FINAL PROJECT

PLANT VS ZOMBIE GAME

2/1/2020
OBJECT ORIENTED PROGRAMMING
DR. TRAN THANH TUNG
GROUP'S NAME

ABSTRACT

This report presents the detailed development and implementation of simple Plant vs Zombie game in endless mode. This project bases on the application of object-oriented programming. The Plant vs Zombie game consists of graphical user interface, self-designed textures, game play components, implemented using C#.

ACKNOWLEDGEMENT

First, we are very grateful for this game project which helps us to have an opportunity to learn more about object-oriented programming and to work as a team.
We are sincerely thankful to Dr. Tran Thanh Tung, for his instruction and his effort in facilitating and helping us in this project in particular and in object-oriented programming subject in general.
Finally, we would like to thank anyone who might not mentioned but have given us great support during our game project. This opportunity with the amazing team has given us unforgettable memory.

GAME CONTENT

In the game, player will become a hero who receives an S.O.S message of the people from the past and back to the ancient time to protect them from many dangerous and calamitous species. He uses the power of nature to rescue the people and accomplish the mission.

GAME RULE

- Defense your garden from zombie's attack.
- Use money from sun to buy plant fighting against zombie
- Zombies appear regularly and more powerful after time.

*** *Plant:***

Sunflower	Pea Shooter	Carnivorous Plant
- Health: 100 - Sun cost: - Sun production: 1	- Health: 100 - Sun cost: - Damage: 20 - Range: straight, 4 tiles	- Health: 100 - Sun cost: 150 - Damage: 100 - Range: 1 tile - Firing rate:
+ Produce extra sun after a period	+ Shoot peas to attack zombies + Cannot be planted near the border of the yard (4 tiles)	+ Eat the first zombie that gets close (even flying zombie) + Need time to digest

*** *Zombie:***

Normal Zombie	Flying Zombie	Lane Jumping Zombie
- Speed: - Health: 100 - Damage:	- Speed: - Health: - Damage:	- Speed: - Health: 100 - Damage:
	+ Dodge all peas until passing over the first plant he meets	+ Change lane while moving

*** *Controller:***

Keyboard	Mouse
- A: choose Sunflower - S: choose Pea Shooter - C: choose Carnivorous Plant	- Right Click: turn back to the mouse - Left Click: collect sun and plant plants

I. CONTRIBUTION

MEMBER	CONTRIBUTION
Phạm Huy Hoàng ITITIU18042	Build classes: BackgroundMusicManager, PlayerManager, LaneJumpingZombie Edit parameters Make powerpoint
Phùng Khánh Linh ITITIU18073	Design the game architecture Build classes: LogicManager, SpawnManager, NormalZombie, PeaShooter, Bullet and other abstract classes 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 ITITUN17030	Help with designing game graphic Build classes: SunFlower, CarnivorousPlant, EndMenu Make powerpoint

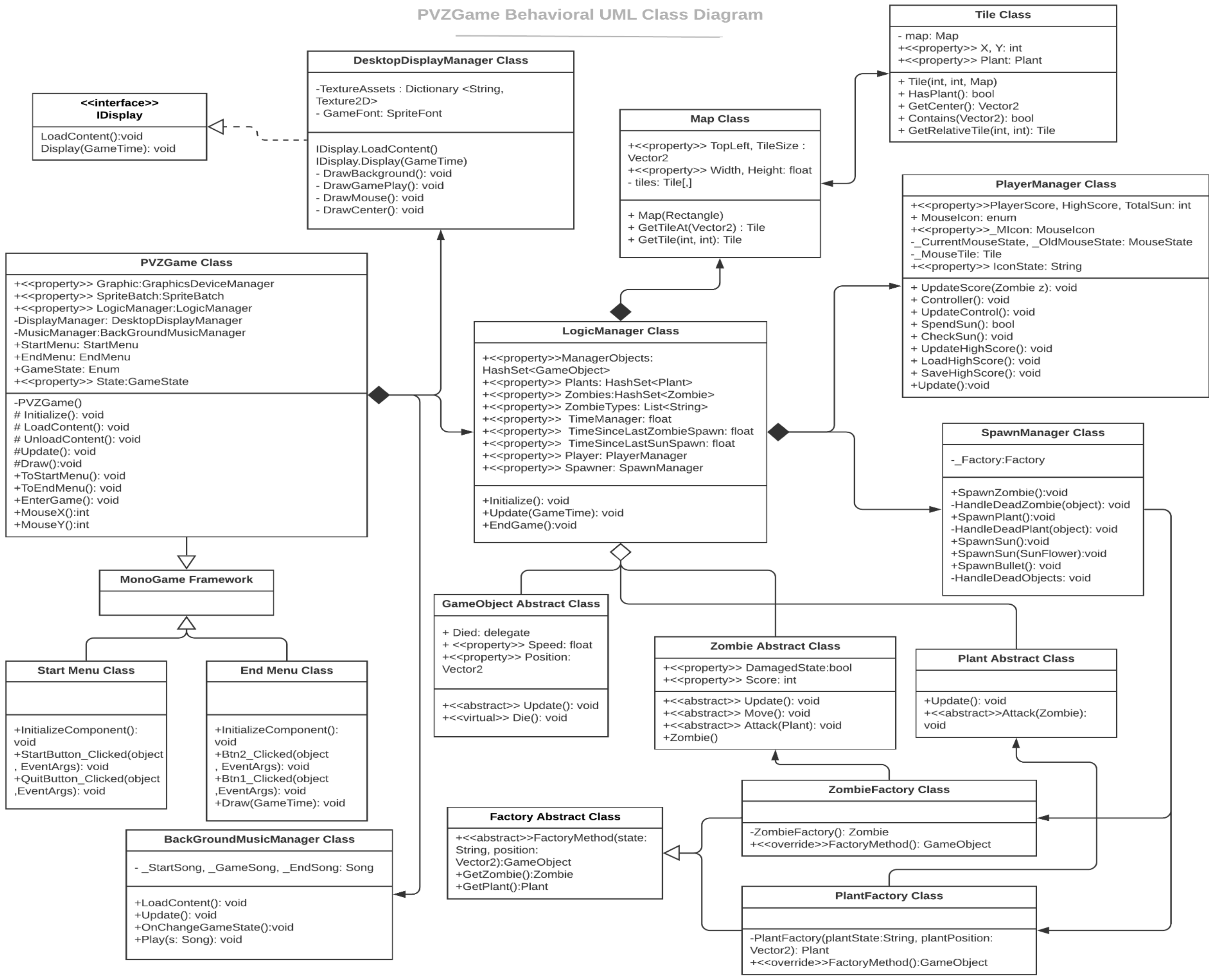
II. ARCHITECTURE AND DESIGN

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.

1. 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 Behavioral UML Class Diagram


Tile Class
- map: Map
+<<property>> X, Y: int
+<<property>> Plant: Plant
+ Tile(int, int, Map)
+ HasPlant(): bool
+ GetCenter(): Vector2
+ Contains(Vector2): bool
+ GetRelativeTile(int, int): Tile
DesktopDisplayManager Class
-TextureAssets : Dictionary <String, Texture2D>
- GameFont: SpriteFont
IDisplay.LoadContent()
IDisplay.Display(GameTime)
- DrawBackground(): void
- DrawGamePlay(): void
- DrawMouse(): void
- DrawCenter(): void
<<interface>> IDisplay
LoadContent():void
Display(GameTime): void
Map Class
+<<property>> TopLeft, TileSize : Vector2
+<<property>> Width, Height: float
- tiles: Tile[,]
+ Map(Rectangle)
+ GetTileAt(Vector2) : Tile
+ GetTile(int, int): Tile
PlayerManager Class
+<<property>>PlayerScore, HighScore, TotalSun: int
+ MouseIcon: enum
+<<property>>_MIcon: MouseIcon
-_CurrentMouseState, _OldMouseState: MouseState
-_MouseTile: Tile
+<<property>> IconState: String
+ UpdateScore(Zombie z): void
+ Controller(): void
+ UpdateControl(): void
+ SpendSun(): bool
+ CheckSun(): void
+ UpdateHighScore(): void
+ LoadHighScore(): void
+ SaveHighScore(): void
+Update():void
PVZGame Class
+<<property>> Graphic:GraphicsDeviceManager
+<<property>> SpriteBatch:SpriteBatch
+<<property>> LogicManager:LogicManager
-DisplayManager: DesktopDisplayManager
-MusicManager:BackGroundMusicManager
+StartMenu: StartMenu
+EndMenu: EndMenu
+GameState: Enum
+<<property>> State:GameState
-PVZGame()
# Initialize(): void
# LoadContent(): void
# UnloadContent(): void
#Update(): void
#Draw():void
+ToStartMenu(): void
+ToEndMenu(): void
+EnterGame(): void
+MouseX():int
+MouseY():int
LogicManager Class
+<<property>>ManagerObjects: HashSet<GameObject>
+<<property>> Plants: HashSet<Plant>
+<<property>> Zombies:HashSet<Zombie>
+<<property>> ZombieTypes: List<String>
+<<property>> TimeManager: float
+<<property>> TimeSinceLastZombieSpawn: float
+<<property>> TimeSinceLastSunSpawn: float
+<<property>> Player: PlayerManager
+<<property>> Spawner: SpawnManager
+Initialize(): void
+Update(GameTime): void
+EndGame():void
SpawnManager Class
-_Factory:Factory
+SpawnZombie():void
-HandleDeadZombie(object): void
+SpawnPlant():void
-HandleDeadPlant(object): void
+SpawnSun():void
+SpawnSun(SunFlower):void
+SpawnBullet(): void
-HandleDeadObjects: void
MonoGame Framework
GameObject Abstract Class
+ Died: delegate
+ <<property>> Speed: float
+<<property>> Position: Vector2
+<<abstract>> Update(): void
+<<virtual>> Die(): void
Zombie Abstract Class
+<<property>> DamagedState:bool
+<<property>> Score: int
+<<abstract>> Update(): void
+<<abstract>> Move(): void
+<<abstract>> Attack(Plant): void
+Zombie()
Plant Abstract Class
+Update(): void
+<<abstract>>Attack(Zombie): void
Start Menu Class
+InitializeComponent(): void
+StartButton_Clicked(object , EventArgs): void
+QuitButton_Clicked(object ,EventArgs): void
End Menu Class
+InitializeComponent(): void
+Btn2_Clicked(object , EventArgs): void
+Btn1_Clicked(object ,EventArgs): void
+Draw(GameTime): void
ZombieFactory Class
-ZombieFactory(): Zombie
+<<override>>FactoryMethod(): GameObject
Factory Abstract Class
+<<abstract>>FactoryMethod(state: String, position: Vector2):GameObject
+GetZombie():Zombie
+GetPlant():Plant
BackGroundMusicManager Class
- _StartSong, _GameSong, _EndSong: Song
+LoadContent(): void
+Update(): void
+OnChangeGameState():void
+Play(s: Song): void
PlantFactory Class
-PlantFactory(plantState:String, plantPosition: Vector2): Plant
+<<override>>FactoryMethod():GameObject

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 three Managers: Logic, Display and BackgroundMusic. 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:

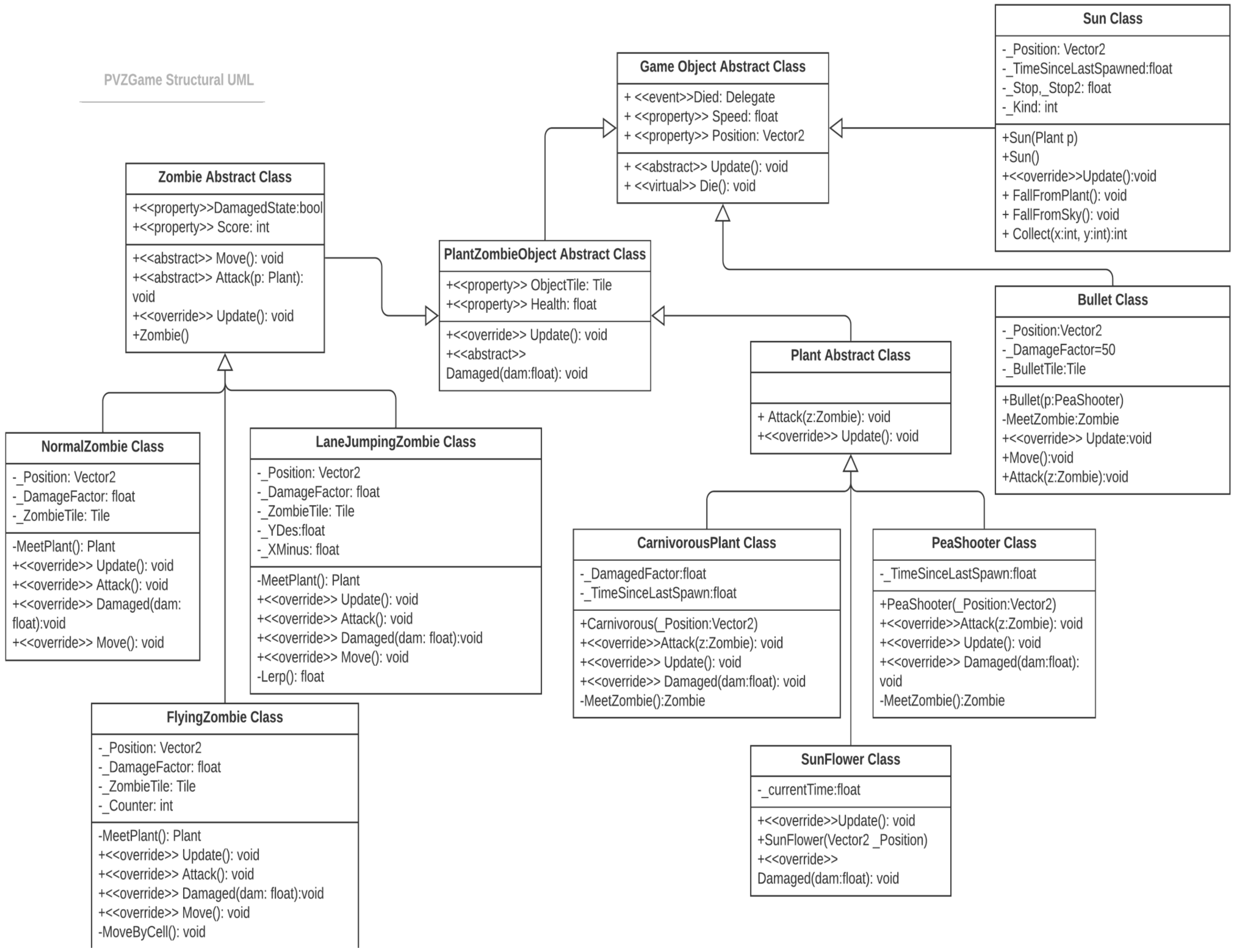
```
public delegate void DieDelegate(object self);

    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 Factory _Factory;
        public void SpawnZombie()
        {
            _Factory = new ZombieFactory();
            Zombie z = _Factory.GetZombie();
            if (z != null)
            {
                z.Died += HandleDeadZombie;
                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.

2. 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:


PVZGame Structural UML
Game Object Abstract Class
+ <<event>>Died: Delegate
+ <<property>> Speed: float
+ <<property>> Position: Vector2
+ <<abstract>> Update(): void
+ <<virtual>> Die(): void
Sun Class
-_Position: Vector2
-_TimeSinceLastSpawned:float
-_Stop,_Stop2: float
-_Kind: int
+Sun(Plant p)
+Sun()
+<<override>>Update():void
+ FallFromPlant(): void
+ FallFromSky(): void
+ Collect(x:int, y:int):int
Zombie Abstract Class
+<<property>>DamagedState:bool
+<<property>> Score: int
+<<abstract>> Move(): void
+<<abstract>> Attack(p: Plant): void
+<<override>> Update(): void
+Zombie()
PlantZombieObject Abstract Class
+<<property>> ObjectTile: Tile
+<<property>> Health: float
+<<override>> Update(): void
+<<abstract>> Damaged(dam:float): void
Bullet Class
-_Position:Vector2
-_DamageFactor=50
-_BulletTile:Tile
+Bullet(p:PeaShooter)
-MeetZombie:Zombie
+<<override>> Update:void
+Move():void
+Attack(z:Zombie):void
Plant Abstract Class
+ Attack(z:Zombie): void
+<<override>> Update(): void
NormalZombie Class
-_Position: Vector2
-_DamageFactor: float
-_ZombieTile: Tile
-MeetPlant(): Plant
+<<override>> Update(): void
+<<override>> Attack(): void
+<<override>> Damaged(dam: float):void
+<<override>> Move(): void
LaneJumpingZombie Class
-_Position: Vector2
-_DamageFactor: float
-_ZombieTile: Tile
-_YDes:float
-_XMinus: float
-MeetPlant(): Plant
+<<override>> Update(): void
+<<override>> Attack(): void
+<<override>> Damaged(dam: float):void
+<<override>> Move(): void
-Lerp(): float
CarnivorousPlant Class
-_DamagedFactor:float
-_TimeSinceLastSpawn:float
+Carnivorous(_Position:Vector2)
+<<override>>Attack(z:Zombie): void
+<<override>> Update(): void
+<<override>> Damaged(dam:float): void
-MeetZombie():Zombie
PeaShooter Class
-_TimeSinceLastSpawn:float
+PeaShooter(_Position:Vector2)
+<<override>>Attack(z:Zombie): void
+<<override>> Update(): void
+<<override>> Damaged(dam:float): void
-MeetZombie():Zombie
FlyingZombie Class
-_Position: Vector2
-_DamageFactor: float
-_ZombieTile: Tile
-_Counter: int
-MeetPlant(): Plant
+<<override>> Update(): void
+<<override>> Attack(): void
+<<override>> Damaged(dam: float):void
+<<override>> Move(): void
-MoveByCell(): void
SunFlower Class
-_currentTime:float
+<<override>>Update(): void
+SunFlower(Vector2 _Position)
+<<override>> Damaged(dam:float): void

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;

}
```

3. Design Patterns:

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

a. *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();
```

b. *Factory Method Pattern:*

The second design pattern to be applied into the game was Factory Method. By following the structure from the book “Design Patterns – Elements of Reusable Object-Oriented Software” (Figure a), the Factory Method was implemented as in the diagram in the Figure b:

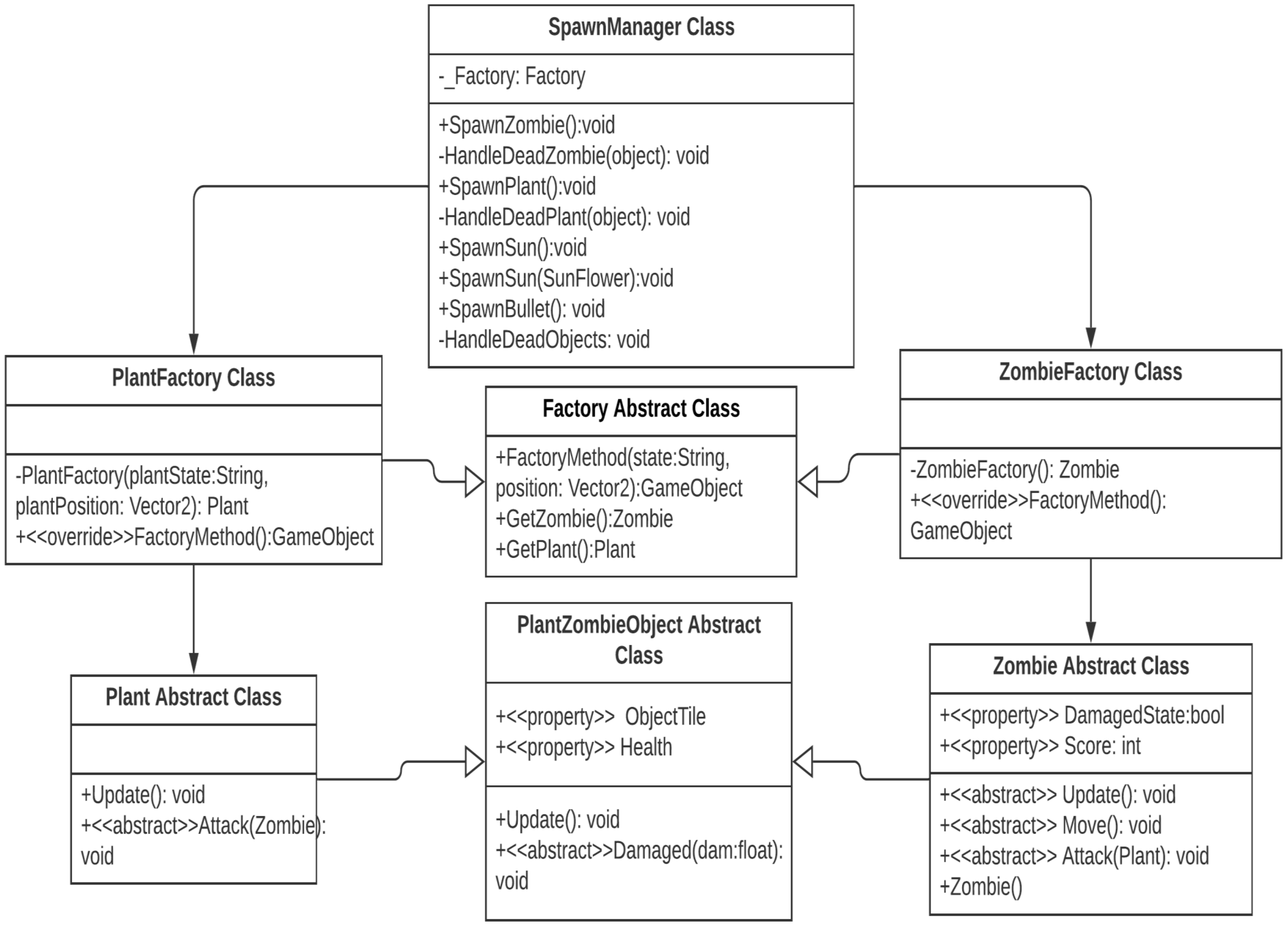

SpawnManager Class
-_Factory: Factory
+SpawnZombie():void
-HandleDeadZombie(object): void
+SpawnPlant():void
-HandleDeadPlant(object): void
+SpawnSun():void
+SpawnSun(SunFlower):void
+SpawnBullet(): void
-HandleDeadObjects: void
PlantFactory Class
-PlantFactory(plantState:String, plantPosition: Vector2): Plant
+<<override>>FactoryMethod():GameObject
Factory Abstract Class
+FactoryMethod(state:String, position: Vector2):GameObject
+GetZombie():Zombie
+GetPlant():Plant
ZombieFactory Class
-ZombieFactory(): Zombie
+<<override>>FactoryMethod(): GameObject
Plant Abstract Class
+Update(): void
+<<abstract>>Attack(Zombie): void
PlantZombieObject Abstract Class
+<<property>> ObjectTile
+<<property>> Health
+Update(): void
+<<abstract>>Damaged(dam:float): void
Zombie Abstract Class
+<<property>> DamagedState:bool
+<<property>> Score: int
+<<abstract>> Update(): void
+<<abstract>> Move(): void
+<<abstract>> Attack(Plant): void
+Zombie()

Figure a

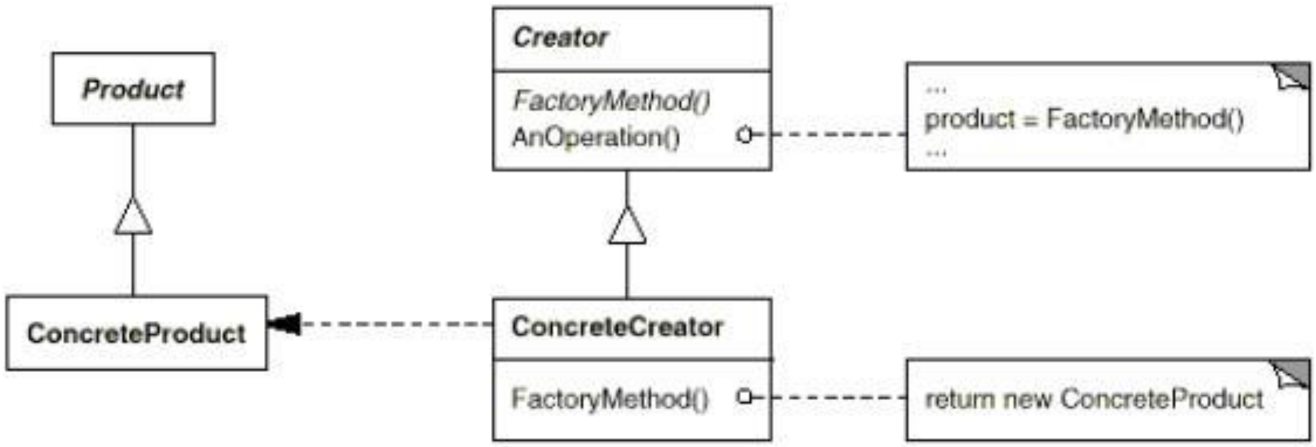


Figure b

This design was to let the subclasses decide what object to be created and 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 Factory abstract class 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.
Participants by mapping the classes in the game to the classes in the structure:

- Creator: Factory Abstract Class
- ConcreteCreator: ZombieFactory and PlantFactory
- Product: Zombie Abstract Class and Plant Abstract Class
- ConcreteProduct: Concrete classes that extend from the abstract classes of Zombie and Plant

The fragility appears in these lines of codes:

```
public abstract class Factory
    {
        public abstract GameObject FactoryMethod(String state, Vector2 position);

        public Zombie GetZombie()
        {
            return (Zombie)FactoryMethod(null, Vector2.Zero);
        }

        public Plant GetPlant(string plantState, Vector2 plantPosition)
        {
            return (Plant)FactoryMethod(plantState, plantPosition);
        }
}

    private Factory _Factory;
    public void SpawnZombie()
    {
        _Factory = new ZombieFactory();
        Zombie z = _Factory.GetZombie();
        if (z != null)
        {
            z.Died += HandleDeadZombie;
            PVZGame.Game.LogicManager.ManagedObjects.Add(z);
            PVZGame.Game.LogicManager.Zombies.Add(z);
        }

    }
```

For each Spawn of Zombie and Plant, the _Factory variable must be reconstructed but someone may call the GetPlant method in the SpawnZombie method and create errors. This weakness should be mended in the future.

III. SPECIFIC WORKS:

Name	Class	Idea	Implementation	
			Variable	Method
Pham Huy Hoang	Lane Jumping Zombie	Create a Zombie type that can jump between the lanes. This type of Zombie can switch between the neighboring lane randomly.	- A private Vector2 **_Position** to store the position of the zombie. - A private float **_DamageFactor** to store the value of the damage the zombie deals to the plant. - A Tile **_ZombieTile** to store the tile where this zombie is on. - A random integer generator **rand**. - A float **yDes** to store the y destination the zombie will go. - A float **xMinus** to store the x distance the zombie has travelled. * Notes: The variables **rand, yDes, xMinus** will be used in Move() method.	- Method **MeetPlant()** to check if the zombie meets a plant. - Method **Update()** to decide whether the zombie will attack the plant or will move. This will use the method **MeetPlant()**; - Method **Move()** to perform the movement of the zombie. When the zombie has reached to the destination tile and the horizontal distance the zombie has travelled is two times greater than horizontal size Tile. The zombie will randomly move to the next relative tile. This method uses the **Lerp()** method to move to another lane. - Method **Lerp()** (linear interpolation), this method returns the value between two float values.
	Player Manager	Create a class to manage the things that related to the user. This class include the game score, the game high score, the mouse icon, the total sun.	- A MouseIcon **MIcon** to store the current mouse icon. - A MouseStste **_CurrentMouseState** to store the current mouse state. - A MouseState **_OldMouseState** to store the previous mouse state. - A Tile **_MouseTile** to get the Tile of the mouse corresponds to the position of the mouse. - An int **TotalSun** to proceed and store the total Sun. - An int **PlayerScore** to proceed and store the game score. - An Int **HighScore** to proceed and store the game high score. - A String **IconState** to proceed and store the state of the **mouseIcon**. **- Enumeration: MouseIcon** to store the collection the mouse icon.	- Method **UpdateScore(Zombie z)** to increase the _PlayerScore which correspond to the Zombie type. - Method **Controller()** set the **MIcon** and correspond **IconState;** - Method **SpendSun()** to check if the **TotalSun** can spend. - Method **UpdateControl()** to update controller of the mouse. - Method **CheckSun()** to increase the **TotalSun** for each sun collected; - Method **UpdateHighScore()** to update new **HighScore** base on the **PlayerScore;** - Method **LoadHighScore()** to initialize the **HighScore** from file *highscore.txt*. - Method **SaveHighScore()** to save the **HighScore** to the file *highscore.txt*. - Method **Update()** to call all Update method.

	Backgro-und Music Manager	Any game should have background music and this game is not an exception. Therefore, we want to play background sound depend on the state of the game.	3 Variables **_MenuSong**, **_GameSong**, **_EndSong** of type **Song**.	- Method **LoadContent()** to load the content from files. - Method **Update()** to play the background music depends on game state. - Method **OnChangeGameState()** to stop the previous song. - Method **Play(Song s)** to play song **s**.

Name	Class	Idea	Implementation	
			Variable	Method
Phung Khanh Linh	NormalZombie	Zombie move normally	- A private Vector2 **_Position** to store the position of the zombie. - A private float **_DamageFactor** to store the value of the damage the zombie deals to the plant. - A Tile **_ZombieTile** to store the tile where this zombie is on.	- Method **MeetPlant()** to check if the zombie meets a plant. - Method **Update()** to decide whether the zombie will attack the plant or will move. This will use the method **MeetPlant()**. When the zombie meets a plant, the variable **_counter** changes to 1 and it's DamagedState changing to true. - Method **Move()** to perform the basic movement of the zombie.

	Pea Shooter	PeaShooter check within its range of shooting, if there is a zombie in that range, it invokes the spawning of bullet	- A private **_TimeSinceLastSpawn** to invoke the spawning of bullet in a specific amount of time	- Method **Update()** to decide the plant attack zombie or not
				`public override void Update()`
				`{`
				
				`   base.Update();`
				`   if (MeetZombie() != null)`
				`   {`
				`       _TimeSinceLastSpawn`
				`+=(float)PVZGame.Game.CurrentGameTime.ElapsedGameTi`
				`me.TotalSeconds;`
				`       if (_TimeSinceLastSpawn >= 5f)`
				`       {`
				`          Attack(null);`
				`          _TimeSinceLastSpawn = 0f;`
				`       }`
				`    }`
				`}`
				
				- Method **MeetZombie()** to check if there is a zombie in its shooting range
				`private Zombie MeetZombie()`
				`{`
				
				`   foreach (var z in`
				`PVZGame.Game.LogicManager.Zombies)`
				`   {`
				`        if(z.ObjectTile != null )`

				`if (ObjectTile.Y == z.ObjectTile.Y && ObjectTile.X < z.ObjectTile.X && z.ObjectTile.X <= ObjectTile.X + 5)`
				`return z;`
				`}`
				`return null; }`
				
				-Method **Attack()** to invoke spawning of bullet
				`public override void Attack(Zombie z)`
				`{`
				`PVZGame.Game.LogicManager.Spawner.SpawnBullet(this);`
				`}`

	Bullet	Damage the zombie when invoked from the PeaShooter	- A private Vector2 **_Position** to store the position of the bullet.
			- A private float **_DamageFactor** to store the value of the damage the plant deals to the plant.
			- A Tile **_BulletTile** to store the tile where this bullet is on.
				-The bullet was born from the position of the PeaShooter
				`public Bullet(PeaShooter p)`
				`{`
				`_Position = p.Position;`
				`this.Position = _Position;`
				`_BulletTile = PVZGame.Game.LogicManager.GameMap.GetTileAt(_Position);`
				`Speed = 0.8f;`
				`}`
				-**MeetZombie()** method returns the zombie if the distance from the bullet to the zombie is smaller or equal the half the width of the tile
				`private Zombie MeetZombie()`
				`{`
				`foreach (var z in PVZGame.Game.LogicManager.Zombies)`
				`{`
				`if (Vector2.Distance(z.Position, Position) <= PVZGame.Game.LogicManager.GameMap.TileSize.X * 0.5)`
				`return z;`
				`}`
				`return null;`
				`}`
				
				After the bullet attack the zombie, the bullet will die.

	SpawnManager	Explained previous	Explained previous	Explained previous

Name	Class	Idea	Implementation	
			Variable	Method
Tran Duc Tri	Flying Zombie	Create a Zombie type that can fly. This type of Zombie cannot receive damage until it passes a plant.	- A private Vector2 **_Position** to store the position of the zombie. - A private float **_DamageFactor** to store the value of the damage the zombie deals to the plant. - A Tile **_ZombieTile** to store the tile where this zombie is on. - A private int **_counter** (the initial value is 0) to check the state of the zombie (whether it passes a plant). * Notes: The variables **_counter** will be used in **Update()** method.	- Method **MeetPlant()** to check if the zombie meets a plant. - Method **Update()** to decide whether the zombie will attack the plant or will move. This will use the method **MeetPlant()**. When the zombie meets a plant, the variable **_counter** changes to 1 and it's DamagedState changing to true. - Method **Move()** to perform the basic movement of the zombie. - Method **MoveByCell()** to perform the movement of the zombie when it meets the first plant. When the zombie meets the first plant, it will fly to the next relative tile.
	Sun	Create sun class which provides player with a tool to manage their planting.	- A private Vector2 **_Position** to store the position of the sun. - A private float **_TimeSinceLastSpawned = 0f** to instantiate the time since the sun is spawned to 0f. - A private int **_Kind** which has two value 0 and 1. Value 0 is the type of sun falling from plants and value 1 is the type of sun falling from the sky. - Two private float **_Stop** and **_Stop2** to set the stop position of the sun from both plants and sky when falling.	- This class has two constructors that one will receive the a plant as an argument and other with empty argument. - Method **Update()** to check if the sun spawned is what kind then using two methods **FallFromPlant()** and **FallFromSky()** corresponding to that kind and this method will make the sun die after a time. - Method **FallFromSky()** and **FallFromPlant()** to perform the basic movement of the sun. - Method **Collect()** to check if the sun is collected or not and return the value of the sun.

Name	Class	Idea	Implementation	
			Variable	Method
Nguyen Ha Van	Carniv -orous Plant	Create an eater-zombie plant which will eat zombie in front of it and it will pending every 5 seconds between each attack.	- A private Vector2 **_Position** to store the position of the plant. - A private float **_DamageFactor** to store the value of the damage the plant deals to the zombie. - A private float **_TimeSinceLastSpawn** to set the time between each attack. - A private **Tile_EatTile** to check the position where the plant can attack the zombie.	- Method **Update()** wil set the time pending between each attack. - Method **Attack()** will make the carnivorous plant attack the zombie. - Method **Damaged()** it will minus the zombie health.
	Sun Flower	Create sun flower that spawning sun each 10 seconds	- private float **_currentTime** to set the current time of the game	- Method **Update()**: to make the sun flower spawn sun each 10 seconds of the game - Method **Damaged()**: to minus health to the sun flower if it get hit by a zombie

Name	Class	Idea	Implementation	
			Variable	Method

Ly Bao Thoai	Map	To manage the lanes' and cells' positions, we created a class name "Map". It uses the given windows dimensions to calculate the positions of the cells to pixels.
		As the trees planted, we wanted to have a class that can store a reference to a plant in order to stop the player from planting another tree in that slot. We named it "Tile" and each tile corresponds to a single cell in the board.
		The map will contain the tiles, these two classes will work together. The map will calculate which tiles at a given pixel and a tile will calculate its position using the map.
			- A two-dimensional array of the tiles.
			- Position of the top left corner of the first cell.
			- public float **Width, Height**: The sizes of the tiles in the map (all tiles have the same sizes).
			- The width and height of the board.
				- Method **GetTileAt()** returns the tile at a given position in pixels.
				`public Tile GetTileAt(Vector2 position)`
				`        {`
				`            Vector2 relativePos =`
				`Vector2.Subtract(position, TopLeft);`
				`            if (relativePos.X < 0 ||`
				`relativePos.X >= Width ||`
				`               relativePos.Y < 0 ||`
				`relativePos.Y >= Height)`
				`                return null;`
				`            int x = (int)(relativePos.X /`
				`TileSize.X);`
				`            int y = (int)(relativePos.Y /`
				`TileSize.Y);`
				`            return tiles[x, y];`
				`        }`
				- Method **GetTile()** returns the tile at a given coordinate of the tiles grid.
				`public Tile GetTile(int x, int y)`
				`        {`
				`            if (x < 0 || x >= tiles.GetLength(0)`
				`|| y < 0 || y >= tiles.GetLength(1))`
				`                return null;`
				`            return tiles[x, y];`
				`        }`

	Tile		- A reference to the map - It's x and y coordinate with respect to the map (integer) - A reference to a plant	- Method **GetCenter()** returns the pixels coordinate of its center. - Method **Contains()** checks if it contains a position in pixels. - Method **GetRelativeTile()** returns a relative tile with an offset. - Method **HasPlant()** returns the plant it stores.

GUI:

Name	Class	Idea	Implementation
Ly Bao Thoai	Drawing & Updating	We want a start menu and an end menu. To do this, we use an extension called *UI.Forms* in Monogame. UI.Forms helps creating the buttons and drawing the buttons to the screen. We decided to have 2 classes named “Start Menu” and “End Menu”, each class extends *ControlManager* from the UI.Forms extension, and the main game will use these two classes. To draw and update the gameplay and the menus properly, the game needs to have another variable called "state" to indicate if the game is in menu or is playing for the updater to update and for the drawer to draw the appropriate part of the game.	- Both 2 classes have an override method named Update() and Draw(). - Those methods will be called by the updater and the drawer.
Tran Duc Tri	Start Menu		StartMenu class contains 2 buttons for playing and exiting the game. - Play Button: Calls the EnterGame method from PVZGame - Exit Button: Calls the Exit method from PVZGame
Nguyen Ha Van	End Menu		EndMenu class contains 2 buttons for replaying and exiting the game. - Play Again Button: Calls the EnterGame method from PVZGame - Exit Button: Calls the Exit method from PVZGame

IV. CONCLUSION AND FUTURE WORKS

Conclusion

This Plant vs Zombie game was runnable. The code is not in the highest efficiency and it may not be perfect as still violating some design principles of object-oriented programming but those problems were a good challenge to solve. Coding the game has demonstrated the utility of many aspects in object-oriented programming.

Project Status and Future Works

The game's basic objects are still lack of animation. Therefore, we are going to develop some kinds of motion for the plants, zombies,... to make the game more lively. Besides, the implementation could be improved to avoid violating some design principles. Furthermore, as we wish to develop a diversity in the game mode and its characters, the team will improve the game at our best if we had any chance.

V. REFERENCES

1. Monogame:Retrieved from http://www.monogame.net/
2. Monogame.UI.Forms: Retrieve from https://www.youtube.com/watch?v=N9whx5Cozog&fbclid=IwAR13ngZqzZFFr8ioVXb7yAqeMf7PirzFhjZOZt7ejivADcLsv9i1pJbf-a8
3. XNA Game Studio 4.0 : Retrieved from https://docs.microsoft.com/en-us/previous-versions/windows/xna/bb200104(v=xnagamestudio.41)?redirectedfrom=MSDN
4. C# documentation : Retrieve from https://docs.microsoft.com/en-us/dotnet/csharp/
5. Visual Studio: Retrieve from https://visualstudio.microsoft.com/
6. Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (2016). Design patterns: elements of reusable object-oriented software. Boston, MA: Addison-Wesley.