INF1009 - Object Oriented Programming

Student Name, Matriculation No., Email:	Chng Yu Qi Bernice, 2302020, 2302020@sit.singaporetech.edu.sg Yeong Chin Liong, 2303296, 2303296@sit.singaporetech.edu.sg Toh Jun Kuan Johnathan, 2301915, 2301915@sit.singaporetech.edu.sg Ool Jun Kai Jacob, 2301828, 2301828@sit.singaporetech.edu.sg Tan Sheng Le Brendan, 2301782, 2301782@sit.singaporetech.edu.sg
Lab Group:	Lab P8 Team 5
Tutor:	Professor Graham Ng
Assignment Title:	Team Project Part 1

Table of Contents

Overall System Design for Game Engine

Purpose of the different Managers

SimulationLifeCycleManager - A central hub for managing different aspects of the game's lifecycle and interactions between all the managers.
AudioManager - Handles audio-related operations and contains the different audios needed for respective purposes in the game.
AIControlManager - To orchestrate how AI entities move in certain directions and behave during the gameplay.
CollisionManager - Handle collision detection between the player and AI entities within a game, serving a critical function in gameplay mechanics by determining when these entities come into proximity within a predefined range.
EntityManager - To create, manage and update all the entities that will be created during the gameplay.
IOManager - Handles all input and output operations from keyboard and mouse within the game.
PlayerControlManager - Focuses on translating user inputs into actions performed by the player or entities with the game.
SceneManager - To create the different screens required and also manages the transitions of screens made during gameplay.

Implementation of Managers

SimulationLifeCycleManager :
By extending libGDX's `Game` class, it utilizes the framework's structured approach to handle different game states and screens, effectively managing transitions such as between gameplay and menus. This class initializes essential resources like `SpriteBatch` and `BitmapFont`, which are crucial for rendering textures and text, respectively. It also composites several manager classes, which are `SceneManager`, `EntityManager`, `IOManager`, `PlayerControlManager`, `AIControlManager`, `AudioManager`, and `CollisionManager`, each dedicated to managing specific game operations such as screen transitions, entity behaviors, input/output processes, audio, and collision detection. Through its lifecycle methods, `create` for initializing resources, `render` for the game loop execution, and `dispose` for resource cleanup. The `SimulationLifeCycleManager` ensures efficient resource management and game operation. Additionally, the class provides getter methods for accessing its managed resources and managers, facilitating a modular and accessible game architecture that promotes a clean separation of concerns and streamlined development.

AudioManager :
By leveraging libGDX's audio capabilities, it establishes a centralized audio management system, crucial for dynamically adjusting to the game's requirements. This is achieved through the use of a `HashMap` to map background music tracks to specific game scenarios, such as "Gameplay" or "MainMenu", allowing for transitions between audio tracks as the game progresses through different stages. Additionally, the class preloads a sound effect for in-game events such as when two entities collide. The implementation ensures that music and sound effects can be accessed and controlled across the game engine, offering methods for retrieving music based on keys and playing sound effects with a simple interface.

AIControlManager :
The class establishes the groundwork for directional control. Its intended role is to facilitate navigation and movement strategies for AI components, by providing a structured approach to accessing directional information. With the method to retrieve directional information (`getDirections`) allows other parts of the game to interact with the `AIControlManager` to dictate or influence AI movement patterns. As such, the class outlines the basic infrastructure for AI control.

CollisionManager :
The manager operates by first setting a collision range which is a distance threshold for considering entities as colliding. Upon initialization, this range is set to zero and requires explicit assignment to activate collision detection. The core functionality, encapsulated in the `checkForCollision` method, involves iterating through the game's entities to identify the player and subsequently detect any AI entities within the collision range. If such a collision is detected, the `CollisionManager` triggers specific game logic to handle the event, which, in this implementation, results in transitioning to a lose screen, thereby signaling the player's defeat. This process is facilitated through interactions with the `SimulationLifeCycleManager`, which provides access to game entities and other components necessary for managing the game's state and responses to collision events. Additionally, the class includes methods to set and retrieve the collision range, allowing for dynamic adjustments to how sensitively the game responds to the proximity of entities.

EntityManager :
Implemented through an `ArrayList<Entity>`, it facilitates the addition and retrieval of entities, enhancing the game's operational efficiency and maintainability. With functionalities like the `addEntity` method for integrating new entities, `checkClass` and `getEntityID` for specific entity retrieval based on class type or unique identifier, the `EntityManager` is responsible for the management of the game's entities. This

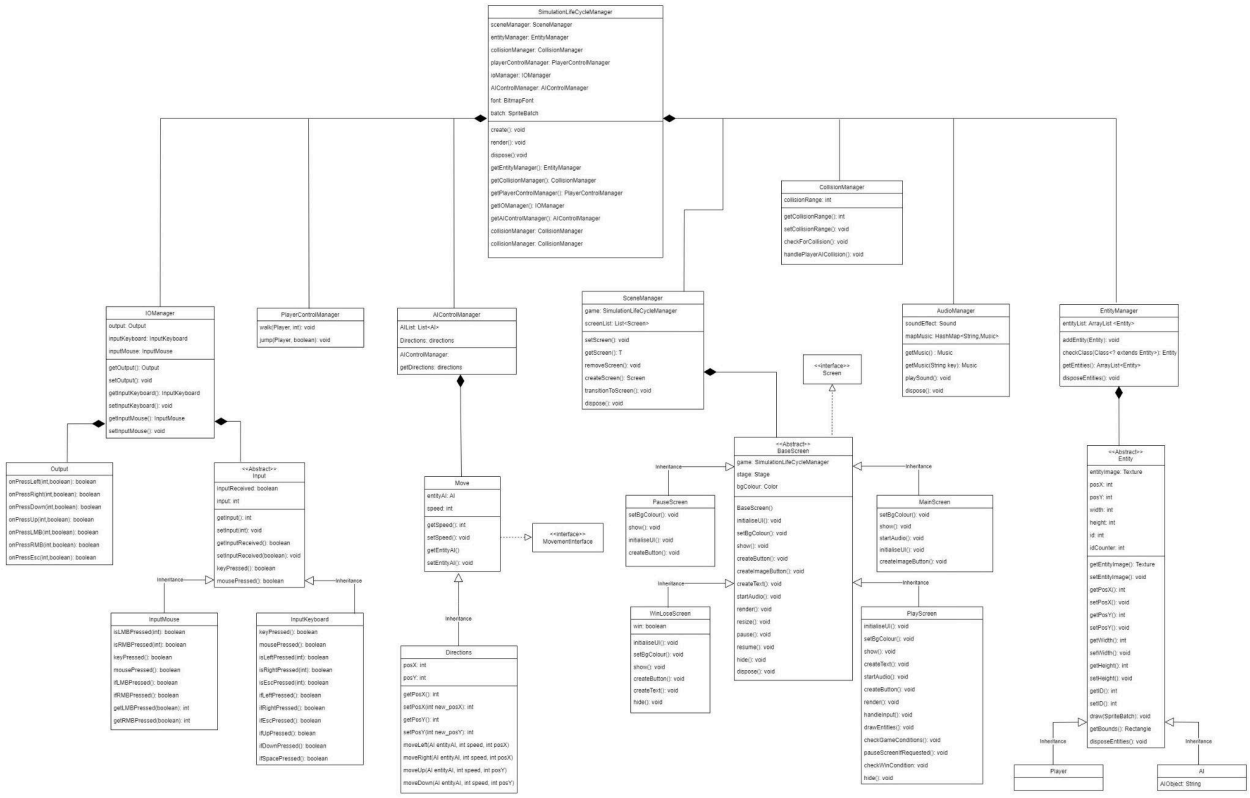
centralized management system not only simplifies the addition and manipulation of entities within the game's ecosystem but also ensures that resources are efficiently utilized and freed when no longer needed, thereby preventing memory leaks and maintaining optimal game performance.

IOManager :
It utilizes listener patterns to detect and process user inputs. When a specific input is detected, such as a key press or mouse click, the IO Manager interprets this action and triggers corresponding events or actions within the game. This could involve moving a character, opening a menu, or any other response to user inputs. The modular nature of the IO Manager allows for easy expansion to support additional input devices or methods, enhancing the game's accessibility and user experience.

PlayerControlManager :
It is responsible for translating processed inputs from the IOManager into actions and movements of the player. Walking is achieved by detecting right or left arrow key inputs to move the player horizontally, whereas jumping is controlled by a boolean flag that dictates vertical movement, simulating a jump or fall using up or down arrow keys. Both actions incorporate `Gdx.graphics.getDeltaTime()` to normalize movement speed across different frame rates. With player control logic encapsulated within its manager, adding new player actions or modifying existing ones becomes more straightforward. This modularity also facilitates code reuse across different parts of the game.

SceneManager :
By maintaining a list of screens, the `SceneManager` allows for efficient management of screen states, ensuring that only the necessary screens are active and available at any given time. This setup facilitates seamless transitions between screens, such as moving from the main menu to gameplay or to a win/loss screen, enhancing the user experience by providing a smooth and logical flow through the game's various stages. The implementation leverages Java's reflective capabilities to create screen instances, adding flexibility in handling a range of screen types without hardcoding their instantiation. Through methods like `setScreen` for changing the current screen, `getScreen` for retrieving an existing screen, and `createScreen` for creating and adding new screens dynamically, the `SceneManager` abstracts the complexity of screen management, making it easier to develop, maintain, and expand the game's user interface and navigational structure.

UML Diagram for Game Engine



A clearer picture of our UML Diagram is on Page 14

Our Game Engine UML comprises the Simulation Lifecycle Manager, Entity Manager, Scene Manager, Input/Output Manager, Player Control Manager and AI Control Manager. As the Simulation Lifecycle Manager's role is to initialize all the other managers, this forms a composition relationship between the Simulation Lifecycle Manager and the other managers because without the Simulation Lifecycle Manager, the other managers will not exist. As the other managers are essential elements of the game, they cannot exist independently without the Simulation Life Cycle Manager, and without these managers, the game will not be able to function effectively as well.

The inheritance relationship is demonstrated in the Input/Output Manager and the SceneManager. Under the Input/Output Manager, the superclass ("Input"), which is also an abstract class, gets the subclasses to create their own implementation of the various abstract methods listed in the superclass. The reason why the "Input" is an abstract class is because for the Inputs, the abstract methods are used to differentiate between a Keyboard input and Mouse input based on the boolean value that is returned by the abstract method. In addition, the manager also forms a composition relationship with the other managers because the Input/Output Manager is responsible for detecting input commands by the user which is essential for the program to process user input. These

inputs will also be passed into the Player Control class to process what kind of response should the program provide once they receive a certain input. For the SceneManager, inheritance will allow the various screens such as ‘PlayScreen’ to reuse similar attributes and methods that are available in the ‘BaseScreen’ class.

The Scene Manager and the Entity Manager forms an aggregation relationship because it is not a requirement for every scene that is created to have an entity in it. For example, in a ‘MainScreen’ class, there will be no Entity-related objects created. This shows that the Scene Manager and Entity Manager can exist independently. ‘BaseScreen’, which has a composition relationship with Scene Manager, is an abstract class because it defines methods for the other screens to implement as they have common behaviors and structures.

For the AI Control Manager, there is a mix of inheritance and dependency relationships between the subclasses. Through the use of parent-child classes as well as interfaces, it creates a unique relationship which the parent-class has to implement ‘Move’ which contains some methods which must be implemented in the parent-class and the child-class will be able to access all of the properties that the superclass holds, including the implementation of the interface.

Object Oriented Programming Principles in Managers

Encapsulation

Encapsulation can be applied by using access modifiers (private, protected, and public) to change the accessibility of each attribute and method of a class. The access modifiers can limit access of attributes/methods that can be accessed by external classes. This promotes security and maintainability of the overall class.

```
public class EntityManager {
    private ArrayList<Entity> entityList;

    public EntityManager() {
        this.entityList = new ArrayList<>();
    }

    public void addEntity(Entity entity) {
        entityList.add(entity);
    }
```

In the EntityManager class above, encapsulation was demonstrated through the use of defining access modifiers such as ‘private’ and ‘public’ to determine the visibility of the

various attributes and methods in the class. The entityList is crucial for the EntityManager to be able to keep track of the entities that are being created. Since every new Entity created will be added into the list, the entityList has to be private so that it prevents any kind of unintended changes made to the list.

```
public abstract class Input {
    private boolean inputReceived;
    private int input;

    // getters and setters
    public int getInput(){
        return this.input;
    }
    public void setInput(int new_input){
        this.input = new_input;
    }
    public boolean getInputReceived(){
        return this.inputReceived;
    }
    public void setInputReceived(boolean new_inputReceived){
        this.inputReceived = new_inputReceived;
    }
}
```

It is also used in the `'Input'` class, where we have private fields for `'input'` and `'inputReceived'`, and public methods to access or modify these fields (i.e. getters and setters). This ensures that the variables `'input'` and `'inputReceived'`cannot be changed directly from outside the class, preserving the integrity of the data.

By doing so, this control over data access helps to prevent unintended side effects and makes the code easier to debug and maintain.

Inheritance

Inheritance promotes a parent-child relationship between two classes. The superclass(parent class) will have a list of attributes and methods which the sub-classes(child class) are able to inherit from, allowing the child classes to have parent-like behaviors even if they are outside of the parent class.

Inheritance is shown through the different Screens that are managed by SceneManager.
The 'BaseScreen' class acts as a base class for different screens in the game. It implements LibGdx's 'Screen' interface which allows for common structure and functionality throughout the different screens.

```
public abstract class BaseScreen implements Screen {

  protected abstract void initialiseUI();

  public void setBgColour(Color colour) {
      this.bgColour = colour;
   }

   protected void createText(String text){
       ...
   }

  //Other functions to used in other screens

}
```

The classes ‘MainScreen’, ‘PauseScreen’, ‘PlayScreen’ and ‘WinLoseScreen’ extend ‘BaseScreen’, inheriting its properties and methods. The following examples shows how the different screens inherit from ‘BaseScreen’

‘MainScreen’ overrides the inherited ‘initialiseUI()’ method to show its own unique interface, while still maintaining common structure with the other screens

```
public class MainScreen extends BaseScreen {
  @Override
  protected void initialiseUI() {
     ...
  }
```

‘PauseScreen’ uses ‘setBgColour()’ method inherited from ‘BaseScreen’

```
public class PauseScreen extends BaseScreen {

  public PauseScreen(SimulationLifeCycleManager game) {
       ...
       setBgColour(Color.TEAL);
   }
}
```

‘PlayScreen’ uses ‘createText()’ method inherited from ‘BaseScreen’

```
@Override
   public void initialiseUI() {
      createText("Play Screen");
   }
```

Another example below shows the ‘InputMouse’ and ‘InputKeyboard’ class inherits from a base ‘Input’ class.

```
public abstract class Input {
  private boolean inputReceived;
  private int input;

   public Input(boolean inputReceived, int input){
       this.inputReceived = inputReceived;
       this.input = input;
   }
   // other common Input methods
}

public class InputKeyboard extends Input {
  // InputKeyboard specific properties and methods
}

public class InputMouse extends Input {
  // InputMouse specific properties and methods
}
```

We can define common properties and behaviors in the ‘Input’ and ‘BaseScreen’ class, and specialized behaviors in the subclasses. This not only reduces code duplication but also facilitates adding new input or screens with minimal changes to the existing codebase.

Abstraction

Abstraction uses abstract classes and interfaces to create a general structure of a class without showing the implementation of abstract methods. It forces the sub-classes to create their own implementation of the abstract methods that were defined in the parent class, reducing complex codes in the abstract class and having unique implementation in the sub-classes.

```
public abstract class Entity{
   protected Texture entityImage;
```

```
    protected int posX; //Y position of entity
    protected int posY; //Y position of entity
    protected int width = 90; // width of entity
    protected int height = 80; // height of entity
    private static int idCounter = 0 ;
    protected int id;
    public Entity(String entityImagePath, int posX, int posY) {
        this.entityImage = new Texture(Gdx.files.internal(entityImagePath));
        this.posX = posX;
        this.posY = posY;
        this.id = idCounter++; // Assign an id to this entity, idCounter++ =
track IDs count
    }
```

The Entity class is defined to be an abstract class so that other classes such as (Player and AI) can inherit the same attributes and methods of the parent class (Entity). Some of the attributes are defined as 'protected' so that it can be accessed by both parent and child subclasses.

```
public abstract class Move implements MovementInterface{
    private int speed;
    private AI entityAI;

    //default constructor
    public Move(){
        this.speed = 0;
        this.entityAI = null;
    }
    public int getSpeed(){
        return this.speed;
    }
    public void setSpeed(int new_speed){
        this.speed = new_speed;
    }
    public AI getEntityAI(){
        return this.entityAI;
    }
    public void setEntityAI(AI newEntityAI){
        this.entityAI = newEntityAI;
    }
}
```

The Move class is using abstraction so that a subclass `'Directions'` and interface `'MovementInterface'` can use its parent class (Move Class) to inherit the same methods and attributes. This specifies what actions are possible without going into

details of how those actions are executed. By implementing an interface, it defines a set of common behaviors that all entities in the game must adhere to.

Polymorphism

Polymorphism consists of method overriding which allows the implementation of a method of a superclass to be overwritten by the child class, allowing the same methods to have different functionality based on the class. Another form of polymorphism is method overloading that allows methods of the same type to have different numbers of parameters. Both methods allow the classes to have code flexibility and different implementation outcomes.

```
public class Player extends  Entity {
   public Player() {
       super("fish.png", 0, 10);
   }
   @Override
   public Rectangle getBounds() {
       return new Rectangle(super.getPosX(), super.getPosY(),
super.getWidth(), super.getHeight());
   }
   @Override
   public void draw(SpriteBatch batch) {
       super.draw(batch);
   }
```

In the above Player Class, polymorphism is achieved using the method overriding. The method `'getBounds()'` is inheriting attributes from a super class. In other words, `'getBounds()'` constructs a new rectangle object using the position and size information retrieved from the superclass (Entity Class) methods such as `'getPosX()'`, `'getPosY()'` etc.

```
//set imgbutton position
   protected void createImageButton(String imagePath, ClickListener
listener, float posX, float posY) {
       Texture buttonTexture = new Texture(Gdx.files.internal(imagePath));
       Drawable drawable = new TextureRegionDrawable(buttonTexture);
       ImageButton.ImageButtonStyle imageButtonStyle = new
ImageButton.ImageButtonStyle();
       imageButtonStyle.up = drawable;
       ImageButton imageButton = new ImageButton(imageButtonStyle);
       imageButton.setPosition(posX, posY);
       imageButton.addListener(listener);
       stage.addActor(imageButton);
```

```
    }

    //set imgbutton size
    protected void createImageButton(String imagePath, ClickListener
listener, float width, float height, float posX, float posY) {
        Texture buttonTexture = new Texture(Gdx.files.internal(imagePath));
        Drawable drawable = new TextureRegionDrawable(buttonTexture);
        ImageButton.ImageButtonStyle imageButtonStyle = new
ImageButton.ImageButtonStyle();
        imageButtonStyle.up = drawable;
        ImageButton imageButton = new ImageButton(imageButtonStyle);
        imageButton.setSize(width, height);
        imageButton.setPosition(posX, posY);
        imageButton.addListener(listener);
        stage.addActor(imageButton);
    }
```

In the above code found in BaseScreen Class, polymorphism is also achieved through the method overloading. The method `'createImageButton()'` is overloaded with two different versions that differ in the number of parameters used. The first method allows the image button to have a specified position using `'posX'` and `'posY'`. The second method extends the first method's functionality by allowing the size to be changed using additional parameters `'width'` and `'height'`.

Reflection on Limitations of Game Engine

Excess Overheads
As there are many manager classes with related classes, this may lead to performance issues such as lapse in receiving output results and increase in memory consumption . The use of polymorphism and inheritance can also lead to slower execution times. As the game grows more complex, there will be an increased overhead of managing the interactions among the classes.

Testing and Debugging
Since objects need to work together to produce a certain outcome, this complicates testing especially when there is a huge number of objects to work with. A huge amount of time will be needed to screen through the classes of codes. Changes made in one class will impact related classes, creating a list of bugs and errors. Debugging can also be a challenge since there are many layers of abstraction and the interactions between objects.

Understanding and Design Complexity
As there are a few principles in OOP, it requires a high-level of understanding of these principles. Without a good understanding, new developers may have a hard time navigating through the classes and cannot implement new ones into the system.

Interdependence
While modularity is beneficial in game engines, it also leads to tight coupling of objects. This makes it hard to modify or scale the system without affecting other parts of the codes. As such, this will lead to an overuse of inheritance or inheritance issues. Especially when it comes to deep inheritance hierarchies, any changes to the superclass will affect subclasses, vice versa. This also makes the codes harder to read and understand.

Team members' contribution

Bernice	Contributed IOManager and PlayerControlManager as well as report writing and slides.
Brendan	Contributed SceneManager and screens as well as report writing and slides
Chin Liong	Contributed CollisionManager, SimulationLifeCycleManager as well as compiling the report and slides.
Johnathan	Contributed EntityManager and AudioManager as well as report writing and slides.
Jacob	Contributed by writing some of the functions in the project as well as the composition of the report and slides.