INF1009

Object Oriented Programming Team Project P1 Team 1

By : SEBASTIAN NUGUID FERNANDEZ
YEO SONG CHEN
JUSTIN TAN YONG AN
NITHIYAPRIYA RAMESH
CHIN QUN ZHEN
FUN KAI JUN

Key aspects of the game engine

Scalability

Ensure easy integration of new entities, scenes and audio assets without extensive modification.

Reusability

Use of interfaces and abstract classes to promote code reusability and modularity.

Design

The design strictly follows Object Oriented Programming principles. (SOLID)

Scalability Aspect of Game Engine

Scalability in a game engine refers to the engine's ability to handle growing amounts of work or its potential to accommodate growth in complexity, size, and performance requirements.

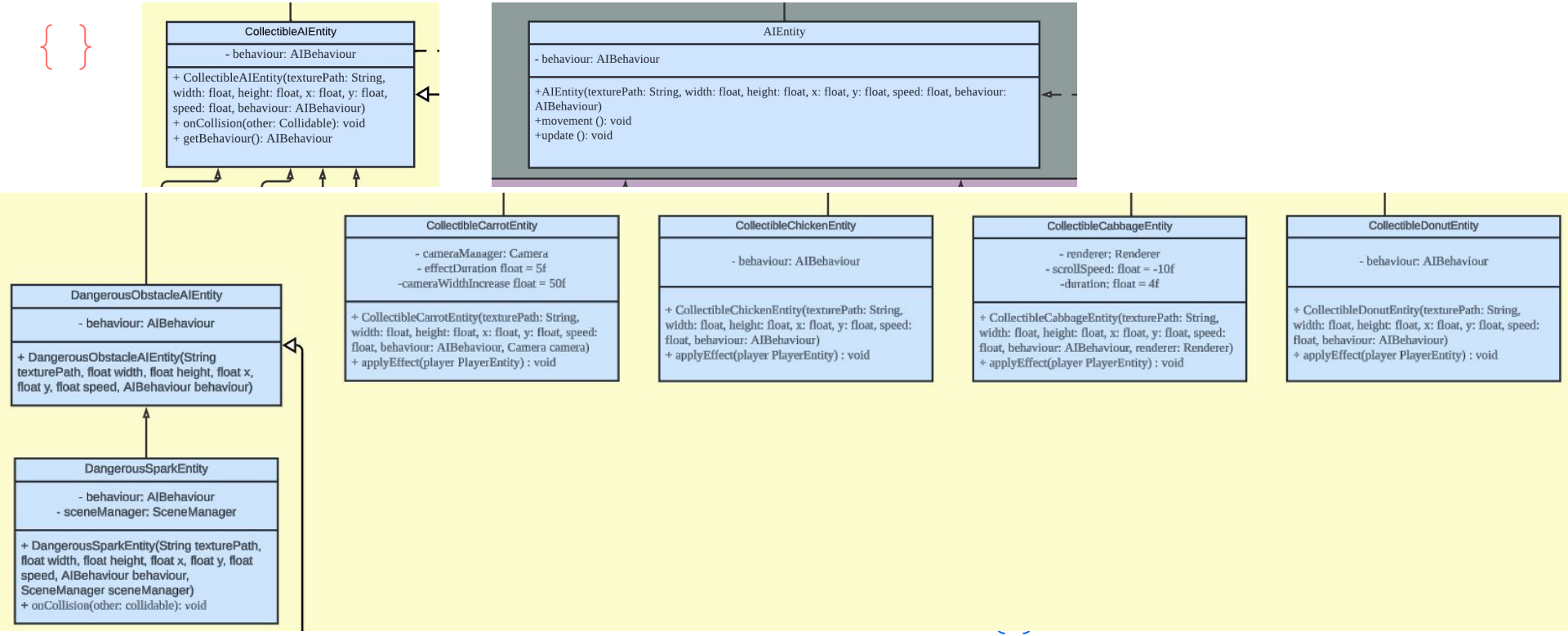
Some examples includes:

New Entities and Content: As a game evolves, there is a need to introduce new characters, items, or obstacles. A scalable game engine allows for the creation and addition of new entities without impacting existing ones.

Game World Expansion: A scalable engine supports enlarging the game world—adding new levels, environments, and worlds—with ease.

Upgrading Visuals and Audio: Updating or enhancing the audio-visual fidelity of the game. A scalable engine will support such enhancements, allowing more detailed textures, complex shaders, or high-fidelity audio without starting from scratch.

Scalability Aspect of Game Engine



Reusability Aspect of Game Engine

Reusability in software development refers to the ability to use parts of the code in different applications or in various parts of the same application without having to significantly rework it

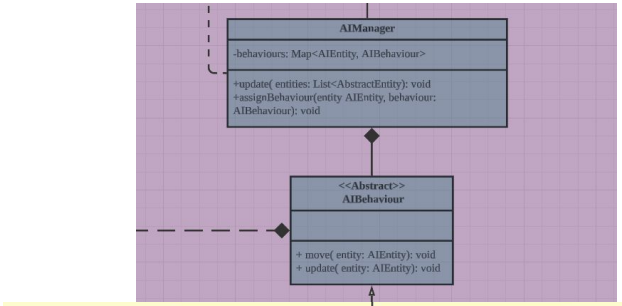
Code Components: Functions, classes, or entire modules designed to perform common tasks can be reused across different projects or different parts of the same game.

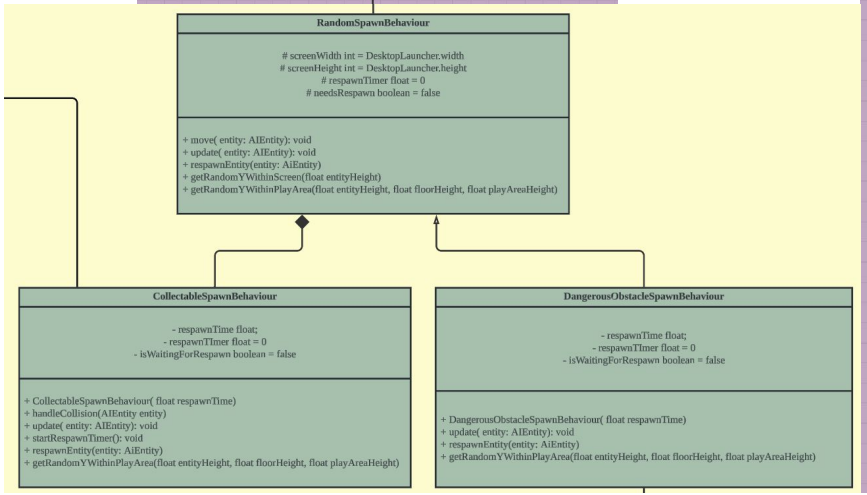
Game Assets: Models, textures, sounds, and animations can often be reused in different levels or even different games, especially when they are generic.

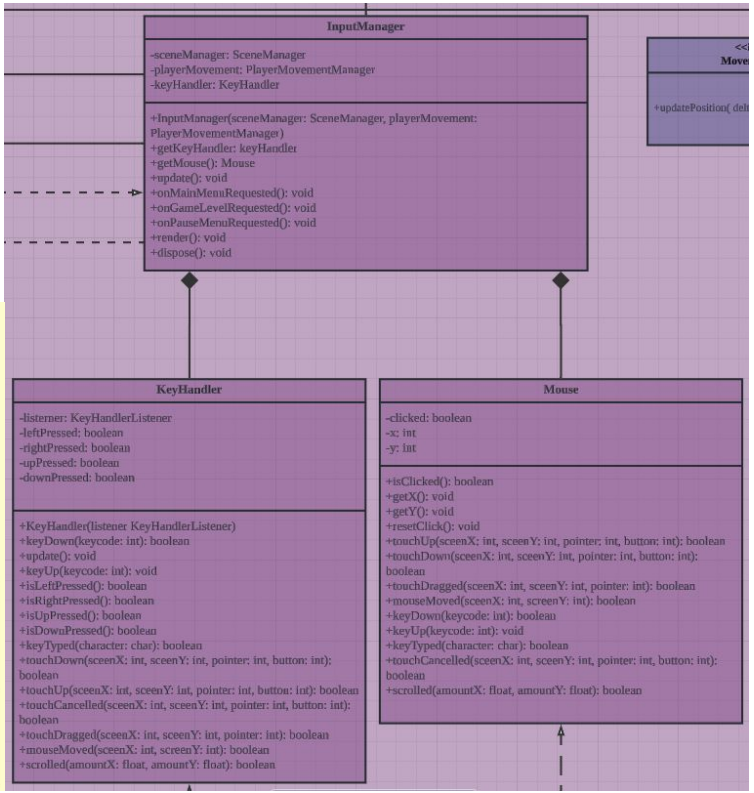
Game Engine Features: Components like physics engines, rendering systems, and input handlers are typically designed to be reusable across different games built with the same engine.

Design Patterns: Common solutions to recurring problems, such as the entity-component system, allow for flexible and reusable architecture within the game's design.

Reusability Aspect of Game Engine



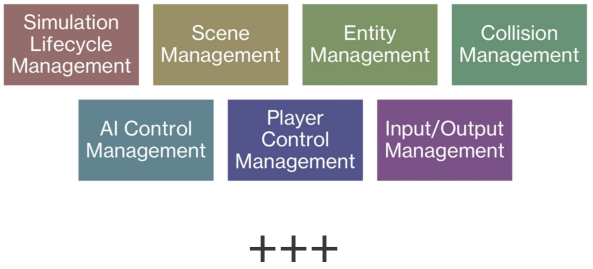




Design Aspect of Game Engine

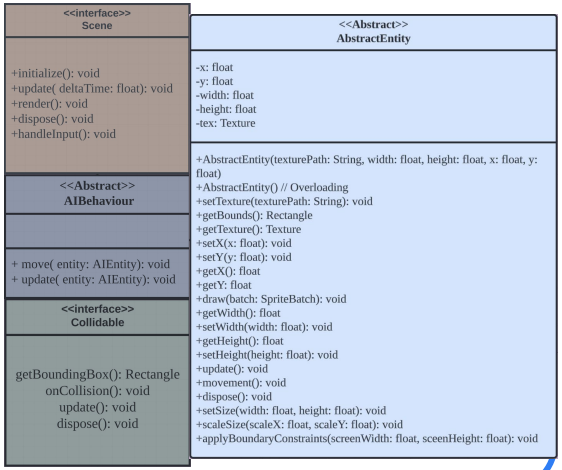
Single Responsibility Principle (SRP)

- Additional dependency Injection for 7 Core Manager
  - Entity Manager-> Entity Factory
  - Scene Manager ->Transition Effect, Scene Factory, Renderer, Background
  - Collision Manager -> Screen Boundary Manager



Open/Closed Principle (OCP)

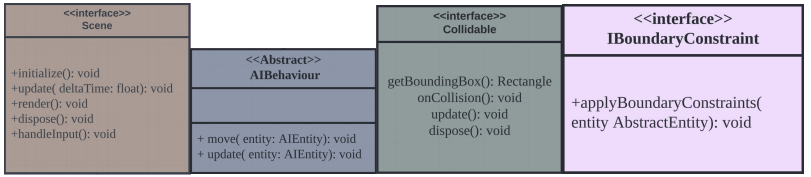
- AbstractEntity, AI Behaviour, ICollidable, Scene ...
- By using a **base class** from which other entities inherit, developers can extend the system with new types of entities, behaviours, scenes without modifying existing code, keeping it open for extension but closed for modification.
  - **Collision Manager**: Interacts with entities through the Collidable interface
  - **Scene Manager**: Interacts with scenes through the Scene interface
  - **AI Manager:** Interacts with AI Entities through Abstract AI Behaviour
  - **Entity Manager:** Interacts with Entities through Abstract Entity



Design Aspect of Game Engine

Interface Segregation (IS)

- **iCollidable**: For implementing Collision
- **iBoundary**: For implementing Boundaries
- **CollectibleEffect**: For implementing collection effects
- **Scene**: For implementing scene

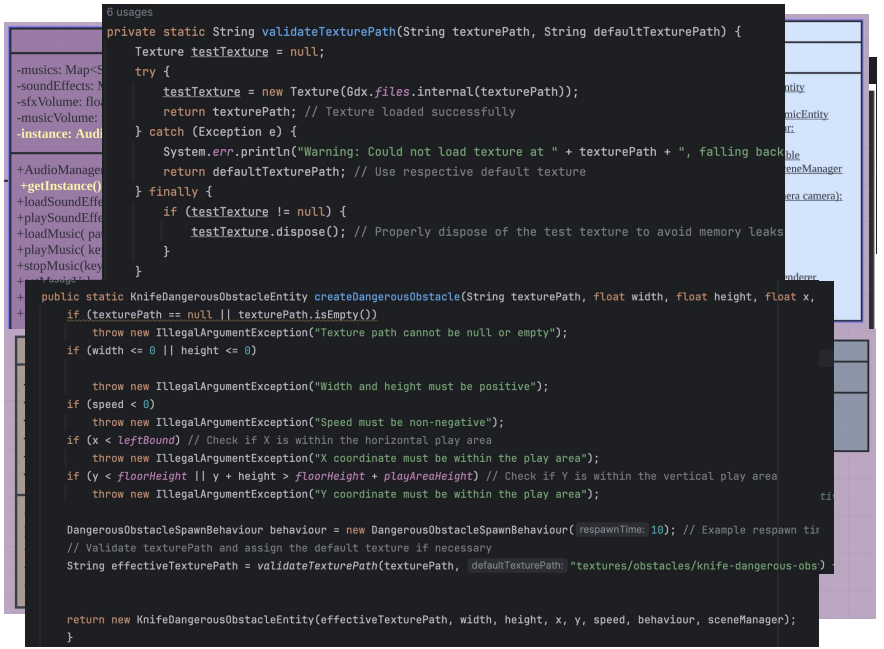


Dependency Inversion Principle (DIP)

- **Entity Manager:** The entity manager is designed to manage entities at a high level without needing to know the specifics of each entity type. It interacts with entities solely through the Abstract Entities Methods
- **Scene Manager:** Manages Scenes similar to Entity Manager's approach

Key Improvements to Game Engine

- Enhanced Design of Managers and Factories:
  - Fails Gracefully with default Assets
- Enhanced Robustness of Managers and Factories:
  - Implemented Stricter Input Validation
  - Incorporated Try and Except catches
- Improved Readability of Game Engine Components
  - Incorporated Documentation and Logs
- Enhanced Packaging and Organisation:
  - Organisation of Assets Improved
  - Organisation of files into feature-based packaging
- Utilised new Design Patterns learned from Module
  - Factory Pattern
  - Strategy
  - Singleton



Design Patterns Implemented

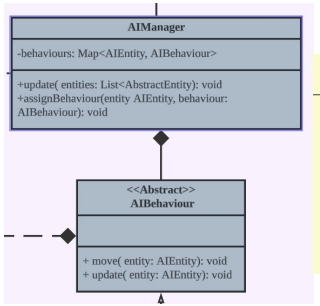
- Factory Patterns:
  - Entity Factory: Responsible for creating various entities within the game, such as characters, objects, or items.
  - Scene Factory: Handles the creation of different scene types and input parameters

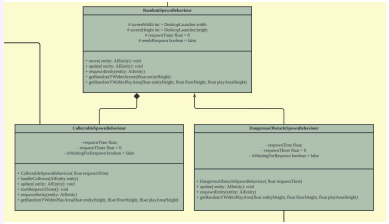
```
1 usage
public static PlayerEntity createPlayerEntity(String texturePath, float width, float height, float x, float y, float speed) {
    if (texturePath == null || texturePath.isEmpty()) // Check if texture path given is empty
        throw new IllegalArgumentException("Texture path cannot be null 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");

    String effectiveTexturePath = validateTexturePath(texturePath, defaultTexturePath: "textures/default/entity-defaults/player-en

    return new PlayerEntity(effectiveTexturePath, width, height, x, y, speed);
}
```

- Singleton Pattern:
  - Audio Manager: Demonstrating the Singleton pattern, the Audio Manager ensures there's only one instance responsible for managing all audio-related tasks, such as playing background music and sound effects.

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

- Strategy Pattern:
  - AI Manager: Utilizing the Strategy pattern, the AI Manager dynamically selects and applies different strategies for non-player character behaviors, enhancing the game's adaptability and challenge.





The Game: Healthy-copter

- Navigate through the kitchen on your trusty helicopter hat and see how far you can reach!
- Collect nutritious delights like **broccoli** and **carrots** for buffs while skillfully dodging dangerous obstacles like **flying knives** and tempting but unhealthy treats such as **fried foods** and **sugary snacks** which debuff the player.
- "Healthy Copter" empowers children to take control of their health while having a blast in the process.

Summary

1. **Architected** on **strong object-oriented programming (OOP) foundations,** emphasizing **scalability**, **reusability**, and **robust design principles.**

2. **Intuitive for developers**, with **clear pathways** for adding new features and content with factories, entity inheritance trees, interfaces, feature-based package organisation, robust managers, input validation and code documentation

3. **Implementation of extra features** such as the introduction and use of the **camera**, dynamic background class and the **failsafe** methods in Entity Factory.

4. Developed an **innovative** an **entertaining** way for kids to learn healthy eating habits