Design Overview for Cosmic Defense

Name: Nguyen Tran Quang Minh
Student ID: 104179687

Summary of Program

In this custom program, I would like to create a custom game based on the popular arcade video game in the 1980s called "Space Invaders". Space Invaders is a classic arcade game where players control a spaceship moving horizontally at the bottom of the screen, shooting at descending rows of pixelated aliens. The objective is to eliminate all the invading aliens while avoiding their counterattacks and dodging obstacles.

I will use SplashKit as the library to draw the game. First, my job is to construct a game background with a title, player's lives, score, and timer. Secondly, I need to create a player as a spaceship and four types of enemies including 3 kinds of aliens that will shoot and approach the player steadily, the UFO that flies around above. Finally, is the signature of Space Invaders, the stationary defense bunkers, in this custom game I will make it different from the original game in that the player will lose scores if hitting the bunkers. I believe that the game's challenging part is creating the logic for enemies' moves, shooting, and collisions.

Required Roles

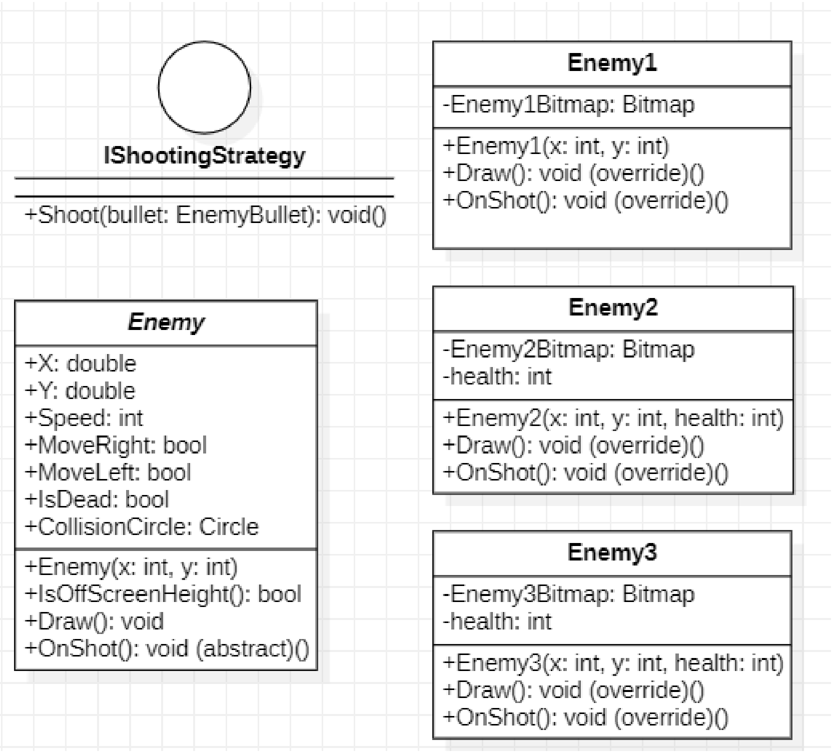
I believe that my program is complex and well-designed due to its effective use of object-oriented principles and design patterns. The program's modular structure and class hierarchy promote code reusability and maintainability. The Strategy pattern allows different enemy types to have unique shooting behaviors, while the Template Method pattern enables the `Game` class to define the overall game flow while allowing individual components to customize specific steps. This design approach results in a flexible and extensible codebase, making it easier to add new features or modify existing ones. Overall, the program demonstrates a thoughtful design that enhances maintainability and scalability.

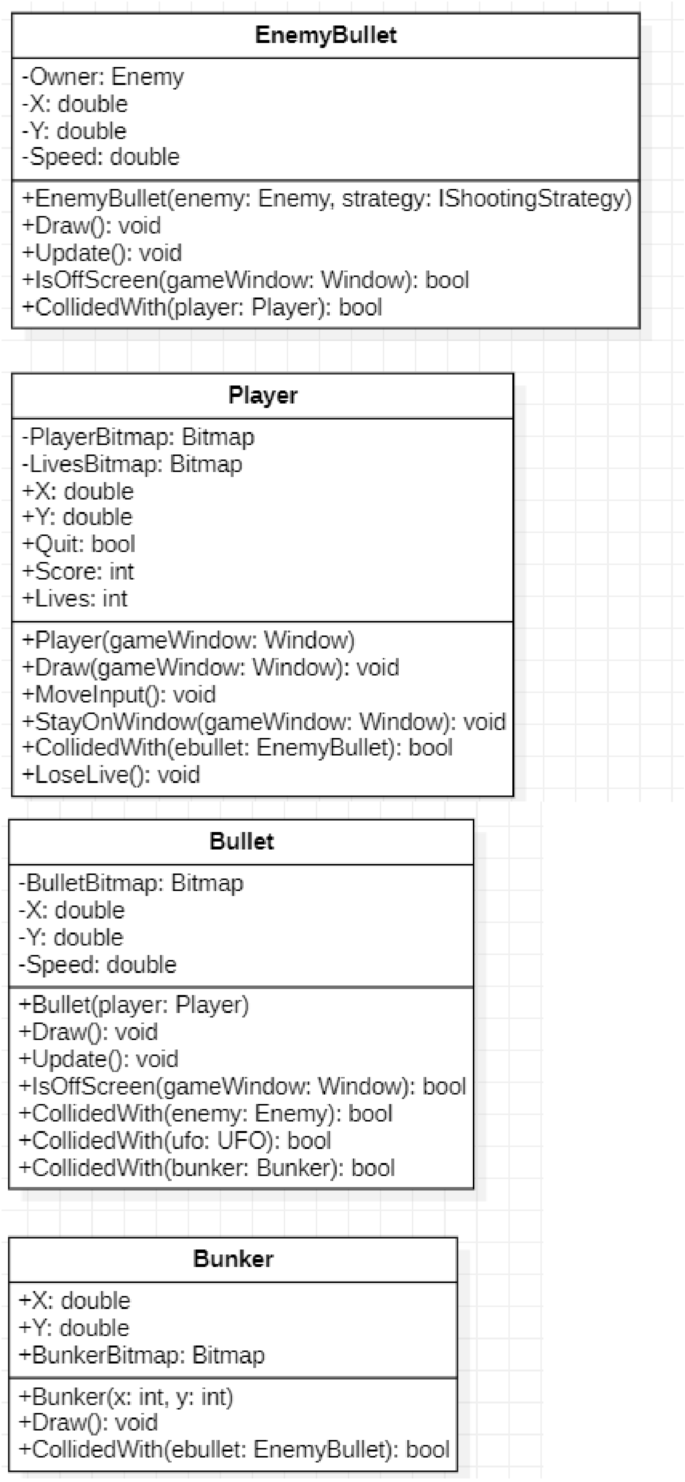
Class	**Responsibilities**	**Type Details and Notes**
Enemy	- Represent enemy objects in the game - Manage the position and movement of enemies - Track health and state of enemies - Provide abstract draw method - Check if enemies are off the screen vertically - Handle when enemies are shot	- Class - Properties: X, Y, MoveRight, MoveLeft, IsDead - Abstract Draw() method - Abstract OnShot() method
Player	- Represent the player character in the game	- Class
- Properties: X, Y, Lives, Score
-

	- Manage the position and movement of the player - Track player's lives and score - Maintain a list of bullets - Draw the player on the screen - Handle player movement and firing input - Check collisions with enemy bullets - Reduce lives when hit by enemy bullets	List<Bullet> for player bullets - Draw() method - MoveInput() method - CollidedWith(EnemyBullet) method - LoseLive() method
EnemyBullet	- Represent alien bullets fired by enemies - Manage position and movement of bullets - Define shooting strategy for bullets - Update bullet position based on strategy - Draw bullets on the screen - Check if bullets are off the screen - Check collision with enemies	- Class - Properties: X, Y, Speed - IShootingStrategy for shooting strategy - Update() method - Draw() method - IsOffScreen(Window) method - Rectangle for collision detection
Bullet	- Represent player bullets fired by the player - Manage position and movement of bullets - Update bullet position based on movement speed - Draw bullets on the screen - Check if bullets are off the screen - Check collision with enemies	- Class - Properties: X, Y, Speed - Update() method - Draw() method - IsOffScreen(Window) method - CollidedWith(Enemy) method
Bunker	- Represent cave blocks that protect the player - Manage position and size of bunkers - Draw bunkers on the screen - Check collision with enemy bullets	- Class - Properties: X, Y, Width, Height - Draw() method - CollidedWith(EnemyBullet) method
Game	- Manage game state and elements	- Class

	- Handle game window and timer - Update and draw game elements - Handle player input and collisions - Display game over and win messages - Manage game reset and quit states	- Properties: GameWindow (Window), myTimer (Timer) - Player (Player), Enemies (List<Enemy>) - UFOs (List<UFO>), Bunkers (List<Bunker>) - Update() method - Draw() method - MoveInput() method - CheckCollision() method - GameOverEffect() and WinEffect() methods - Reset and Quit flags
IShootingStrategy	- Interface for shooting strategy used by enemy bullets - Define the Shoot method for bullet behavior	- Interface - Method: Shoot(EnemyBullet)

Class Diagram


IShootingStrategy
+Shoot(bullet: EnemyBullet): void()
Enemy1
-Enemy1Bitmap: Bitmap
+Enemy1(x: int, y: int)
+Draw(): void (override)()
+OnShot(): void (override)()
Enemy
+X: double
+Y: double
+Speed: int
+MoveRight: bool
+MoveLeft: bool
+IsDead: bool
+CollisionCircle: Circle
+Enemy(x: int, y: int)
+IsOffScreenHeight(): bool
+Draw(): void
+OnShot(): void (abstract)()
Enemy2
-Enemy2Bitmap: Bitmap
-health: int
+Enemy2(x: int, y: int, health: int)
+Draw(): void (override)()
+OnShot(): void (override)()
Enemy3
-Enemy3Bitmap: Bitmap
-health: int
+Enemy3(x: int, y: int, health: int)
+Draw(): void (override)()
+OnShot(): void (override)()



EnemyBullet

-Owner: Enemy
-X: double
-Y: double
-Speed: double

+EnemyBullet(enemy: Enemy, strategy: IShootingStrategy)
+Draw(): void
+Update(): void
+IsOffScreen(gameWindow: Window): bool
+CollidedWith(player: Player): bool

Player

-PlayerBitmap: Bitmap
-LivesBitmap: Bitmap
+X: double
+Y: double
+Quit: bool
+Score: int
+Lives: int

+Player(gameWindow: Window)
+Draw(gameWindow: Window): void
+MoveInput(): void
+StayOnWindow(gameWindow: Window): void
+CollidedWith(ebullet: EnemyBullet): bool
+LoseLive(): void

Bullet

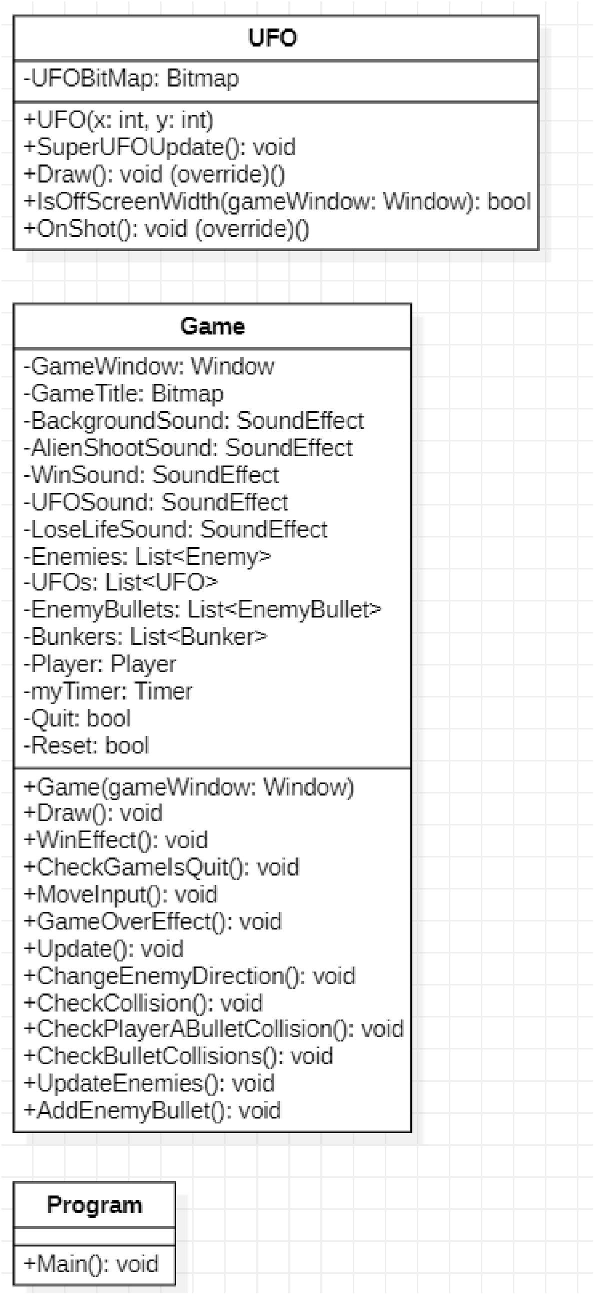
-BulletBitmap: Bitmap
-X: double
-Y: double
-Speed: double

+Bullet(player: Player)
+Draw(): void
+Update(): void
+IsOffScreen(gameWindow: Window): bool
+CollidedWith(enemy: Enemy): bool
+CollidedWith(ufo: UFO): bool
+CollidedWith(bunker: Bunker): bool

Bunker

+X: double
+Y: double
+BunkerBitmap: Bitmap

+Bunker(x: int, y: int)
+Draw(): void
+CollidedWith(ebullet: EnemyBullet): bool



UFO

-UFOBitMap: Bitmap

+UFO(x: int, y: int)
+SuperUFOUpdate(): void
+Draw(): void (override)()
+IsOffScreenWidth(gameWindow: Window): bool
+OnShot(): void (override)()

Game

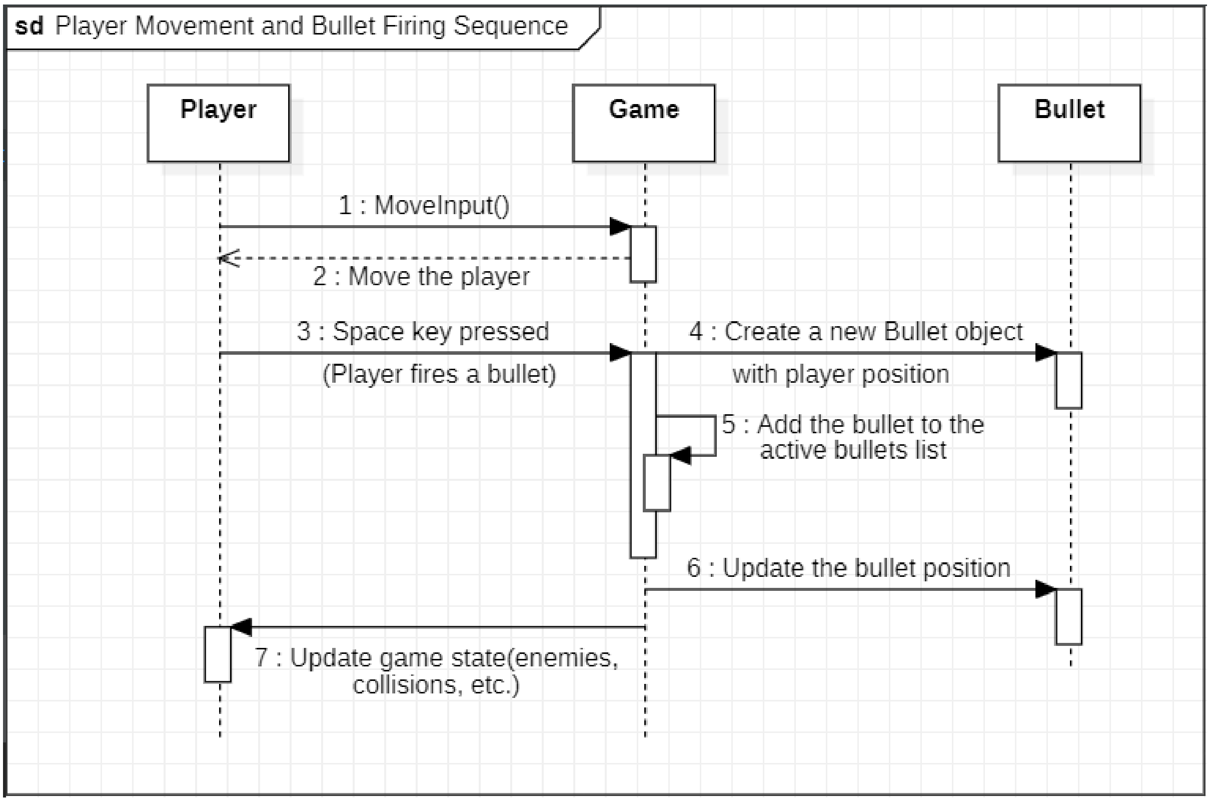
-GameWindow: Window
-GameTitle: Bitmap
-BackgroundSound: SoundEffect
-AlienShootSound: SoundEffect
-WinSound: SoundEffect
-UFOSound: SoundEffect
-LoseLifeSound: SoundEffect
-Enemies: List<Enemy>
-UFOs: List<UFO>
-EnemyBullets: List<EnemyBullet>
-Bunkers: List<Bunker>
-Player: Player
-myTimer: Timer
-Quit: bool
-Reset: bool

+Game(gameWindow: Window)
+Draw(): void
+WinEffect(): void
+CheckGameIsQuit(): void
+MoveInput(): void
+GameOverEffect(): void
+Update(): void
+ChangeEnemyDirection(): void
+CheckCollision(): void
+CheckPlayerABulletCollision(): void
+CheckBulletCollisions(): void
+UpdateEnemies(): void
+AddEnemyBullet(): void

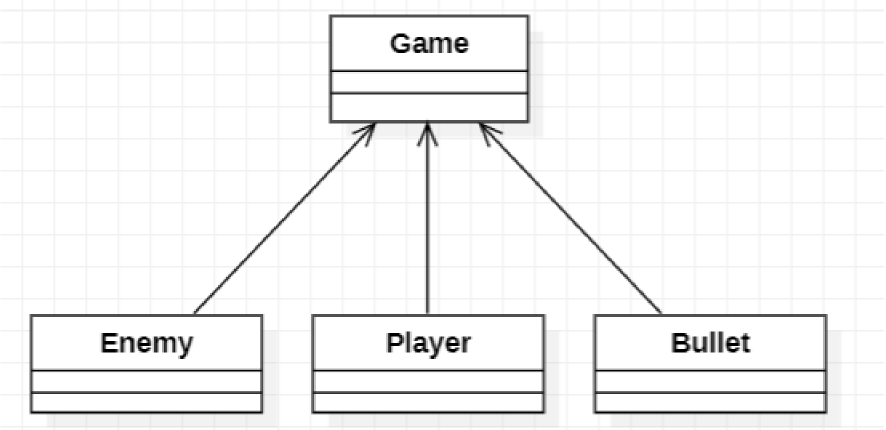
Program

+Main(): void

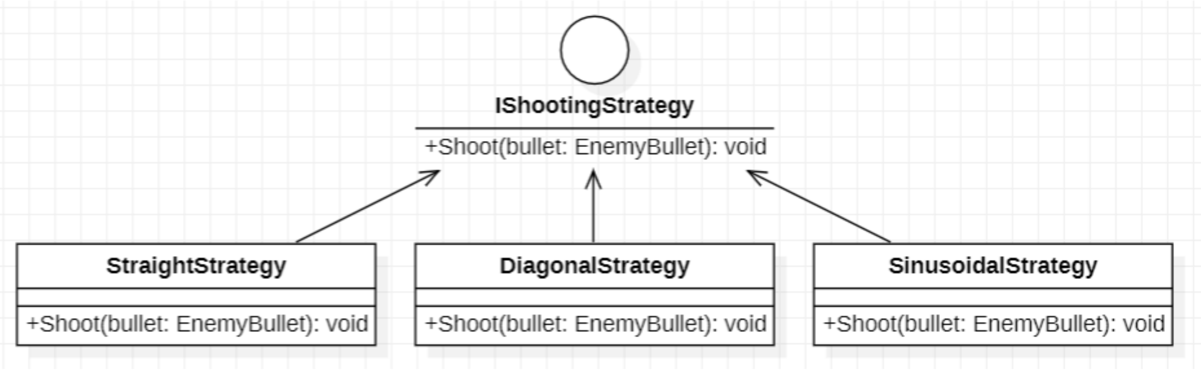
Sequence Diagram


sd Player Movement and Bullet Firing Sequence
Player
Game
Bullet
1 : MoveInput()
2 : Move the player
3 : Space key pressed
(Player fires a bullet)
4 : Create a new Bullet object
with player position
5 : Add the bullet to the
active bullets list
6 : Update the bullet position
7 : Update game state(enemies,
collisions, etc.)

Design Pattern



1. Template Method Pattern: The Template Method pattern is used in the Game class. The Update method in the Game class can be considered a template method. It defines the skeleton of an algorithm but allows subclasses (EnemyBullet, Bullet, UFO, etc.) to override some steps of the algorithm without changing its structure.


IShootingStrategy
+Shoot(bullet: EnemyBullet): void
StraightStrategy
+Shoot(bullet: EnemyBullet): void
DiagonalStrategy
+Shoot(bullet: EnemyBullet): void
SinusoidalStrategy
+Shoot(bullet: EnemyBullet): void

2. Strategy Pattern: The Strategy pattern is used to define a family of algorithms, encapsulate each one, and make them interchangeable. In the context of your program, the Enemy class uses the Strategy pattern to define different shooting strategies for different types of enemies. The IShootingStrategy interface represents the family of shooting algorithms, and the concrete classes StraightShootingStrategy, DiagonalShootingStrategy, and SinusoidalShootingStrategy encapsulate the individual shooting behaviors.