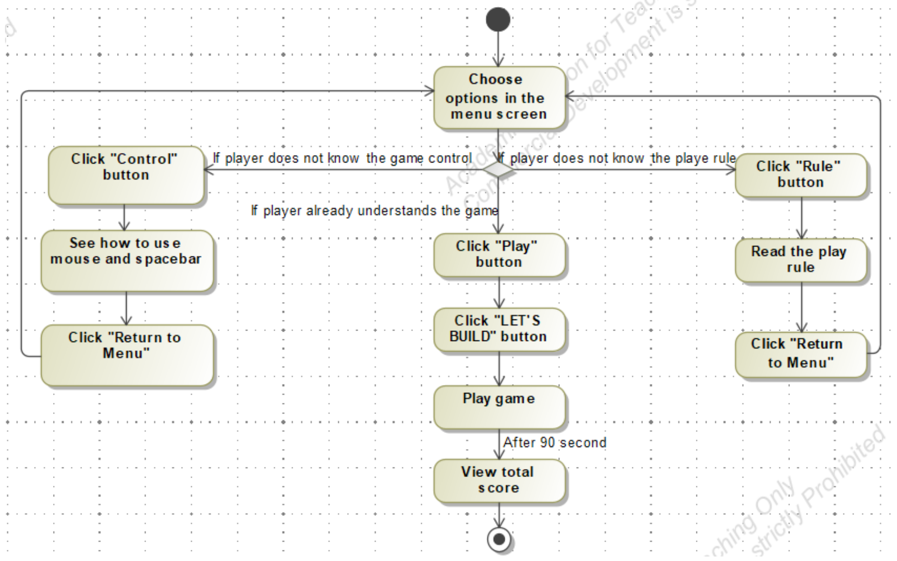

Choose options in the menu screen
If player does not know the game control
If player does not know the playe rule
If player already understands the game
Click "Control" button
See how to use mouse and spacebar
Click "Return to Menu"
Click "Play" button
Click "LET'S BUILD" button
Play game
After 90 second
View total score
Click "Rule" button
Read the play rule
Click "Return to Menu"

**Fig. 6.1** Make "N" Break Activity Diagram

The activity diagram illustrates the gaming process for a player. Upon entering the game, the player is presented with a menu screen offering three options. If the player is unfamiliar with the game's rules, they can click on the "Rule" button to access the play rule information. After reading the rules, the player can click the "Return to Menu" button to return to the menu screen. In case the player is unsure about the control scheme, they can click on the "Control" button to learn how to play the game. If the player already knows how to play, they can initiate the game by clicking the "Play" button and subsequently selecting the "LET'S BUILD" option. This allows the player to begin constructing the desired structure. After a duration of 90 seconds, the player will receive their total score.

### 6.3.2 Class Diagram

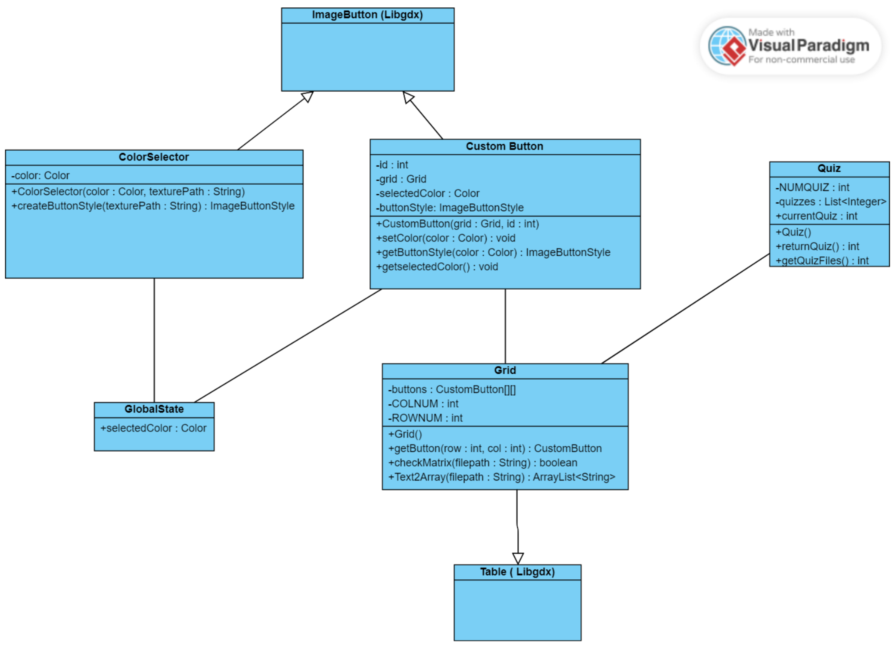

ImageButton (Libgdx)
Made with Visual Paradigm For non-commercial use
ColorSelector
-color: Color
+ColorSelector(color : Color, texturePath : String)
+createButtonStyle(texturePath : String) : ImageButtonStyle
Custom Button
-id : int
-grid : Grid
-selectedColor : Color
-buttonStyle: ImageButtonStyle
+CustomButton(grid : Grid, id : int)
+setColor(color : Color) : void
+getButtonStyle(color : Color) : ImageButtonStyle
+getselectedColor() : void
Quiz
-NUMQUIZ : int
-quizzes : List<Integer>
+currentQuiz : int
+Quiz()
+returnQuiz() : int
+getQuizFiles() : int
GlobalState
+selectedColor : Color
Grid
-buttons : CustomButton[][]
-COLNUM : int
-ROWNUM : int
+Grid()
+getButton(row : int, col : int) : CustomButton
+checkMatrix(filepath : String) : boolean
+Text2Array(filepath : String) : ArrayList<String>
Table ( Libgdx)

**Fig. 6.2** Make 'N' Break game Class Diagram

This class diagram shows the intricacies of different classes in our implementation of the game.

- **Table**: **Table** is a versatile layout container used for organizing and arranging UI elements such as buttons, labels, text fields, and images in a structured manner. This class is an native class in Lidgdx library.
- **Grid**: a custom class that extends the **Table** class. It represents a grid of buttons that can be interacted with. The grid is defined by a 2D array of **CustomButton**

objects. Objects from this class has the role of a container for create and manipulate a grid of buttons. It also provides methods necessary to compare the current state of the grid to the corresponding prepared answer from the **Quiz** class.

- **ImageButton**: **ImageButton** is a sub-class of **Button** and both of them are `Actors`. It provides a convenient way to create interactive elements in your user interface, such as clickable icons or graphical buttons.This class is an native class in Lidgdx library.
- **ColorSelector**: The main role for this class is to create area for the player to choose the color pieces and put it in the grid. It inherited from **ImageButton** so it would have all the functionalities a button. Whenever player click in a **ColorSelector** object, it would set the **GlobalState** - the state of color - to the respective color of the object itself.
- **CustomButton**: Also extends the **ImageButton** class, objects created with this class will acted like atomic area to place the color pieces in. Using many instances of this class in **Grid**, we have visualized a playable area for this game. Inside the grid, 3 objects of this class will form the pieces and each of them have the ability to change color to which there are clicked with - the current color. The methods **setColor** and **getButtonStyle** combined with algorithm are responsible for making the changes to the buttons.

# Chapter 7
# Conclusions and Future Work

## 7.1 How was the team work

- This is the first time we are working together as a team; therefore, difficulties are inevitable. Thuan, Phuc chose Android Studio and Kha chose IntelliJ since his Android Studio has problems. Luckily, everythings worked stably because we use Github to upload and download code after we finished a specific part.
- There were some heated arguments but we came to an agreement in a respectable manner. When things settled down, the interpersonal relationship between the group members was drastically improve.

## 7.2 What you have learned

- Throughout the process of creating a Java game project, we have acquired valuable skills and knowledge. One aspect that significantly contributed to our growth was learning how to push and commit code to GitHub. By mastering this essential skill, our teamwork has greatly improved as we can now collaborate more efficiently and effectively.
- Additionally, we delved into the world of game development using the LibGDX framework. This endeavor enhanced our understanding of the Java development process by providing us with hands-on experience in creating games. As a result, we gained valuable insight into game mechanics, graphics rendering, and user interaction.
- However, during the development process, we encountered performance issues caused by our limited understanding of certain functions within the LibGDX framework. Thankfully, we were able to identify the root causes of lag and implement the necessary fixes. This experience not only improved our problem-solving abilities but also sharpened our troubleshooting skills.

- Moreover, our project's scope extended beyond game development as we explored the use of LaTeX for document creation. By enhancing our understanding of LaTeX, we acquired the ability to create professional and aesthetically pleasing documents. This newfound knowledge further broadened our skill set and made us more versatile in our programming endeavors.
- Lastly, as we dived into Java, we discovered its object-oriented syntax, which closely aligns with other popular programming languages such as JavaScript, C++, and Python. This familiarity allowed us to quickly grasp the concepts and syntax required to build our game project. Furthermore, we learned that in Java, text strings are enclosed within double quotes, providing us with a fundamental understanding of string manipulation.
- Overall, our journey in creating a Java game project has been fruitful. We have not only improved our teamwork and collaboration skills through version control but also gained expertise in the LibGDX framework, troubleshooting performance issues, utilizing LaTeX for document creation, and understanding the object-oriented nature of Java. These skills and knowledge will undoubtedly prove invaluable in our future programming endeavors.

## 7.3 Ideas for the future development of game

After we have spent a great amount of time on reviewing the game, we relized that there are a number of improvements that we could have made:

- Design new cursor since we want the cursor in our game to be eye-catching, helping players feel a bit interesting when performing gameplay operations.
- Design new game operations because our game operations can be difficult for some people, especially those who do not use a mouse but use a laptop touchpad to play. We plan to use a keyboard key to change the direction of the brick.
- Adding local area network (LAN) playing option. Player can join up to maximum number of 4 for a multiplayer experience.
- Implement a Elo system, or a number that determine how good the player is at the game. It could use for competitive reason.
- Create an option where player can decide the dimensions of the grid. The card/ quiz is then generated by AI. The system core functionality will operate the same apart from the grid and the question.
- Add sound effect for invalid move and choosing color
- Leaderboard for players to take turn playing the game.
- Monetization: Advertisement on the menu screen; public game on Steam.
- Set game window to an adjustable state since we want players can play game in a bigger screen size.

## 7.4 Summary

In summary, despite facing various difficulties, our game was a resounding success and highly appreciated by our peers. During the poster viewing session, many classmates flocked to our group's device to play the game, engaging in friendly competition to achieve higher scores. Furthermore, our game attracted interest from others who downloaded the source code from our group's Github to implement the game on their personal laptops. This widespread positive reception demonstrates the triumph of our project and its broad appeal.