OBJECT ORIENTED ANALYSIS AND DESIGN USING JAVA

MINI PROJECT
PACMAN

NAME	SRN
VRINDA S GIRIMAJI	PES1UG20CS514
VRINDA KORE	PES1UG20CS513
SAMSKRUTHI S DINESH	PES1UG20CS522
VIJITHA M	PES1UG20CS501

THIS DOCUMENT CONTAINS

Project Synopsis (Problem statement definition)
Use case diagrams
Use case specifications of four important use cases
Class models
Architecture Patterns
Design Principles and Design patterns used (use case + desc)
Github link to code base
Screenshots of input and output
Individual contributions of team members

PROJECT SYNOPSIS

Our game project called the “Better Pacman” is a classic arcade-style game that involves navigating a maze and collecting points while avoiding enemies. The project is typically implemented using Java programming language.

The game mechanics involve a player controlling two characters (Gabe and Mani) who move around a maze collecting dots and power-ups while avoiding enemy characters (two wizards). The player wins by collecting all the dots, and loses if one of the characters collides with one of the wizards.

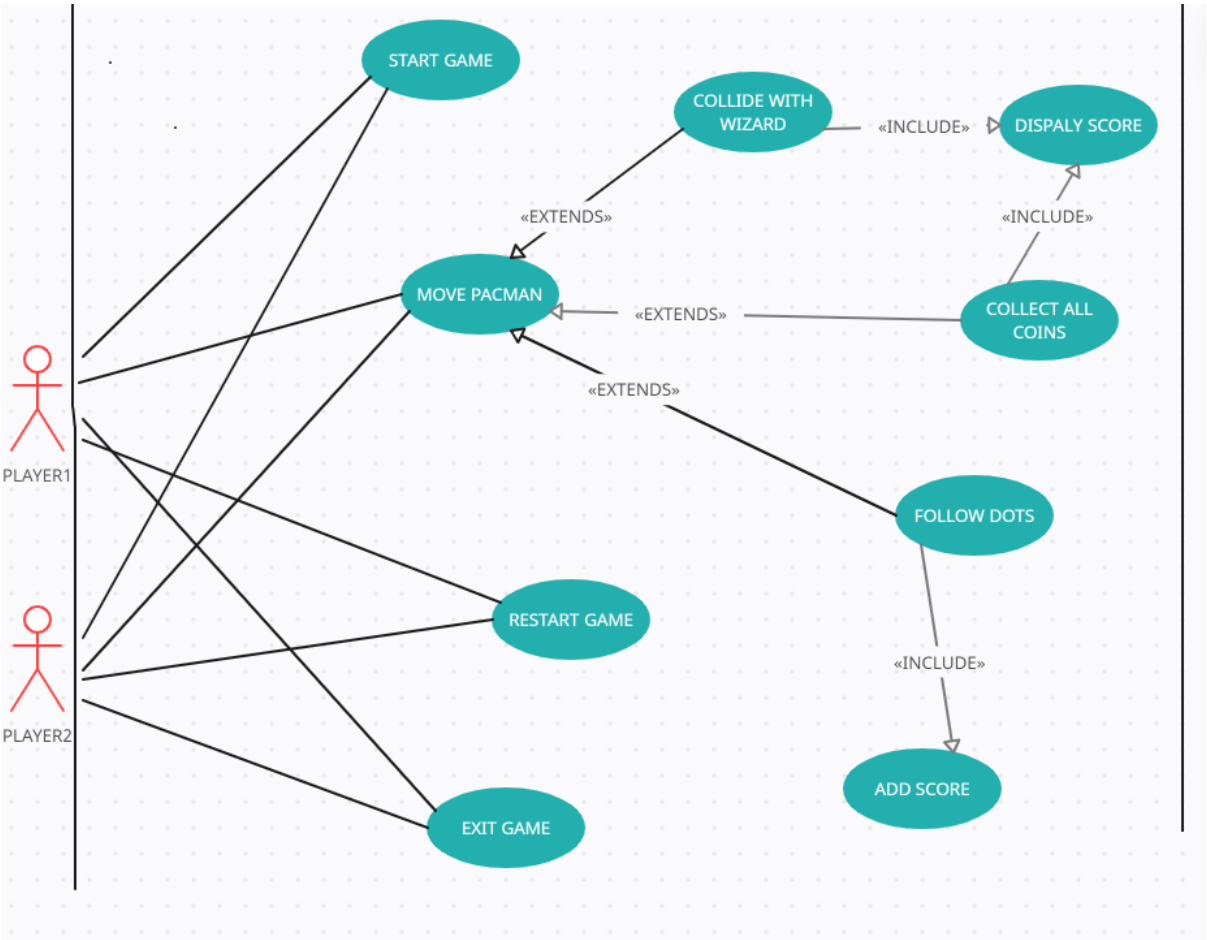
The project involves several components, including graphics rendering, user input handling, and game logic. Some of the key features of the game include:

1. A maze that is generated each time the game is played.
2. Enemy characters that move around the maze, attempting to intercept and collide with the characters.
3. Scoring system that keeps track of points earned by collecting p
4. High score tracking that records the highest scores achieved by players.

The project can be implemented using Java's built-in graphics libraries, for our project we have used two frameworks one is libGDX which is a cross-platform Java game development framework based on OpenGL (ES) that works on Windows, Linux, macOS, Android, your browser and iOS and other is Tiles which is a free open-sourced templating framework for modern Java applications. Based upon the Composite pattern it is built to simplify the development of user interfaces.

Overall, the Better Pacman game project is a fun and challenging way to practise object-oriented programming and game development skills using Java.

USE CASE DIAGRAMS



USE CASE SPECIFICATIONS OF FOUR IMPORTANT USE CASES(vk):

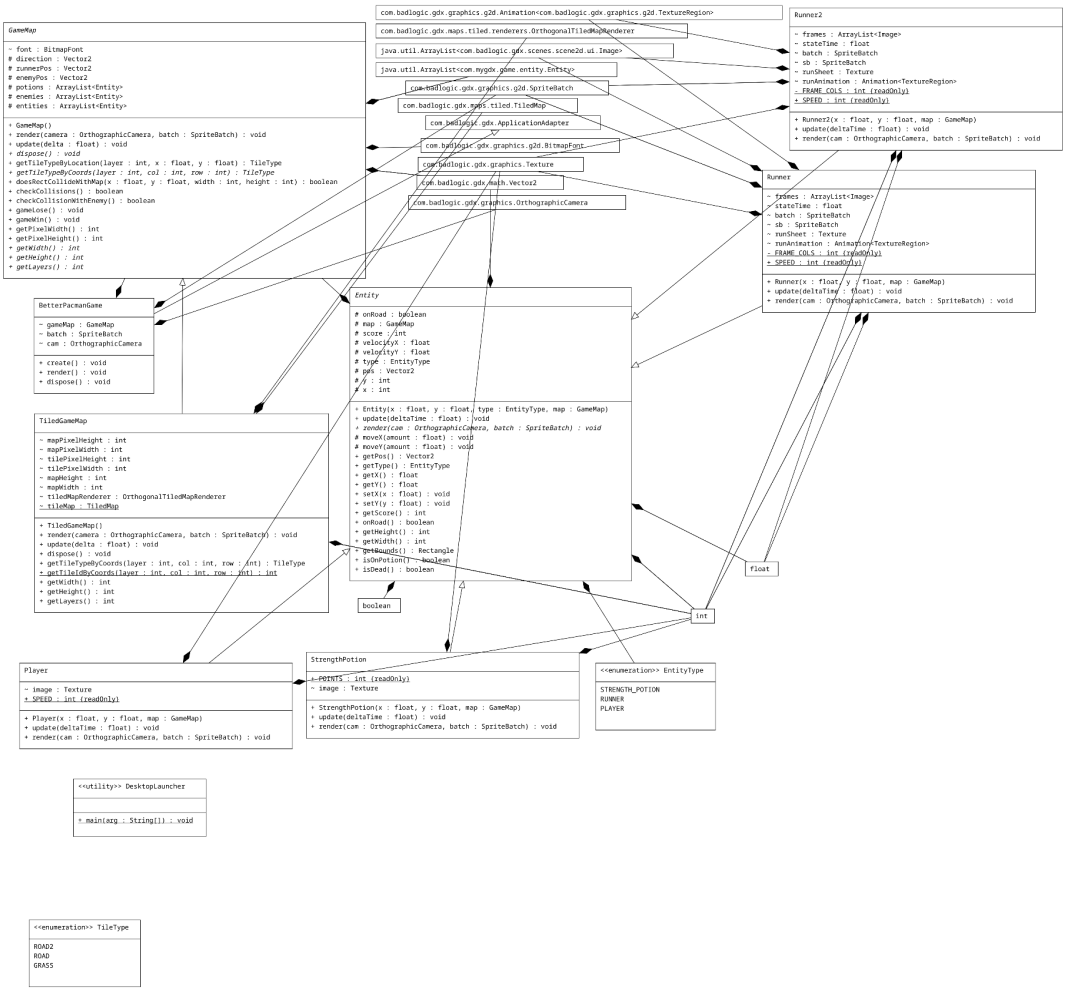
1. Name: Move Pacman
   Summary/Overview: This use case involves moving 2 Pacman through the maze using the arrow keys and the AWSD controls.
   Actor: User
   Pre-conditions: The game is running and Pacman is visible on the screen, along with the 2 wizards ready to collide with the 2 pacmans.
   Description: The user presses the arrow keys or other controls to move the Pacmans through the maze simultaneously. The game updates the position of Pacmans on the screen based on the user's input. If Pacman collides with a power pellet, the game updates the score(+10) and removes the pellet from the screen.
   Exception: If Pacman collides with a wizard, the game updates the game points accordingly and ends the game. If Pacman reaches the end of the maze without the wizard colliding and collecting all the points, the game ends.

Post-conditions: Pacmans have moved to a new position on the screen, and the game state has been updated to reflect any changes in the score.

2. Name: Collect Dots
   Summary/Overview: This use case involves collecting dots as Pacman moves through the maze.
   Actor: User
   Pre-conditions: The game is running and Pacman is visible on the screen.
   Description: As Pacman moves through the maze, he collects dots that are scattered throughout the maze. The game keeps track of the number of dots collected and updates the score accordingly(+10 for each dot).
   Exception: If Pacman collides with an enemy, the game reduces the number of lives and updates the game state accordingly. If Pacman reaches the end of the maze, the game ends.
   Post-conditions: The score has been updated based on the number of dots collected.
3. Name: Collide with wizard
   Summary/Overview: This use case involves Pacman getting hit by a wizard and the game ending.
   Actor: User
   Pre-conditions: The game is running and Pacman is visible on the screen. There are 2 wizards on the screen.
   Description: If Pacman collides with a wizards, the game ends and updates the game score accordingly.
   Exception: If Pacman reaches the end of the maze without getting collided by the maze, the game ends with the points displayed.
   Post-conditions: The game state has been updated to reflect any changes in the point collected.
4. Name: Displaying the Score
   Summary/Overview: This use case involves displaying the score on the side-screen.
   Actor: User
   Pre-conditions: The game is running and Pacman is visible on the screen. The pacman is either collided by the wizard, or has completed taking all the power pellets.

Description: The game keeps track of the score as Pacman collects dots. The game displays the updated score on the screen after each dot is collected.
Exception: None.
Post-conditions: The updated score will be displayed on the screen.

CLASS MODELS:



ARCHITECTURE PATTERNS:

1. Model-View-Controller (MVC) Pattern: This pattern separates the application into three interconnected components: the model (data and business logic), the view (user interface), and the controller (handles user input and updates the model and view). This pattern can be used to separate the game logic from the user interface and make the game easier to maintain and modify.
2. Layered Pattern:

The Layered Pattern can be used to divide the game logic into different layers:

- Presentation Layer: This layer is responsible for the user interface of the game, including the graphics and user input. It interacts with the other layers to display the game state and receive user input.
- Business Layer: This layer is responsible for the game logic, including the movement of the Pacman and the wizards, the collision detection, and the scoring system. It interacts with the other layers to update the game state and respond to user input.
- Application Layer: This layer acts as a medium for communication between the Presentation Layer and the Data Layer. It handles the communication between the different components of the game and ensures that the game logic is executed correctly.
- Data Layer: This layer is responsible for temporary data storage and retrieval, including the game state, the points collected till now, and the configuration settings. It interacts with the other layers to store and retrieve data as needed.

DESIGN PRINCIPLES

The design principles of Pac-Man like game include minimalist design and a highly-immersive game environment.Pac-Man like game is a revolutionized gaming and is at the root of a rich design tradition that goes beyond detailed graphics and fluid controls.The enduring appeal of Pac-Man like game is attributed to its design innovation that created meaning and shaped playing experiences.

1.Inheritance is a principle of class hierarchy that allows one object to take on the states, behaviors, and functionality of another.In game, inheritance is used to create sound inheritance hierarchies that are not being abused by code that introduces bugs that are hard to fix.Inheritance is also used to create reusable code and to make it easier to manage game object.

2.The Single Responsibility Principle (SRP) is a software engineering principle that states that every module or class should have only one responsibility and that responsibility should be entirely encapsulated within the class.In game , the SRP can be applied to ensure that each game object or system has a single responsibility and

that it is encapsulated within that object or system.This can help to improve the maintainability and flexibility of the game codebase.

3.The Open-Closed Principle (OCP) is a software design principle that states that software entities should be open for extension but closed for modification. In game development, the OCP can be applied to ensure that game objects or systems can be extended without modifying the existing codebase. This can help to improve the maintainability and flexibility of the game codebase. The OCP can be achieved through the use of interfaces, abstract classes, and other design patterns that allow for extension without modification.

4. Dependency Inversion Principle (DIP) is a software design principle that promotes two objectives: high-level modules should not depend on low-level modules, and abstractions should not depend on details, but details should depend on abstractions. In game development, the DIP can be applied to ensure that game objects or systems are decoupled from each other and that they depend on abstractions rather than concrete implementations. This can help to improve the maintainability, flexibility, and testability of the game codebase. The DIP can be achieved through the use of interfaces, abstract classes, and other design patterns that allow for decoupling and abstraction

DESIGN PATTERNS

There are several design patterns that can be used to implement the Pacman game project in Java. Here are some examples:

1. Singleton Pattern: The Singleton pattern can be used to ensure that only one instance of certain classes, such as the game controller or maze generator, is created. This can help to simplify the code and prevent unexpected behaviour.
2. Observer Pattern: The Observer pattern can be used to update different game entities when the state of the game changes. For example, when the character collects the strength potion, this can trigger an event that updates the behaviour of the wizards.
3. Factory Pattern: The Factory pattern can be used to create different types of game entities, such as the characters,wizards, and strength potion, without

exposing the details of their implementation. This can help to simplify the code and make it easier to add new types of entities.

4. Command Pattern: The Command pattern can be used to encapsulate actions taken by the player, such as moving the characters or using a power-up, into objects that can be executed later. This can help to simplify the code and make it easier to add new player actions.

GITHUB LINK: https://github.com/flamethrower775/fictional-dollop

SCREENSHOTS OF THE GAME

- **The screenshot below shows the starting of the game. The two wizards (enemies) start from the same point. Each of them follows and tries to touch one player.**

- **The screenshot below shows the game once it has been played for a while. We can see that the players' scores have increased (10 points per potion), before the wizards caught up with them.**

INDIVIDUAL CONTRIBUTIONS OF TEAM MEMBERS

VRINDA S GIRIMAJI

I have developed a Pacman-like game called Better Pacman using libGDX . Here is a high-level overview of the process:

1. Set up the development environment: To start, I downloaded and installed libGDX and set up the development environment.

2. Create the game assets: The game will require various assets, such as images for the maze,the characters, and the wizards. I created these assets using the tiles framework. Some of the character animations are shown below. (more details about this are in the description of the README in the github link)

3. Build the game entities: In libGDX, game entities are represented using classes. I created classes for runners, and other game entities. These classes should contain information such as the entity's position, size, and behaviour.

Class Runner (most code is omitted for brevity. See the source code on GitHub):

```
public class Runner extends Entity {

   public static final int SPEED = 50;

   private static final int FRAME_COLS = 7;

   Animation<TextureRegion> runAnimation;

   Texture runSheet;

   SpriteBatch sb;

   SpriteBatch batch;

   float stateTime;

   ArrayList<Image> frames;
```

Class Player (the wizard/enemy):

```
public class Player extends Entity {

   public static final int SPEED = 50;


   Texture image;
```

Class strengthPotion:

```
public class StrengthPotion extends Entity{

   Texture image;

   public static final int POINTS = 10;
```

4. Implement the game logic: The game logic determines how the game entities behave. I implemented the rules for how the characters move around the

maze, how the wizards chase the characters, and how points are scored. Some of the code is pasted below. It is not an exhaustive list, but everything can be found in the source code.

Logic for moving the wizard:

```
enemyPos = new Vector2(enemies.get(0).getX(),
enemies.get(0).getY());

   runnerPos = new Vector2(entities.get(0).getX(),
entities.get(0).getY());



   direction = new Vector2();



   direction.x = (runnerPos.x + 10) - (enemyPos.x + 10);

   direction.y = (runnerPos.y + 10) - (enemyPos.y + 10);

   direction.nor();



   float lala1 = enemies.get(0).getX();

   enemies.get(0).setX(lala1 + direction.x * 0.5f);



   float lala2 = enemies.get(0).getY();

   enemies.get(0).setY(lala2 + direction.y * 0.5f);




   enemyPos = new Vector2(enemies.get(1).getX(),
enemies.get(1).getY());

   runnerPos = new Vector2(entities.get(1).getX(),
entities.get(1).getY());



   direction = new Vector2();
```

```
    direction.x = (runnerPos.x + 10) - (enemyPos.x + 10);

    direction.y = (runnerPos.y + 10) - (enemyPos.y + 10);

    direction.nor();


    float lala3 = enemies.get(1).getX();

    enemies.get(1).setX(lala3 + direction.x * 0.5f);


    float lala4 = enemies.get(1).getY();

    enemies.get(1).setY(lala4 + direction.y * 0.5f);

}
```

Collision Logic:

```
public boolean doesRectCollideWithMap(float x, float y, int width, int
height) {

    if ( x < 0 || y < 0 || x + width >= getPixelWidth() || y + height >=
getPixelHeight()) {

        return true;

    }


    for (int row = (int)Math.floor(y / TileType.TILE_SIZE) + 1; row <
Math.ceil(((y + height) / TileType.TILE_SIZE) - 0.5f); row ++) {

        for (int col = (int)Math.floor(x / TileType.TILE_SIZE) + 1; col
< Math.ceil(((x + width) / TileType.TILE_SIZE) - 0.5f); col ++) {

            int type = TiledGameMap.getTileIdByCoords(0, col, row);

            if (type != -1) return true;

        }

    }

    return false;

}
```

5. Create the game view: The game view is responsible for rendering the game entities and displaying the game state to the user. I used libGDX's built-in graphics libraries to create the game view.
6. Test and refine: Once the game is implemented, I tested it to ensure that it works as expected.

VRINDA KORE

Our team has developed a 2-player pacman game, which is different from the conventional pacman, in terms of the number of players. My specific responsibilities on the team included placing power points on the game map, writing the code for scoring functionality and displaying the score points of both the PacMans.

- To place the power points, I, along with a discussion with my fellow teammates, decided on the optimal locations for the power points, taking into account factors such as the difficulty of reaching them, their distance from other objects in the game, and their impact on gameplay.

```
public GameMap() {

    entities = new ArrayList<Entity>();
    entities.add(new Runner(100, 70, this));
    entities.add(new Runner2(150, 70, this));

    enemies = new ArrayList<Entity>();
    enemies.add(new Player(346, 415, this));
    enemies.add(new Player(346, 415, this));

    potions = new ArrayList<Entity>();
    potions.add(new StrengthPotion(12.f, 84, this));
    potions.add(new StrengthPotion(83,  84, this));
    potions.add(new StrengthPotion(152, 84, this));

    potions.add(new StrengthPotion(177, 30, this));
    potions.add(new StrengthPotion(235, 30, this));
    potions.add(new StrengthPotion(291, 30, this));

    potions.add(new StrengthPotion(291, 60, this));
    potions.add(new StrengthPotion(291, 100, this));
    potions.add(new StrengthPotion(291, 150, this));

    potions.add(new StrengthPotion(126, 139, this));
    potions.add(new StrengthPotion(82, 215, this));
```

```
    potions.add(new StrengthPotion(127, 215, this));

    potions.add(new StrengthPotion(176, 215, this));
    potions.add(new StrengthPotion(225, 215, this));
    potions.add(new StrengthPotion(286, 215, this));

    potions.add(new StrengthPotion(346, 215, this));
    potions.add(new StrengthPotion(395, 215, this));
    // and so on
```

- Once the power points were placed, I wrote the code for the scoring functionality. Each time a Pac-Man character collected a power point, their score increased by 10 points. This was achieved through a function that tracked the number of power points collected and incremented the score accordingly. The scores of both players were stored temporarily in the backend until the game ended and individual scores for each player were displayed.

```
if (isOnPotion()) {
    this.score += 10;
    System.out.println("GOT IT!!!" + this.score);
}
```

- The game concludes when both players collected all the points on the map without being caught by two wizards present in the game. This is achieved through a function that checks whether all the power points has been collected and whether either player has collided with a wizard (function endthegame()). If both players have collected all the power points and have not collided with a wizard, the game ends and individual scores for each player are displayed.

```
public boolean checkCollisions() {
    if(potions.isEmpty()) {
        gameWin();
    }

    for (Entity player : entities) {
        Rectangle playerBox = player.getBounds();
        for (Entity potion : potions) {
            Rectangle potionBox = potion.getBounds();
```

```
            if (playerBox.intersects(potionBox)) {
                potions.remove(potion);
                return true;
            }
        }
    }
    return false;
}
```

```
public void gameLose() {
    System.out.println("You Lose :(");
}

public void gameWin() {
    System.out.println("You Win!");
}
```

```
public void render(OrthographicCamera camera, SpriteBatch batch) {
    font.draw(batch, "Better Pacman!", 515, 475);

    String lala = Integer.toString(entities.get(0).getScore());
    font.draw(batch, "Mani: ", 515, 430);

    String lala2 = Integer.toString(entities.get(1).getScore());
    font.draw(batch, "Gabe: ", 515, 400);

    font.draw(batch, lala, 580, 430);
    font.draw(batch, lala2, 580, 395);

    for (Entity entity : entities) {
        entity.render(camera, batch);
    }
    for (Entity entity : enemies) {
        entity.render(camera, batch);
    }
    for (Entity entity : potions) {
        entity.render(camera, batch);
    }

}
```

- If either Pac-Man character collides with a wizard, the game ends, and the score of both the pacmans up until that point is displayed. This is achieved through a function that checks for collisions between the Pac-Man characters

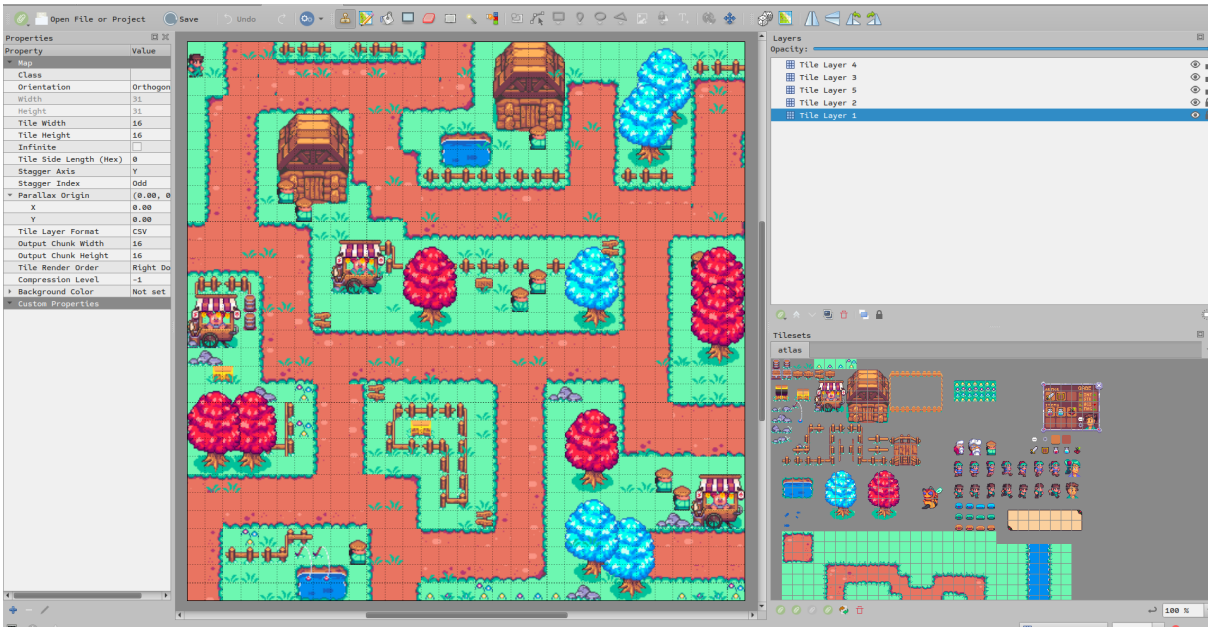
and the wizards. If a collision occurs, the score up until that point is displayed and the game ends.

```
public boolean checkCollisions() {
    if(potions.isEmpty()) {
        gameWin();
    }

    for (Entity player : entities) {
        Rectangle playerBox = player.getBounds();
        for (Entity potion : potions) {
            Rectangle potionBox = potion.getBounds();

            if (playerBox.intersects(potionBox)) {
                potions.remove(potion);
                return true;
            }
        }
    }
    return false;
}
```

SAMSKRUTHI S DINESH

I in the project contributed towards building a game maze with Tile and integrate it with a desktop launcher, one can use a game development engine like Unity or Unreal Engine to create the game mechanics and graphics. In game, a tile is an image that is used to create other, bigger images (such as a platform) in a 2D game. From a gameplay point of view, a tile is the minimum unit of movement for a game object in many games. A tile-based video game is a type of video game where the playing area consists of small square (or, much less often, rectangular, parallelogram) tiles. Tile-based refers to the method of building levels in a game, where the code will layout tiles in specific locations to cover the intended area. Tilemaps are a popular technique in 2D game development, consisting of building the game world or level map out of small, regular-shaped tiles.. Once the game is developed, it can be packaged as a desktop application using tools like Electron. The customizable part of the Electron app is straightforward, and the window width and height can be set . The game can then be copied into the same folder as the Electron app and launched in the command line. To distribute the game, the dist folder can be created, and the game can be run by double-clicking electron.exe. Finally, the desktop launcher can be used to launch the game. Which later on algorithms can be built and integrated with the desktop launcher.

Tiled:



The map in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.10" tiledversion="1.10.1" orientation="orthogonal"
renderorder="right-down" width="31" height="31" tilewidth="16"
tileheight="16" infinite="0" nextlayerid="6" nextobjectid="1">
<tileset firstgid="1" source="atlas.tsx"/>
<layer id="1" name="Tile Layer 1" width="31" height="31"
locked="1">
 <data encoding="csv">
0,0,742,547,743,0,0,0,0,0,0,0,678,547,679,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,
0,0,742,719,547,770,770,770,770,770,770,771,772,547,547,711,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,787,788,647,547,547,547,547,547,547,547,547,547,774,775,0,0,0,
0,0,514,515,516,0,0,0,0,614,615,615,
0,614,547,547,821,802,803,804,805,547,547,547,805,805,805,807,0,0,
0,0,0,814,547,815,0,0,0,0,646,547,547,
0,646,547,548,0,0,0,0,0,678,547,679,0,0,0,0,0,0,0,0,846,547,847,
0,0,0,0,678,547,547,
0,678,547,548,0,0,0,0,0,678,547,711,0,0,0,0,0,0,0,0,878,547,783,
0,0,0,0,678,547,679,
0,646,547,548,0,0,0,0,0,710,547,547,620,0,0,0,0,0,0,0,846,547,81
5,0,0,0,0,678,547,679,
0,678,547,548,0,0,0,0,0,578,547,547,711,0,0,0,0,0,0,0,878,547,84
7,0,0,0,0,678,547,547,
```

```
0,710,547,548,0,0,0,0,0,0,710,547,547,771,772,772,772,772,770,771,
772,547,547,547,773,773,773,773,547,547,547,
0,742,547,548,0,0,0,0,0,0,717,579,547,547,547,547,547,547,547,547,
547,547,547,547,547,547,547,547,774,547,547,
773,547,547,547,617,618,516,0,0,0,0,0,717,803,804,804,804,804,804,
804,579,579,579,579,547,547,547,579,579,579,720,
547,547,547,547,547,547,711,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,723,
547,751,0,0,0,0,
648,649,650,650,547,547,679,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,755,
547,783,0,0,0,0,
0,0,0,0,646,547,711,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,787,547,815,
0,0,0,0,
0,0,0,0,678,547,743,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,755,547,847,
0,0,0,0,
0,0,0,0,710,547,775,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,787,547,751,
0,0,0,0,
0,0,0,0,742,547,547,770,771,772,773,770,771,772,773,773,770,771,77
2,773,770,771,772,773,547,547,783,0,0,0,0,
0,0,0,0,710,547,547,547,547,547,547,547,547,547,547,547,547,547,54
7,547,547,547,547,547,547,547,815,0,0,0,0,
0,0,0,0,578,803,804,547,547,803,804,805,802,803,804,805,802,547,54
7,547,802,803,804,805,547,547,847,0,0,0,0,
0,0,0,0,0,0,0,678,547,711,0,0,0,0,0,0,0,646,547,679,0,0,0,0,723,54
7,879,0,0,0,0,
0,0,0,0,0,0,0,710,547,743,0,0,0,0,0,0,0,678,547,711,0,0,0,0,755,54
7,547,771,772,772,772,
0,0,0,0,0,0,0,710,547,775,0,0,0,0,0,0,0,710,547,743,0,0,0,0,787,54
7,547,547,547,547,547,
0,0,0,0,0,0,0,646,547,711,0,0,0,0,0,0,0,710,547,775,0,0,0,0,819,80
3,820,821,822,823,820,
0,0,0,0,0,0,0,678,547,743,0,0,0,0,0,0,0,646,547,743,0,0,0,0,0,0,0,
0,0,0,0,
615,616,617,618,619,618,619,547,54
```

Code that gets the ID of the tiles on the map:

```
public TileType getTileTypeByCoords(int layer, int col, int row) {
   TiledMapTileLayer.Cell cell =
((TiledMapTileLayer)tileMap.getLayers().get(layer)).getCell(col,
row);

   if (cell != null) {
       TiledMapTile tile = cell.getTile();
       if (tile != null) {
           int id = tile.getId();
           return TileType.getTileTypeById(id);
       }
```

```
    }
    return null;
}

public static int getTileIdByCoords(int layer, int col, int row) {
    TiledMapTileLayer.Cell cell =
((TiledMapTileLayer)tileMap.getLayers().get(layer)).getCell(col,
row);

    if (cell != null) {
        TiledMapTile tile = cell.getTile();
        if (tile != null) {
            return tile.getId();
        }
    }
    return -1;
}
```

VIJITHA M

In the context of the Better Pacman game,I have implemented the controller which is responsible for handling user input and updating the game state accordingly. The controller is typically implemented as a separate class that communicates with the other components of the game, such as the view and the model.

Here are some of the key responsibilities of the controller in the Pacman game project:

1. Handling user input: The controller listens for user input, such as keyboard or mouse events, and translates these inputs into actions that can be performed by the game entities.

```
if (Gdx.input.isKeyPressed(Input.Keys.RIGHT) && !(isDead())) {

     // handle animation, and move right. GitHub for full code

}

if (Gdx.input.isKeyPressed(Input.Keys.LEFT) && !(isDead())) {

    TextureRegion currFrame = runAnimation.getKeyFrame(stateTime,
true);

    if (!currFrame.isFlipX()) {
```

```
        currFrame.flip(true, false);
    }
    sb.begin();

    if (map.doesRectCollideWithMap
            (pos.x - 0.45f, pos.y, getWidth(), getHeight())) {
        pos.x = pos.x - 0.45f;
    }
    sb.draw(currFrame, pos.x, pos.y);
    sb.end();
if (Gdx.input.isKeyPressed(Input.Keys.UP) && !(isDead())) {
    moveY(SPEED * deltaTime);
}
if (Gdx.input.isKeyPressed(Input.Keys.DOWN) && !(isDead())) {
    moveY(-SPEED * deltaTime);
}
```

2. Updating the game state: The controller updates the state of the game, such as the position of characters and the wizards, based on the actions performed by the user and the current state of the game.
3. Collision detection: The controller detects collisions between different game entities, such as when one of the characters collides with a wizard or collects a strength potion.

```
public boolean doesRectCollideWithMap(float x, float y, int width,
int height) {
    if ( x < 0 || y < 0 || x + width >= getPixelWidth() || y +
height >= getPixelHeight()) {
        return true;
    }
```

```
   for (int row = (int)Math.floor(y / TileType.TILE_SIZE) + 1; row
< Math.ceil(((y + height) / TileType.TILE_SIZE) - 0.5f); row ++) {
       for (int col = (int)Math.floor(x / TileType.TILE_SIZE) + 1;
col < Math.ceil(((x + width) / TileType.TILE_SIZE) - 0.5f); col
++) {
           int type = TiledGameMap.getTileIdByCoords(0, col, row);
           //System.out.println(type + " " + x + " " + y);
           if (type != -1) return true;
       }
   }
   return false;
}

public boolean checkCollisions() {
   if(potions.isEmpty()) {
       gameWin();
   }

   for (Entity player : entities) {
       Rectangle playerBox = player.getBounds();
       for (Entity potion : potions) {
           Rectangle potionBox = potion.getBounds();

           if (playerBox.intersects(potionBox)) {
               potions.remove(potion);
               return true;
           }
```

```
        }
    }
    return false;
}



public boolean checkCollisionWithEnemy() {
    for (Entity player : entities) {
        Rectangle playerBox = player.getBounds();
        //System.out.println(playerBox.x + " " + playerBox.y);
        for (Entity enemy : enemies) {
            Rectangle enemyBox = enemy.getBounds();


            if (playerBox.intersects(enemyBox)) {
                //System.out.println("lala");
                gameLose();
                return true;
            }
        }
    }
    return false;
}
```

4. Managing game events: The controller triggers events that update the behaviour of the game entities, such as when the character collects a power-up or when the game is over.

```
public void dispose() {
    tileMap.dispose();
```

```
}
```

5. Providing feedback to the user: The controller updates the view to provide feedback to the user, such as displaying the score.