VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

PROJECT REPORT

Object-Oriented Programming

TOPIC: ESCAPE FROM OOP (GAME)

A. PROJECT INFORMATION

A1. Project Title:

Escape from OOP is a mini game project based on SOLID principles in Object-Oriented Programming.

A2. Team Leader:

Full name : Nguyễn Cao Anh Tiến

Major: Computer Science

Student ID: ITCSIU24085

A3.Other Team Member:

No.	Full Name	Student ID	Major
1	Nguyễn Cao Anh Tiến	ITCSIU24085	Leader
2	Đỗ Đăng Quang	ITITIU24049	Handle Physic and Movement
3	Trịnh Như Uyên	ITDSIU24056	Design characters and items
4	Trần Nguyễn Trúc Mai	ITCSIU24055	Draw and design game resources

B. Introduction

Project Title

Escape from OOP is a 2D game project. It shows how SOLID rules work in Object-Oriented Programming. SOLID rules are simple ideas to make code better. The game is a learning tool. It explains how good design patterns and rules make code easy to keep and add to.

Project Overview

"Escape from OOP" is a fun 2D game. Players control a character named Shin. They move through a map, collect coins, use items, avoid enemies, and save Nanako. The game has a full loop with win and lose conditions. It uses physics for movement, checks for hits, and manages game states. The project focuses on SOLID rules. It

shows Interface Segregation Principle well. This means splitting big tasks into small ones. It also shows areas to improve in other rules like Open/Closed, Dependency Inversion, and Single Responsibility.

C.PROGRAMMING LANGUAGE

- **Programming language**: Java (JDK 8 or higher)
- **IDE**: Visual Code Studio
- **Build System**: Manual build using javac with bash scripts (run.sh for Unix/Linux, run.bat for Windows)
- **Dependencies**: Basic Java libraries (javax.swing for GUI, java.awt for graphics)

D. SYSTEM DESIGN AND ARCHITECTURE

Project Structure Overview

The "Escape from OOP" project has an organized structure. It is split into logical groups that keep things separate and easy to manage. The code is in these main folders:

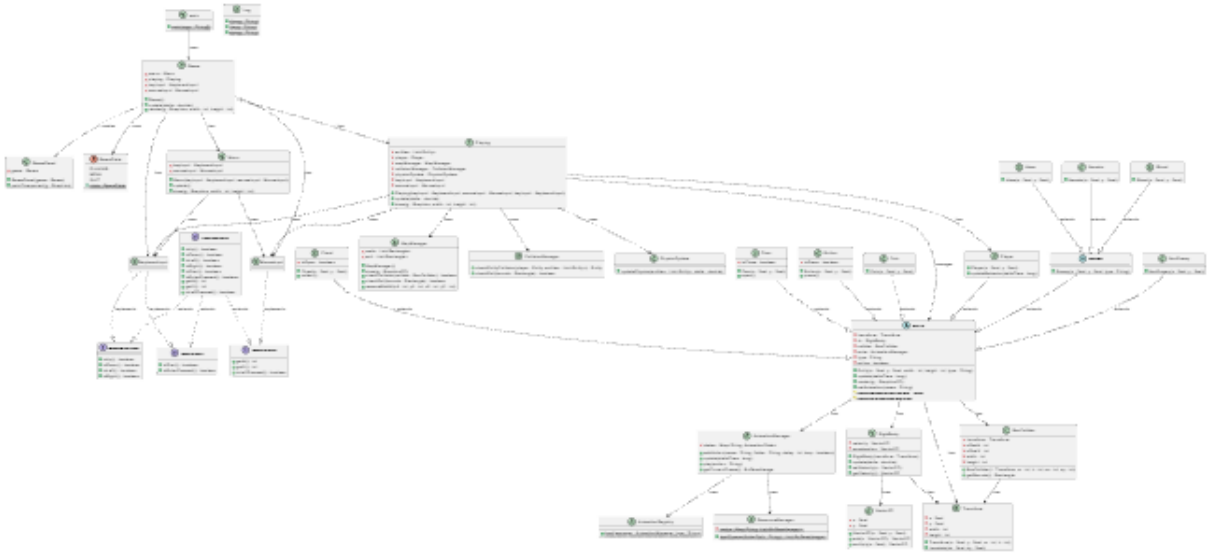
D1. Overall Architecture

The game's system is designed using a multi-layered structure:

- ❖ Game Logic
  - ➢ Responsible for handling movement, collision detection, coin collection, win conditions, and enemy behaviors.
- ❖ Input
  - ➢ Responsible for reading and processing the player's key inputs.
- ❖ Rendering
  - ➢ Responsible for rendering and drawing the game’s images.
- ❖ Abstraction

➢ Utilizes interfaces and abstract classes to reduce dependencies and ensure a more modular design.

D2. UML Class Diagram



Link to Diagram for detail: https://bit.ly/4ap087a

D3. Component Descrip tion

1. **Core Architecture**

- **core/**: Main game control classes
- `Game.java`: Main game manager for menu, playing, and quit states
- `GamePanel.java`: JPanel part that runs the game loop and draws at 60 FPS
- `PhysicsSystem.java`: Updates physics for all things in the game world
- `MapManager.java`: Handles map parts like walls, exits, and changes
- `CollisionManager.java`: Checks for hits between things and map parts
-

Entity System

- **entities/**: Uses the Entity-Component-System pattern. This is a way to organize game objects.
- `Entity.java`: Base class with common features for all game objects

- **characters/**: Player and enemy types
    - `Player.java`: Shin character with movement and action logic
    - `Enemy.java`: Base enemy class
    - `Ghost.java`, `Hima.java`, `Nanako.java`: Specific enemy types
    - `NonEnemy.java`: Base class for friendly characters
- **items/**: Things to interact with
    - `Chest.java`, `Door.java`, `Button.java`, `Coin.java`: Things to collect or use

Graphics and Animation

- **graphics/**: Handles drawing and visual effects
- **ui/**: User interface parts
    - `Menu.java`: Main menu screen
    - `Playing.java`: Gameplay UI overlay
    - `GameOverOverlay.java`, `GameWinOverlay.java`: End-game screens
- **animation/**: Animation system
    - `AnimationManager.java`: Manages frame-based animations
    - `AnimationRegistry.java`: Main list for animation resources
    - `ResourceManager.java`: Loads and stores game assets

Input System

- **input/**: Split input handling following ISP. ISP means Interface Segregation Principle, which splits big interfaces into small ones.
- `IMovementInput.java`: Interface for movement controls (WASD/arrows)
- `IMenuInput.java`: Interface for menu navigation (Enter, Escape, etc.)
- `IMouseInput.java`: Interface for mouse actions
- `KeyboardInput.java`: Handles keyboard movement and menu inputs

- `MouseInput.java`: Handles mouse position and clicks

Physics Engine

- **physics/**: Custom 2D physics system
- `RigidBody.java`: Manages speed, acceleration, and physics updates
- `Transform.java`: Handles position, rotation, and size
- `Vector2D.java`: 2D vector math and operations
- `BoxCollider.java`: Axis-Aligned Bounding Box (AABB) collision check.

  AABB is a simple box shape for hit detection.

Utilities

- **utils/**: Helper tools
- `GameState.java`: List of game states (MENU, PLAYING, QUIT)
- `Log.java`: Logging tool (basic now)

Resources

- **resources/**: Asset organization
- **audio/**: Sound effects for win/lose
- **characters/**: Images for player and enemies
- **items/**: Images for things to use
- **map/**: Map layout and wall images

E.GAME DESIGN

- **Gameplay Mechanics**: Player moves with sticky controls, collects coins/items, avoids enemies, presses buttons to open doors, leads Nanako to exit.
- **Levels**: Single map with walls, enemies, and collectibles.
- **UI/UX**: Menu for start, Playing for gameplay, overlays for win/lose.
- **Art Style**: 2D sprites with animations.
- **Sound**: Basic audio for win/lose (not implemented in code).

E1. Algorithm Analysis (Math, Physics, Collision)

E1.1 Player Movement Algorithm

- Sticky Movement: Player stops immediately when no input, no sliding.
- Code: In Player.updateBehavior(), check input; if no keys, set velocity to 0.
- Math: Velocity = input_direction * speed (speed = 200 units/s).
- Physics: No friction; instant stop for "sticky" feel.

E1.2 Enemy AI (Chasing Algorithm)

- Vector-Based Chasing: Enemies calculate direction to player and move.
- Math: Direction vector = (player_pos - enemy_pos).normalize()
- Velocity = direction * enemy_speed (e.g., 100 units/s for Ghost).
- Code: In Enemy.updateBehavior(), compute vector, set velocity.
- Analysis: Simple pursuit; no pathfinding, assumes direct line.

E1.3 Collision Detection (AABB - Axis-Aligned Bounding Box)

- Algorithm: Check overlap on X and Y axes.
- Math: Two boxes overlap if (box1.x < box2.x + box2.w) && (box1.x + box1.w > box2.x) && same for Y.
- Code: In BoxCollider, getBounds() returns Rectangle; CollisionManager checks overlaps.
- Efficiency: O(1) per pair; used for walls, entities.

E1.4 Physics Calculations

- RigidBody Update:
- Acceleration: Gravity = (0, 300) units/s² downward.
- Velocity: v = v + a * dt
- Position: pos = pos + v * dt
- Code: RigidBody.update(double dt)
- Transform: Position (x,y), size (w,h); translate adds to pos.
- Vector2D: 2D vector ops: add, multiply (scalar), normalize (unit vector).

E1.5 Win/Lose Logic

- Win: countCoins >= 6 && nanakoFollow && isOpenChest && collision with exit.

- Lose: Collision with enemy (hitEntity instanceof Enemy).
- Reset: Re-init entities, set countCoins=0, stop audio.

E1.6 Animation and Rendering

- Frame-Based Animation: AnimationManager cycles frames at delay (ms).
- Rendering: Draw entities in order; UI overlays on top.
- This ensures realistic physics, efficient collisions, and balanced gameplay.

F. IMPLEMENTATION DETAILS (SOLID PRINCIPLES)

1. A: Single Responsibility Principle (SRP):

The Single Responsibility Principle means that each class should have only one responsibility, or task, to handle:

- AnimationManager.java: Manages animations only.
- KeyboardInput.java: Handles key inputs only.
- MouseInput.java: Handles mouse inputs only.
- MapManager.java: Handles map control and management only.

These classes demonstrate SRP clearly by ensuring each one is responsible for a single task or functionality.

1. B: SRP Code Excerpts :

→

G.Project Goal

1.Performance

- Correct gameplay as predicted
- Smooth movement
- Real-time update correctly
- Collision working properly

2.Quality

- Follow SOLID
- High cohesion
- Easy to fix, add, test new features
- Codes and code structures are easy to understand

H.Conclusion

By applying SOLID design patterns, we transformed a standard game implementation into an extensible system. The clear separation of concerns ensures that the codebase remains easy to navigate and modify. This project demonstrates that even classic game mechanics benefit significantly from a rigorous, clean-code approach."