INTERNATIONAL UNIVERSITY
VIETNAM NATIONAL UNIVERSITY, HCM CITY

School of Computer Science & Engineering

FALLEN KINGDOM TD PROJECT

Advisor: Tran Thanh Tung
Course: Object – Oriented Programming

Group members:

Nguyễn Đặng Lan Anh	ITITIU20155
Lê Nguyễn Bảo Long	ITITIU20138
Nguyễn Huy Thanh	ITITIU20308
Nguyễn Quang Duy	ITITIU20196

Contents

I. Introduction:

Tower defense (TD) is a sub-genre of strategy games where the goal is to defend a player's territories or possessions by obstructing the enemy attackers or by stopping enemies from reaching the exits, usually achieved by placing defensive structures on or along their path of attack.

This typically means building a variety of different structures that serve to automatically block, impede, attack or destroy enemies.

Tower defense is seen as a sub-genre of real-time strategy video games, due to its real-time origins, even though many modern tower defense games include aspects of turn-based strategy.

Strategic choice and positioning of defensive elements is an essential strategy of the genre.

This project was designed based on Object – Oriented Programming method by Java language.

II. Properties of Tower Defence:

1. Goal:

Create a complete game in Java.

Graphic design for the game.

Gain the basic knowledge on animations, game loop, path finding, even handling, and rendering animations.

2. Rule:

The player must use "obstruction", which can inflict damage or cause them to move more slowly, to protect the "base" from incoming waves of "enemy".

"Base" have a limited "heart" - only a certain number of "enemy" can pass through the "base".

There are many kinds of "enemies", some of which can be on the ground or in the air and require different "obstructions" to deal with. There are also "enemies" with faster movement, tougher "enemies",... etc.

Due to the numerous “enemies”, there are various different types of “obstructions” that cost different “currencies”.

The player will receive a little amount of “currency” to purchase certain “obstructions” at the beginning of the first wave, “currency” also can get by killing an “enemy”, by time, or losing “heart”.

When a specific number of “enemies” are killed or the timer expires, the wave ends.

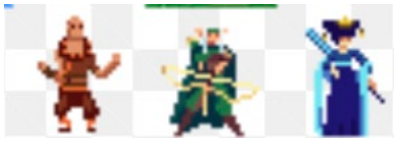
III. Literature:

1) Objects:

There are 2 types of characters: Enemy, Tower (protector).

Enemy types: there are Golem, Witch, Knight, and Vampire. They are invaders who try to go along the road to destroy the base. they have different speed and health.

Tower types: there are Cannon, Archer, Wizard. They are defined as the protectors who are arranged by the player to defend the enemies.



Weapon types that Tower use to kill the enemies included arrow, witchcraft, boom.

2) Other functions:

a) Game loop:

In class Game:we make game loop, it's purpose was for running the game in any device with the same speed (FPS). Moreover, we also created update() method to move object, check event, check collision for later.

```
private void updateGame() {
    switch (GameStates.gameState) {
    case EDIT:
        editing.update();
        break;
    case MENU:
        break;
    case PLAYING:
        playing.update();
        break;
    case SETTINGS:
        break;
    default:
        break;
    }
}
```

b) Load Image:

In the sprieatlast image was contains all the necessary Images of enemy, tower and tile for game design.

```
private void loadEnemyImgs() {
    BufferedImage atlas = LoadSave.getSpriteAtlas();
    for (int i = 0; i < 4; i++)
        enemyImgs[i] = atlas.getSubimage(i * 32, y: 32, w: 32, h: 32);

}
```

```java
private void loadTowerImgs() {
    BufferedImage atlas = LoadSave.getSpriteAtlas();
    towerImgs = new BufferedImage[3];
    for (int i = 0; i < 3; i++)
        towerImgs[i] = atlas.getSubimage((4 + i) * 32, y: 32, w: 32, h: 32);
}
```

```java
public class TileManager {

    public Tile GRASS, WATER, ROAD_LR, ROAD_TB, ROAD_B_TO_R, ROAD_L_TO_B, ROAD_L_TO_T,
            ROAD_T_TO_R, BL_WATER_CORNER, TL_WATER_CORNER,
            TR_WATER_CORNER, BR_WATER_CORNER, T_WATER, R_WATER, B_WATER,
            L_WATER, TL_ISLE, TR_ISLE, BR_ISLE, BL_ISLE;

    private BufferedImage atlas;
    public ArrayList<Tile> tiles = new ArrayList<>();

    public ArrayList<Tile> roadsS = new ArrayList<>();
    public ArrayList<Tile> roadsC = new ArrayList<>();
    public ArrayList<Tile> corners = new ArrayList<>();
    public ArrayList<Tile> beaches = new ArrayList<>();
    public ArrayList<Tile> islands = new ArrayList<>();

    public TileManager() {

        loadAtalas();
        createTiles();

    }
```

c) Draw image:

```java
public void paintComponent(Graphics g) {
    super.paintComponent(g);

    game.getRender().render(g);

}
```

d) Set Font:

e) Input: (keyboard and mouse)

Key press

```
23      @Override
24      public void keyPressed(KeyEvent e) {
25          if (GameStates.gameState == EDIT)
26              game.getEditor().keyPressed(e);
27          else if (GameStates.gameState == PLAYING)
28              game.getPlaying().keyPressed(e);
29
30      }
```

Mouse click

```java
@Override
public void mouseClicked(MouseEvent e) {
    if (e.getButton() == MouseEvent.BUTTON1) {
        switch (GameStates.gameState) {
        case MENU:
            game.getMenu().mouseClicked(e.getX(), e.getY());
            break;
        case PLAYING:
            game.getPlaying().mouseClicked(e.getX(), e.getY());
            break;
        case SETTINGS:
            game.getSettings().mouseClicked(e.getX(), e.getY());
            break;
        case EDIT:
            game.getEditor().mouseClicked(e.getX(), e.getY());
            break;
        case GAME_OVER:
            game.getGameOver().mouseClicked(e.getX(), e.getY());
            break;
        default:
            break;
        }
    }
}
```

Mouse press

```java
@Override
public void mousePressed(MouseEvent e) {
    switch (GameStates.gameState) {
    case MENU:
        game.getMenu().mousePressed(e.getX(), e.getY());
        break;
    case PLAYING:
        game.getPlaying().mousePressed(e.getX(), e.getY());
        break;
    case SETTINGS:
        game.getSettings().mousePressed(e.getX(), e.getY());
        break;
    case EDIT:
        game.getEditor().mousePressed(e.getX(), e.getY());
        break;
    case GAME_OVER:
        game.getGameOver().mousePressed(e.getX(), e.getY());
        break;
    default:
        break;
    }
}
```

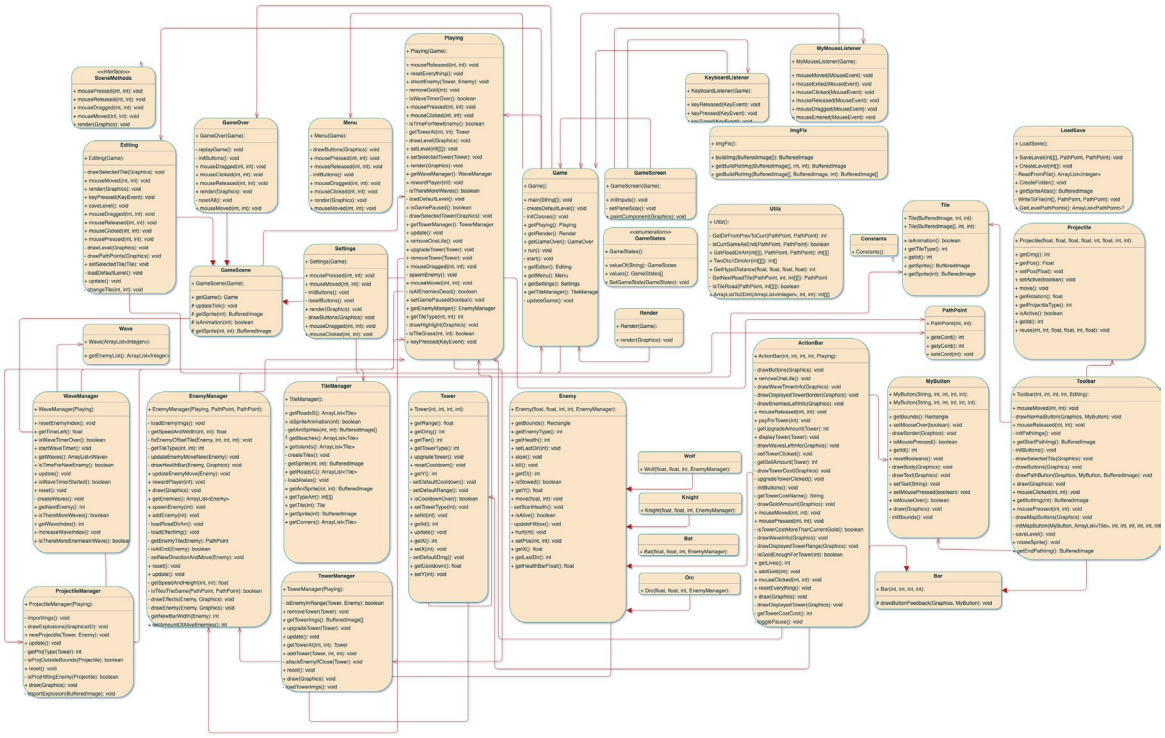
Mouse drag

```
18      @Override
19      public void mouseDragged(MouseEvent e) {
20          switch (GameStates.gameState) {
21          case MENU:
22              game.getMenu().mouseDragged(e.getX(), e.getY());
23              break;
24          case PLAYING:
25              game.getPlaying().mouseDragged(e.getX(), e.getY());
26              break;
27          case SETTINGS:
28              game.getSettings().mouseDragged(e.getX(), e.getY());
29              break;
30          case EDIT:
31              game.getEditor().mouseDragged(e.getX(), e.getY());
32              break;
33          default:
34              break;
35          }
36
37      }
```

Mouse release

```
@Override
public void mouseReleased(MouseEvent e) {
    switch (GameStates.gameState) {
    case MENU:
        game.getMenu().mouseReleased(e.getX(), e.getY());
        break;
    case PLAYING:
        game.getPlaying().mouseReleased(e.getX(), e.getY());
        break;
    case SETTINGS:
        game.getSettings().mouseReleased(e.getX(), e.getY());
        break;
    case EDIT:
        game.getEditor().mouseReleased(e.getX(), e.getY());
        break;
    case GAME_OVER:
        game.getGameOver().mouseReleased(e.getX(), e.getY());
        break;

    default:
        break;
    }
}
```

IV. Project UML:



(https://drive.google.com/file/d/1WdzJ85sfy8xMxWY5CBL-8hLDK0uZKfDb/view?usp=sharing)

V. Methodology:

For the better understanding about the project, we have the project Tower Defence is a combination of 9 packages. Each of them represents a different function of the game.

List of packages:

- Main
- Enemy
- Managers
- Objects
- Helpz
- Scenes
- Event
- UI

Here are the overview of some packages which we considered as the remakable functions for game process.

First, in package main, it contained some function like Virtual Screen size, game screen, fps and responsible for the first setting of game.

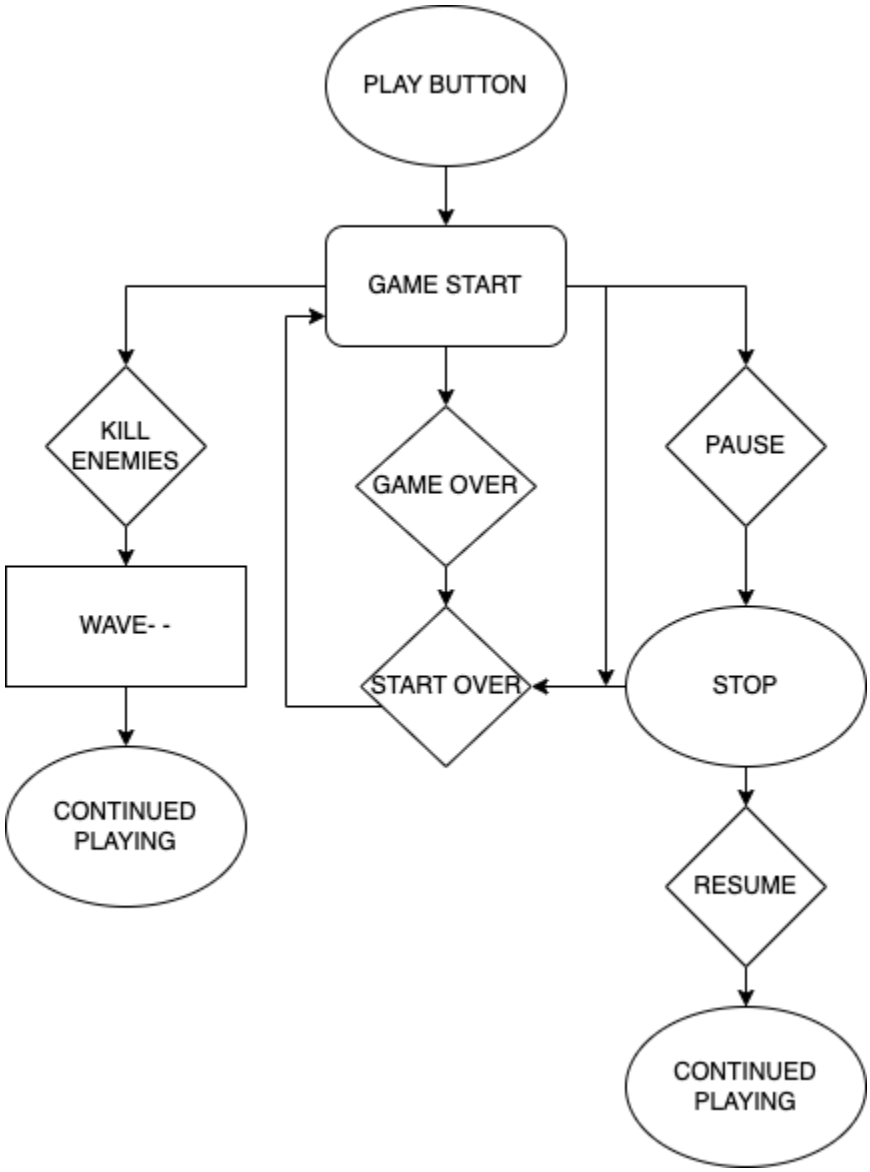
Next, we build the input in package Inputs, it recognized activity from the keyboard and mouse: click, hold, press, ... this was for game playing. In addition, our game is mainly playing by using mouse, and keyboard is use for switching between clients and setting game.

Package managers , it included the methods to create and load the images of enemy, tower (or protector) and tiles into the game for game design later. it also had the movement of the objects as well.

Furthermore, package Scenes, the main function of this package is to design and set the game client which contained game theme, playground, setting,...

Finally, we connect them logically, and put it in method Main for game operating.

Tower Defence 's Algorithm



V. Main class - Function - Explaination:

1) Characters and it's properties:

We have defined characters in class Constant (package helpz) and set the ID for each of them.

Class Enemies:

```
public static final int GOLEM = 0;
public static final int WITCH = 1;
public static final int KNIGHT = 2;
public static final int VAMPIRE = 3;
```

Method GetSpeed: is the method that initialized the speed of the enemies

Method GetStartHealth: it assigned the health for each enemies.

To make the game balanced, the properties of each enemy types are different because they have the distinctive role.

```
public static float GetSpeed(int enemyType) {
    switch (enemyType) {
    case GOLEM:
        return 0.5f;
    case WITCH:
        return 0.7f;
    case KNIGHT:
        return 0.45f;
    case VAMPIRE:
        return 0.85f;
    }
    return 0;
}
```

```
public static int GetStartHealth(int enemyType) {
    switch (enemyType) {
    case GOLEM:
        return 85;
    case WITCH:
        return 100;
    case KNIGHT:
        return 400;
    case VAMPIRE:
        return 125;
    }
    return 0;
}
```

Class Towers:

```
public static final int CANNON = 0;
public static final int ARCHER = 1;
public static final int WIZARD = 2;
```

In package helpz class Constants, we first initialized the distinctive properties of each tower: Damage, Range, Cool down.

Method GetStartDmg: assumed the different damage for each towers type. It's the amount of damage a tower can hit an enemy at a limit time.

```
public static int GetStartDmg(int towerType) {
    switch (towerType) {
    case CANNON:
        return 15;
    case ARCHER:
        return 5;
    case WIZARD:
        return 0;
    }

    return 0;
}
```

Method GetDefaultRange: the range which allows a tower can work and hit the enemies. If the enemy is not within that range then the tower will stop attacking.

```
public static float GetDefaultRange(int towerType) {
    switch (towerType) {
    case CANNON:
        return 75;
    case ARCHER:
        return 120;
    case WIZARD:
        return 100;
    }

    return 0;
}
```

Method GetDefaultCooldown: we created this method to prevent the tower from dealing continuous damage to enemies, which mean the attacking speed of the towers, after firing they will count down a certain amount of time to fire the next shot

```
public static float GetDefaultCooldown(int towerType) {
    switch (towerType) {
    case CANNON:
        return 120;
    case ARCHER
        return 35;
    case WIZARD:
        return 50;
    }

    return 0;
}
```

Method GetTowerCost: in order to use the towers, the player must purchase them with the original default amount. Because each tower has its own strengths and weaknesses, they are sold at different prices.

```
public static int GetTowerCost(int towerType) {
    switch (towerType) {
    case CANNON:
        return 65;
    case ARCHER:
        return 35;
    case WIZARD:
        return 50;
    }
    return 0;
}
```

```
public void upgradeTower() {
    this.tier++;

    switch (towerType) {
    case ARCHER:
        dmg += 2;
        range += 20;
        cooldown -= 5;
        break;
    case CANNON:
        dmg += 5;
        range += 20;
        cooldown -= 15;
        break;
    case WIZARD:
        range += 20;
        cooldown -= 10;
        break;
    }
}
```

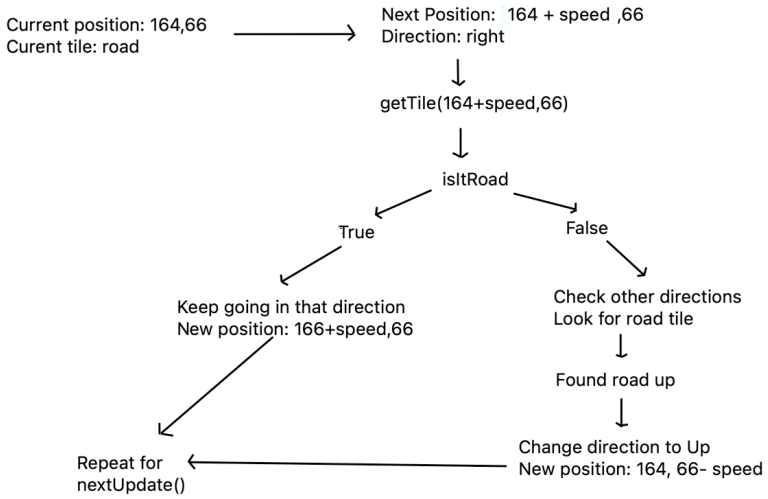
2) Enemy move:

In order to get every single enemies, we created the loop:

```
private void loadEnemyImgs() {
    BufferedImage atlas = LoadSave.getSpriteAtlas();
    for (int i = 0; i < 4; i++)
        enemyImgs[i] = atlas.getSubimage(i * 32, y: 32, w: 32, h: 32);

}
```

Then, add method draw to display the enemies into the screen.

Direction's algorithm explanation:



Folow the alrogithm above, base on the current position of the enemies, we created the method for checking the surrounding tile whether it is a road or not, if it is, the enemies will go along that direction, then repeat the checking process for the following movement.

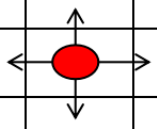
In package enemies class enemy, we defined the fourth direction and assumed the last direction of the enemies as the current position, then checking on each tile.

+If the road is in the LEFT, x will be decreased which mean the enemies will move to the left direction , similar to UP.

+If the road is in the RIGHT, x will be increased which mean the enemies will move to the right direction, similar to Down.

This is an illustration of the map and the red circle is the enemy.

y \ x	0	1	2	...	n
0	0,0	0,1	0,2	...	0,n
1	1,0				
2	2,0				
.	.				
.	.				
n	n,0				



```java
public void move(float speed, int dir) {
    lastDir = dir;

    if (slowTick < slowTickLimit) {
        slowTick++;
        speed *= 0.5f;
    }

    switch (dir) {
    case LEFT:
        this.x -= speed;
        break;
    case UP:
        this.y -= speed;
        break;
    case RIGHT:
        this.x += speed;
        break;
    case DOWN:
        this.y += speed;
        break;
    }

    updateHitbox();
}
```

(Method updateHitBox is for later feature)

In package manager class enemyManager, we created the method setNewDirectionAndMove. It is responsibility for checking the next tile, lastDir equal to dir assume the dir when the enemy move to lastDir one by one.

```
private void setNewDirectionAndMove(Enemy e) {
    int dir = e.getLastDir();

    int xCord = (int) (e.getX() / 32);
    int yCord = (int) (e.getY() / 32);

    fixEnemyOffsetTile(e, dir, xCord, yCord);

    if (isAtEnd(e))
        return;

    if (dir == LEFT || dir == RIGHT) {
        int newY = (int) (e.getY() + getSpeedAndHeight(UP, e.getEnemyType()));
        if (getTileType((int) e.getX(), newY) == ROAD_TILE)
            e.move(GetSpeed(e.getEnemyType()), UP);
        else
            e.move(GetSpeed(e.getEnemyType()), DOWN);
    } else {
        int newX = (int) (e.getX() + getSpeedAndWidth(RIGHT, e.getEnemyType()));
        if (getTileType(newX, (int) e.getY()) == ROAD_TILE)
            e.move(GetSpeed(e.getEnemyType()), RIGHT);
        else
            e.move(GetSpeed(e.getEnemyType()), LEFT);

    }
}
```

the method updateEnemyMove, after moving, this method will set the new position of the enemy to the current. The purpose is updating the current move and preparing for the next checking.

```
private void updateEnemyMoveNew(Enemy e) {
    PathPoint currTile = getEnemyTile(e);
    int dir = roadDirArr[currTile.getyCord()][currTile.getxCord()];

    e.move(GetSpeed(e.getEnemyType()), dir);

    PathPoint newTile = getEnemyTile(e);

    if (!isTilesTheSame(currTile, newTile)) {
        if (isTilesTheSame(newTile, end)) {
            e.kill();
            playing.removeOneLife();
            return;
        }
        int newDir = roadDirArr[newTile.getyCord()][newTile.getxCord()];
        if (newDir != dir) {
            e.setPos(newTile.getxCord() * 32, newTile.getyCord() * 32);
            e.setLastDir(newDir);
        }
    }

}
```

3) Action:

This part included hp bar, shoot effect, projectile path.

a) Hp bar:

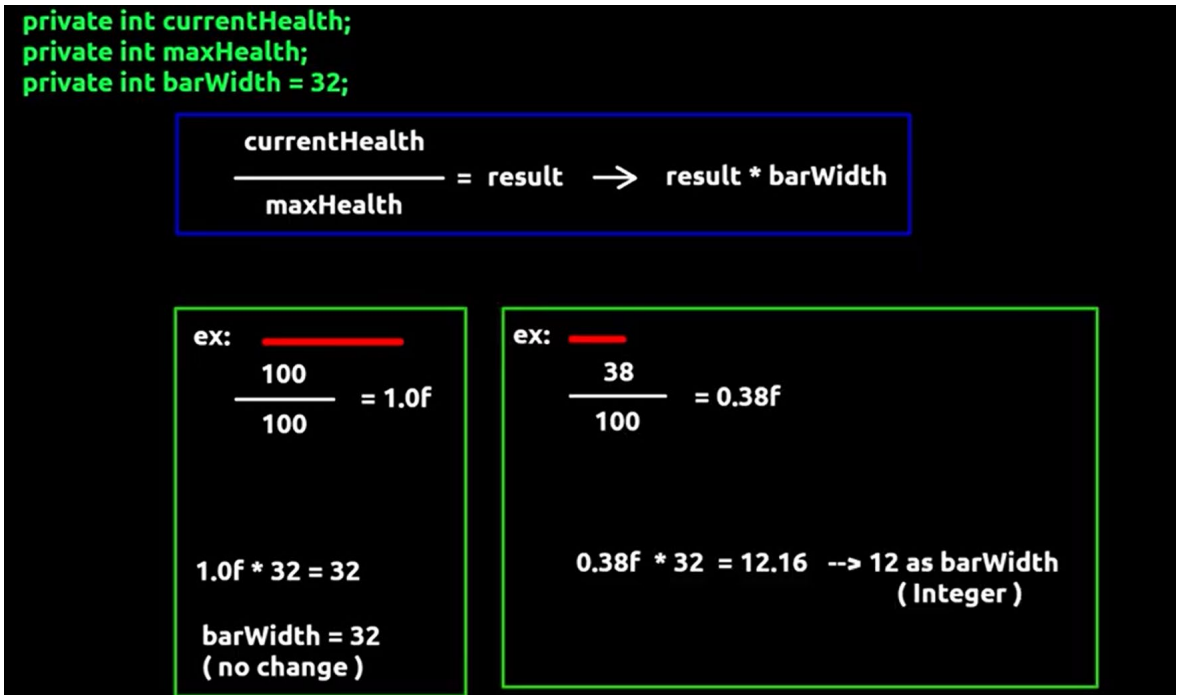
- Health bar: it's a bar that shows the enemy's health. When the tower shoots at the enemy, the amount of health will be deducted equal to the amount of the enemy's health. When their losing health the bar gets smaller and smaller until the bar disappears which mean the enemy has been killed.

The way to do this is we declared two value : current health and max health. At the beginning, the starting health is equal to the max health. Let current health divided by max health equals result. If the enemy is attacked, the current health would been lost the value and the health bar gotten smaller and smaller.

Example:

Before the collision, the result equals to 1 which mean the default health is 100%. Thus, there is no change in bar width.

After the collision, the current health is decreased making the result to be reduced as well. When the result the result of multiplying the bar, the bar will be shortened.



For calculation, we set the setStartHealth method in the enemies package Enemy class.

```
private void setStartHealth() {
    health = helpz.Constants.Enemies.GetStartHealth(enemyType);
    maxHealth = health;
}
```

-Then we generated the method below to get the health bar and cast maxHealth to float:

```
public float getHealthBarFloat() {
    return health / (float) maxHealth;
}
```

-In EnemyManager class, we added these method to method draw. The intention is to draw the health bar on the top of the head of the enemies.

```
public void draw(Graphics g) {
    for (Enemy e : enemies) {
        if (e.isAlive()) {
            drawEnemy(e, g);
            drawHealthBar(e, g);
            drawEffects(e, g);
        }
    }
}
```

-Then we set the color and the bar when the health of enemy being lower:

```
rivate void drawHealthBar(Enemy e, Graphics g) {
    g.setColor(Color.red);
    g.fillRect((int) e.getX() + 16 - (getNewBarWidth(e) / 2), (int) e.getY() - 10, getNewBarWidth(e), height: 3);
```

b) Shoot effect:

Projectile:

- It's an animation show what towers are shooting. We don't use a hit box because it's so small so we just use distance.

- We created a projectile, in there we made a boolean method that it will active if the enemies in range.

In Constant class, we creatde a new static method of projectile to get the speed of the weapons:

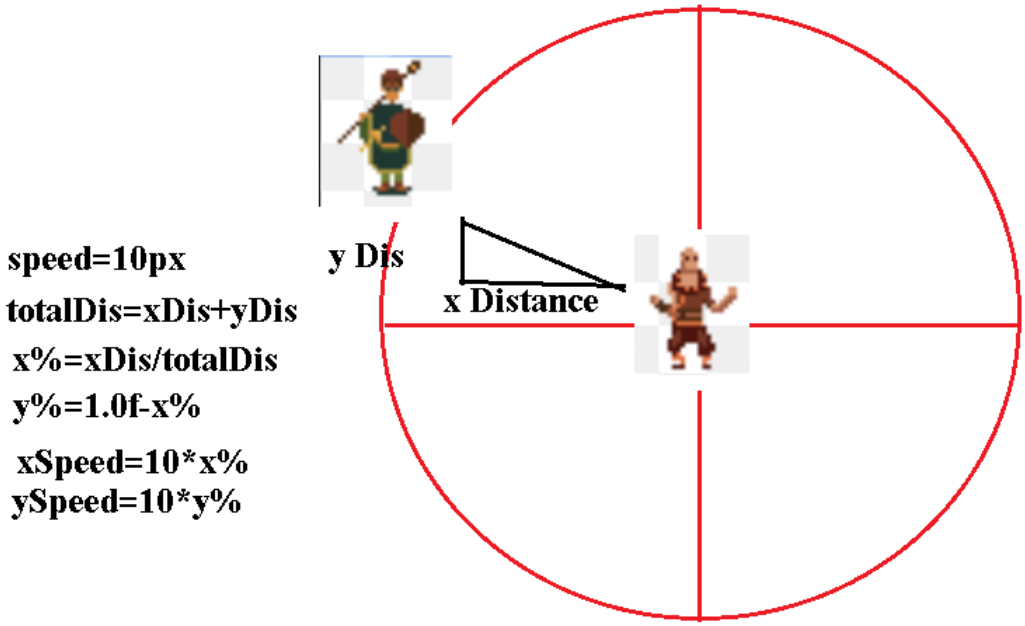
```java
public static class Projectiles {
    public static final int ARROW = 0;
    public static final int CHAINS = 1;
    public static final int BOMB = 2;

    public static float GetSpeed(int type) {
        switch (type) {
        case ARROW:
            return 8f;
        case BOMB:
            return 4f;
        case CHAINS:
            return 6f;
        }
        return 0f;
    }
}
```

In class ProjectileManager, we use the image that we have inserted before:

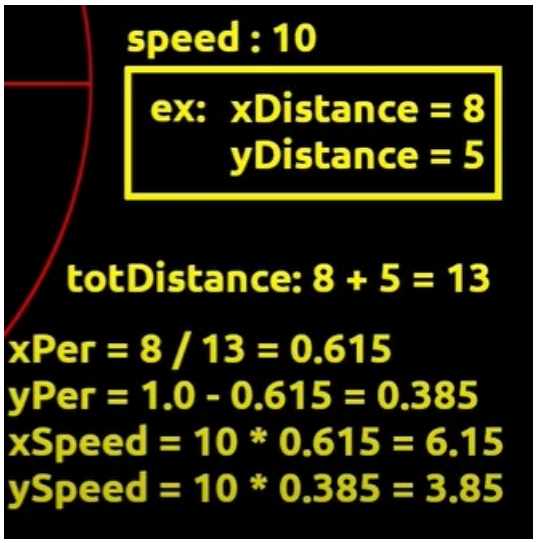
```java
private void importImgs() {
    BufferedImage atlas = LoadSave.getSpriteAtlas();
    proj_imgs = new BufferedImage[3];

    for (int i = 0; i < 3; i++)
        proj_imgs[i] = atlas.getSubimage((7 + i) * 32, y: 32, w: 32, h: 32);
    importExplosion(atlas);
}
```

Since enemies are not always going to be in any of those directions so we had some ways do deal with this issue.

At the beginning, we have a tower and an enemy with some random position. We need to know how the projectile go in x distance and y distance . Now we need the total distance in length by x dis plus y distance. So that, we can divide x distance with the total distance to get the percentage of the total distance for x distance. Then we get the x speed by multiplying speed and also with y speed.



Example:



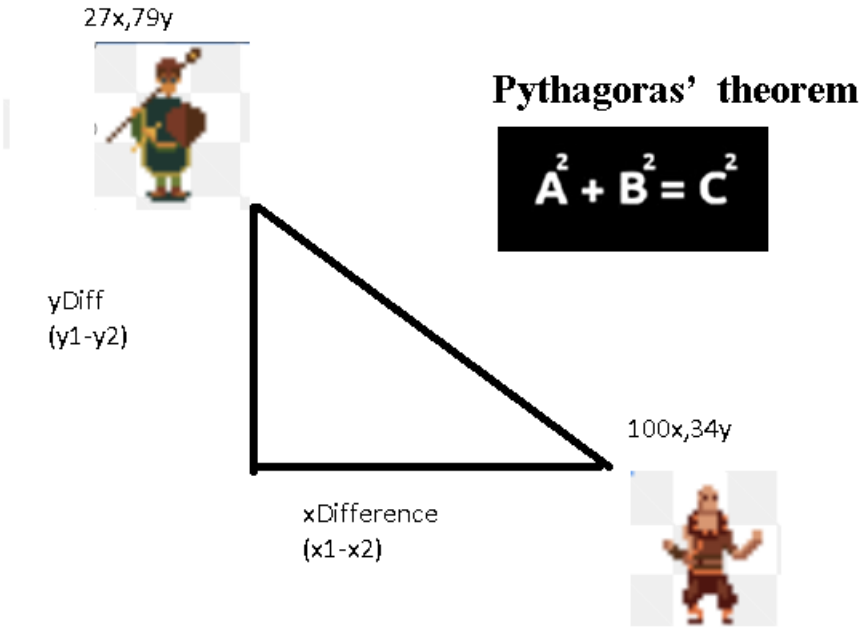
Then we created new variables of xSpeed and ySpeed, brought them into our projectile and added them in constructor. Then in ProjectileManager, we created a new method to show the projectile around the tower.

4) Combat:

a) Range checking:

+First, we have to checking if the enemies are close enough for tower to shot. We used Pythagoras' theorem to get the actual real distance between the tower and the enemy by subtracting the positions with each other to get the hypotenuse. We counted the distance of the enemy and tower if it is less than the actual range of the tower, the tower can hit the enemy.

+We used math.abs which is absolute math to make sure that the distance is always positive. If x1 is larger than the other when we do a subtraction then it might end up a negative number. So we converted that to positive and that's what math.abs and we used this method in order to get the actual real distance.



Now in the managers package TowerManager class, inside the update method, we added a method called attackEnemyIfClose to check the range. We used loop to access our enemies array.

Then check if the enemies is alive or and in range, the tower will shoot them, then the tower will cool down. When cool down is over the tower will repeat the process until the enemies out the range.

```
private void attackEnemyIfClose(Tower t) {
    for (Enemy e : playing.getEnemyManger().getEnemies()) {
        if (e.isAlive())
            if (isEnemyInRange(t, e)) {
                if (t.isCooldownOver()) {
                    playing.shootEnemy(t, e);
                    t.resetCooldown();
                }
            } else {
                // we do nothing
            }
    }

}
```

b) Explosions:

In the class ProjectileManagear, we created a method to make no rotate to bombs and chains. In here, we used the image that we inputed before having the image explosion.

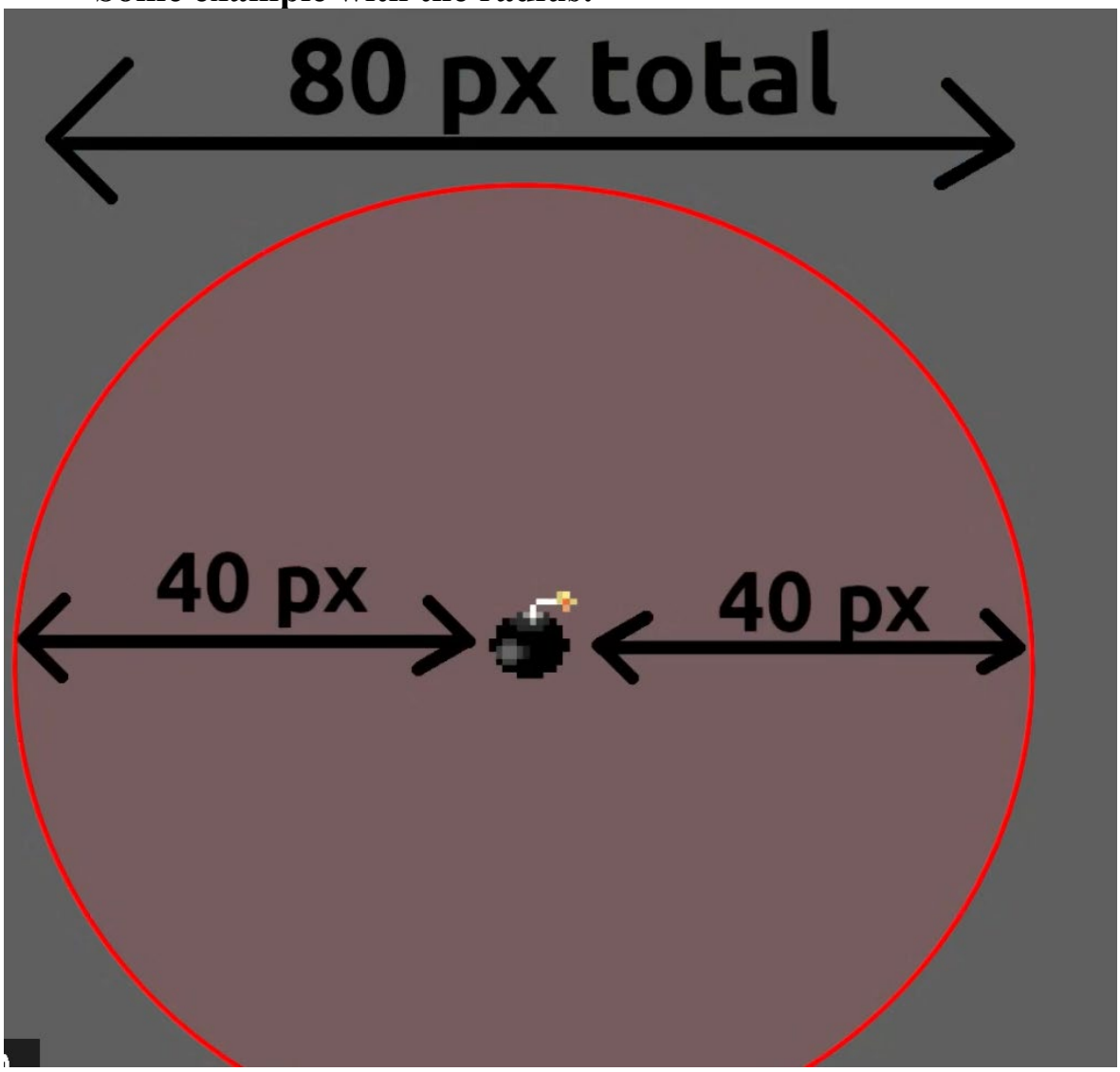
```
public void draw(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;

    for (Projectile p : projectiles)
        if (p.isActive()) {
            if (p.getProjectileType() == ARROW) {
                g2d.translate(p.getPos().x, p.getPos().y);
                g2d.rotate(Math.toRadians(p.getRotation()));
                g2d.drawImage(proj_imgs[p.getProjectileType()], -16, -16, observer: null);
                g2d.rotate(-Math.toRadians(p.getRotation()));
                g2d.translate(-p.getPos().x, -p.getPos().y);
            } else {
                g2d.drawImage(proj_imgs[p.getProjectileType()], (int) p.getPos().x - 16, (int) p.getPos().y - 16,
            }
        }

    drawExplosions(g2d);

}
```

Then, we got a inner explosion class in ProjectileManager with a constructor.

```
public class Explosion {

    private Point2D.Float pos;
    private int exploTick, exploIndex;

    public Explosion(Point2D.Float pos) {
        this.pos = pos;
    }

    public void update() {
        exploTick++;
        if (exploTick >= 6) {
            exploTick = 0;
            exploIndex++;
        }
    }

    public int getIndex() {
        return exploIndex;
    }

    public Point2D.Float getPos() {
        return pos;
    }
}
```

- AOE effect : we made a radius for our explosion and created float value radius.

Some example with the radius:



Then we checked the distance when enemies is closer than the radius. If the enemies are closer, it will take the damage. We had some method to do that .

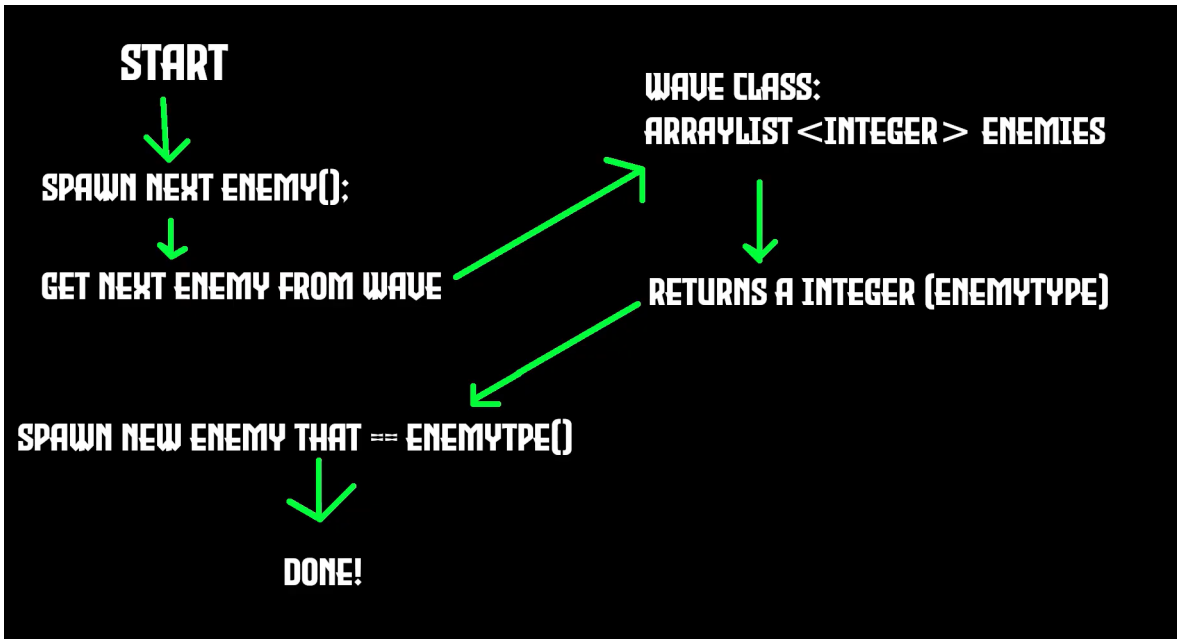
```java
private void explodeOnEnemies(Projectile p) {
    for (Enemy e : playing.getEnemyManger().getEnemies()) {
        if (e.isAlive()) {
            float radius = 40.0f;

            float xDist = Math.abs(p.getPos().x - e.getX());
            float yDist = Math.abs(p.getPos().y - e.getY());

            float realDist = (float) Math.hypot(xDist, yDist);

            if (realDist <= radius)
                e.hurt(p.getDmg());
        }

    }
```

5) Event:

In here, we control how much enemies will arrive, the amount of enemies, the types of enemies, the delay of each wave, the time left to next waves and the amount of enemies left.

First, we created a wave class. this hold a list of integers that represent the type of enemies and when actives it loops in the EnemyManager and go to the next integers in that list and then create more enemies in that integers.

This is the algorithm of how the waves work:



We created WaveManager class and wrote some constructor in here. Then turn back to the EnemyManager to create a new method for spawning enemies:

```
private boolean isTimeForNewEnemy() {
    if(playing.getWaveManager().isTimeForNewEnemy()) {

    }

    return false;
```

6) UI

MyButton class:

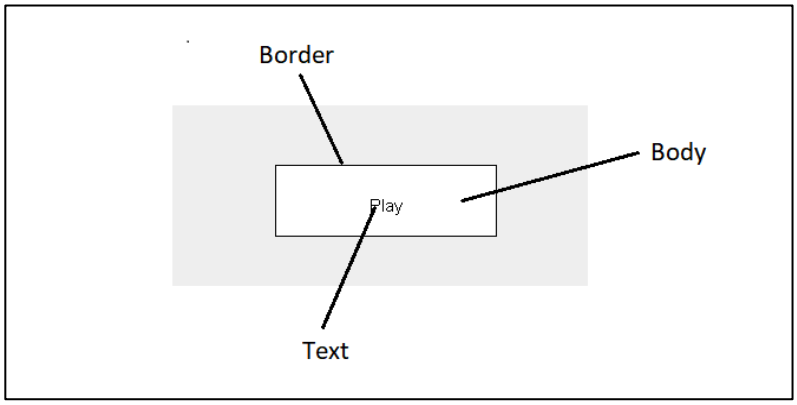
To create a button, we need coordination(x,y), width, height and the text of the button in the constructor

```java
public MyButton(String text, int x, int y, int width, int height) {
    this.text = text;
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.id = -1;

    initBounds();
}
```

In the constructor, there is also the method call “initBounds” , which we use to create a new Rectangle class to check if the mouse is inside the button.

```java
private void initBounds() {
    this.bounds = new Rectangle(x, y, width, height);
}
```

And to draw the button, we divided into 3 method to draw 3 parts of a button which are body, border and text.



```java
public void draw(Graphics g) {
    // Body
    drawBody(g);

    // Border
    drawBorder(g);

    // Text
    drawText(g);
}
```

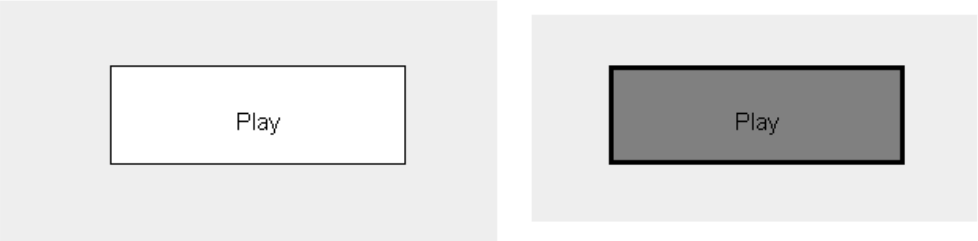
For easier for user to realize the mouse is inside the button, we decided that the body of the button will turn into gray instead of white when the mouse is inside the button. The mouseOver is boolean type whose value equal true if the mouse is inside the button.

```java
private void drawBody(Graphics g) {
    if (mouseOver)
        g.setColor(Color.gray);
    else
        g.setColor(Color.WHITE);
    g.fillRect(x, y, width, height);

}
```

Similarly, we want the border will be bold if the button being pressed.The mousePress is boolean type whose value will equal true if the mouse press the button.

```java
private void drawBorder(Graphics g) {

    g.setColor(Color.black);
    g.drawRect(x, y, width, height);
    if (mousePressed) {
        g.drawRect(x + 1, y + 1, width - 2, height - 2);
        g.drawRect(x + 2, y + 2, width - 4, height - 4);
    }

}
```

And here is the result.



Bar, ToolBar,ActionBar class:

The bar which can allow the user interact with the game are used for Playing and Editing game state are quite similar.So we create the class “Bar” and other two classes “ActionBar” and “ToolBar” will inheritance from it

In the class “Bar”, there is the construct similar to the class “Mybutton” ,and the method to draw the button called “drawButtonFeedback” .

```
protected void drawButtonFeedback(Graphics g, MyButton b) {
    // MouseOver
    if (b.isMouseOver())
        g.setColor(Color.white);
    else
        g.setColor(Color.BLACK);

    // Border
    g.drawRect(b.x, b.y, b.width, b.height);

    // MousePressed
    if (b.isMousePressed()) {
        g.drawRect(b.x + 1, b.y + 1, b.width - 2, b.height - 2);
        g.drawRect(b.x + 2, b.y + 2, b.width - 4, b.height - 4);
    }
}
```

In Action Bar,use in Playing game state, we want to show the user about gold, wave, lives, the tower that the can buy, and also the button that allow that return to menu or pause the game.

```
public void draw(Graphics g) {

    // Background
    g.setColor(new Color(220, 123, 15));
    g.fillRect(x, y, width, height);

    // Buttons
    drawButtons(g);

    // DisplayedTower
    drawDisplayedTower(g);

    // Wave info
    drawWaveInfo(g);

    // Gold info
    drawGoldAmount(g);

    // Draw Tower Cost
    if (showTowerCost)
        drawTowerCost(g);

    // Game paused text
    if (playing.isGamePaused()) {
        g.setColor(Color.black);
        g.drawString("Game is Paused!", 110, 790);
    }

    // Lives
    g.setColor(Color.black);
    g.drawString("Lives: " + lives, 110, 750);

}
```

And in ToolBar, use in Editing game state,we want show user the ground , road , water,start and end path, menu and save button, which they can modify the map.

```java
private void initButtons() {

    bMenu = new MyButton("Menu", 2, 642, 100, 30);
    bSave = new MyButton("Save", 2, 674, 100, 30);

    int w = 50;
    int h = 50;
    int xStart = 110;
    int yStart = 650;
    int xOffset = (int) (w * 1.1f);
    int i = 0;

    bGrass = new MyButton("Grass", xStart, yStart, w, h, i++);
    bWater = new MyButton("Water", xStart + xOffset, yStart, w, h, i++);

    initMapButton(bRoadS, editing.getGame().getTileManager().getRoadsS(), xStart, yStart, xOffset, w, h, i++);
    initMapButton(bRoadC, editing.getGame().getTileManager().getRoadsC(), xStart, yStart, xOffset, w, h, i++);
    initMapButton(bWaterC, editing.getGame().getTileManager().getCorners(), xStart, yStart, xOffset, w, h, i++);
    initMapButton(bWaterB, editing.getGame().getTileManager().getBeaches(), xStart, yStart, xOffset, w, h, i++);
    initMapButton(bWaterI, editing.getGame().getTileManager().getIslands(), xStart, yStart, xOffset, w, h, i++);

    bPathStart = new MyButton("PathStart", xStart, yStart + xOffset, w, h, i++);
    bPathEnd = new MyButton("PathEnd", xStart + xOffset, yStart + xOffset, w, h, i++);

}
```

7) Other function development:

Goal:

+Starting with the fix amount of money: 100 gold

+Receiving reward gold, when an enemy get killed

+Increasing the amount of money every three seconds: 1 gold

+Selling the tower and get the money back

+Upgrading the tower.

a) The default budget:

In order to use the towers, the player must purchase them with the original default amount. Because each tower has

its own strengths and weaknesses, they are sold at different prices. We have built the method GetTowerCost:

```
public static int GetTowerCost(int towerType) {
    switch (towerType) {
    case CANNON:
        return 65;
    case ARCHER:
        return 35;
    case WIZARD:
        return 50;
    }
    return 0;
}
```

Method payForTower: After purchasing an tower, the amount available will be deducted as the cost of the Tower above.

```
public void payForTower(int towerType) {
    this.gold -= helpz.Constants.Towers.GetTowerCost(towerType);

}
```

Since the starting budget is 100, the towers can not be purchased unlimited. So we have generated the method isGoldEnoughForTower to check if the available gold is enough for purchasing, then added some code to the method mouseClick to prevent the player from buying over the budget which mean if the gold is enough then the player can select the Tower and put it on the grass.

```
private boolean isGoldEnoughForTower(int towerType) {

    return gold >= helpz.Constants.Towers.GetTowerCost(towerType);
}
```

```
for (MyButton b : towerButtons) {
    if (b.getBounds().contains(x, y)) {
        if (!isGoldEnoughForTower(b.getId()))
            return;

        selectedTower = new Tower(x: 0, y: 0, -1, b.getId());
        playing.setSelectedTower(selectedTower);
        return;
    }
}
```

After killing an enemy, the player will receive a reward as the gold to keep purchase and upgrade tower.

b) Reward:

In package helpz method Constant, we defined the money of each enemy.

After killing an enemy, the player will receive a reward as the gold to keep purchase and upgrade tower.

```
public static int GetReward(int enemyType) {
    switch (enemyType) {
    case GOLEM:
        return 5;
    case WITCH:
        return 5;
    case KNIGHT:
        return 25;
    case VAMPIRE:
        return 10;
    }
    return 0;
}
```

To get the reward, we created the method rewardPlayer and added it in method hurt, when an enemy get killed it will return to gold.

```
public void rewardPlayer(int enemyType) {
    playing.rewardPlayer(enemyType);
}
```

```
public void hurt(int dmg) {
    this.health -= dmg;
    if (health <= 0) {
        alive = false;
        enemyManager.rewardPlayer(enemyType);
    }

}
```

In package ui method actionBar for display, we created method addGold

to add the gold when the enemy get killed the same amount of gold as assigned gold to budget.

```
public void addGold(int getReward) {
    this.gold += getReward;
}
```

Then call method rewardPlayer to playing class and finished this section.

```java
public void rewardPlayer(int enemyType) {
    actionBar.addGold(helpz.Constants.Enemies.GetReward(enemyType));
}
```

c) Gold increasing:

We declared the goldTick in playing class and added some code for gold increasing:

```java
public void update() {
    if (!gamePaused) {
        updateTick();
        waveManager.update();

        // Gold tick
        goldTick++;
        if (goldTick % (60 * 3) == 0)
            actionBar.addGold(getReward: 1);

        if (isAllEnemiesDead()) {
            if (isThereMoreWaves()) {
                waveManager.startWaveTimer();
                if (isWaveTimerOver()) {
                    waveManager.increaseWaveIndex();
                    enemyManager.getEnemies().clear();
                    waveManager.resetEnemyIndex();

                }
            }
```

Every 3 second the gold will be automately added 1 gold.

d) Tower selling:

In package manages class TowerManager, we built the method removeTower for players to remove any tower which they have purchased. If the tower id from the information box equals to the tower id on the game scene, it will be remove.

```
public void removeTower(Tower displayedTower) {
    for (int i = 0; i < towers.size(); i++)
        if (towers.get(i).getId() == displayedTower.getId())
            towers.remove(i);
}
```

Then call it to class Playing to apply it in game process.

```
public void removeTower(Tower displayedTower) {
    towerManager.removeTower(displayedTower);
}
```

In package UI class ActionBar, we created method sellTowerClicked to sell tower by clicking the button “sell” on the information box at the same time they generate the disappear tower into gold (money).

+Gold: is the amount of money that the player received from selling the tower. However, player only get half of the original cost. So the player must think carefully before selling.

+DisplayedTower equals to null make the tower disappear after selling.

```
private void sellTowerClicked() {
    playing.removeTower(displayedTower);
    gold += helpz.Constants.Towers.GetTowerCost(displayedTower.getTowerType()) / 2;

    int upgradeCost = (displayedTower.getTier() - 1) * getUpgradeAmount(displayedTower);
    upgradeCost *= 0.5f;
    gold += upgradeCost;

    displayedTower = null;

}
```

(UpgradeCost will be exlain below)

e) Tower upgrading:

In package manager class towerManager, we created method updradeTowerClicked for the player to update the Tower and make its stronger while dealing with the huge waves of enemies.

Likewise, the method updradeTower has been add to complete the very first step of updating Tower.

```
public void upgradeTower(Tower displayedTower) {
    for (Tower t : towers)
        if (t.getId() == displayedTower.getId())
            t.upgradeTower();
}
```

The same as selling function, we had to call it to the playing class to make the code valid.

```
public void upgradeTower(Tower displayedTower) {
    towerManager.upgradeTower(displayedTower);

}
```

It is obviously that the current gold of the player will be

deducted according to each tower type.

```
private void upgradeTowerClicked() {
    playing.upgradeTower(displayedTower);
    gold -= getUpgradeAmount(displayedTower);

}
```

In package objects class Tower, we made a method

upgradeTower, the purpose is when player updates any kind of tower that is fitting with the case below, it will increase the damage, range and reduce cooldown of that tower.

```
public void upgradeTower() {
    this.tier++;

    switch (towerType) {
    case ARCHER:
        dmg += 2;
        range += 20;
        cooldown -= 5;
        break;
    case CANNON:
        dmg += 5;
        range += 20;
        cooldown -= 15;
        break;
    case WIZARD:
        range += 20;
        cooldown -= 10;
        break;
    }
}
```

Everytime the tower get upgraded, it takes 30% cost of the tower, we had method getUpgradeAmount.

```
private int getUpgradeAmount(Tower displayedTower) {
    return (int) (helpz.Constants.Towers.GetTowerCost(displayedTower.getTowerType()) * 0.3f);
}
```

Furthermore, we have limited the updating within two times for each tower. To do this, we added these code int to draw method of ActionBar class to make sure that the upgrade no more than 3. Then the “upgrade“ button will be disappeared.

```
if (displayedTower.getTier() < 3 && gold >= getUpgradeAmount(displayedTower)) {
    upgradeTower.draw(g);
    drawButtonFeedback(g, upgradeTower);
}
```

Since there are two times of upgrade, everytime the player upgrade the tower the cost will increase. If the player sell the tower, he/she will gain half of the price plus the upgrade cost (for the upgrade tower only).

```
private int getSellAmount(Tower displayedTower) {
    int upgradeCost = (displayedTower.getTier() - 1) * getUpgradeAmount(displayedTower);
    upgradeCost *= 0.5f;

    return helpz.Constants.Towers.GetTowerCost(displayedTower.getTowerType()) / 2 + upgradeCost;
}
```

8) Game Status:

There are 5 states in the game : PLAYING , MENU , SETTINGS, EDIT,GAME_OVER.

```java
public enum GameStates {

    PLAYING, MENU, SETTINGS, EDIT, GAME_OVER;

    public static GameStates gameState = MENU;

    public static void SetGameState(GameStates state) {
        gameState = state;
    }

}
```

The game state being changed when meet some conditions

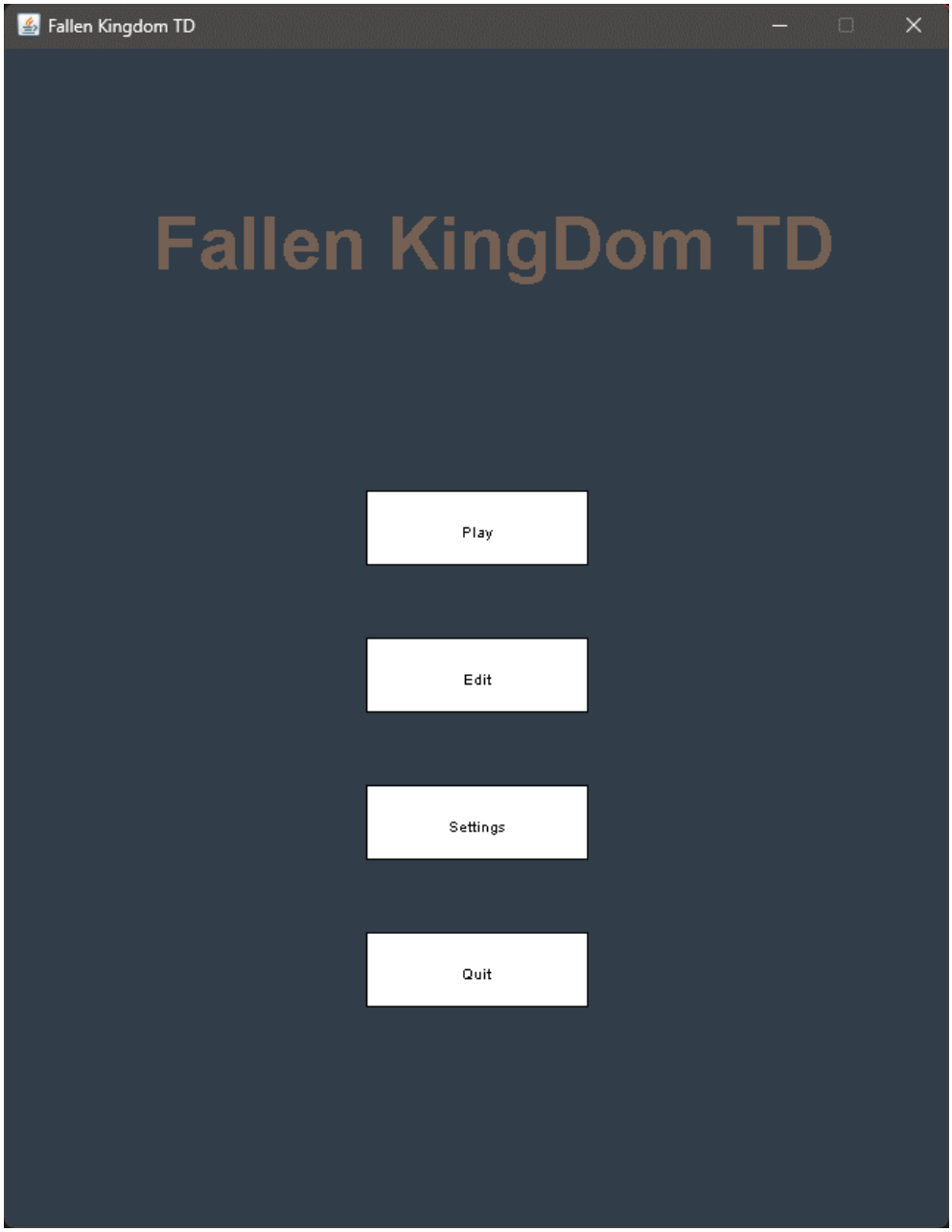
State	Condition
PLAYING	Click the playing button at the menu
MENU	Being set as default or click the menu button while playing or editing
SETTINGS	Click the settings button at the menu
EDIT	Click the edit button at the menu
GAME_OVER	Lives = 0

VI. Result – Limited – Conclusion:

1. **Result**

Based on our understanding of the Java programming language, OOP principles, we have completed our game with basic rules and logic. All of the classes in our projects are logically presented and successfully output to a user interface where the user can choose place the tower using the mouse.

In the menu screen where the user can choose start the game, edit the map, modify the settings or quit the game. When pressing the “Start” button, the user can get to the game.



After entering the main game screen, the player can place any

tower using mouse if they can afford the price of that tower. And after the tower is placed, it will automatically attack enemies in its range.Tower can be also upgrade to enhance its properties such as attack damage.

2. Limitation

With the scope of this project, the game still lacks varieties of game map, levels, enemies, towers, tower skins for each tower and the sound effect. Despite lacking such features, our code definitely allows room for these upgrades.

3. Conclusion

In conclusion, by building the Tower Defense game based on Object-Oriented Programming techniques, the process is much easier, and the code is arranged in a very logical way. The project has shown many properties of Object-Oriented Programming, such as polymorphism, inheritance,encapsulation, data abstraction, etc.

Besides practicing all the OOP techniques, learning more knowledge outside the course limit, such as working with a framework, getting used to Git is one of the most important things to do while working on this project.