VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY

INTERNATIONAL UNIVERSITY SCHOOL
OF COMPUTER SCIENCE AND ENGINEERING

FINAL REPORT

Artifact Project

Course: *OBJECT – ORIENTED – PROGRAMMING – IT0791U*

ADVISOR : ***Assoc. PhD. Lê Duy Tân***

Ho Chi Minh City, 22th December 2025

VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY

INTERNATIONAL UNIVERSITY SCHOOL
OF COMPUTER SCIENCE AND ENGINEERING

GROUP 10 PROJECT'S MEMBER

No.	Full Name	ID	Role	Contribution
1	Hồ Ngọc An	ITITIU21146	Team Leader	40%
2	Trần Viết Trung	ITITWE22116	Member	30%
3	Lê Bá Khánh Hoàng	ITITWE22134	Member	30%

LIST OF FIGURES

TABLE OF CONTENTS

Chapter **Page**

ABSTRACT

Artifact game is a 2D platformer game that blends contemporary gameplay principles with aspects of vintage side-scrolling games. Players take control of a brave character in this colorful and lively game as they try to stop an evil sorcerer from destroying their community.

There are several stages in the game, and each has its own special difficulties and barriers. To advance through these stages, players must acquire power-ups, vanquish opponents, and solve riddles. Thanks to the responsive and easy-to-use controls, players of all skill levels may enjoy this game.

The dynamic difficulty system of Artifact games is one of its main characteristics. The game offers a difficult yet equitable experience by adjusting to the player's skill level. The game's richness and replayability are further increased by the assortment of mini-games and side missions it offers.

In conclusion, Artifact game is an entertaining and captivating 2D platformer game that provides a distinctive fusion of traditional and contemporary gameplay components. Adventure Quest is likely to keep gamers of all ages entertained for hours on end with its simple controls, varied difficulty structure, and endearing artwork.

CHAPTER 1. INTRODUCTION

1.1. Plan

An overview of an artifact project is what this paper aims to convey. Stakeholders engaged in its creation, implementation, and maintenance are the target audience for this paper, which includes design, implementation, and assessment. For anyone who would like to know more about the Artifact Project, the report can be a useful reference.

WEEK	TASK
1 - 4	Learn Basics of Javascript-ES6 and decide a type of game, Gameplay Mechanics, Art Style.
5 - 8	Design characters, level and use interface
9 - 12	• Implement the gameplay mechanics, character movements, collision detection, etc. • Create or source the art assets for your game, including characters, backgrounds, and UI elements.
13 (Deadline)	Play the game ourself and make adjustments based on our experience. Have others play your game and provide feedback

Table 1: Table of timeline

1.2. Requirements

The process of making a 2D game is thrilling and starts with a well-defined concept and design that specify the genre, plot, characters, and aesthetic.

A thorough grasp of a programming language appropriate for game creation, such as Python and Pygame, JavaScript and Phaser, or C# and Unity, is necessary for this procedure. By offering necessary tools and frameworks, game

engines such as Godot, Unity, or GameMaker may greatly expedite the creation process.

Characters, locations, and in-game items are all rendered in two dimensions utilizing graphic design tools, which adds to the game's visual attractiveness. Sound effects and music are added to further improve the immersive game experience.

Thorough testing is essential to guarantee seamless gaming and to find and fix any faults. When a game is flawless, it's prepared for release on an appropriate platform. This may be a web-based platform for browser games, a PC platform like Steam, or a mobile app store. Making a 2D game only takes time, patience, and a will to learn, but it can be a really fulfilling process despite the difficulties.

CHAPTER 2: STORY

Stage 1: Once upon a time, in a distant kingdom, there existed an evil sorcerer who sought to conquer the world. Using dark magic, he spread suffering and fear among the people. However, there was a young man named Arthur, born with a courageous heart and an indomitable spirit. Arthur decided to embark on a quest to find the legendary sword, which the legends said was the only weapon capable of defeating the sorcerer.

Stage 2: On his journey, Arthur encountered many companions: a fierce female warrior, a wise monk, and a small, flying dragon. Together, they overcame numerous challenges, from dense forests and treacherous mountains to monster-filled caves. When they finally reached the sorcerer's fortress, Arthur and his companions faced deadly traps and fierce minions.

Stage 3: In the end, with bravery and the unity of his friends, Arthur confronted the sorcerer head-on. Using the legendary sword and the support of his companions, Arthur defeated the sorcerer, freeing the world from the clutches of evil. The people rejoiced, the kingdom returned to peace, and Arthur and his friends became heroes remembered forever.

CHAPTER 3. TUTORIAL

3.1. How to play

- **"Play"** to start the game.
- **"Help"** for knowledge about how to play game.
- **"Quit"** to close the game.

Figure 1: Menu

- Hold **E** and the button →,← to run.

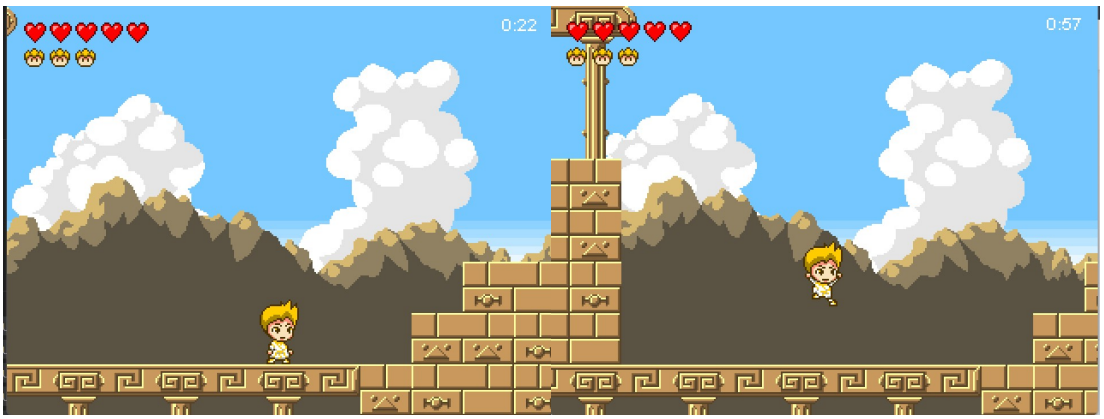

Figure 2: Move

- You can kill all enemies by F in a straight line when using dash

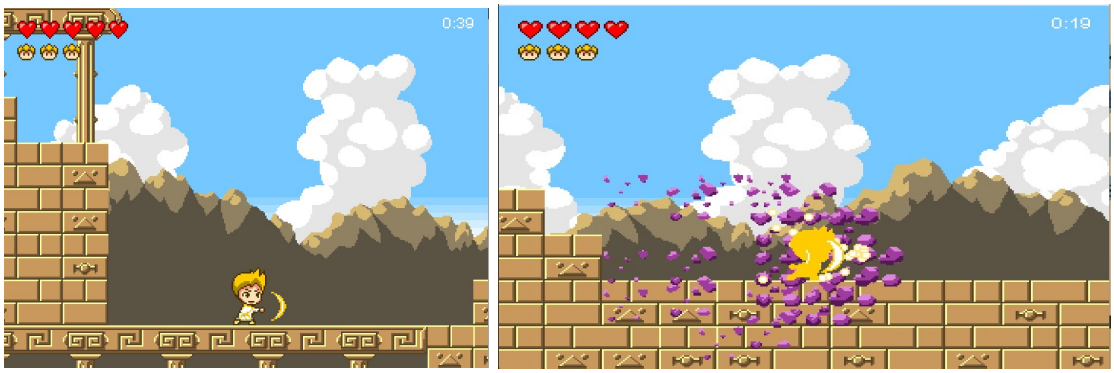

Figure 3: Attack and defeat enemies

- Fall into an abyss the game will start over

Figure 4: Fall into

3.2. Goals

- Monster alert

Figure 5: Hey, adds inc.

- Defeat enemies and boss

Figure 6: Enemies and Boss

- The place to move to the next stage

Figure 7: Place to move

CHAPTER 4. DESIGN & NEW FEATURES

4.1 Design

- BufferedImage for image files and pixel color manipulation.
- Pixel colors for game components with RGB channels.
- Level class for level data and game elements.
- Google resources for optimization.

4.2 New Features

- Keyboard control for movement and attack.
- Slashing move until hitting tiles.
- Tutorial at the start.
- Bosses at the end.
- Jumping costs energy

4.3 Summary

- State pattern for modes and transitions in game states package.
- Constants class for game constants.
- Single Responsibilities pattern applied.
- Design principle for readability and maintainability.

CHAPTER 5. PROPERTIES OF OOP

5.1 Encapsulation:

- **Classes Involved:** Player, Enemy, Game.
- **Attributes Hidden:** Attributes like health, position, and score are made private in the Player class.
- **Methods Hidden:** Functions such as updateHealth() or move() are kept internal.
- **Getters and Setters:** These provide controlled access to attributes, ensuring the internal state isn't directly modified.

5.2 Abstraction:

- **Subclasses:** Subclasses like EnemyTypeA and EnemyTypeB extend the Enemy class.
- **GameState Interface:** Defines abstract methods like start(), pause(), and end().
- **How It Helps:** Shared methods like takeDamage() or move() are abstract in the base class Enemy but implemented differently in subclasses.

5.3 Inheritance:

- Create subclasses for code reuse and hierarchy.
- For example, MapObject and GameState.
- Reducing code duplication.
- Promoting modular code.

5.4 Polymorphism:

- The EnemyManager class contains 2 checkEnemyHit methods.
- Depend on the type of collision.

CHAPTER 6. SOLID PRINCIPLE

6.1. Single-Responsibility Principle

The background class is designed to do one main task only: to load an image as the background and provide various functionalities to manage the appearance, position, scaling, and movement. Each method, like setPosition, setScale, and setVector allow the background to be created in different ways, such as using a full image, a scaled image.

```
public class Background {

    private BufferedImage image;

    private double x;
    private double y;
    private double dx;
    private double dy;

    private int width;
    private int height;

    private double xscale;
    private double yscale;

    public Background(String s) {
        this(s, 0.1);
    }

    public Background(String s, double d) {
        this(s, d, d);
    }

    public Background(String s, double d1, double d2) {
        try {
            image = ImageIO.read(
                getClass().getResourceAsStream(s)
            );
            width = image.getWidth();
            height = image.getHeight();
            xscale = d1;
            yscale = d2;
        }
        catch(Exception e) {
            e.printStackTrace();
        }
```

Figure 8: The first part of Background code

```java
public Background(String s, double ms, int x, int y, int w, int h) {
    try {
        image = ImageIO.read(
            getClass().getResourceAsStream(s)
        );
        image = image.getSubimage(x, y, w, h);
        width = image.getWidth();
        height = image.getHeight();
        xscale = ms;
        yscale = ms;
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

public void setPosition(double x, double y) {
    this.x = (x * xscale) % width;
    this.y = (y * yscale) % height;
}

public void setVector(double dx, double dy) {
    this.dx = dx;
    this.dy = dy;
}

public void setScale(double xscale, double yscale) {
    this.xscale = xscale;
    this.yscale = yscale;
}

public void setDimensions(int i1, int i2) {
    width = i1;
    height = i2;
}
```

Figure 9: The second part of the background

6.2 Open/Closed Principle

The Open/Closed Principle states that a class should be open for extension but closed for modification. This means you should be able to add new functionality to a class without changing its existing code, ensuring stability and reducing the risk of breaking existing behavior. In the provided code, the Enemy and EnemyProjectile classes follow the Open/Closed Principle by allowing extensions through inheritance. Developers can create new types of enemies or projectiles by extending these classes and overriding methods like update or setHit, without altering the base class implementation. This ensures the base functionality remains intact while allowing for new behaviors.

```
public class Enemy extends MapObject {

    protected int health;
    protected int maxHealth;
    protected boolean dead;
    protected int damage;
    protected boolean remove;

    protected boolean flinching;
    protected long flinchCount;

    public Enemy(TileMap tm) {
        super(tm);
        remove = false;
    }

    public boolean isDead() { return dead; }
    public boolean shouldRemove() { return remove; }

    public int getDamage() { return damage; }

    public void hit(int damage) {
        if(dead || flinching) return;
        JukeBox.play("enemyhit");
        health -= damage;
        if(health < 0) health = 0;
        if(health == 0) dead = true;
        if(dead) remove = true;
        flinching = true;
        flinchCount = 0;
    }

    public void update() {}

}
```

Figure 10: The Code of Enemy class

```java
public abstract class EnemyProjectile extends MapObject {

    protected boolean hit;
    protected boolean remove;
    protected int damage;

    public EnemyProjectile(TileMap tm) {
        super(tm);
    }

    public int getDamage() { return damage; }
    public boolean shouldRemove() { return remove; }

    public abstract void setHit();

    public abstract void update();

    public void draw(Graphics2D g) {
        super.draw(g);
    }

}
```

Figure 11: The Code of EnemyProjectile

6.3 Liskov Substitution Principle

The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. This means that a subclass must fully implement the behavior expected of its superclass so that it can be used interchangeably. In the provided code, the GameState abstract class enforces Liskov Substitution Principle by defining abstract methods like init, update, draw, and handleInput. Subclasses of GameState must implement these methods, ensuring they can be used wherever a GameState object is expected, maintaining consistent behavior and compatibility.

```java
package com.neet.GameState;

import java.awt.Graphics2D;

public abstract class GameState {

    protected GameStateManager gsm;

    public GameState(GameStateManager gsm) {
        this.gsm = gsm;
    }

    public abstract void init();
    public abstract void update();
    public abstract void draw(Graphics2D g);
    public abstract void handleInput();

}
```

Figure 12: The Code of GameState class

6.4 Interface Segregation Principle:

The Interface Segregation Principle (ISP) is a principle that helps ensure that classes are not forced to implement methods they do not use. In other words, instead of creating a large interface (fat interface) with many methods that not all classes need, you should

break it down into smaller interfaces, each containing only the methods relevant to a specific set of tasks.

6.5 Dependency Inversion Principle (DIP)

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules; instead, both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions. This principle promotes loose coupling and makes the code more flexible and easier to maintain. In the provided code, the Enemy class relies on abstractions like MapObject and potentially other interfaces or base classes, ensuring that changes in the low-level implementation do not directly affect the Enemy class.

```java
public class Enemy extends MapObject {  7 usages  Ho Ngoc An

    protected int health;  4 usages
    protected int maxHealth;  no usages
    protected boolean dead;  4 usages
    protected int damage;  1 usage
    protected boolean remove;

    protected boolean flinching;  2 usages
    protected long flinchCount;  1 usage

    public Enemy(TileMap tm) {  no usages  Ho Ngoc An
        super(tm);
        remove = false;
    }

    public boolean isDead() { return dead; }  no usages  Ho Ngoc An
    public boolean shouldRemove() { return remove; }  no usages  Ho Ngoc An

    public int getDamage() { return damage; }  no usages  Ho Ngoc An

    public void hit(int damage) {  no usages  Ho Ngoc An
        if(dead || flinching) return;
        JukeBox.play("enemyhit");
        health -= damage;
        if(health < 0) health = 0;
        if(health == 0) dead = true;
        if(dead) remove = true;
        flinching = true;
        flinchCount = 0;
    }

    public void update() {}  Ho Ngoc An

}
```

Figure 13: Dependency Invesion Principle used in the project

CHAPTER 7. UML AND CLASSES

7.1 Audio

JukeBox class is designed to handle audio functionalities such as playing, stopping, and looping sound clips. It includes various attributes and methods that manage these audio operations.

JukeBox class is responsible for managing audio playback within the application. Its primary functions include:

- ***Loading and Storing Audio Clips***: New clips are loaded into this collection using the load method, and all audio clips are stored in the clips property.
- ***Playing Control:*** The ability to start, stop, and resume the playing of audio clips is provided via controls like play, stop, and resume.
- ***Looping:*** With control over start and finish positions, the loop techniques offer many ways to loop audio clips for a predetermined number of times or forever.
- ***Position Management:*** You may precisely control where playback begins and ends by using methods like getPosition, setPosition, and getFrames to manage the playback position of audio clips.
- ***Muting:*** The audio playing can be muted or unmuted using the mute property.

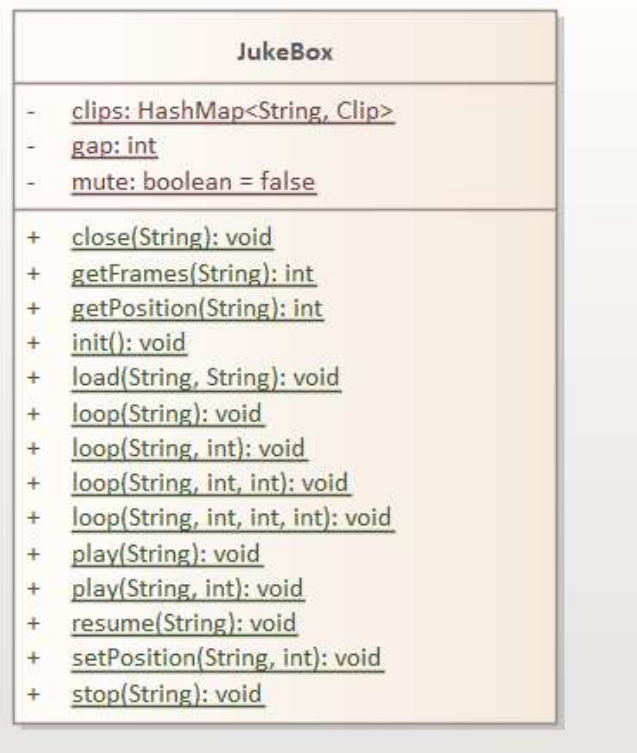

Figure 14: Audio

7.2 Enemy

- ***DarkEnergy Class:*** Depicts an energy-based adversary that may be either transient or permanent and travels in a bouncing pattern under the effect of gravity.
- ***Gazer Class:*** Describes an adversary with particular timing and sprite modifications that may employ beams or gaze assaults.
- ***GelPop Class:*** This type of opponent resembles a blob and is capable of interacting with the player and changing its state in response to actions.
- ***Tengu Class:*** Possibly modeled after a mythological monster, this class depicts a multifaceted adversary capable of both attacking and jumping.

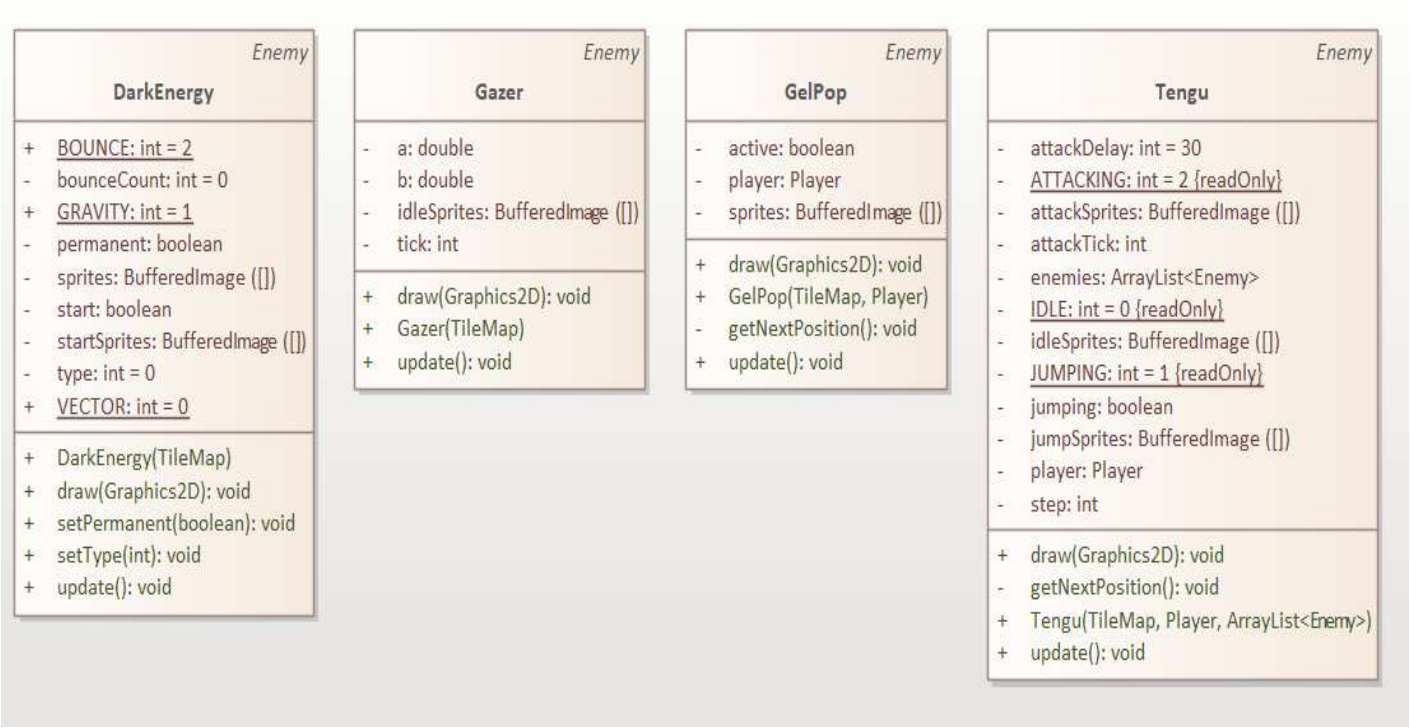

Figure 15: Enemies

7.3 Handlers

- ***Content Class:*** in charge of controlling the loading and storing of picture assets within the application.
- ***Keys Class:*** this class handles updating and preserving key input statuses so the program can respond to user keyboard and mouse inputs.

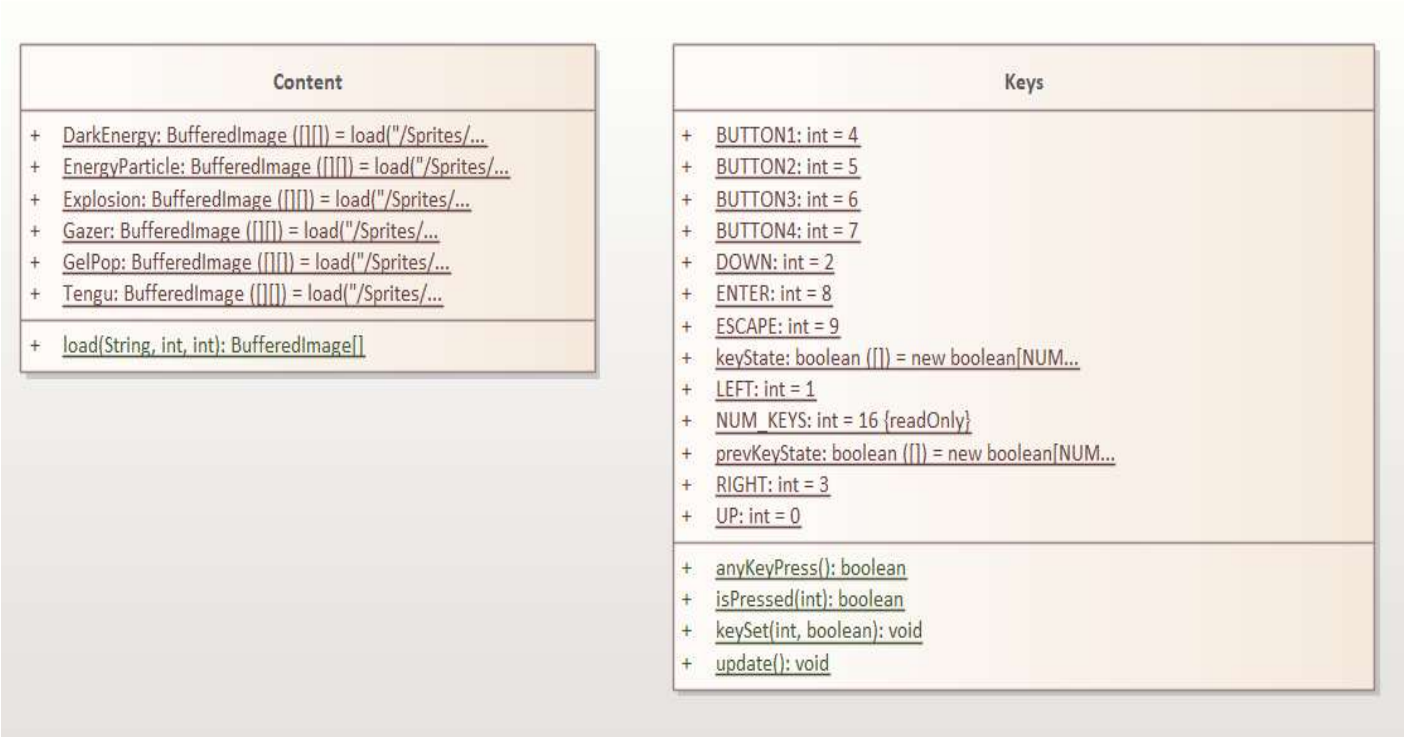

Figure 16: Handlers

7.4 Main

- ***Game class***: serves as the starting point of the application with its main method.
- ***GamePanel class***: is the core component that handles the game loop, rendering, and input. It contains various attributes for managing the game's state and methods for initializing, updating, drawing, and handling keyboard events.

Figure 17: Main Class

```java
package com.neet.Main;

import javax.swing.JFrame;

public class Game {  no usages  Ho Ngoc An

    public static void main(String[] args) {  Ho Ngoc An
        JFrame window = new JFrame("Artifact");
        window.add(new GamePanel());
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setResizable(false);
        window.pack();
        window.setLocationRelativeTo(null);
        window.setVisible(true);
    }

}
```

Figure 18: Main class code

```java
@SuppressWarnings("serial")  Ho Ngoc An
public class GamePanel extends JPanel implements Runnable, KeyListener{

    // dimensions
    public static final int WIDTH = 320;
    public static final int HEIGHT = 240;
    public static final int SCALE = 2;  4 usages

    // game thread
    private Thread thread;  3 usages
    private boolean running;  2 usages
    private int FPS = 60;  1 usage
    private long targetTime = 1000 / FPS;  1 usage

    // image
    private BufferedImage image;  5 usages
    private Graphics2D g;  2 usages

    // game state manager
    private GameStateManager gsm;  3 usages

    // other
    private boolean recording = false;  3 usages
    private int recordingCount = 0;  2 usages
    private boolean screenshot;  3 usages

    public GamePanel() {  Ho Ngoc An
        super();
        setPreferredSize(new Dimension(WIDTH * SCALE, HEIGHT * SCALE));
        setFocusable(true);
        requestFocus();
    }

    public void addNotify() {  no usages  Ho Ngoc An
        super.addNotify();
        if(thread == null) {
            thread = new Thread(this);
            addKeyListener(this);
            thread.start();
        }
    }
```

Figure 19: GamePanel Class

7.5 Entity

7.5.1 Artifact

- ***BottomLeftPiece:*** This symbol designates a section of the map situated in the lower-left corner. This particular section of the map is rendered and updated by this class.
- ***BottomRightPiece:*** This element of the map denotes the area at the bottom-right corner. This particular section of the map is rendered and updated by this class.
- ***TopLeftPiece:*** This symbol designates a section of the map situated in the upper-left corner. This particular section of the map is rendered and updated by this class.
- ***TopRightPiece:*** This symbol designates a section of the map situated in the upper-right corner. This particular section of the map is rendered and updated by this class.

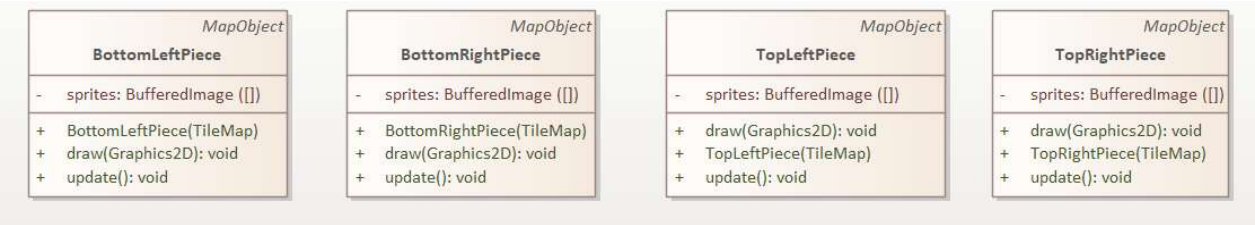

Figure 20: Artifact

```java
public class BottomLeftPiece extends MapObject {  1 usage  Ho Ngoc An

    private BufferedImage[] sprites;  3 usages

    public BottomLeftPiece(TileMap tm) {  no usages  Ho Ngoc An
        super(tm);
        try {
            BufferedImage spritesheet = ImageIO.read(
                getClass().getResourceAsStream("/Sprites/Other/Artifact.gif")
            );
            sprites = new BufferedImage[1];
            width = height = 4;
            sprites[0] = spritesheet.getSubimage(0, 10, 10, 10);
            animation.setFrames(sprites);
            animation.setDelay(-1);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

    public void update() {  Ho Ngoc An
        x += dx;
        y += dy;
        animation.update();
    }

    public void draw(Graphics2D g) { super.draw(g); }

}
```

Figure 21: The BottomLeftPiece Code

7.5.2 Enemies

1. Player Class

Function: Acts as a representative of the player character and controls its actions, interactions, and status in the game environment.

Main Duties:

- Manages player activities including dashing, jumping, and attacking.
- Controls player states, such as health, score, and other states (such as dead or flinching).
- Refreshes the player's location, interacts with other objects, and looks for collisions.

```
public Player(TileMap tm) {   ± Ho Ngoc An *
    super(tm);
    ar = new Rectangle(0, 0, 0, 0);
    ar.width = 30;
    ar.height = 20;
    aur = new Rectangle((int)x - 15, (int)y - 45, 30, 30);
    cr = new Rectangle(0, 0, 0, 0);
    cr.width = 50;
    cr.height = 40;
    width = 30;
    height = 30;
    cwidth = 15;
    cheight = 38;
    moveSpeed = 1.6;
    maxSpeed = 1.6;
    stopSpeed = 1.6;
    fallSpeed = 0.15;
    maxFallSpeed = 4.0;
    jumpStart = -4.8;
    stopJumpSpeed = 0.3;
    doubleJumpStart = -3;
    damage = 2;
    chargeDamage = 1;
    facingRight = true;
    lives = 3;
    health = maxHealth = 5;
    // load sprites
```

Figure 22: First Part of player class

```
    try {
        BufferedImage spritesheet = ImageIO.read(
            getClass().getResourceAsStream(
                "/Sprites/Player/PlayerSprites.gif"
            )
        );
        int count = 0;
        sprites = new ArrayList<BufferedImage[]>();
        for(int i = 0; i < NUMFRAMES.length; i++) {
            BufferedImage[] bi = new BufferedImage[NUMFRAMES[i]];
            for(int j = 0; j < NUMFRAMES[i]; j++) {
                bi[j] = spritesheet.getSubimage(
                    j * FRAMEWIDTHS[i],
                    count,
                    FRAMEWIDTHS[i],
                    FRAMEHEIGHTS[i]
                );
            }
            sprites.add(bi);
            count += FRAMEHEIGHTS[i];
        }
        // emotes
        spritesheet = ImageIO.read(getClass().getResourceAsStream(
            "/HUD/Emotes.gif"
        ));
        confused = spritesheet.getSubimage(
            0, 0, 14, 17
        );
        surprised = spritesheet.getSubimage(
            14, 0, 14, 17
        );
    }
    catch(Exception e) {
        e.printStackTrace();
    }
    energyParticles = new ArrayList<EnergyParticle>();
    setAnimation(IDLE);
    JukeBox.load("/SFX/playerjump.mp3", "playerjump");
    JukeBox.load("/SFX/playerlands.mp3", "playerlands");
    JukeBox.load("/SFX/playerattack.mp3", "playerattack");
    JukeBox.load("/SFX/playerhit.mp3", "playerhit");
    JukeBox.load("/SFX/playercharge.mp3", "playercharge");

}
```

Figure 23: Second Part of player class

2. Enemy Class:

Function: Acts as a representative for antagonistic figures in the game, controlling their activities and interactions with the player.

Main Duties:

- Controls adversary conditions, including health, location, and assault patterns.
- Manages player interactions, updates opponent position, and looks for collisions.
- Regulates enemy-specific behaviors and conditions (such as explosions upon death).

```
    public Enemy(TileMap tm) {  no usages  Ho Ngoc An
        super(tm);
        remove = false;
    }

    public boolean isDead() { return dead; }  no usages  Ho Ngoc An
    public boolean shouldRemove() { return remove; }  no usages  Ho Ngoc An

    public int getDamage() { return damage; }  no usages  Ho Ngoc An

    public void hit(int damage) {  no usages  Ho Ngoc An
        if(dead || flinching) return;
        JukeBox.play("enemyhit");
        health -= damage;
        if(health < 0) health = 0;
        if(health == 0) dead = true;
        if(dead) remove = true;
        flinching = true;
        flinchCount = 0;
    }

    public void update() {}  Ho Ngoc An

}
```

Figure 24: The code of Enemy class

3. MapObject Class:

Function: Acts as the foundational class for all objects on the game map that communicate with their surroundings.

Main Duties:

- Offers fundamental characteristics for location, size, and motion.
- Controls the game map's collision detection and reaction.
- Describes changes in state and movement behaviors (e.g., falling, leaping).

```
public MapObject(TileMap tm) {   Ho Ngoc An
    tileMap = tm;
    tileSize = tm.getTileSize();
    animation = new Animation();
    facingRight = true;
}

public boolean intersects(MapObject o) {   no usages   Ho Ngoc An
    Rectangle r1 = getRectangle();
    Rectangle r2 = o.getRectangle();
    return r1.intersects(r2);
}

public boolean intersects(Rectangle r) { return getRectangle().intersects(r); }

public boolean contains(MapObject o) {   no usages   Ho Ngoc An
    Rectangle r1 = getRectangle();
    Rectangle r2 = o.getRectangle();
    return r1.contains(r2);
}

public boolean contains(Rectangle r) { return getRectangle().contains(r); }

public Rectangle getRectangle() {   6 usages   Ho Ngoc An
    return new Rectangle(
            (int)x - cwidth / 2,
            (int)y - cheight / 2,
            cwidth,
            cheight
    );
}
```

Figure 25: The code of MapObject class

4. Title Class:

Function: Stands in for individual tiles that comprise the game map.

Main Duties:

- Keeps track of tile attributes including type and blocking status.
- Provides tools for managing the attributes of tiles and drawing them.

```
public class Title {  2 usages  ▲ Ho Ngoc An

    public BufferedImage image;

    public int count;  2 usages
    private boolean done;  4 usages
    private boolean remove;  2 usages

    private double x;  8 usages
    private double y;  2 usages
    private double dx;  3 usages

    private int width;  6 usages

    public Title(String s) {  no usages  ▲ Ho Ngoc An

        try {
            image = ImageIO.read(getClass().getResourceAsStream(s));
            width = image.getWidth();
            x = -width;
            done = false;
        }
        catch(Exception e) {
            e.printStackTrace();
        }

    }

    public Title(BufferedImage image) {  no usages  ▲ Ho Ngoc An
        this.image = image;
        width = image.getWidth();
        x = -width;
        done = false;
    }
```

Figure 26: The code of Title Class

5. HUD Class:

Function: Organizes and maintains the Heads-Up Display (HUD), which presents player-specific data like score and health.

Main Duties:

- Sketches HUD components into the screen.
- Adjusts the display according to game events and player status.

```
public class HUD {  3 usages  ± Ho Ngoc An

    private Player player;  4 usages

    private BufferedImage heart;  2 usages
    private BufferedImage life;  2 usages

    public HUD(Player p) {  no usages  ± Ho Ngoc An
        player = p;
        try {
            BufferedImage image = ImageIO.read(
                getClass().getResourceAsStream(
                    "/HUD/Hud.gif"
                )
            );
            heart = image.getSubimage(0, 0, 13, 12);
            life = image.getSubimage(0, 12, 12, 11);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
```

Figure 27: The code of HUD class

6. EnergyParticle Class:

Function: Describes particles produced by certain game events or activities (e.g., assaults' consequences).

Principal Duties:

- Controls the motion and location of particles.
- Updates the status of particles and draws them.
- Establishes the appropriate time for particle removal.

```
public class EnergyParticle extends MapObject {  3 usages  ▲ Ho Ngoc An

    private int count;  3 usages
    private boolean remove;  2 usages

    private BufferedImage[] sprites;  2 usages

    public static int UP = 0;  1 usage
    public static int LEFT = 1;  1 usage
    public static int DOWN = 2;  1 usage
    public static int RIGHT = 3;  no usages

    public EnergyParticle(TileMap tm, double x, double y, int dir) {  no usages  ▲ Ho Ngoc An
        super(tm);
        this.x = x;
        this.y = y;
        double d1 = Math.random() * 2.5 - 1.25;
        double d2 = -Math.random() - 0.8;
        if(dir == UP) {
            dx = d1;
            dy = d2;
        }
        else if(dir == LEFT) {
            dx = d2;
            dy = d1;
        }
        else if(dir == DOWN) {
            dx = d1;
            dy = -d2;
        }
        else {
            dx = -d2;
            dy = d1;
        }

        count = 0;
        sprites = Content.EnergyParticle[0];
        animation.setFrames(sprites);
        animation.setDelay(-1);
    }
```

Figure 28: The code of EnergyParticle class

7. Explosion Class:

Function: Acts as the game's representation of explosive effects.

Main Duties:

- Controls animation frames and explosion position.
- Creates and updates the effect of an explosion.
- Manages an explosion's lifespan, such as by eliminating it after it's finished.

```
public Explosion(TileMap tm, int x, int y) {  no usages  ▲ Ho Ngoc An

    super(tm);

    this.x = x;
    this.y = y;

    width = 30;
    height = 30;

    speed = 2;
    diagSpeed = 1.41;

    sprites = Content.Explosion[0];

    animation.setFrames(sprites);
    animation.setDelay(6);

    points = new Point[8];
    for(int i = 0; i < points.length; i++) {
        points[i] = new Point(x, y);
    }

}
```

Figure 29: The code of Explosion class

8. EnemyProjectile Class:

Function: Stands in for enemy projectiles.

Main Duties:

- Controls projectile location, motion, and condition.
- Responds to impacts with other things, such as players.
- Draws the projectile and updates it.

```
public abstract class EnemyProjectile extends MapObject {  2 usages  ▲ Ho Ngoc An

    protected boolean hit;  no usages
    protected boolean remove;
    protected int damage;  1 usage

    public EnemyProjectile(TileMap tm) { super(tm); }

    public int getDamage() { return damage; }  no usages  ▲ Ho Ngoc An
    public boolean shouldRemove() { return remove; }  no usages  ▲ Ho Ngoc An

    public abstract void setHit();  no usages  ▲ Ho Ngoc An

    public abstract void update();  ▲ Ho Ngoc An

    public void draw(Graphics2D g) { super.draw(g); }

}
```

Figure 30: The code of EnemyProjectile class

9. Portal Class:

Function: Stands in for portals that the player may utilize to get between the game's many sections.

Main Duties:

- Controls the portal's status (open, closed, etc.).
- Creates and modifies the animations for portals.
- Manages the player's portal interactions.

```
public Portal(TileMap tm) {  no usages  ▲ Ho Ngoc An

    super(tm);

    width = 81;
    height = 111;

    try {

        BufferedImage spritesheet = ImageIO.read(
            getClass().getResourceAsStream("/Sprites/Other/Portal.gif")
        );

        closedSprites = new BufferedImage[1];
        closedSprites[0] = spritesheet.getSubimage(0, 0, width, height);

        openingSprites = new BufferedImage[6];
        for(int i = 0; i < openingSprites.length; i++) {
            openingSprites[i] = spritesheet.getSubimage(
                i * width, height, width, height
            );
        }

        openedSprites = new BufferedImage[3];
        for(int i = 0; i < openedSprites.length; i++) {
            openedSprites[i] = spritesheet.getSubimage(
                i * width, 2 * height, width, height
            );
        }

        animation.setFrames(closedSprites);
        animation.setDelay(-1);

    }
    catch(Exception e) {
        e.printStackTrace();
    }

}
```

Figure 31: The code of Portal class

10. FireBall Class:

Function: Stands for fireball projectiles that the player or adversaries can utilize.

Main Duties:

- Controls the location, motion, and collision detection of fireballs.
- Draws the fireball and refreshes it.
- Manages how fireballs behave when they hit (e.g., delivering damage, removing after contact).

```
public FireBall(TileMap tm, boolean right) {  no usages  ± Ho Ngoc An *
    super(tm);
    facingRight = right;
    moveSpeed = 3.8;
    if(right) dx = moveSpeed;
    else dx = -moveSpeed;
    width = 30;
    height = 30;
    cwidth = 14;
    cheight = 14;
    // load sprites
    try {
        BufferedImage spritesheet = ImageIO.read(
            getClass().getResourceAsStream(
                "/Sprites/Player/fireball.gif"
            )
        );
        sprites = new BufferedImage[4];
        for(int i = 0; i < sprites.length; i++) {
            sprites[i] = spritesheet.getSubimage(
                i * width,
                0,
                width,
                height
            );
        }
        hitSprites = new BufferedImage[3];
        for(int i = 0; i < hitSprites.length; i++) {
            hitSprites[i] = spritesheet.getSubimage(
                i * width,
                height,
                width,
                height
            );
        }
        animation.setFrames(sprites);
        animation.setDelay(4);
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

Figure 32: The code of FireBall class

11. PlayerSave Class:

Function: organizes the loading and storing of player status information.

Main Duties:

- Keeps track of player information, such as score, lives, and health.
- Allows for the saving and loading of this data, enabling game persistence.

```
public class PlayerSave {  4 usages  Ho Ngoc An

    private static int lives = 3;  3 usages
    private static int health = 5;  3 usages
    private static long time = 0;  3 usages

    public static void init() {  Ho Ngoc An
        lives = 3;
        health = 5;
        time = 0;
    }

    public static int getLives() { return lives; }  no usages  Ho Ngoc An
    public static void setLives(int i) { lives = i; }  no usages  Ho Ngoc An

    public static int getHealth() { return health; }  no usages  Ho Ngoc An
    public static void setHealth(int i) { health = i; }  no usages  Ho Ngoc An

    public static long getTime() { return time; }  no usages  Ho Ngoc An
    public static void setTime(long t) { time = t; }  no usages  Ho Ngoc An

}
```

Figure 33: The Code of PlayerSave class

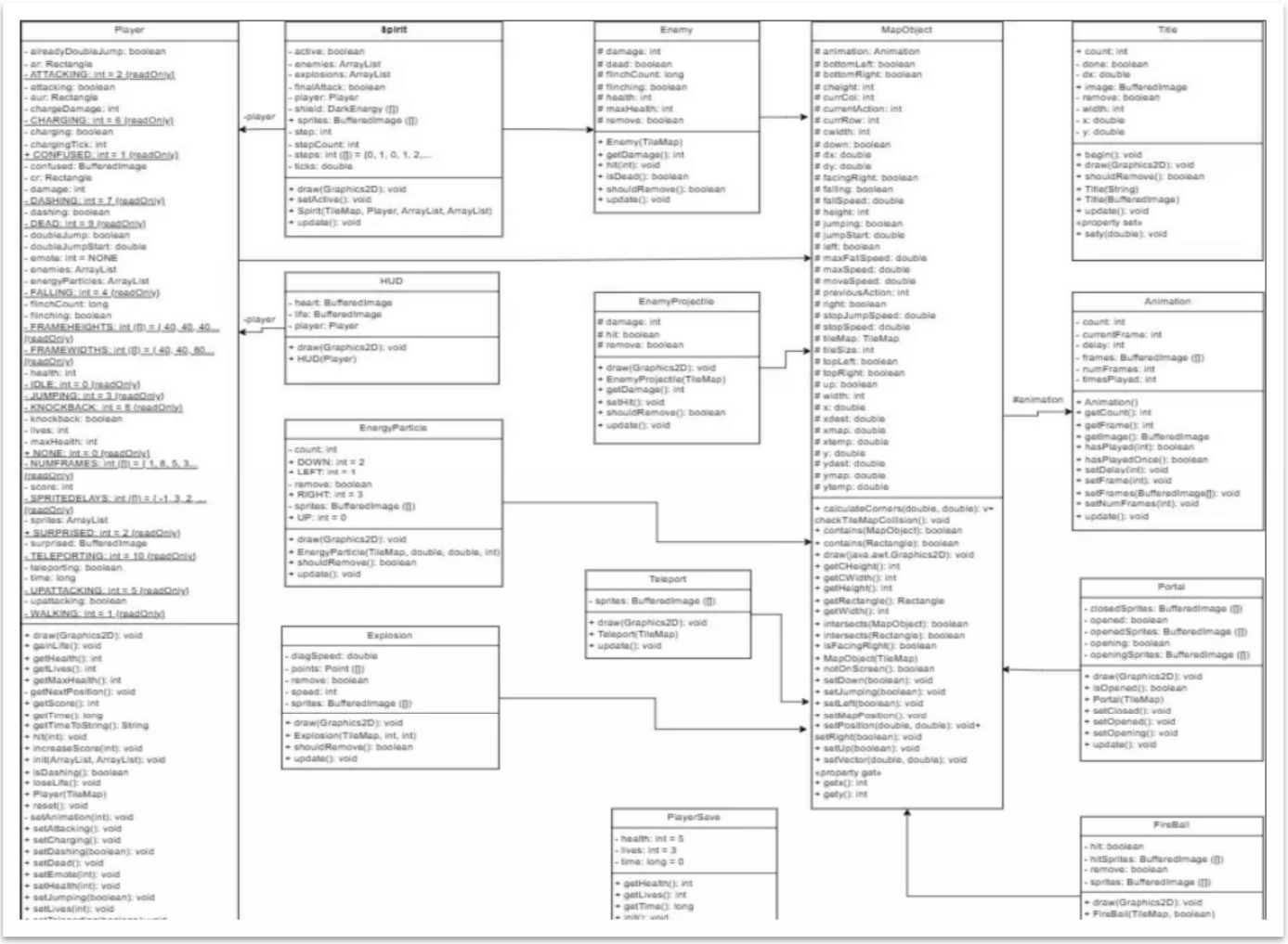

Figure 34: Enemies

7.6 GameState

7.6.1 AcidState:

Manages the state where the game includes acid-based mechanics or visuals.

7.6.2 PauseState:

Manages the game state when the game is paused.

7.6.3 GameStateManager:

Manages all game states, handling transitions and updates between them.

7.6.4 MenuState:

Manages the main menu of the game.

7.6.5 Level1State:

Manages the first level of the game.

7.6.6 Level2State:

Manages the second level of the game.

7.6.7 Level3State:

Manages the third level of the game.

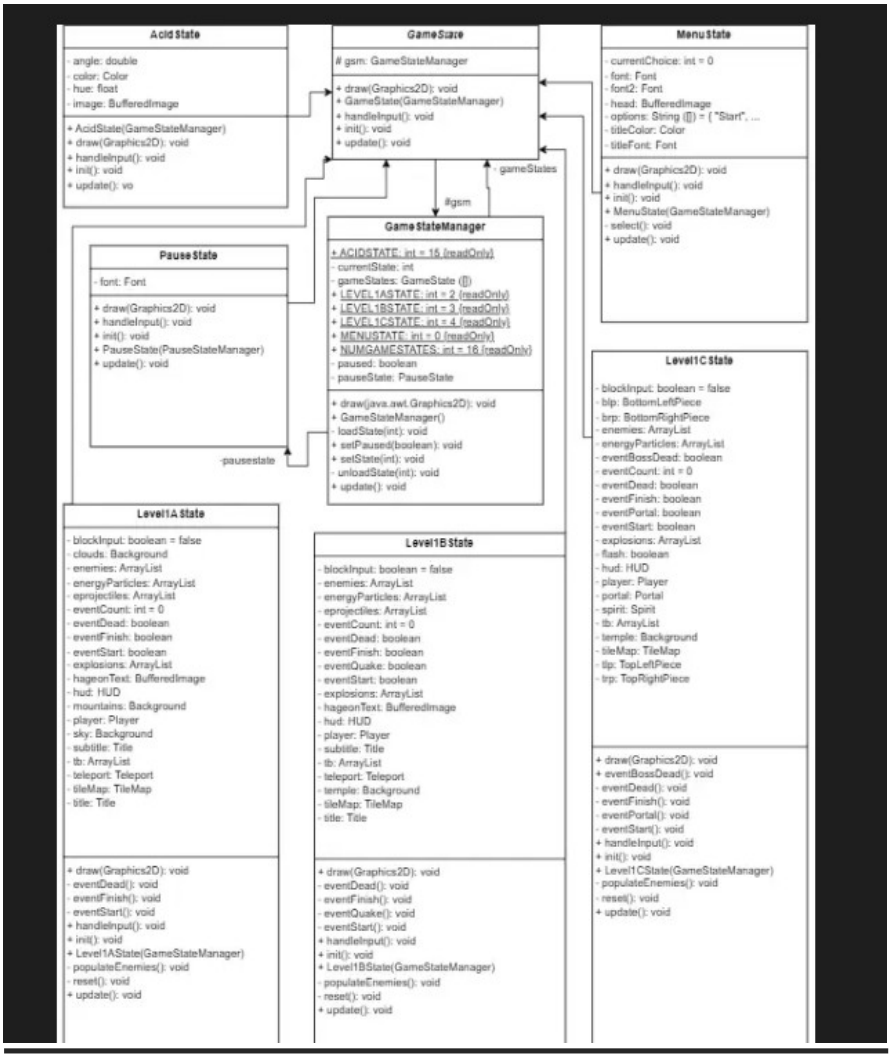

Figure 35: Game State

CHAPTER 8. CONCLUSION

8.1. Future Work:

Our current application still has vital problems that the workers are hoping to address in future implementation. Here are some of them:

8.1.1 Implementing more User Interface (UI):

The user interface (UI) is a bit off since we had not got a user setting site, adding ticket, commenting page to function properly. We modified the login and sign up forms to store the user email in the database, had we not had abundant time to do everything we originally planned. Our team wanted to use ReactJS [5] Framework to make features like forms, buttons, and menus for easier user interaction.

8.1.2 Complete the security system:

The existing application is susceptible to security vulnerabilities due to the absence of decentralized security measures. This means that a customer's token, once acquired, grants unrestricted access and modification privileges to all data in the database, posing a significant risk if the token falls into the wrong hands.

In simple terms, unauthorized possession of a user's token could enable malicious manipulation of the database, resulting in potential property damage and complications. This flaw may lead to the exploitation of crucial information, precipitating serious legal and financial consequences. To address these security concerns, upcoming versions of the database will implement a stringent user access control policy, mitigating the risk of unauthorized access and manipulation.

8.1.3 Packaging:

The current project faces compatibility challenges across platforms, requiring users to manually set up and install necessary dependencies. This process is error-prone and time-consuming, leading to complications and unreliability during the setup. Running the application in diverse settings becomes challenging due to these issues. To streamline this process, we aim to leverage Postman[3] to bundle our application.

By utilizing Postman[3], we intend to simplify the setup process and enhance the application's compatibility across various platforms. This approach aims to provide

users with a more user-friendly and consistent experience, reducing the likelihood of errors and saving time during the installation phase. Postman's capabilities will contribute to a more seamless and reliable deployment of the application across different environments.

8.1.4 Deployment:

Currently, the application is limited to running on localhost, restricting users from accessing it from different locations. To address this limitation and enhance accessibility, there are plans to deploy the application on Amazon Web Services (AWS). By leveraging AWS, we aim to make the application accessible over the internet, providing users with the flexibility to use it from various locations. This deployment strategy not only expands the reach of the application but also contributes to its scalability and availability by utilizing cloud infrastructure and services provided by AWS.

8.2 Summary:

To conclude, despite some shortcomings, we have achieved the main goal of the project. We have followed all the use cases in our Use Case Diagram and created a Database that meets the Third Normal Form. If you want to learn how to use our application for your own project, please contact us!

CHAPTER 9. GITHUB

To ensure transparency and collaboration in the development of the Artifact game, we have maintained a dedicated GitHub repository. This repository serves as the central hub for tracking our progress, managing source code, and organizing project-related assets.

You can explore our development journey, including code iterations, feature additions, and bug fixes, by visiting the following link:
https://github.com/KyungUwU/Artifact

This repository is continuously updated as the project evolves, offering insights into the development process and the underlying codebase that powers the Artifact game. Whether you're looking for inspiration, technical implementation details, or want to contribute, the repository is open for exploration.

LIST OF REFERENCES

1. LibGDX. Introduction to LibGDX. Available at: https://libgdx.com/ (Accessed: 22 December 2024).
2. LibGDX GitHub. Official GitHub Repository. Available at: https://github.com/libgdx/libgdx (Accessed: 22 December 2024).
3. Godot Engine. Official Documentation. Available at: https://docs.godotengine.org/en/stable/ (Accessed: 22 December 2024).
4. LWJGL Wiki. Learn OpenGL for Java. Available at: https://github.com/LWJGL/lwjgl3-wiki/wiki/Introduction (Accessed: 22 December 2024).
5. Tiled Map Editor. Tilemap Design Tool. Available at: https://www.mapeditor.org/ (Accessed: 22 December 2024).
6. Lospec. Pixel Art for Beginners. Available at: https://lospec.com/pixel-art-tutorials (Accessed: 22 December 2024).
7. Box2D. Physics for 2D Games. Available at: https://box2d.org/ (Accessed: 22 December 2024).
8. Gamasutra. Designing Difficulty Levels. Available at: https://www.gamasutra.com/view/feature/166972/dynamic_difficulty_in_video_games.php (Accessed: 22 December 2024).
9. Jumpman GitHub. Open Source Java Platformer. Available at: https://github.com/Apollo-Developers/Jumpman (Accessed: 22 December 2024).
10. Reddit. r/GameDev Community. Available at: https://www.reddit.com/r/gamedev/ (Accessed: 22 December 2024).