|  |  |
| --- | --- |
|  |  |
| FINAL PROJECT PLANT VS ZOMBIE GAME |  |
|  |  |
|  | 2/1/2020OBJECT ORIENTED PROGRAMMING |
|  | DR. TRAN THANH TUNGGROUP’S NAME |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | ABSTRACTLorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. | |  |
|  | ACKNOWLEDGEMENT Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.  Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.  Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. | Image |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | GAME CONTENT – NEEDS EDITING |  |  |
|  | HEADING Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. | |  |

### CONTRIBUTION

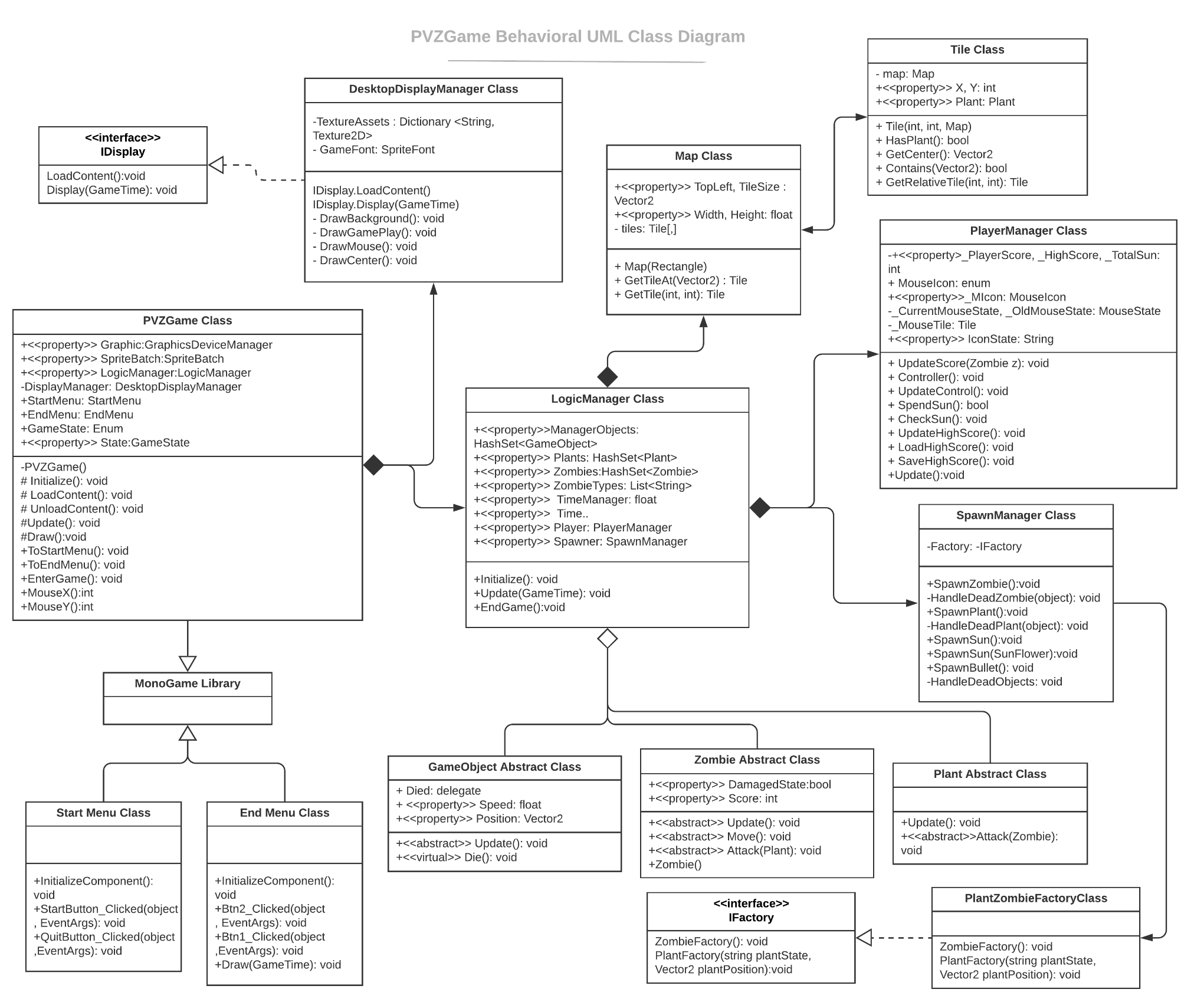
|  |  |
| --- | --- |
| MEMBER | CONTRIBUTION |
| Phạm Huy Hoàng  ITITIU18042 | Build classes: PlayerManager, LaneJumpingZombie  Edit parameters  Make powerpoint |
| Phùng Khánh Linh | Design the game architecture  Build classes: LogicManager, SpawnManager, NormalZombie, PeaShooter, Bullet  Apply design patterns  Write report |
| Lý Bảo Thoại  ITITIU18122 | Design the game graphic and GUI  Build classes: DesktopDisplayManager, Map, Tile  Make powerpoint |
| Trần Đức Trí  ITITIU18132 | Build classes: FlyingZombie, Sun, StartMenu  Write report |
| Nguyễn Hà Văn | Help with designing game graphic  Build classes: SunFlower, CarnivorousPlant, EndMenu  Make powerpoint |

### ARCHITECTURE AND DESIGN

#### In this section, the architecture and design of the game is discussed and reasoned to show the compromise of the team. This section is divided into three parts: behavioral architecture, structural architecture and design patterns used in the game.

## **Behavioral architecture:**

The main efforts while trying to build this behavioral side of architecture was to avoid passing arguments and making bi-directional association between two classes. Such efforts bring our team to the architecture as in the UML diagram below:



PVZGame Class extends from the class Game of MonoGame framework. Since this class must obey the structure required by the framework, it contains both game logical functions such as Initialize function, Load and Unload Content functions, Update function and draw function. As the team trying to compromise with Single Responsibility Principle, we are unable to separate these functions since they are protected. Hence, the team came to the solution of creating two Managers: Logic and Display. These Managers are the components of the PVZGame class. Then, PVZGame will only control the properties that are related to the game state and MonoGame frameworks.

* DisplayManager, or DesktopDisplayManager implements the IDisplay interface and manage the drawing methods of the game. This implementation not only obeys the Single Responsibility Principle, but also provides an openness for extension in other displayers (webs, phones, etc.).   
  Furthermore, reflection by using TextureAssets Dictionary from a String value to a Texture2D value and GetType methods is applied in order to draw the textures of the object without having to create concrete Draw classes or violating Open/Close Principle.
* LogicManager control the game loop and be the composite of most of the game objects. Other objects try to “know” each other by this Manager.

It is worthwhile to mention the SpawnManger class. This class uses a concrete class which implements from the IFactory interface. This step follows the Factory Method design patterns, this pattern will be discussed in detail in the Design Patterns used in the game.   
The PlayerManager class shown in the diagram controls all features related to the players, namely: score, sun (to plant plants), mouse and keyboard controls. This class although controls many aspects of the player, since these aspects are unlikely to extend, it is compromised to put these aspects in to one class for the reduction of complexity.

In the abstract class Game Object, we applied the delegate and event from C# for the die event of objects. The main purpose for this event was that we wish to kill the object without calling any Die methods directly when that objects meet our wished requirements:

public delegate void DieDelegate(object self);

public abstract class GameObject

{

public event DieDelegate Died;

public abstract void Update();

public virtual void Die() // all objects must be in the form of this die function

{

if (Died != null)

Died(this);

}

}

// this is when we need to call the Die method

public abstract class PlantZombieObject:GameObject

{

public Tile ObjectTile {get; set;}

public float Health {get; set;} = 100;

public override void Update()

{

if (Health <= 0)

Die();

}

public abstract void Damaged(float dam);

}

public class SpawnManager

{

private IFactory \_Factory = new PlantZombieFactory();

public void SpawnZombie()

{

Zombie z = \_Factory.ZombieFactory();

if (z != null)

{

z.Died += HandleDeadZombie; //The event will add the HandleDeadZombie and execute this methods after the Zombie’s die method.

PVZGame.Game.LogicManager.ManagedObjects.Add(z);

PVZGame.Game.LogicManager.Zombies.Add(z);

}

}

private void HandleDeadZombie(object self)

{

PVZGame.Game.LogicManager.ManagedObjects.Remove((GameObject)self);

PVZGame.Game.LogicManager.Zombies.Remove((Zombie)self);

PVZGame.Game.LogicManager.Player.UpdateScore((Zombie)self);

}

Delegate is similar to the function pointer in C++ as it used to pass methods as arguments to other methods and can be chained together. As we only need delegate to handle the die event only, delegate is a simple alternative way to Command Design Pattern.

## **Structural architecture:**

In this section, only the GameObject Abstract Class and its subclasses are discussed. The structural architecture is built as in the diagram below:

A close up of a piece of paper

Description automatically generated

Since there are concrete methods that need implementing in the abstract classes in order to control what should be handle in the subclasses, the usage of abstract classes rather than interface was a reasonable choice. Furthermore, if we chose to use the interfaces as the alternative to the abstract classes, there would be more unnecessary complexity to the structure.

The ObjectTile in the PlantZombieObject was used by the Plant and Zombie objects of the game. Such implementation of the Tile violates the initial regulation of the game: All objects aggregated in Logic Manager must not know each other directly but through the manager. Nevertheless, this only violation during the whole game creates an easy-to-use implementation to conduct the interactions between Plant and Zombie objects. For instance, this is the application in the PeaShooter class:

private Zombie MeetZombie()

{

foreach (var z in PVZGame.Game.LogicManager.Zombies)

{

if (ObjectTile.Y == z.ObjectTile.Y && ObjectTile.X <= z.ObjectTile.X && z.ObjectTile.X <= ObjectTile.X + 5)

return z;

}

return null;

}

## **Design Patterns:**

The two design patterns were to be implemented in the game are Singleton Pattern and Factory Method.

1. *Singleton Pattern:*

Since there should be only one instance that control the running loop of the whole game, the Singleton Pattern was applied to the PVZGame class to create one and only static instance Game:

private PVZGame()

{

Graphic = new GraphicsDeviceManager(this);

Content.RootDirectory = "Content";

}

public static readonly PVZGame Game = new PVZGame();

1. *Factory Method:*

The second design pattern to be applied into the game was Factory Method. The Factory Method was implemented as in the diagram below:

A screenshot of a cell phone

Description automatically generated

This design was to follow the Single Responsibility Principle, when we wish to handle the spawn object and the logic of the spawn methods separately. Another reasonable idea when this design was implemented is the logic of spawning plants and zombies were hidden from other classes. Hence, if anyone wish to adjust the spawning logic, they do not have to mind the SpawnManager Class. Furthermore, the IFactory interface allows other developers to add new Factory methods as they wish to Spawn other objects, for instance, decorators or new types of sun or bullet.

Although the Plant Factory and the Zombie Factory should be in separate classes, this implementation reduces the program’s complexity, and does not violate the Single Responsibility Principle since the actor is still only handling the Factory methods.

However, there is one weakness that the team found out from the Factory Methods Pattern in our game: developers still need to add new line of codes whenever they add new types of Plants or Zombies. This weakness should be mended in the future.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Lorem Ipsum is simply dummy text of the printing and typesetting industry. |
|  |