**Game: MARIO GAME   
Software re engineering**

**Assignment 2**

**GROUP MEMEBERS:**

**Waniya Wajahat**

**Adeeba Hanif**

**Saira Ramzan  
  
  
Code Smells and Solutions**

1. **Long Method**:
   * **Smell**: The run() method in the GamePanel class is handling multiple responsibilities, including updating the game state and rendering graphics.
   * **Solution**: Refactor the run() method into smaller methods that focus on specific tasks, such as updateGameState() and renderGraphics().
   * **Justification**: Following the **Single Responsibility Principle** (SRP) makes the code more modular and easier to maintain. Each method can be tested independently, improving code quality.

**Refactored Example**:

java

Copy code

@Override

public void run() {

while (running) {

updateGameState(); // Handle game state updates

renderGraphics(); // Handle rendering

try {

Thread.sleep(16); // ~60 FPS

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

private void updateGameState() {

mario.update();

}

private void renderGraphics() {

repaint();

}

1. **Magic Numbers**:
   * **Smell**: The code contains hardcoded values (e.g., 100, 450, and 16) without context or explanation.
   * **Solution**: Define constants at the top of the class to improve readability and maintainability.
   * **Justification**: Using constants gives meaningful names to values, making it easier to understand their purpose. It also makes future adjustments easier, adhering to the **DRY (Don't Repeat Yourself)** principle.

**Refactored Example**:

java

Copy code

private static final int PLAYER\_WIDTH = 50;

private static final int PLAYER\_HEIGHT = 50;

private static final int GROUND\_Y\_POSITION = 450;

private static final int TARGET\_FPS = 60;

// In updateGameState or other methods, replace numbers with constants:

if (y >= GROUND\_Y\_POSITION) {

y = GROUND\_Y\_POSITION;

}

1. **Poor Naming Conventions**:
   * **Smell**: Variable and method names like dx, dy, updateGame(), and renderGraphics() could be more descriptive.
   * **Solution**: Use more meaningful names that convey purpose and functionality. For example, rename dx to horizontalVelocity and dy to verticalVelocity.
   * **Justification**: Clear naming improves code readability and understanding, which is essential for long-term maintenance. It aligns with the **self-documenting code** principle.

**Refactored Example**:

java

Copy code

private int horizontalVelocity;

private int verticalVelocity;

// Usage in update method

horizontalVelocity = left ? -5 : right ? 5 : 0;

1. **God Object**:
   * **Smell**: The GamePanel class may become a "God Object" if it takes on too many responsibilities (e.g., handling rendering, input, game logic).
   * **Solution**: Consider splitting the logic into separate classes or using a **Game Manager** class to oversee different aspects of the game (input handling, rendering, and game state).
   * **Justification**: This separation of concerns leads to better-organized code, easier testing, and increased flexibility for future changes, aligning with the **Separation of Concerns** principle.

**Refactored Example**:

java

Copy code

public class GameManager {

private Player mario;

private GamePanel gamePanel;

public GameManager(GamePanel gamePanel) {

this.gamePanel = gamePanel;

this.mario = new Player(100, GROUND\_Y\_POSITION);

}

public void update() {

mario.update();

}

public void render(Graphics g) {

mario.draw(g);

}

}

1. **Tight Coupling**:
   * **Smell**: The GamePanel is tightly coupled with the Player class, making it difficult to modify or extend either class independently.
   * **Solution**: Use interfaces or abstract classes to define behaviors that can be implemented by different classes. For example, create a GameObject interface for all game entities.
   * **Justification**: Reducing coupling enhances modularity and makes the system easier to modify and extend. This follows the **Dependency Inversion Principle**, one of the SOLID principles of object-oriented design.

**Refactored Example**:

java

Copy code

public interface GameObject {

void update();

void draw(Graphics g);

}

public class Player implements GameObject {

@Override

public void update() {

// player update logic

}

@Override

public void draw(Graphics g) {

// player drawing logic

}

}

**Conclusion**

By addressing these code smells, the Mario game can be made more maintainable, understandable, and adaptable to future changes. Software reengineering principles such as **modularity**, **clarity**, and **separation of concerns** play a crucial role in enhancing the overall quality of the code. This leads to more robust software, easier collaboration, and a better foundation for expanding the game's features and capabilities.

Implementing these changes will also simplify future enhancements, making it easier to introduce new game elements like levels, enemies, or power-ups without significant overhauls.