BACHELOR OF APPLIED ARTIFICIAL INTELLIGENCE (WITH HONOURS)

INF1009, AY 2024 Trimester 2

Project 2: Game

Done by:

Student Name	Student ID
SEBASTIAN NUGUID FERNANDEZ	2302811
YEO SONG CHEN	2302785
JUSTIN TAN YONG AN	2302675
NITHIYAPRIYA RAMESH	2302704
CHIN QUN ZHEN	2302814
FUN KAI JUN	2303556

Declaration:

I/we hereby pledge that this assignment is not plagiarised and has been written wholly as a result of our own research and compilation of information.

Signed: P1 – Team 1

Dated: 28th March 2024

Introduction

This report details the development and execution of our game under the chosen topic, **Healthy Eating For Kids**, for our Object-Oriented Programming course, applying OOP principles, design patterns, and architectural frameworks. Aimed at constructing a compelling and educational gameplay experience, our project adheres to SOLID principles and integrates design patterns to foster a robust, scalable, and maintainable codebase. The project's scope encompasses the strategic design and development of the game, emphasising the application of object-oriented techniques, effective pattern implementation, and maintaining a distinct division between engine and gameplay code, with the final evaluation focusing on the quality of code and implementation of OOP methodologies.

Overall System Design

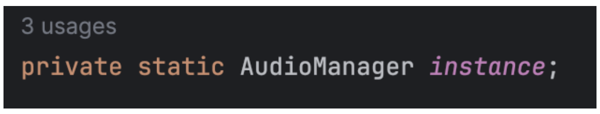
Overview and Purpose

This section outlines the key improvements and the incorporation of design patterns within a game development project to enhance robustness, maintainability, and scalability. Focusing on the specific implementations of Singleton, Factory and other design patterns, it discusses their application in managing game audio, collisions, and AI behaviour. The report also highlights general improvements in modularization, error handling, and the implementation of fallback mechanisms and input validation.

Design Patterns Implemented

Audio Manager Singleton:

Implementation Details:



```
3 usages
private static AudioManager instance;
```

Figure 1: AudioManager Singleton Design Pattern

The Singleton design pattern in the `**AudioManager**` class is instantiated through a private static variable `instance` and controlled via the public static `getInstance()` method. This setup guarantees that only one instance of the AudioManager is created, accessible throughout the game. If the `instance` doesn't exist upon a call to `getInstance()`, it initialises a new

AudioManager; otherwise, it returns the existing one. This ensures that audio management is consistent and centralised, avoiding the complications of multiple, disparate audio control systems.

Impact of Implementation:

Centralised Management: The game benefits from having a single, unified point for managing all audio resources, like sound effects and music tracks. This central management facilitates easier adjustments and streamlining of audio controls, leading to a more organised and manageable codebase.

Consistency: A single instance of the AudioManager ensures uniformity in audio settings across the entire game, from volume levels to playback states. This uniformity helps in avoiding inconsistencies that could disrupt the gaming experience, ensuring that sound levels and states are the same game-wide, thereby eliminating confusion and bugs associated with disparate audio conditions.

Resource Efficiency: Implementing the Singleton pattern contributes significantly to efficient resource management, an essential aspect in game development. By having only one instance of the AudioManager, the game minimises memory and resource usage. Efficient handling of audio resources ensures that sounds and music are loaded, played, and disposed of correctly, which is vital for maintaining optimal game performance and reducing overhead.

Global Access: The global accessibility of the AudioManager, facilitated by the Singleton pattern, means that any part of the game can easily control audio without the need to pass around specific references to the audio manager. This decouples the game components from the audio management logic, allowing for more flexible game development and easier integration of sound effects and music. It enhances the modular nature of the game's architecture, allowing developers to add or modify audio elements without extensive reconfiguration.

Entity Factory Pattern Class:

Implementation Overview:

The EntityFactory class in our game project is an implementation of the Factory design pattern. This class provides static methods for creating various game entities, such as player characters, static objects, collectibles, and enemies, each with specified properties and behaviours. The use of static methods allows for straightforward creation of entities without needing to instantiate the factory itself, simplifying the game entity creation process.

Key Features and Functions:

Type-Specific Entity Creation: The factory offers methods like createPlayerEntity, createStaticEntity, and createCollectibleCoin, among others, allowing for the instantiation of different types of game entities with customised attributes such as texture, dimensions, position, and speed.

Entity-Specific Behaviours: For dynamic entities like collectibles, the factory assigns behaviour patterns (e.g., CollectableSpawnBehaviour), illustrating an adaptation of the Strategy pattern to allow for varied entity behaviour.

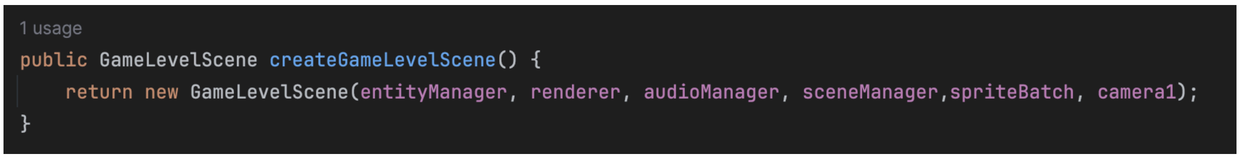
Impact of Implementation:

Modularization and Reusability: Centralising entity creation in a single class enhances code modularity and reusability. Developers can add new entity types or modify existing ones without affecting other parts of the game.

Consistency and Maintainability: The uniform approach to entity creation ensures consistency across the game's ecosystem, making the codebase easier to understand and maintain.

Robustness: Input and texture validation within the factory methods contribute to the game's robustness, preventing crashes and other issues related to invalid entity states or missing resources.

Scene Factory Pattern Class:



```
1 usage
public GameLevelScene createGameLevelScene() {
    return new GameLevelScene(entityManager, renderer, audioManager, sceneManager,spriteBatch, camera1);
}
```

Figure 2: Scene Factory

Implementation Overview:

The SceneFactory class employs the Factory design pattern to create different scenes such as main menu, game level, and game end scenes. This encapsulation of scene creation enhances modularity and decouples scene instantiation from usage.

Key Features and Functions:

Scene Creation Methods: Includes methods like createMainMenuScene, createGameLevelScene, and createGameEndScene, each returning an instance of the respective scene initialised with necessary components such as Renderer, AudioManager, and SceneManager.

Impact of Implementation:

Enhanced Flexibility: By abstracting scene creation, new scenes can be added or existing ones modified with minimal impact on other game systems such as the scene manager

Improved Cohesion: Each scene is constructed with all necessary components, ensuring that every part of the game functions cohesively within its intended context.

Simplified Scene Management: Centralising scene creation in the SceneFactory removes the responsibility from scene manager, enabling it to focus on the game's different states and transitions, contributing to a cleaner and more organised codebase.

AIManager Strategy Pattern Class:

Implementation Overview:

The AIManager class implements the **Strategy Pattern** by managing AI entities and their behaviours within the game. This approach centralises AI behaviour control, allowing for dynamic updates and assignment of behaviours to different entities based on game logic.

Key Features and Functions:

Behaviour Management: Manages a mapping between AI entities and their corresponding behaviours (AIBehaviour), facilitating easy access and updates. This setup enables the dynamic assignment and swapping of behaviours for individual AI entities.

Dynamic Updates: The update method iterates over all AI entities, updating each based on its assigned behaviour. This regular update cycle allows behaviours to change in response to game events or conditions, making AI entities' actions adaptable and context-sensitive.

Impact of Implementation:

Behavioural Flexibility: Utilising the Strategy Pattern allows for a high degree of flexibility in how AI entities behave. Developers can easily introduce new behaviours or modify existing ones without altering the entities themselves or the overarching AI management framework.

Decoupling and Modularity: AI entities are decoupled from their specific behaviours, promoting modularity. This separation allows developers to add, remove, or modify behaviours independently of the entities, simplifying maintenance and scaling.

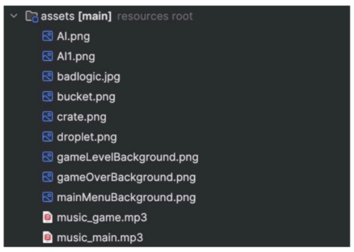
Streamlined AI Management: Centralising AI behaviour management within a single class streamlines the process of updating and controlling AI entities across different game scenes and states, contributing to a more organised and manageable code structure.

Key Improvements to our Game Engine

In our part 1 project, we received positive feedback on our implementations on the key managers, their relationships and the upholding of the SOLID principles. However despite this we also highlighted our limitations such as the robustness issue due to the lack of input validation, exception handling and documentation. With the help of the feedback from our Professors and looking back on our shortcomings, we implemented the following key improvements to ensure robustness, readability and reusability for our revamped game engine.

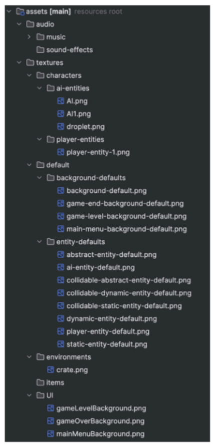
Enhanced Asset Management:

Organisation and Packaging: As seen in Figure 1, the assets are efficiently categorised into designated folders based on type ensuring easy access during deployment. This organised approach enhances overall management, simplifies asset retrieval throughout the project's lifecycle and follows the principle of modularization, ensuring maintainability.



```
assets [main] resources root
    AI.png
    AI1.png
    badlogic.jpg
    bucket.png
    crate.png
    droplet.png
    gameLevelBackground.png
    gameOverBackground.png
    mainMenuBackground.png
    music_game.mp3
    music_main.mp3
```

Old Assets Packaging



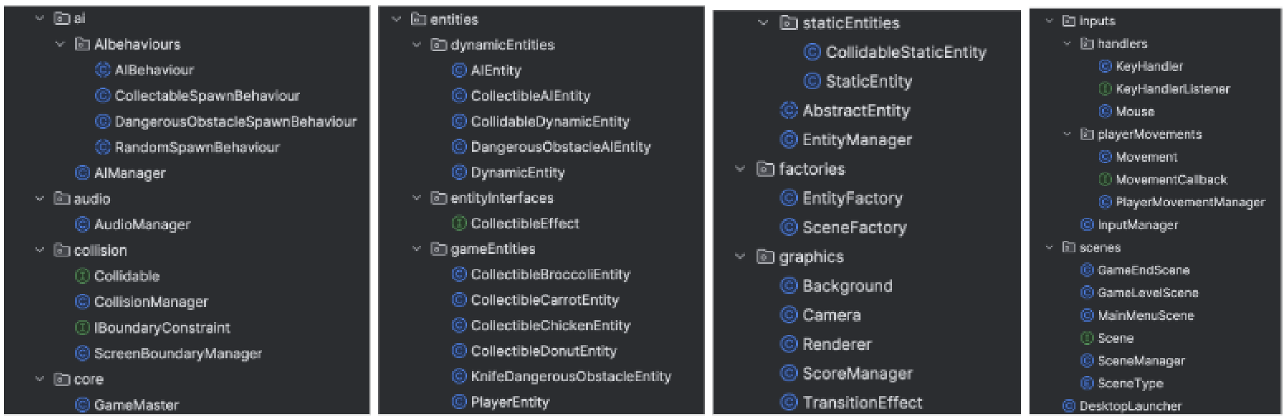
```
assets [main] resources root
  audio
    music
    sound-effects
  textures
    characters
      ai-entities
        AI.png
        AI1.png
        droplet.png
      player-entities
        player-entity-1.png
    default
      background-defaults
        background-default.png
        game-end-background-default.png
        game-level-background-default.png
        main-menu-background-default.png
      entity-defaults
        abstract-entity-default.png
        ai-entity-default.png
        collidable-abstract-entity-default.png
        collidable-dynamic-entity-default.png
        collidable-static-entity-default.png
        dynamic-entity-default.png
        player-entity-default.png
        static-entity-default.png
    environments
      crate.png
    items
    UI
      gameLevelBackground.png
      gameOverBackground.png
      mainMenuBackground.png
```

New Assets Packaging

Figure 3:. Organisation and Packaging

Enhanced System Structure (via Packages):

Organisation and Packaging:



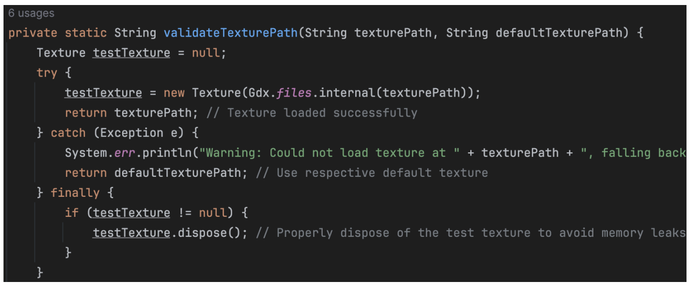
```
ai
  AIbehaviours
    AIBehaviour
    CollectableSpawnBehaviour
    DangerousObstacleSpawnBehaviour
    RandomSpawnBehaviour
  AIManager
audio
  AudioManager
collision
  Collidable
  CollisionManager
  IBoundaryConstraint
  ScreenBoundaryManager
core
  GameMaster
entities
  dynamicEntities
    AIEntity
    CollectibleAIEntity
    CollidableDynamicEntity
    DangerousObstacleAIEntity
    DynamicEntity
  entityInterfaces
    CollectibleEffect
  gameEntities
    CollectibleBroccoliEntity
    CollectibleCarrotEntity
    CollectibleChickenEntity
    CollectibleDonutEntity
    KnifeDangerousObstacleEntity
    PlayerEntity
  staticEntities
    CollidableStaticEntity
    StaticEntity
  AbstractEntity
  EntityManager
factories
  EntityFactory
  SceneFactory
graphics
  Background
  Camera
  Renderer
  ScoreManager
  TransitionEffect
inputs
  handlers
    KeyHandler
    KeyHandlerListener
    Mouse
  playerMovements
    Movement
    MovementCallback
    PlayerMovementManager
  InputManager
scenes
  GameEndScene
  GameLevelScene
  MainMenuScene
  Scene
  SceneManager
  SceneType
DesktopLauncher
```

Figure 4: Demonstration of Enhanced System Structure

Our overall code adopts a **feature-based approach**, meticulously segmenting each core aspect of game functionality into intuitive packages that foster a coherent and maintainable codebase.

The **`ai` package** contains sub-packages for various AI behaviours and an AI manager for cohesive control over AI operations. The **`audio` package** encapsulates audio management, essential for creating an immersive game environment. **Collision detection** is centralised within the **collision package**, which houses interfaces and managers dedicated to maintaining the integrity of game physics. The core package contains the GameMaster, the game's command centre. **Entity classes** are neatly categorised under the **entities package**, with further subdivisions for dynamic entities, entity interfaces, game entities, and static entities; this organisation supports straightforward entity management and scalability. **Factory patterns** are utilised within the factories package to streamline the creation of entities and scenes. The **graphics package** holds the rendering components, ensuring that the visual aspect of the game is crisp and responsive. **Input handling** is designated to the inputs package, with sub-packages for handlers and player movements, reflecting a clear division of responsibilities. Finally, the scenes package comprises all scene-related classes and a scene manager, which oversees the transition between different game states. Overall our approach to organising game engine components promotes maintainability and ease of understanding for developers.

Enhanced Robustness/Readability/Reusability of Entity and Scene Factory:



```
6 usages
private static String validateTexturePath(String texturePath, String defaultTexturePath) {
    Texture testTexture = null;
    try {
        testTexture = new Texture(Gdx.files.internal(texturePath));
        return texturePath; // Texture loaded successfully
    } catch (Exception e) {
        System.err.println("Warning: Could not load texture at " + texturePath + ", falling back
        return defaultTexturePath; // Use respective default texture
    } finally {
        if (testTexture != null) {
            testTexture.dispose(); // Properly dispose of the test texture to avoid memory leaks
        }
    }
```

Figure 5: Exception Handling

Exception Handling and Resource Management: In Figure 5, the method validateTexturePath now not only checks if the texture exists but also handles any exceptions thrown during the texture loading process. This is a key improvement in robustness because it ensures that our game can gracefully handle missing or corrupted texture files without crashing. This separation of texture validation into its own method (validateTexturePath) follows the principle of separation of concerns. In addition to this is the finally block which ensures that any texture instances created for validation are properly disposed of. This addresses potential memory leak issues, which is crucial for games that handle many assets.

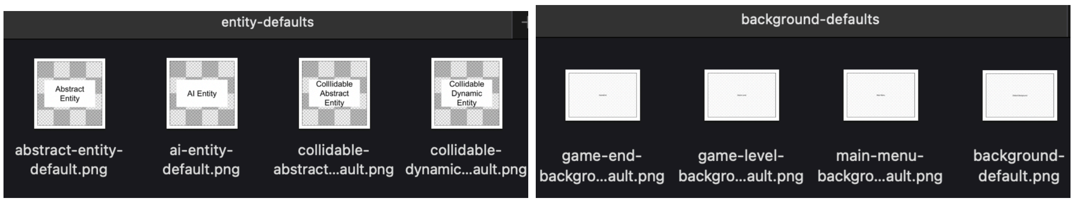
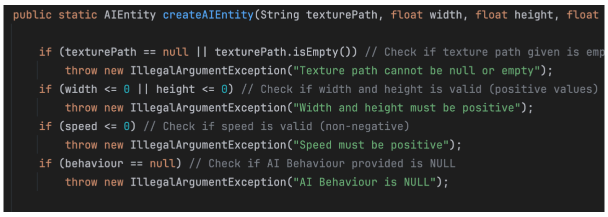

entity-defaults
Abstract Entity
AI Entity
Collidable Abstract Entity
Collidable Dynamic Entity
abstract-entity-default.png
ai-entity-default.png
collidable-abstract...ault.png
collidable-dynamic...ault.png
background-defaults
game-end-backgro...ault.png
game-level-backgro...ault.png
main-menu-backgro...ault.png
background-default.png

Figure 6: Fallback Mechanism

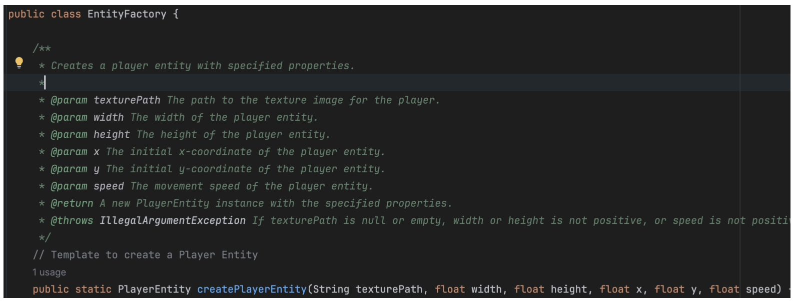
Fallback Mechanism: In Figure 6, the factory methods now include a fallback mechanism for textures. If the specified texture cannot be loaded, the entity will use a default texture instead. This ensures that the game can continue running even if some assets are missing or incorrect, improving the user experience and making debugging easier.



```
public static AIEntity createAIEntity(String texturePath, float width, float height, float

    if (texturePath == null || texturePath.isEmpty()) // Check if texture path given is emp
        throw new IllegalArgumentException("Texture path cannot be null or empty");
    if (width <= 0 || height <= 0) // Check if width and height is valid (positive values)
        throw new IllegalArgumentException("Width and height must be positive");
    if (speed <= 0) // Check if speed is valid (non-negative)
        throw new IllegalArgumentException("Speed must be positive");
    if (behaviour == null) // Check if AI Behaviour provided is NULL
        throw new IllegalArgumentException("AI Behaviour is NULL");
```

Figure 7: Input Validation

Input Validation: The updated EntityFactory now includes more rigorous checks for entity placement within the game area, specifically validating the x and y coordinates against predefined game boundaries (leftBound, floorHeight, playAreaHeight) in addition to the texture path, width, height, and speed. This enhancement ensures that all entities are spawned within the visible game world, preventing errors and improving game integrity.

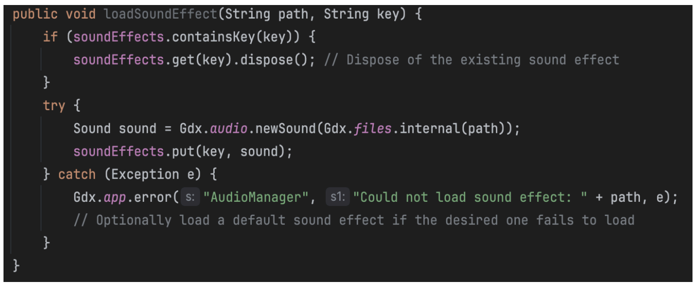


```
public class EntityFactory {

    /**
     * Creates a player entity with specified properties.
     *
     * @param texturePath The path to the texture image for the player.
     * @param width The width of the player entity.
     * @param height The height of the player entity.
     * @param x The initial x-coordinate of the player entity.
     * @param y The initial y-coordinate of the player entity.
     * @param speed The movement speed of the player entity.
     * @return A new PlayerEntity instance with the specified properties.
     * @throws IllegalArgumentException If texturePath is null or empty, width or height is not positive, or speed is not positi
     */
    // Template to create a Player Entity
    1 usage
    public static PlayerEntity createPlayerEntity(String texturePath, float width, float height, float x, float y, float speed)
```

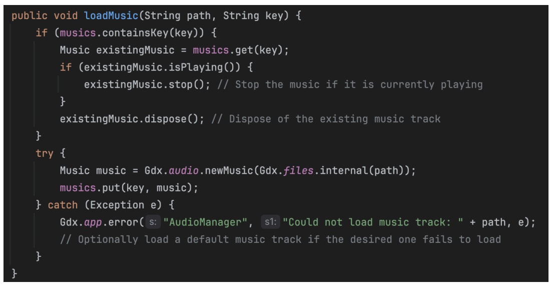
Figure 8: Code Documentation and comments for Readability

Code Documentation: The addition of comments within our methods improves readability and maintainability. Comments explaining the checks and fallback mechanisms helps future developers to understand the purpose and function of each part of the code more quickly.

Enhanced Robustness/Readability of Audio Manager Class:



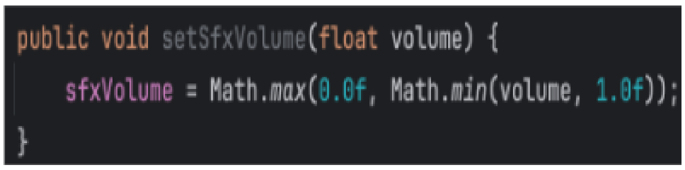
```
public void loadSoundEffect(String path, String key) {
    if (soundEffects.containsKey(key)) {
        soundEffects.get(key).dispose(); // Dispose of the existing sound effect
    }
    try {
        Sound sound = Gdx.audio.newSound(Gdx.files.internal(path));
        soundEffects.put(key, sound);
    } catch (Exception e) {
        Gdx.app.error( s: "AudioManager",  s1: "Could not load sound effect: " + path, e);
        // Optionally load a default sound effect if the desired one fails to load
    }
}
```



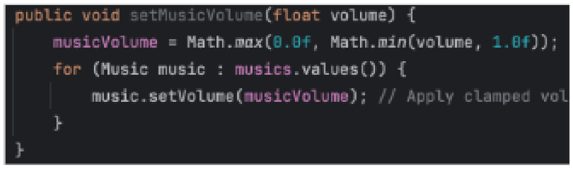
```
public void loadMusic(String path, String key) {
    if (musics.containsKey(key)) {
        Music existingMusic = musics.get(key);
        if (existingMusic.isPlaying()) {
            existingMusic.stop(); // Stop the music if it is currently playing
        }
        existingMusic.dispose(); // Dispose of the existing music track
    }
    try {
        Music music = Gdx.audio.newMusic(Gdx.files.internal(path));
        musics.put(key, music);
    } catch (Exception e) {
        Gdx.app.error( s: "AudioManager",  s1: "Could not load music track: " + path, e);
        // Optionally load a default music track if the desired one fails to load
    }
}
```

Figure 9: Error Handling

Error Handling: In Figure 9, the 'loadSoundEffect' and 'loadMusic' methods employ try-catch blocks to handle exceptions that may occur during the loading process. By catching potential errors, these methods ensure graceful handling of these issues. Logging error messages using 'Gdx.app.error()' facilitates prompt debugging and resolution, enhancing robustness of the system. This approach enables the game to handle missing or corrupted audio files without crashing.



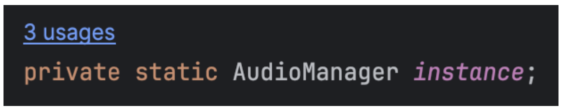
```
public void setSfxVolume(float volume) {
    sfxVolume = Math.max(0.0f, Math.min(volume, 1.0f));
}
```



```
public void setMusicVolume(float volume) {
    musicVolume = Math.max(0.0f, Math.min(volume, 1.0f));
    for (Music music : musics.values()) {
        music.setVolume(musicVolume); // Apply clamped vol
    }
}
```

Figure 10: Volume Clamping

Volume Clamping: As shown in Figure 10, the volume settings are clamped between 0.0 to 1.0 to maintain audio levels within a valid range. By enforcing this constraint, the system ensures consistent and appropriate audio levels throughout the game.

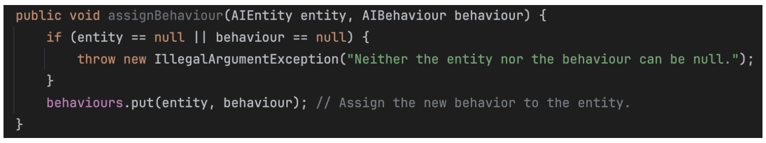


```
3 usages
private static AudioManager instance;
```

Figure 11. Singleton Pattern

Singleton Pattern: Singleton pattern is implemented to ensure one instance of the Audio Manager exists throughout the game's lifecycle. This approach is crucial for essential global audio resource management, allowing centralised control and coordination of audio-related operations.

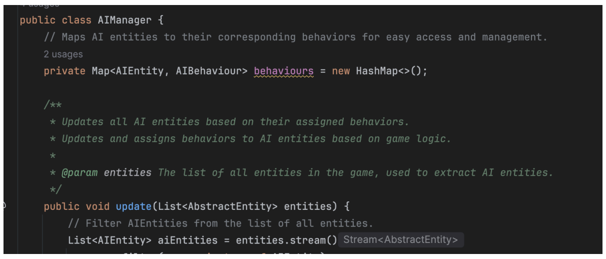
Enhanced Robustness and Readability of AI Manager Class:



```
public void assignBehaviour(AIEntity entity, AIBehaviour behaviour) {
    if (entity == null || behaviour == null) {
        throw new IllegalArgumentException("Neither the entity nor the behaviour can be null.");
    }
    behaviours.put(entity, behaviour); // Assign the new behavior to the entity.
}
```

Figure 12. Exception Handling & Error Messages

Exception Handling: In Figure 12, critical checks are added in the 'AssignBehaviour' method to eliminate null values from entering the system. These checks safeguards against runtime issues and ensure valid behaviour assignments for AI entities.

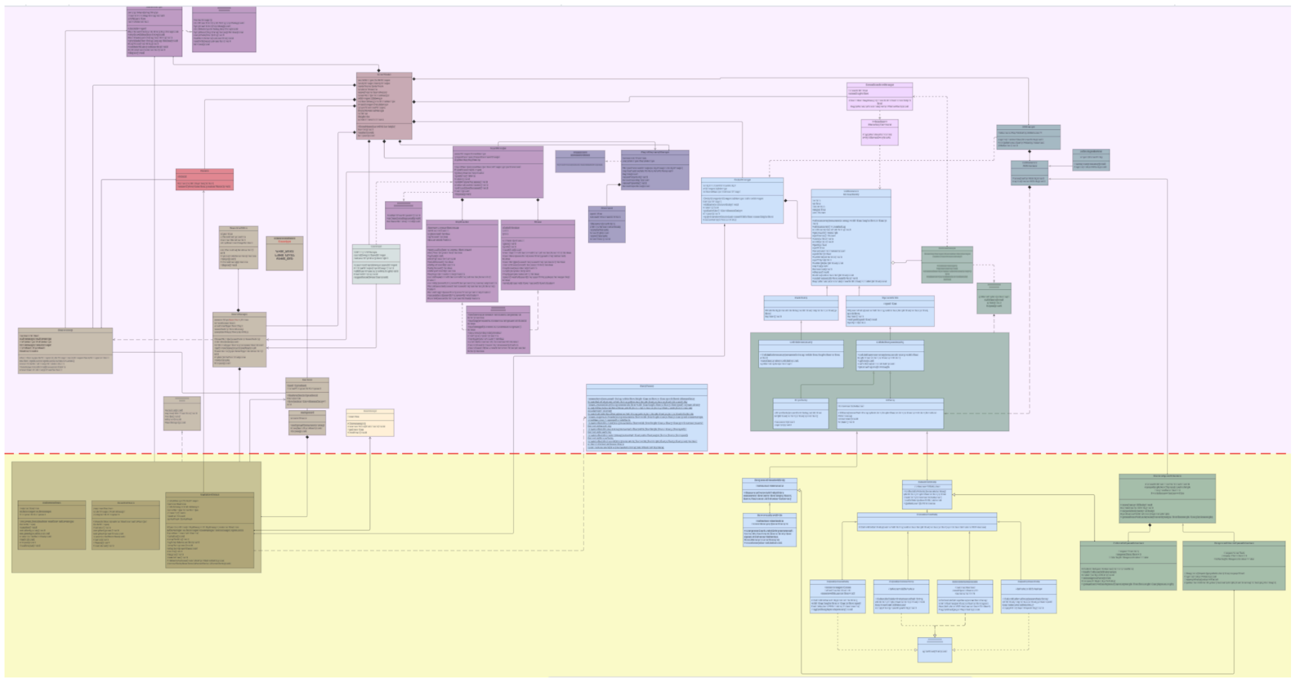


```
public class AIManager {
    // Maps AI entities to their corresponding behaviors for easy access and management.
    2 usages
    private Map<AIEntity, AIBehaviour> behaviours = new HashMap<>();

    /**
     * Updates all AI entities based on their assigned behaviors.
     * Updates and assigns behaviors to AI entities based on game logic.
     *
     * @param entities The list of all entities in the game, used to extract AI entities.
     */
    public void update(List<AbstractEntity> entities) {
        // Filter AIEntities from the list of all entities.
        List<AIEntity> aiEntities = entities.stream() Stream<AbstractEntity>
```

Figure 13. Code Documentation and comments for Readability

Code Documentation: The addition of comments within our methods improves readability and maintainability. Comments explaining the checks and the filtering feature to future developers to understand the purpose and function of each part of the code more quickly.

Layered Architecture



(Click here to access the full UML diagram)

Separation of Concerns: The game engine layer (purple) handles the underlying technical complexities such as rendering graphics, processing input, and managing audio. This separation allows the game layer (yellow) to focus on game-specific logic, like gameplay mechanics, storylines, and world-building.

Reusability: Since the game engine layer provides generic functionalities, it can be reused for other games or projects, which is one of the main advantages of a layered architecture. **Maintainability and Scalability**: With a clear boundary between the game and the engine, maintaining and scaling either layer becomes more manageable. For example, if a new feature or optimization is needed in the rendering process, we can update the game engine without affecting the game logic.

Dependency Direction: In layered architectures, higher layers depend on lower layers but not vice versa. Our game (yellow) depends on services provided by the game engine (purple), but the engine does not depend on the game. This ensures that the engine can operate independently of the game built on top of it.

OOP Principles adopted and justification

Encapsulation: All managers effectively encapsulate their internal states by using private fields and providing controlled access through public methods. This design choice hides implementation details and ensures that the managers are responsible for managing their respective functionalities.

Inheritance: Inheritance is evident in the Entity hierarchy, where subclasses like 'CollectibleDonutEntity', 'CollectibleCabbageEntity', ‘CollectibleChicken’ and 'ColllectibleCarrotEntity' extend the ‘CollectibleCoinEntity’ class. By inheriting from 'AbstractEntity', they gain attributes and behaviours, promoting code reusability. For instance, ‘CollectibleChicken’ utilises the 'super' keyword to inherit common properties from the 'AbstractEntity' superclass.

Polymorphism: The ‘CollectibleEffect’ interface enables polymorphism by allowing different types of entities to be treated uniformly when checking for different effects upon collision. The ‘applyEffect’ method is called on different types of entities through polymorphic behaviour.

Abstraction: Abstraction is represented by the AIBehaviour and Collidable interfaces, along with the AbstractEntity class. AIBehaviour abstracts the concept of AI behaviour, defining the contract that all concrete AI behaviours must follow with methods like move() and update(). The same applies to the Collidable interface, which abstracts the functionality related to collision handling with methods like collision() and getBoundingBox(). This allows different entities (such as CollectibleEntity, DangerousEntity, etc.) to implement these interfaces and provide their own concrete behaviours, promoting reusable and maintainable code.

Composition: The relationship between ‘EntityManager’ and ‘AbstractEntity’ encompasses composition. The composite, ‘EntityManager’ contains objects of another class ‘AbstractEntity’ instances to form a complex object. This design promotes reusability by allowing ‘AbstractEntity’ and its subclasses to be used in different contexts without modification.

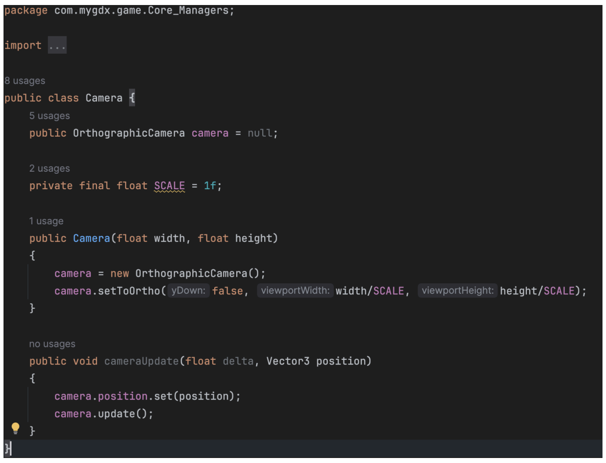
Aggregation: The relationship between EntityManager and AbstractEntity is an example of aggregation. The EntityManager class has a list of AbstractEntity objects, which suggests that it manages these entities, but the lifecycle of the entities is not strictly tied to the EntityManager. AbstractEntities can exist independently of the EntityManager, which is indicative of an aggregation relationship.

Single Responsibility Principles: Each manager adheres to the single responsibility principle by focusing on specific responsibilities. For example, ‘SceneManager’ manages scenes, ‘EntityManager’ handles entities, ‘AudioManager’ manages audio-related functions and ‘InputManager’ handles user input. This modular design enhances readability, maintainability, and ease of future modifications.

Open/Closed Principle: The Open/Closed Principle (OCP) is applied across the design of the system, particularly in the use of abstraction and polymorphism through classes like ‘AbstractEntity’, ‘AIBehaviour’, and interfaces such as ‘Collidable’. This principle states that software entities should be open for extension but closed for modification which can be seen in the The AIManager and AIBehaviour classes illustrate the OCP. The AIManager is open for extension but closed for modification. New behaviours are added by creating new classes that implement the AIBehaviour interface without modifying the AIManager's code. This way, the AIManager can support a wide range of behaviours without needing changes to its existing codebase. These designs promote reusability and flexibility, allowing for easy addition of new features or behaviours with minimal changes to the existing codebase, aligning with the OCP.

Other Features and Game Implementation

New Camera Class:

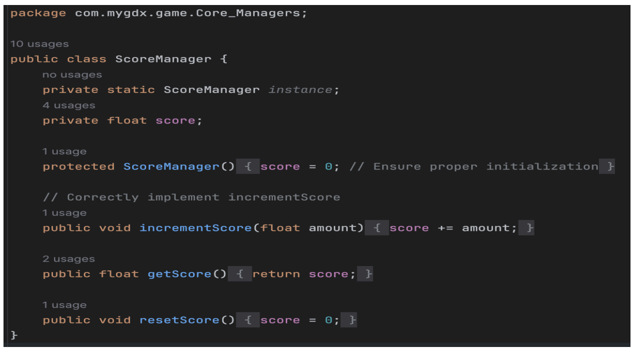


```
package com.mygdx.game.Core_Managers;

import ...

8 usages
public class Camera {
    5 usages
    public OrthographicCamera camera = null;

    2 usages
    private final float SCALE = 1f;

    1 usage
    public Camera(float width, float height)
    {
        camera = new OrthographicCamera();
        camera.setToOrtho( yDown: false,  viewportWidth: width/SCALE,  viewportHeight: height/SCALE);
    }

    no usages
    public void cameraUpdate(float delta, Vector3 position)
    {
        camera.position.set(position);
        camera.update();
    }
}
```

Figure 14: New Camera Class

The Camera class has been designed to encapsulate the game's viewpoint functionalities, utilising LibGDX's OrthographicCamera. Initialised with the game’s dimensions, the camera ensures that the game world is appropriately scaled and rendered. The cameraUpdate function is a critical component of this class, tasked with realigning the camera's position to centre on a designated target within the game environment, typically the player's character. This method not only enhances the dynamic interaction within the game by keeping the player in the viewport's focus but also maintains the integrity of the game's visual representation.

New ScoreManager Class:



```
package com.mygdx.game.Core_Managers;

10 usages
public class ScoreManager {
    no usages
    private static ScoreManager instance;
    4 usages
    private float score;

    1 usage
    protected ScoreManager() { score = 0; // Ensure proper initialization }

    // Correctly implement incrementScore
    1 usage
    public void incrementScore(float amount) { score += amount; }

    2 usages
    public float getScore() { return score; }

    1 usage
    public void resetScore() { score = 0; }
}
```

Figure 15: New ScoreManager Class

The ScoreManager is designed with encapsulation and the Single Responsibility Principle in mind, central concepts of object-oriented programming. It exclusively manages the game's scoring, preventing unintended score changes and enabling a clear focus on score-related functionality. This design facilitates easy updates to scoring rules, like adding multipliers, while keeping the impact of such changes contained within the ScoreManager itself, thus maintaining its modularity and ease of extension in the game's architecture.

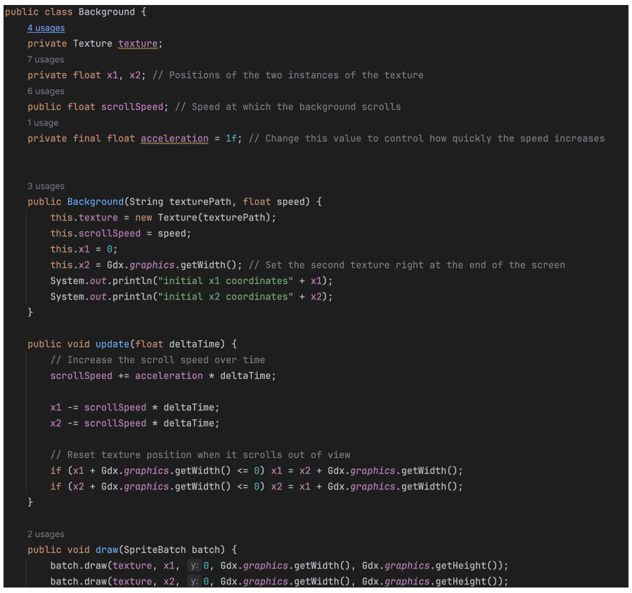
Extended Entity Classes Inheritance/Polymorphism for Game Layer:

The addition of new AI entities, such as CollectibleCarrotEntity, CollectibleCoinEntity, and DangerousObstacleEntity, diversifies the game's challenges and rewards. These entities, categorised under AIEntity, contribute to a richer game environment and player experience by offering a variety of objectives and hazards.

Extended AIBehaviour Classes Inheritance/Polymorphism for Game Layer:

The implementation of a hierarchical AI behaviour structure, with RandomSpawnBehaviour serving as a base for specific behaviours like DangerousObstacleSpawnBehaviour and CollectibleSpawnBehaviour, introduces a more organised and scalable approach to AI development. This hierarchy allows common functionalities to be centralised in the base class while providing flexibility to extend distinct behaviours for different entity types, enhancing the AI's adaptability and efficiency.

Dynamic Scrolling Background Class:



```java
public class Background {
    4 usages
    private Texture texture;
    7 usages
    private float x1, x2; // Positions of the two instances of the texture
    6 usages
    public float scrollSpeed; // Speed at which the background scrolls
    1 usage
    private final float acceleration = 1f; // Change this value to control how quickly the speed increases


    3 usages
    public Background(String texturePath, float speed) {
        this.texture = new Texture(texturePath);
        this.scrollSpeed = speed;
        this.x1 = 0;
        this.x2 = Gdx.graphics.getWidth(); // Set the second texture right at the end of the screen
        System.out.println("initial x1 coordinates" + x1);
        System.out.println("initial x2 coordinates" + x2);
    }

    public void update(float deltaTime) {
        // Increase the scroll speed over time
        scrollSpeed += acceleration * deltaTime;

        x1 -= scrollSpeed * deltaTime;
        x2 -= scrollSpeed * deltaTime;

        // Reset texture position when it scrolls out of view
        if (x1 + Gdx.graphics.getWidth() <= 0) x1 = x2 + Gdx.graphics.getWidth();
        if (x2 + Gdx.graphics.getWidth() <= 0) x2 = x1 + Gdx.graphics.getWidth();
    }

    2 usages
    public void draw(SpriteBatch batch) {
        batch.draw(texture, x1,  y: 0, Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
        batch.draw(texture, x2,  y: 0, Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
```

Figure 16: New Dynamic Background Class

The updated Background class adds a significant visual improvement to our game. It makes the game's background move, creating a feeling of the player entity moving along the background. This feature is particularly useful for making the game world feel more lively and engaging. The class uses two images of the background that move across the screen. When one image moves out of view, it reappears on the other side, creating a continuous scrolling effect. This makes the game's backdrop seem endless and more realistic.

Limitations

There are a few limitations on the game that potentially can be improved in the future. Those limitations will be explained in the section below.

Lack of 'Settings' feature:

The 'Settings' feature has not been integrated into the game for the users to adjust various parameters, such as username, voice and resolution. Without implementing this feature, users can only stick to the preset settings throughout the entire game play.

Users are not able to record their highest score:

Owing to the design structure of the game, the system has not been designed to record down the highest score obtained by the users. Score for each round will be reset to zero after users are in contact with the obstacle. Thus, there is a potential to record down the highest score obtained by all users by connecting the system to the database. This is able to enhance the users experience by comparing their highest score obtained with other players in the community.

Lack of 'Pause' features implemented:

There is no pause button implemented in the game which allows users to halt the game immediately. This is crucial for a single player game to stop any action in the game for various reasons, such as answering phone calls or taking a break from the game. By incorporating a pause button in the game will be able to enhance users' experience by allowing them to have more control on the game flow while preserving their score.

Team Members Participation

Members:	Participation:
Sebastian Nuguid Fernandez	Responsible for leading the team and played an integral role in the design of both the game engine and game.
Yeo Song Chen	Responsible for the design of the game entities
Justin Tan Yong An	Responsible for the design of the collision of entities
Nithiyapriya Ramesh	Responsible for the design of the user input controls
Chin Qun Zhen	Responsible for the design of the scene for the game
Fun Kai Jun	Responsible for the design of player movement for the game