Design Overview for Alien Blaster Program

Name: Hai Nam Ngo
Student ID: 103488515

Summary of Program

"Alien Blaster" is an interactive C# gaming project where players are tasked with controlling a spaceship to shoot aliens within a specified time to accrue points. As players successfully shoot more aliens, the point tally increases. However, they must exercise caution to avoid contact with monsters and prevent monster invasions at the base.

Incorporating fundamental programming concepts, the game employs abstraction, allowing the spaceship to shoot without player knowledge of the shooting mechanism's intricacies. The architecture is built around an Alien Class (parent), capable of moving and shooting. This foundational class spawns three distinct Alien Types (children), each inheriting the Alien Class's capabilities while introducing additional unique abilities, effectively demonstrating polymorphism in the game's design.

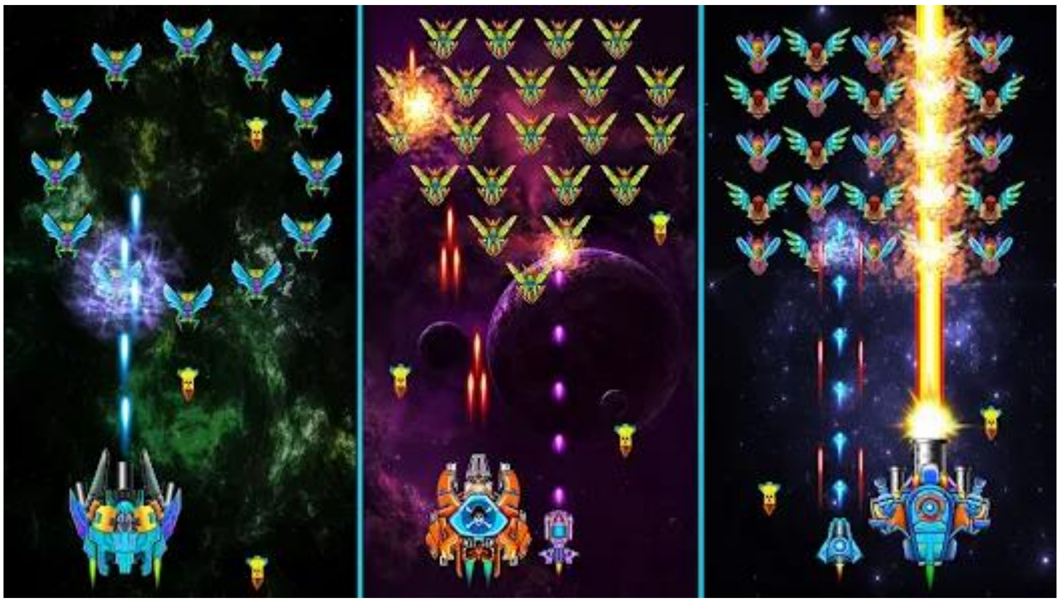


Figure 1 Sample Output (Galaxy Attack: Shooting Game)

Required Roles

Describe each of the classes, interfaces, and any enumerations you will create. Use a different table to describe each role you will have, using the following table templates.

Table 1: GameObject Class details

Responsibility	Type Details	Notes
_gameImage	PictureBox	

_speed	int	
GameObject()	constructor	Takes PictureBox as a parameter
Move()	<<abstract>> method	Abstract method to move the object
Attack()	<<abstract>> method	Abstract method for the object to perform an attack

Table 2: SpaceShip Class details

Responsibility	Type Details	Notes
pressedKeys	HashSet<Keys>	To store keys pressed
_bullets	List<Bullet>	Bullets shot by the spaceship
SpaceShip()	Constructor	
Move()	void	Move the spaceship
Attack()	void	Spaceship attacks
GetBullets()	List<Bullet>	Returns the bullets associated with ship

Table 3: Alien Class details

Responsibility	Type Details	Notes
_random	Random	
Alien()	Constructor	
Move()	Void	Abstract method, moves the Alien
Attack()	Void	Abstract method, Alien attacks

Table 4: FireMonster Class details

Responsibility	Type Details	Notes
FireMonster()	Constructor	
Move()	Void	Override abstract Move()
Attack()	void	Override abstract Attack()

Table 5: AcidMonster Class details

Responsibility	Type Details	Notes
AcidMonster()	Constructor	
Move()	Void	Override abstract Move()
Attack()	void	Override abstract Attack()

Table 6: FlashMonster Class details

Responsibility	Type Details	Notes
FlashMonster()	Constructor	
Move()	Void	Override abstract Move()
Attack()	void	Override abstract Attack()

Table 7: Form1 (or GameSystem) Class details

Responsibility	Type Details	Notes
SingleButton_Click()	Event Method	Method for Single Mode button click event
MultiButton_Click()	Event Method	Method for Multi Mode button click event

Table 8: MultiMode Class details

Responsibility	Type Details	Notes
MultiMode_KeyDown()	Event Method	Handles key down event in MultiMode
MultiMode_KeyUp()	Event Method	Handles key up event in MultiMode
AlienSpawn()	void	Spawns aliens in the MultiMode game
CheckCollisions()	Void	Checks for collisions in the game
EndGame()	void	Ends the MultiMode game

Table 9: SingleMode Class details

Responsibility	Type Details	Notes
SingleMode_KeyDown()	Event Method	Handles key down event in SingleMode
SingleMode_KeyUp()	Event Method	Handles key up event in SingleMode
AlienSpawn()	void	Spawns aliens in the SingleMode game
CheckCollisions()	Void	Checks for collisions in the game
EndGame()	void	Ends the SingleModegame

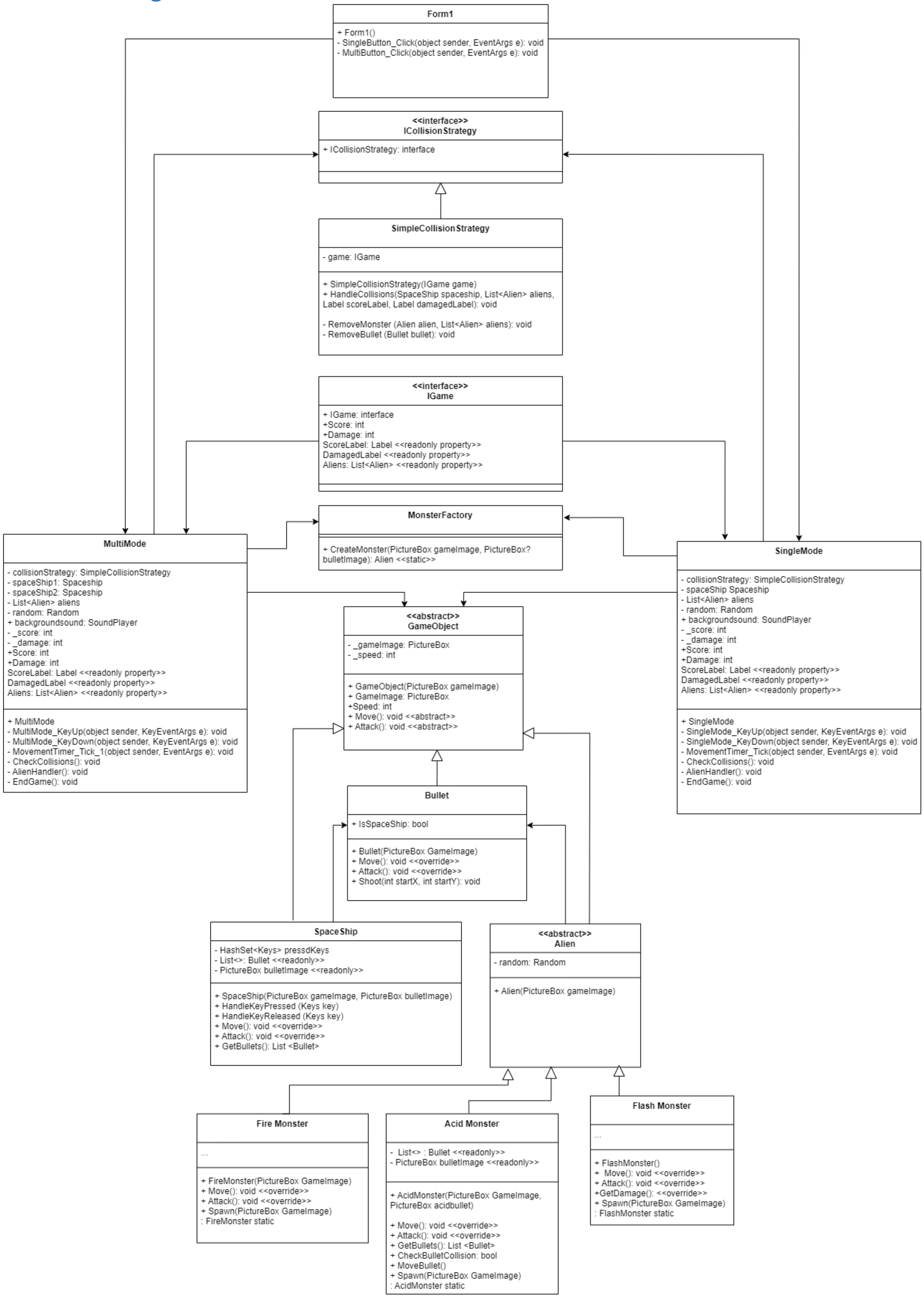
Table 10: Bullet Class details

Responsibility	Type Details	Notes
IsSpaceShip	bool	Determines if bullet belongs to a spaceship
Bullet()	constructor	Initializes bullet with PictureBox
Move()	void	Moves the bullet
Attack()	void	Shoots the bullet

Table 11: Alien details

Value	Notes
Fire Monster	Normal Monster
Acid Monster	Shoot bullet
Flash Monster	High speed to crash the spaceship

HD Class Diagram


Form1
+ Form1()
- SingleButton_Click(object sender, EventArgs e): void
- MultiButton_Click(object sender, EventArgs e): void
<<interface>>
ICollisionStrategy
+ ICollisionStrategy: interface
SimpleCollisionStrategy
- game: IGame
+ SimpleCollisionStrategy(IGame game)
+ HandleCollisions(SpaceShip spaceship, List<Alien> aliens, Label scoreLabel, Label damagedLabel): void
- RemoveMonster (Alien alien, List<Alien> aliens): void
- RemoveBullet (Bullet bullet): void
<<interface>>
IGame
+ IGame: interface
+Score: int
+Damage: int
ScoreLabel: Label <<readonly property>>
DamagedLabel <<readonly property>>
Aliens: List<Alien> <<readonly property>>
MonsterFactory
+ CreateMonster(PictureBox gameImage, PictureBox? bulletImage): Alien <<static>>
MultiMode
- collisionStrategy: SimpleCollisionStrategy
- spaceShip1: Spaceship
- spaceShip2: Spaceship
- List<Alien> aliens
- random: Random
+ backgroundsound: SoundPlayer
- _score: int
- _damage: int
+Score: int
+Damage: int
ScoreLabel: Label <<readonly property>>
DamagedLabel <<readonly property>>
Aliens: List<Alien> <<readonly property>>
+ MultiMode
- MultiMode_KeyUp(object sender, KeyEventArgs e): void
- MultiMode_KeyDown(object sender, KeyEventArgs e): void
- MovementTimer_Tick_1(object sender, EventArgs e): void
- CheckCollisions(): void
- AlienHandler(): void
- EndGame(): void
SingleMode
- collisionStrategy: SimpleCollisionStrategy
- spaceShip Spaceship
- List<Alien> aliens
- random: Random
+ backgroundsound: SoundPlayer
- _score: int
- _damage: int
+Score: int
+Damage: int
ScoreLabel: Label <<readonly property>>
DamagedLabel <<readonly property>>
Aliens: List<Alien> <<readonly property>>
+ SingleMode
- SingleMode_KeyUp(object sender, KeyEventArgs e): void
- SingleMode_KeyDown(object sender, KeyEventArgs e): void
- MovementTimer_Tick(object sender, EventArgs e): void
- CheckCollisions(): void
- AlienHandler(): void
- EndGame(): void
<<abstract>>
GameObject
- _gameImage: PictureBox
- _speed: int
+ GameObject(PictureBox gameImage)
+ GameImage: PictureBox
+Speed: int
+ Move(): void <<abstract>>
+ Attack(): void <<abstract>>
Bullet
+ IsSpaceShip: bool
+ Bullet(PictureBox GameImage)
+ Move(): void <<override>>
+ Attack(): void <<override>>
+ Shoot(int startX, int startY): void
SpaceShip
- HashSet<Keys> pressdKeys
- List<>: Bullet <<readonly>>
- PictureBox bulletImage <<readonly>>
+ SpaceShip(PictureBox gameImage, PictureBox bulletImage)
+ HandleKeyPressed (Keys key)
+ HandleKeyReleased (Keys key)
+ Move(): void <<override>>
+ Attack(): void <<override>>
+ GetBullets(): List <Bullet>
<<abstract>>
Alien
- random: Random
+ Alien(PictureBox gameImage)
Fire Monster
...
+ FireMonster(PictureBox GameImage)
+ Move(): void <<override>>
+ Attack(): void <<override>>
+ Spawn(PictureBox GameImage)
: FireMonster static
Acid Monster
- List<> : Bullet <<readonly>>
- PictureBox bulletImage <<readonly>>
+ AcidMonster(PictureBox GameImage, PictureBox acidbullet)
+ Move(): void <<override>>
+ Attack(): void <<override>>
+ GetBullets(): List <Bullet>
+ CheckBulletCollision: bool
+ MoveBullet()
+ Spawn(PictureBox GameImage)
: AcidMonster static
Flash Monster
...
+ FlashMonster()
+ Move(): void <<override>>
+ Attack(): void <<override>>
+GetDamage(): <<override>>
+ Spawn(PictureBox GameImage)
: FlashMonster static

Additional features

To elevate the gaming experience and introduce more intricate layers to the application's design, I've integrated a few significant enhancements:

Multiplayer Mode:

I've incorporated a multiplayer mode. This was achieved by developing an additional form, allowing players to engage in a more competitive setting, challenging not just the game's AI but also their peers.

Rich Soundscapes:

Understanding the importance of auditory feedback and its role in player immersion, I've embedded background soundtracks for both the main menu and in-game environments. Leveraging the capabilities of System.Media, players are now greeted with a more holistic sensory experience, making each game session more memorable and engaging.

By incorporating these enhancements, the game not only provides more gameplay options but also resonates more deeply with its players through enriched audio-visual feedback.

Opportunities for applying design patterns

Strategy Pattern:

The movement and attack patterns of game characters (such as the spaceship and various monsters) are designed using the Strategy design pattern. With this pattern, different algorithms for collision are handled. The Strategy pattern promotes flexibility and maintainability, enabling the game mechanics to be smoothly adapted as needed. By defining strategy classes, the game logic can be kept modular and extensible.

Factory Patterns: The code implements the Factory pattern uniquely by using static "Spawn" methods in the FireMonster, FlashMonster, and AcidMonster classes to instantiate objects of those monster types. This design follows the factory method pattern, centralizing object creation into designated factory methods. With this approach, monster creation is streamlined and consistent, making it simpler to add new monster types later. The Factory pattern brings extensibility benefits and promotes a clean, organized code structure. By encapsulating monster creation logic within static factory methods, the code stays modular and extendable when new monster varieties are introduced. Overall, the strategic use of factory methods for monster spawning makes the object creation process more robust and maintainable.

What makes my new program design more complex

Adding Function ability:

One of the key design considerations I focused on was the ease of introducing new game elements. For example, if I want to create a new Monster class, maybe a boss for the game.

- Creating new Monster classes like a boss is easy by following the expected monster interface. This simplifies expanding the variety of monsters.
- Updating the factory to handle new monster types keeps core game logic untouched. The modular approach makes adding new game features straightforward.

Or if I want to make some boost item that will make spaceship stronger for a limited time, I can add the Item class and then put it into collision handler to activate its function.

Flexibility in Component Modification:

- The design enables flexibility in modifying spaceship and monster behaviours by swapping strategies.
- This facilitates fine-tuning gameplay without extensive rewriting. The program is adaptable to changing requirements.
- For example, creating a new game mode could be done by simply tweaking the strategies for certain objects, without having to change the core code.

Design Reusability for Diverse Applications:

- The code structure and organization promote reusability beyond gaming.
- The Strategy and Factory patterns can be applied in many software systems needing interchangeable algorithms and dynamic object creation.
- It could be reused for other games with similar concepts of a main character and enemies by leveraging the same strategy classes.