# ***OBJECT-ORIENTED PROGRAMMING***

OOP GAME REPORT

***Game project based on the Object - Oriented Principle*  
\_\_\_**



**Name:** Dương Minh Nhựt **Student ID:** ITITUN19002

**Name:** Lê Hữu Sỹ **Student ID:** ITITIU19067

**Name:** Nguyễn Trí Nhân **Student ID:** ITITIU19170

**OVERVIEW**

This report will show the brief overview of the remake, which contains 6 parts:

* Section I: Introduction
* Section II: Github link
* Section III: Game rules
* Section IV: Class diagram
* Section V: Subjective assessment
* Section VI: Future improvements

**SECTION I: INTRODUCTION**

The project is about a small replica of the game The legend of Zelda, which released in 1986 by Nintendo.

The project codes are all written in Java.

**SECTION II: GITHUB LINK**

Click here to move on to our github link which will help you understand clearly about our game.

**SECTION III: GAME RULES**

You will play a role: “*Link*”.

Player will use 5 buttons including 4 arrow keys and the space button on the keyboard to control the character:

* **UP**: to move up
* **DOWN**: to move down
* **LEFT:** to move left
* **RIGHT**: to move right
* **SPACE:** to attack

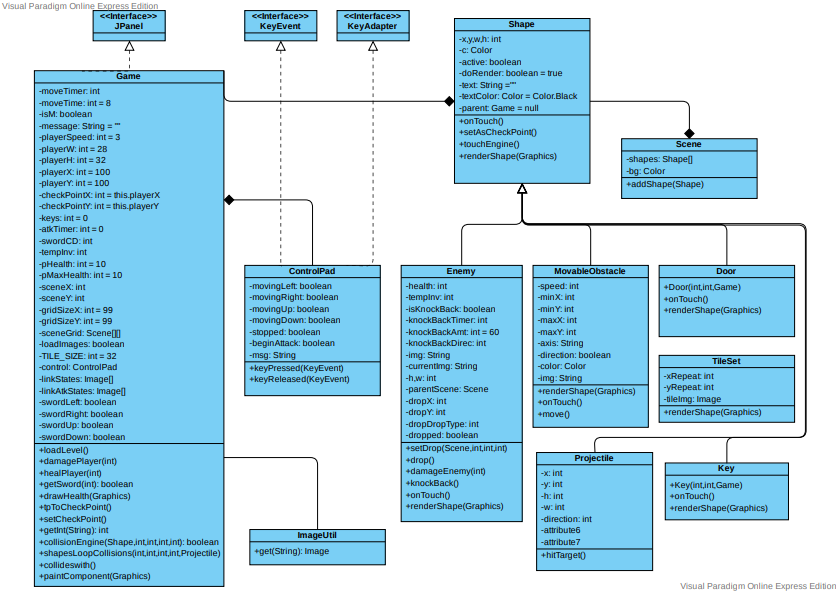
Player’s main goals are to maneouvre through the dungeon while battling against enemy along the way and overcoming the obstacles throughout the map to get to the end of the dungeon.

Player have to finish the current dungeon in order to move on to the next stage and the difficulty of game will increase with each stage.

Player will start with 10 hearts which represent for the lives of the character and it will be deducted when the character is attacked by obstacles and enemies, for detail:

* Enemy: deduct 1 HP/each collision
* Moving Obstacles: deduct 1 HP/each hit

**SECTION IV: CLASS DIAGRAM**



**SECTION V: SUBJECT ASSESSMENT**

1. ***Character*:**

* Character’s principle for movement:
* Getter/Setter:

Let the project have the continuity of the theory of Uniform Access, controls for the important properties that we often use and require exact values.

By using getter/setter, we define the values of orientation, which helps the user to direct and move the character correctly.

**public** Projectile(**int** x, **int** y, **int** direction, Game game) {

**this**.x = x;

**this**.y = y;

**this**.direction = direction;

**this**.parent = game;

}

**public** **int** getDirection() {

**return** **this**.direction;

}

**public** **int** getX() {

**return** **this**.x;

}

**public** **int** getY() {

**return** **this**.y;

}

**public** **int** getW() {

**return** **this**.w;

}

**public** **int** getH() {

**return** **this**.h;

}

* Inheritance: inherit and reuse properties and methods based on the old class. Besides, inheritance is exploited as a way to minimize the boilerplate codes

**import** java.awt.Graphics;

**public** **class** Enemy **extends** Shape {

**private** **int** health;

**private** **int** tempInv;

**private** **boolean** isknockBack;

**private** **int** knockBackTimer;

**private** **final** **int** knockBackAmt = 60;

**private** **int** knockBackDirec;

**private** **final** String img;

**private** String currentImg;

**private** **final** **int** h, w;

**private** Scene parentScene;

**private** **int** dropX;

**private** **int** dropY;

**private** **int** dropType;

**private** **boolean** dropped;

**public** **class** Shape {

**private** **int** x;

**private** **int** y;

**private** **int** w;

**private** **int** h;

**private** Color c;

**private** **boolean** active;

**private** **boolean** doRender = **true**;

**private** String text = "";

**private** Color textColor = Color.BLACK;

**private** Game parent = **null**;

**public** Shape(**int** x, **int** y, **int** w, **int** h, **boolean** block, Color c, **boolean** isActive, Game rpg) {

**this**.x = x;

**this**.y = y;

**this**.w = w;

**this**.h = h;

**this**.c = c;

**this**.active = isActive;

**this**.parent = rpg;

}

1. ***Mistakes*:**

* Due to the disadvantage of “Decorator Design Pattern”, our project is a close principle, letting the user/ programmar have posibility to fix future problem when developing 🡪 which let our program is becoming “Spaghetti Code” 🡪 Find it difficuty to debug and also for implementing or adding a new function/features.

**SECTION VI. FUTURE IMPROVEMENT**

As this is the first time we literally develop a game, therefore, there is an array of mistakes and shortages.

To make this game perfectly, we think these criterias can improve our problems:

1. **Expand the scenes of game:** In our game, there is just 6 scenes for players' ongoing interaction so that later on, we need to grow our game with numerous rounds and levels to improve the players' encounters. Plus, we need to make a chief or a test toward the finish of each round to make the challenges to animate players**.**
2. **Create another feature for ostacles and also character:** In the future, we need to implement and upgrade the movement or the act of character. For examples, the character can collect weapons, use the gun or the enemies can move faster and spit fire.
3. **Adjust the quality of game image:** We need to apply more function related to GUI, Unity as well… to develop a more attractive game.
4. **Create a Boss:** we also tried to create boss, but there was an array of mistakes due to anti-pattern, so the boss wouldn’t be death. So in the future, we need to find a way to obtimize the code, the design-pattern and find out the way to implement bosses at the end of each rounds.
5. **Create some remaining station after expanding game:** There are numerous rounds to play, so we should have some save scenes to make players consoled about their records. The instrument is that players will stroll through over the save station to save the game. There will be a new class: Savepoints, and also an boolean functions check is.death.
6. **Optimize the code:** We will apply appropriate Design Pattern to debug the code easily and also optimize codes for more convenient when implement a function or a new feature as well