INTERNATIONAL UNIVERSITY

VIETNAM NATIONAL UNIVERSITY, HCM CITY

School of Computer Science & Engineering

Object - Oriented Programming

Topic: Adventure 2D Game)

Full name	Student ID	Contribution
Nguyễn Anh Minh	ITITDK20002	21%
Vũ Trà My	ITITIU21247	21%
Hoàng Minh Tiến	ITDSIU21038	16%
Nguyễn Thị Mai Phương	ITDSIU20080	21%
Lê Phương Nghi	ITITIU21253	21%

Content Table

I. Introduction

The game is called The Adventure which brings the player on a fascinating trip filled with challenges. The game has a distinctive and charming retro-inspired 2D pixelated universe. Each level introduces fresh obstacles and riddles that test players' problem-solving skills, ensuring long-term interest in the game. To offer an overview of this thrilling adventure game, this report summarizes the game's primary features, gameplay, and overall experience. However, the adventure game was modified from the original version on YouTube.

II. Instruction, Rules, and Gameplay

1. Start the game

"Welcome to 'Game 2D' - The ultimate adventure through an uncharted forest! In this epic journey, you must explore the forest alone, face monsters. And find the shortest path to save the princess. Are you ready to explore the forest? Pack your bags, because you're in for an interstellar adventure like no other!

Setup Instructions:

- Download our game at:
- https://github.com/MinhNgywn2318/Games2D-projects-main
- Open the file a.exe to play

2. Movements

Press 'W' for forward, 'A' for left, 'S' for backward, and 'D' for right on the keyboard

3. Understanding the different types of characters

In the "Game 2D," players have two distinct character classes, each with its own unique abilities.

Lava: Grants you the ability to move through lava blocks without obstruction

Water: Grants you the ability to move through water blocks without obstruction

Choosing a character class is a critical decision that influences your play style and strategy throughout the game. Select wisely, and may your chosen path lead you to victory!

4. Questions on the road

Engage with NPCs (non-playable characters) by approaching them closely

Answer True/False questions; if you provide an incorrect answer, you cannot proceed until you provide the correct one.

5. Mechanics:

Mechanics: In "Game 2D," mastering fundamental game mechanics is key to surviving the fantastical world. Get acquainted with the following core elements:

Combat: Engage in real-time combat against formidable foes. Players must have enough knowledge to correctly answer True/False questions

Resource Management: Apple increases character speed during battles and do not forget to collect keys to pass the door. Explore the environment to find hidden resources that aid in your journey.

Chest: Contains objects that can either harm or benefit the player.

Apple: Increase your speed

Poison: Push you away

Key: use to unlock the door

Door: Requires a sufficient number of keys to open the door

As you delve into "Game 2D," remember that mastering these mechanics will not only enhance your gameplay experience but also unlock the true potential of your character.

6. Keyboard Shortcuts and In-Game Button:

Familiarize yourself with essential shortcuts for a smoother gaming experience.

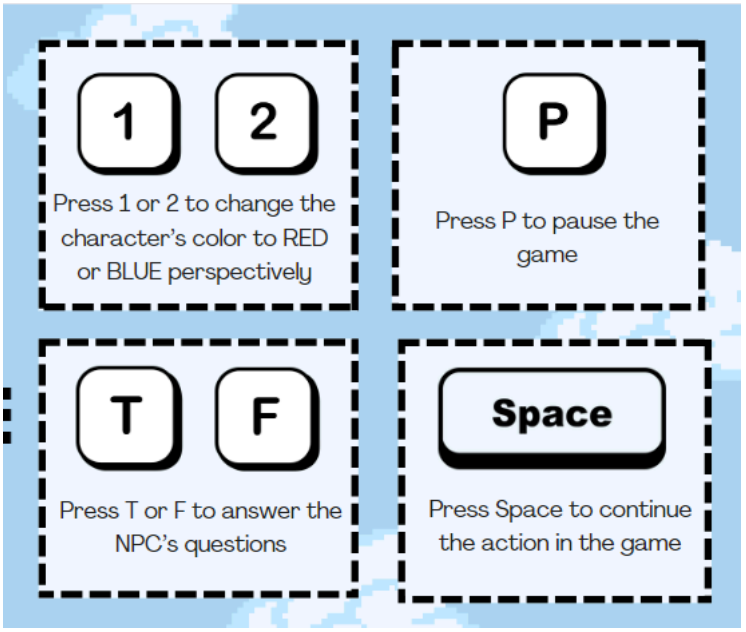


Figure 1.

Remember to explore your surroundings, complete quests, and interact with other players or NPCs to fully enjoy the game. Adapt your playstyle based on the challenges you encounter and always be ready for unexpected twists in the storyline. Enjoy your gaming experience!

III. Design Pattern

1. Command Pattern

We generated a ‘KeyCommand’ interface defining two methods: ‘keyPressed’ and ‘keyReleased’, which represent the actions to be performed when a key is pressed or released, respectively. This interface acts as a command contract, allowing numerous commands to be defined for different key events.

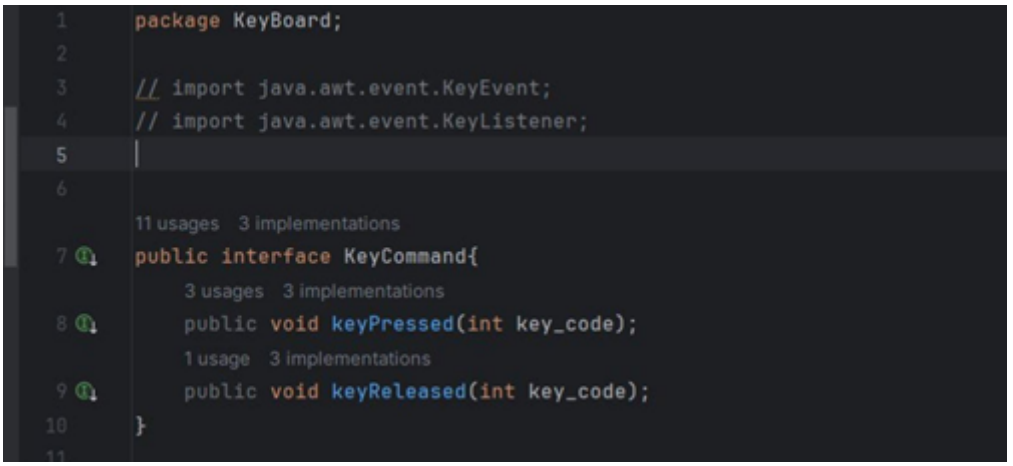
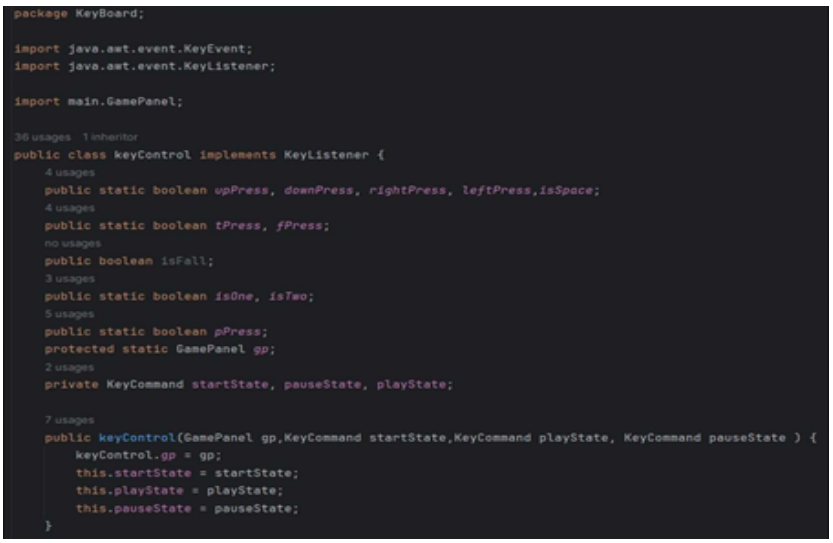


Figure 2. KeyCommand code structure

Then, we created the ‘keyControl’ class implements the ‘keyListener’ interface to manage key events. The ‘keyControl’ class acts as the **invoker**. It delegates the handling of key events to the appropriate ‘keyCommand’ objects based on the current game state. It also maintains references to different ‘keyCommand’ objects (e.g., startState, playState) and dynamically assigns them based on the game state.



```
package KeyBoard;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

import main.GamePanel;

36 usages  1 inheritor
public class keyControl implements KeyListener {
    4 usages
    public static boolean upPress, downPress, rightPress, leftPress,isSpace;
    4 usages
    public static boolean tPress, fPress;
    no usages
    public boolean isFall;
    3 usages
    public static boolean isOne, isTwo;
    5 usages
    public static boolean pPress;
    protected static GamePanel gp;
    2 usages
    private KeyCommand startState, pauseState, playState;

    7 usages
    public keyControl(GamePanel gp,KeyCommand startState,KeyCommand playState, KeyCommand pauseState ) {
        keyControl.gp = gp;
        this.startState = startState;
        this.playState = playState;
        this.pauseState = pauseState;
    }
```

Figure 3. KeyControl code structure

In the end, we made the Concrete Command Classes : ‘startKey’, ‘pauseKey’, ‘playKey’ which implement the ‘keyCommand’ interface. These classes encapsulate the logic for handling key events in different game states. Each concrete class represents a specific command and is responsible for executing the corresponding action.

```
public class pauseKey implements KeyCommand {
    30 usages
    private GamePanel gp;
    1 usage
    public pauseKey(GamePanel gp) {
        this.gp = gp;
    }
```

Figure 4. pauseKey code structure

Figure 5. startKey code structure

Figure 6. playKey code structure

Functionality of the Command Pattern in the Code:

+ **Dynamic Handling of Key Events:**

The Command Pattern allows for the dynamic handling of key events based on the current game state. Different sets of commands (concrete command classes) are assigned and invoked dynamically.

+ **Decoupling of Sender and Receiver:**

The sender (‘keyControl’) is decoupled from the receivers (‘keyCommand’ implementations). This separation facilitates easy addition or modification of commands without altering the sender's code.

+ **Flexibility and Extensibility:**

The implementation provides flexibility in managing key events in different game states. It allows for the easy addition of new game states or modification of existing ones without affecting the core functionality.

+ **Encapsulation of Behavior:**

Each concrete command class encapsulates specific behavior associated with key events, adhering to the principle of encapsulation and contributing to a modular and maintainable design.

+ **Readability and Maintainability:**

The Command Pattern enhances code readability by organizing key event-handling logic into separate command classes. This makes the codebase more understandable and maintainable.

+ **Adaptability to Changes:**

The dynamic assignment of commands based on the game state provides adaptability to changes in game requirements. New commands can be easily added to accommodate new features or interactions.

+ **Potential for Undoable Operations (Future Consideration):**

The Command Pattern lays the foundation for incorporating undoable operations if needed in the future. By extending the pattern, a command history mechanism could be implemented.

Overall, the Command Pattern in the provided code effectively organizes and manages key event handling, promoting a flexible, modular, and extensible design. It facilitates the adaptation of the system to different game states and

supports future enhancements. The separation of concerns between the sender and receivers contributes to a clean and maintainable architecture.

2. Singleton Pattern

The Singleton Pattern conventionally provides a method to retrieve the single instance of the class. In this code, it's the ‘getInstance’ method.

We made a ‘Player’ class which is implemented as a singleton using the ‘getInstance’ method, ensuring that only one instance of the ‘Player’ class exists throughout the application. The constructor of the ‘Player’ is made private to prevent external classes from creating instances directly, The ‘getInstance’ method is a public, static method, allowing other classes to access the single instance of ‘Player’ without instantiating it directly.

Figure 7. Player code structure

<u>Functionality of the Singleton Pattern in the Code:</u>

+ **Global Player Instance:**

The Singleton Pattern ensures that there is only one instance of the ‘Player’ class throughout the game. This provides a global point of access to the player entity from different components of the game.

+ **Controlled Instantiation:**

The private constructor of the ‘Player’ class prevents external classes from directly instantiating the ‘Player’. The only way to obtain the player instance is through the getInstance method.

+ **Consistent Player State:**

The Singleton Pattern ensures that all parts of the game refer to the same player instance. This consistency in the player state is crucial for maintaining a coherent game state.

+ **Flexible Access:**

The ‘getInstance’ method provides a flexible way for other classes, such as ‘GamePanel’, to access the player instance. This allows for efficient communication between different game components.

Therefore**,** the Singleton Pattern in this context enhances the maintainability and consistency of the game code by ensuring that there is only one instance of the ‘Player’ class, which is globally accessible and appropriately initialized. It plays a crucial role in managing the player entity and its interactions within the game architecture.

IV. UML class diagram

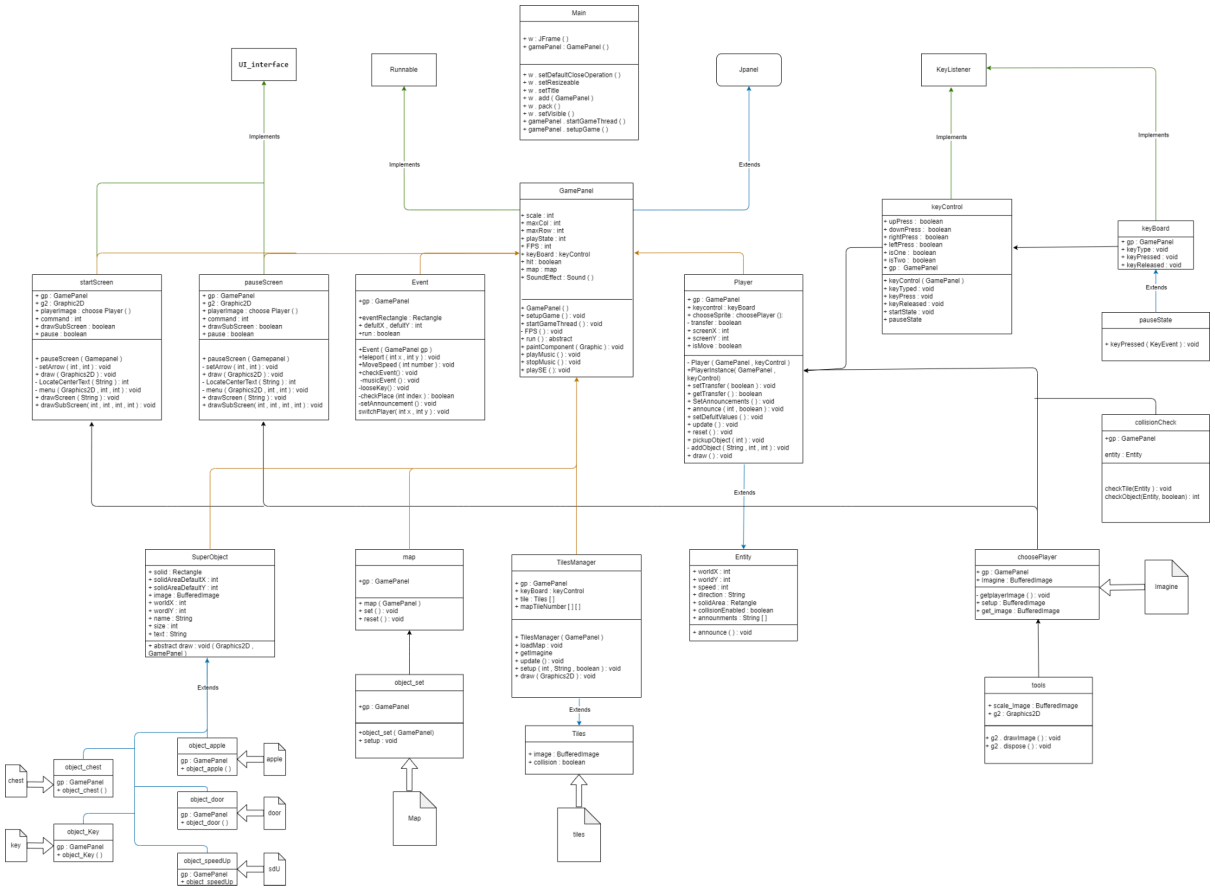


Figure 8. Class Diagram

V. Polymorphism

1. Inheritance in Game.

In the provided game source code, we observe the application of Polymorphism through implementing the KeyListener interface and utilizing objects from different classes implementing KeyCommand. Below is a subsection detailing Polymorphism through Inheritance

a. Interface “KeyListener“.

The keyControl class implements the KeyListener interface. This is an example of employing polymorphism through inheritance via interfaces in Java.

The “KeyListener“ interface provides “keyPressed“,“ keyReleased“, and “keyTyped“ methods, and the keyControl class implements these methods to handle keyboard events.

```
public class keyControl implements KeyListener {
    public static boolean upPress, downPress, rightPress, leftPress,isSpace;
    public static boolean tPress, fPress;
```

Figure 9.

b. Using Objects of Different Classes.

The keyControl class uses objects from different classes belonging to KeyCommand (such as “startState“, “pauseState“, and “playState“).

The use of objects from different subclasses of KeyCommand demonstrates the polymorphic inheritance. These subclasses can implement methods differently depending on the specific needs of each game state.

```
public keyControl(GamePanel gp,KeyCommand startState,KeyCommand playState, KeyCommand pauseState ) {
    keyControl.gp = gp;
    this.startState = startState;
    this.playState = playState;
    this.pauseState = pauseState;
}
public keyControl() {
    super();
}
```

Figure 10

c. Flexibility and Extensibility.

The use of polymorphic inheritance makes the source code flexible and extensible. You can easily add new game states by simply introducing a new subclass of “KeyCommand“ and implementing the corresponding method.

2. Encapsulation

a.Private Fields and Getters/Setters.

In classes like SuperObject, private fields such as image, worldX, worldY, collision, name, size, and text are used.

These values are not directly accessed from outside the class; instead, you've created getter and setter methods to ensure encapsulation. This helps control access and modification of field values.

```
public abstract class SuperObject{

    public Rectangle solidArea = new Rectangle(0, 0,48,48);
    public int solidAreaDefaultX,solidAreaDefaultY;
    public BufferedImage image;
     public int worldX, worldY;
     public boolean collision = false;
     public String name;
     public int size;
     public String text;
```

Figure 11.

b. Protected Fields

In subclasses like NPC2, the protected keyword is used to protect the solidArea field. This allows the field

to be accessed only by subclasses, maintaining security and encapsulation.

```
public class NPC2 extends SuperObject{
    public NPC2(){
        name = "NPC2";
        try {
            image = ImageIO.read(getClass().getResourceAsStream("/picture/object/NPC2.png"));
        } catch (Exception e) {
            e.printStackTrace();
        }
        collision = true;
        size = 48;


    }
}
```

Figure 12.

```
public abstract class SuperObject{

    public Rectangle solidArea = new Rectangle(0, 0,48,48);
    public int solidAreaDefaultX,solidAreaDefaultY;
```

Figure 13.

In the SuperObject class, the solidArea field is declared with the protected keyword. This means that it can be accessed by subclasses (like NPC2), ensuring that only subclasses have direct access to this field. This is an example of encapsulation and access control in object-oriented programming.

3. Abstraction

Abstraction refers to the concept of simplifying complex systems by modeling classes based on the essential properties and behaviors they share. Abstraction involves focusing on the relevant aspects of an object while ignoring unnecessary details.

The SuperObject class is declared as an abstract class using the abstract keyword. Abstract classes are a form of abstraction, as they cannot be instantiated on their own and may contain abstract methods that must be implemented by concrete subclasses.

```java
package object;

import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.image.BufferedImage;
import main.GamePanel;

public abstract class SuperObject {

    // Other fields...

    protected Rectangle solidArea = new Rectangle(0, 0, 48, 48);
    // Other fields...

    public abstract void draw(Graphics2D g, GamePanel gp);

    // Other methods...
}
```

Figure 14.

VI. SOLID Principles

1. Single Responsibility Principle (SRP)

This principle states that a class should have only one reason to change. In the UML, you can see that there are many classes, each with a specific responsibility, such as Player, KeyControl... Each class has a single responsibility, making it easier to maintain and modify.

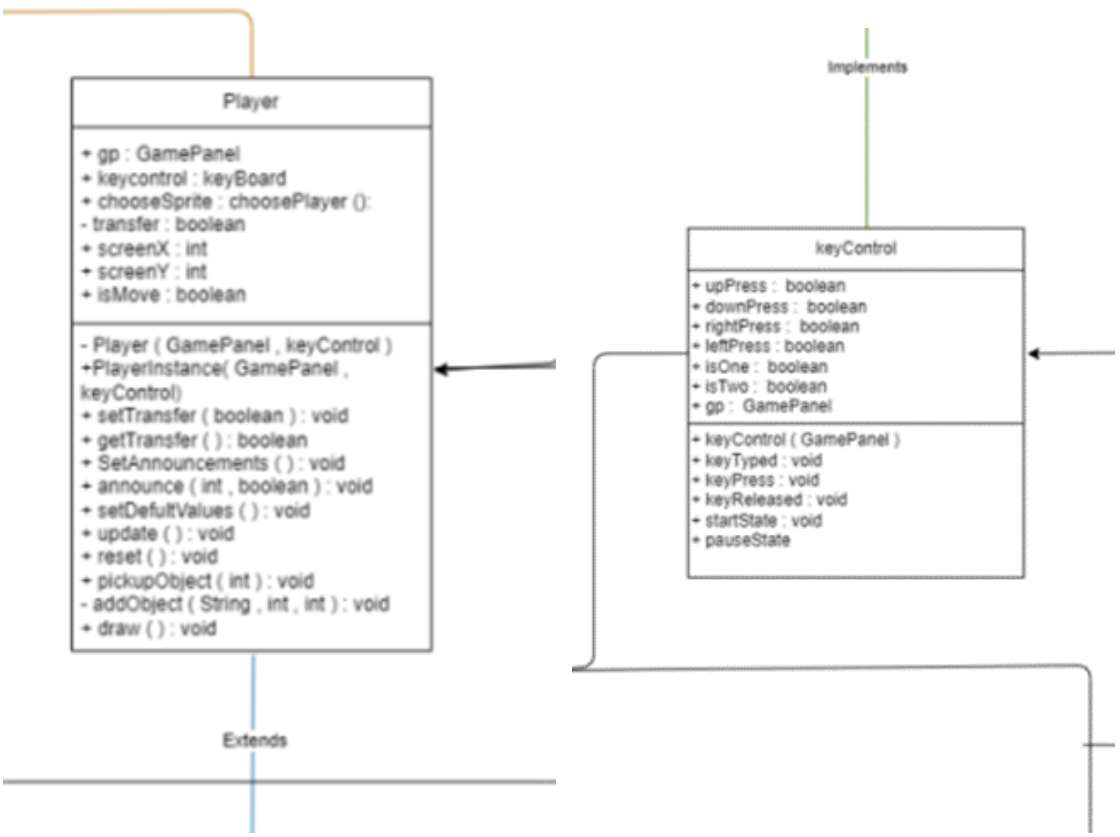

Player
+ gp : GamePanel
+ keycontrol : keyBoard
+ chooseSprite : choosePlayer ():
- transfer : boolean
+ screenX : int
+ screenY : int
+ isMove : boolean
- Player (GamePanel , keyControl)
+PlayerInstance(GamePanel , keyControl)
+ setTransfer (boolean) : void
+ getTransfer ( ) : boolean
+ SetAnnouncements ( ) : void
+ announce (int , boolean) : void
+ setDefultValues ( ) : void
+ update ( ) : void
+ reset ( ) : void
+ pickupObject (int) : void
- addObject (String , int , int) : void
+ draw ( ) : void
Extends
Implements
keyControl
+ upPress : boolean
+ downPress : boolean
+ rightPress : boolean
+ leftPress : boolean
+ isOne : boolean
+ isTwo : boolean
+ gp : GamePanel
+ keyControl (GamePanel)
+ keyTyped : void
+ keyPress : void
+ keyReleased : void
+ startState : void
+ pauseState

Figure 15.

2. Open-Closed Principle (OCP)

This principle states that classes should be open for extension but closed for modification. In the UML, the SuperObject class is an abstract class that can be extended by other classes like object_chest, object_apple, and object_door. This allows for new object types to be added without modifying the existing code.

Figure 16.

3. Liskov Substitution Principle (LSP)

This principle states that objects of a superclass should be able to be replaced by objects of a subclass without affecting the correctness of the program. In the UML, the Entity class is the superclass, and Player and SuperObject are subclasses. The Player and SuperObject classes can be used interchangeably in the code without causing any issues.

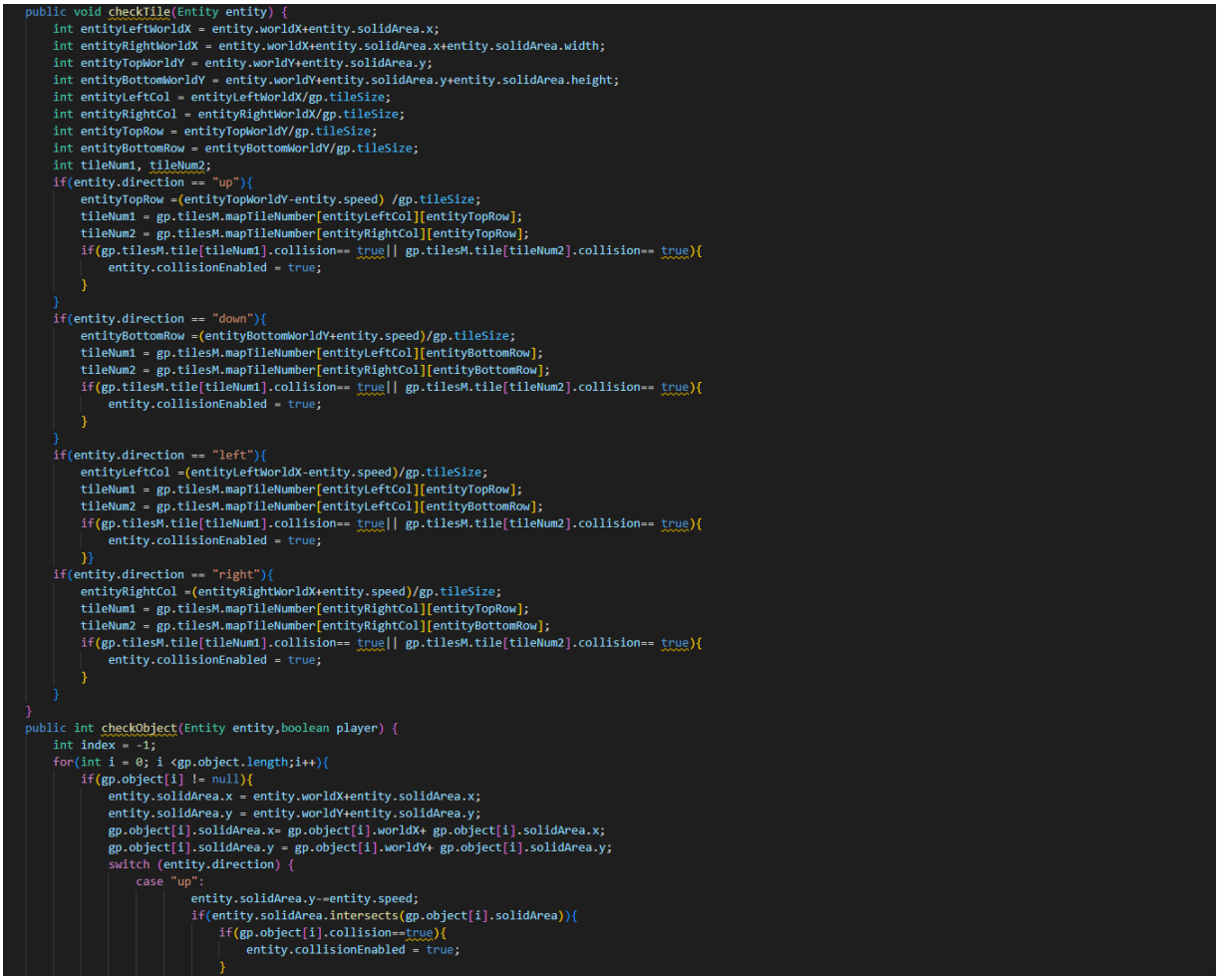
```
public class Player extends Entity {
    choosePlayer chooseSprite = new choosePlayer();
    private boolean transfer = false;
    public int screenX;
    public int screenY;
    private int Key_count = 0;
    public boolean isMove = true;
    private String objectName;
    GamePanel gp;
    public keyControl keyBoard;
    public int global_index = 0;

    private static Player player = null;

    private Player(GamePanel gp, keyControl keyBoard) {
        this.gp = gp;
        this.keyBoard = keyBoard;
        DefultValues();
        setDefultValues();
    }

    public static synchronized Player getInstance(GamePanel gp, keyControl keyBoard) {
        if (player == null) {
            player = new Player(gp, keyBoard);
        }
        return player;
    }

    public void DefultValues() {
        screenX = gp.screenWidth / 2 - (gp.tileSize / 2);
        screenY = gp.screenHeight / 2 - (gp.tileSize / 2);
        solidArea = new Rectangle();
        solidArea.x = 8;
        solidArea.y = 10;
        solidArea.width = 32;
        solidArea.height = 32;
        solidAreaDefaultX = solidArea.x;
        solidAreaDefaultY = solidArea.y;

    }

    public void setDefultValues() {
        worldX = gp.tileSize * 15;// 15
        worldY = gp.tileSize * 11;// 11
        speed = 3;
        direction = "down";
        transfer = true;

    }

    public void update() {
        gp.eventH.checkEvent(worldX, worldY);
        // keyBoard player

        if ((gp.gamestate == gp.announceState || gp.gamestate == gp.playState)) {
            if (keyControl.pPress == true) {
                gp.gamestate = gp.pauseState;
                gp.stopMusic();
```

Figure 17.

4. Interface segregation

Focused interfaces: I'll look for interfaces that define cohesive sets of methods, avoiding unnecessary coupling.

Client-specific interfaces: I'll check if interfaces are tailored to the specific needs of their clients, minimizing dependencies.



```
public void checkTile(Entity entity) {
    int entityLeftWorldX = entity.worldX+entity.solidArea.x;
    int entityRightWorldX = entity.worldX+entity.solidArea.x+entity.solidArea.width;
    int entityTopWorldY = entity.worldY+entity.solidArea.y;
    int entityBottomWorldY = entity.worldY+entity.solidArea.y+entity.solidArea.height;
    int entityLeftCol = entityLeftWorldX/gp.tileSize;
    int entityRightCol = entityRightWorldX/gp.tileSize;
    int entityTopRow = entityTopWorldY/gp.tileSize;
    int entityBottomRow = entityBottomWorldY/gp.tileSize;
    int tileNum1, tileNum2;
    if(entity.direction == "up"){
        entityTopRow =(entityTopWorldY-entity.speed) /gp.tileSize;
        tileNum1 = gp.tilesM.mapTileNumber[entityLeftCol][entityTopRow];
        tileNum2 = gp.tilesM.mapTileNumber[entityRightCol][entityTopRow];
        if(gp.tilesM.tile[tileNum1].collision== true|| gp.tilesM.tile[tileNum2].collision== true){
            entity.collisionEnabled = true;
        }
    }
    if(entity.direction == "down"){
        entityBottomRow =(entityBottomWorldY+entity.speed)/gp.tileSize;
        tileNum1 = gp.tilesM.mapTileNumber[entityLeftCol][entityBottomRow];
        tileNum2 = gp.tilesM.mapTileNumber[entityRightCol][entityBottomRow];
        if(gp.tilesM.tile[tileNum1].collision== true|| gp.tilesM.tile[tileNum2].collision== true){
            entity.collisionEnabled = true;
        }
    }
    if(entity.direction == "left"){
        entityLeftCol =(entityLeftWorldX-entity.speed)/gp.tileSize;
        tileNum1 = gp.tilesM.mapTileNumber[entityLeftCol][entityTopRow];
        tileNum2 = gp.tilesM.mapTileNumber[entityLeftCol][entityBottomRow];
        if(gp.tilesM.tile[tileNum1].collision== true|| gp.tilesM.tile[tileNum2].collision== true){
            entity.collisionEnabled = true;
        }}
    if(entity.direction == "right"){
        entityRightCol =(entityRightWorldX+entity.speed)/gp.tileSize;
        tileNum1 = gp.tilesM.mapTileNumber[entityRightCol][entityTopRow];
        tileNum2 = gp.tilesM.mapTileNumber[entityRightCol][entityBottomRow];
        if(gp.tilesM.tile[tileNum1].collision== true|| gp.tilesM.tile[tileNum2].collision== true){
            entity.collisionEnabled = true;
        }
    }
}
public int checkObject(Entity entity,boolean player) {
    int index = -1;
    for(int i = 0; i <gp.object.length;i++){
        if(gp.object[i] != null){
            entity.solidArea.x = entity.worldX+entity.solidArea.x;
            entity.solidArea.y = entity.worldY+entity.solidArea.y;
            gp.object[i].solidArea.x= gp.object[i].worldX+ gp.object[i].solidArea.x;
            gp.object[i].solidArea.y = gp.object[i].worldY+ gp.object[i].solidArea.y;
            switch (entity.direction) {
                case "up":
                        entity.solidArea.y-=entity.speed;
                        if(entity.solidArea.intersects(gp.object[i].solidArea)){
                            if(gp.object[i].collision==true){
                                entity.collisionEnabled = true;
                            }
```

Figure 18.

5. Dependency inversion

Dependency on abstractions: I'll identify cases where high-level classes depend on abstractions (interfaces or abstract classes) rather than concrete implementations.

Loose coupling: I'll assess the level of coupling between classes and how dependencies are managed.

VII. Conclusion

In conclusion, the successful completion of this Object-Oriented Programming project marks a significant milestone in our journey toward creating robust and maintainable software solutions. Through the application of OOP principles such as encapsulation, inheritance, and polymorphism, we have achieved a modular and extensible codebase that promotes code reuse and enhances readability.

Throughout the project, we adhered to best practices, such as design patterns and SOLID principles, to ensure a high level of code quality. Regular code reviews and collaborative discussions played a crucial role in maintaining consistency and identifying areas for improvement

VIII. Limitation

IX. Reference