ITERATOR Pattern

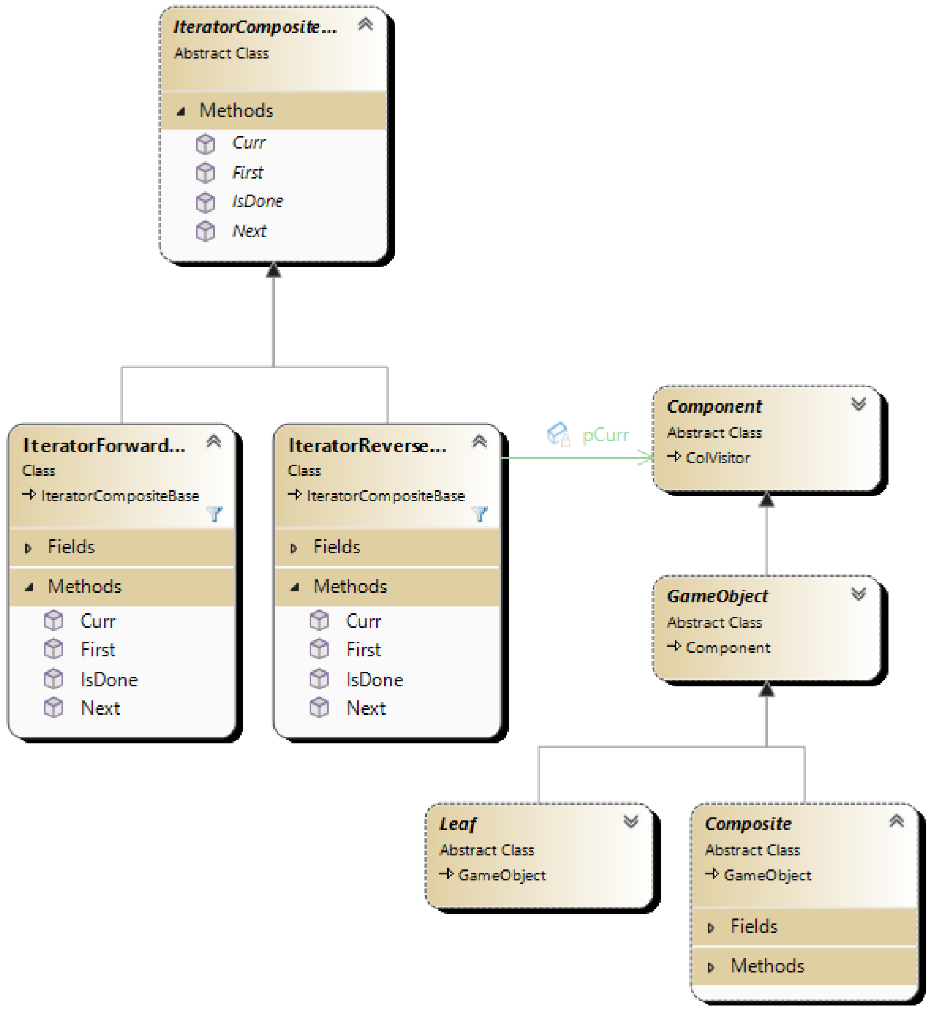
Challenge

Problem:

In the Space Invaders game, we need a way to efficiently handle traversals over collections of game objects used in the game with different data structures such as linked lists (used mostly) or tree structures (composites) which are alien grid and shields. These objects are organized into hierarchical structures. Creating a unique traversal logic for each of the structures would cause redundancy and bring more complexity to the code logic.

Solution:

To solve this issue, we implemented the Iterator pattern, creating a consistent and simple mechanism to traverse complex object collections without exposing their internal representations. This pattern allows us to iterate through each object without the knowledge of their underlying structures.



Pattern description

The Iterator Pattern provides a mechanism to sequentially access elements in a collection without exposing the underlying structure. It is especially beneficial in encapsulating traversal logic, enabling iterate over collections seamlessly without concern for how the data is stored internally.

Our implementation provides iterators designed to traverse different data structures like tree structures used in the game such as composite patterns and linked list data structure. These iterators abstract the details of the data structures, allowing code to remain clean not focusing on any data structure manipulation. Only need to call Curr() for current node, First() for the first element, Next() to move to the next node and IsDone() to check if the Curr() is still valid.

Key Object Oriented mechanics

Encapsulation ensures the internal representation of data structures remains hidden.

Abstraction allows various data structures like trees or linked lists to be iterated over uniformly. Polymorphism allows easy interchangeability of different iterators which use the same base class, enabling flexible traversal strategies like forward iterator or reverse iterator.

The traversal logic is centralized within the iterator, allowing it to iterate through elements without knowledge of their underlying structure.

How it is used in Space Invaders

In the Space Invaders game, the Iterator pattern simplifies traversal through collections of game objects such as Alien Grids and Shields. For instance, when moving aliens horizontally or vertically, the iterator iterates through each element in the composite hierarchy by reaching the children first, then moving to siblings and lastly to the parent without explicit knowledge of the internal structure.

Iterator pattern also used for the linked list structure. For example, when iterating through SpriteNodes, the iterator starts to iterate through the head and goes until the end of the linked list. Erase method in the iterator used for the TimerEventMan update function to remove the timer events from the list when the time is done for that timer event.

FACTORY Pattern

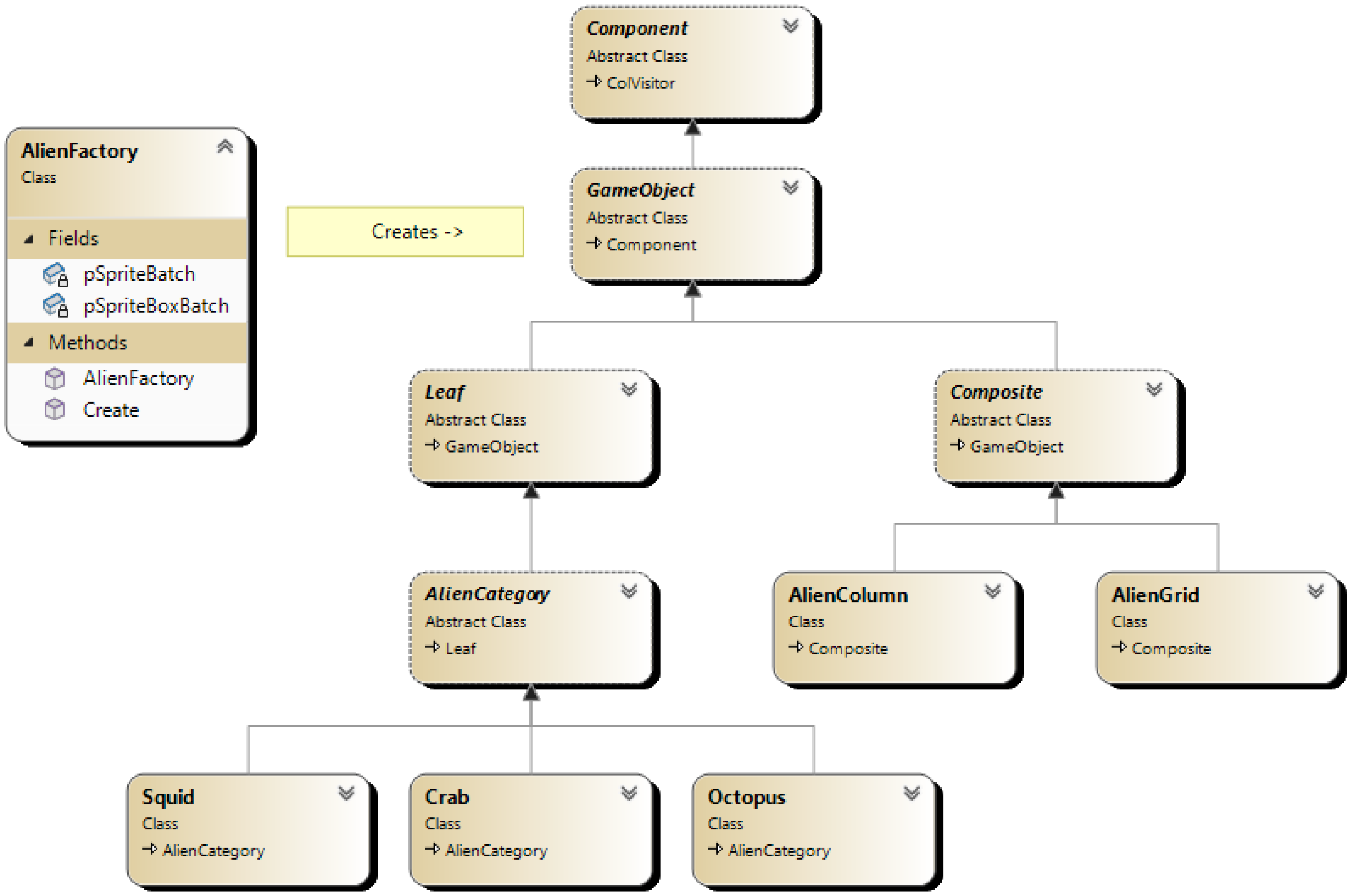
Challenge

Problem:

In the Space Invaders game, we need an efficient and reusable way for the creation of various game objects such as aliens or shields etc. Each alien type like Squid, Crab, and Octopus required instantiation multiple times in different positions within a column and a grid structure. Total of 55 aliens in a 5 x 11 grid with 11 Squids, 22 Crabs and 22 Octopuses. Similarly, shields required instantiation multiple times in different positions. Creating these objects manually would lead to code redundancy and code duplications.

Solution:

To solve this problem, we implemented the Factory design pattern for providing a centralized process for object creation. By doing so, rather than calling the new directly while creating a new alien or a shield type, we create those objects inside the Factory class that calls the new under the hood with the necessary parameters, creates the objects within the class and returns correct object. The Factory pattern simplifies object creation, reduces code redundancy and code duplications.



Pattern description

The Factory Pattern is responsible for the object creation process, providing a common class for creating game objects without exposing instantiation logic. This abstraction separates the code from the complexities of object creation (hard coding), allowing new object types to be integrated with minimal changes to the existing code.

Our implementation involves an AlienFactory class specifically designed to handle the creation of different alien types (Squid, Crab, Octopus, Column and Grid). When requested, this factory generates the required alien instances, sets their respective attributes, such as position(x,y) and returns them without exposing the instantiation details. Similarly, ShieldFactory class specifically designed for creation of different shield types.

Key Object Oriented mechanics

The Factory pattern uses encapsulation and abstraction. Encapsulation hides the internal complexity of object creation, encapsulating it within the factory class itself.

Abstraction simplifies the object creation process by providing a generic creation interface. The code simply requests an object, delegating the specifics of instantiation such as which type of sprite is created to the factory. This significantly reduces coupling between different parts of the code, enhancing maintainability and flexibility.

How it is used in Space Invaders

In our Space Invaders game, the Factory pattern is particularly valuable for creating alien and shield objects. The AlienFactory class encapsulates logic for instantiating aliens based on provided parameters such as type Squid, Crab and Octopus and position (x, y coordinates). When a request to create an alien is made, the factory internally determines the appropriate game object to instantiate based on the provided type parameter. We used Alien Factory to create the alien grid with 11 columns and 55 aliens. The ShieldFactory class is used to create bricks of the shield based on the given type, name and position(x,y). Similarly, when a shield is requested, the factory determines which object to instantiate based on the given type. We used the Shield Factory to create 4 shield grids with columns and bricks with different positions.

PROXY Pattern

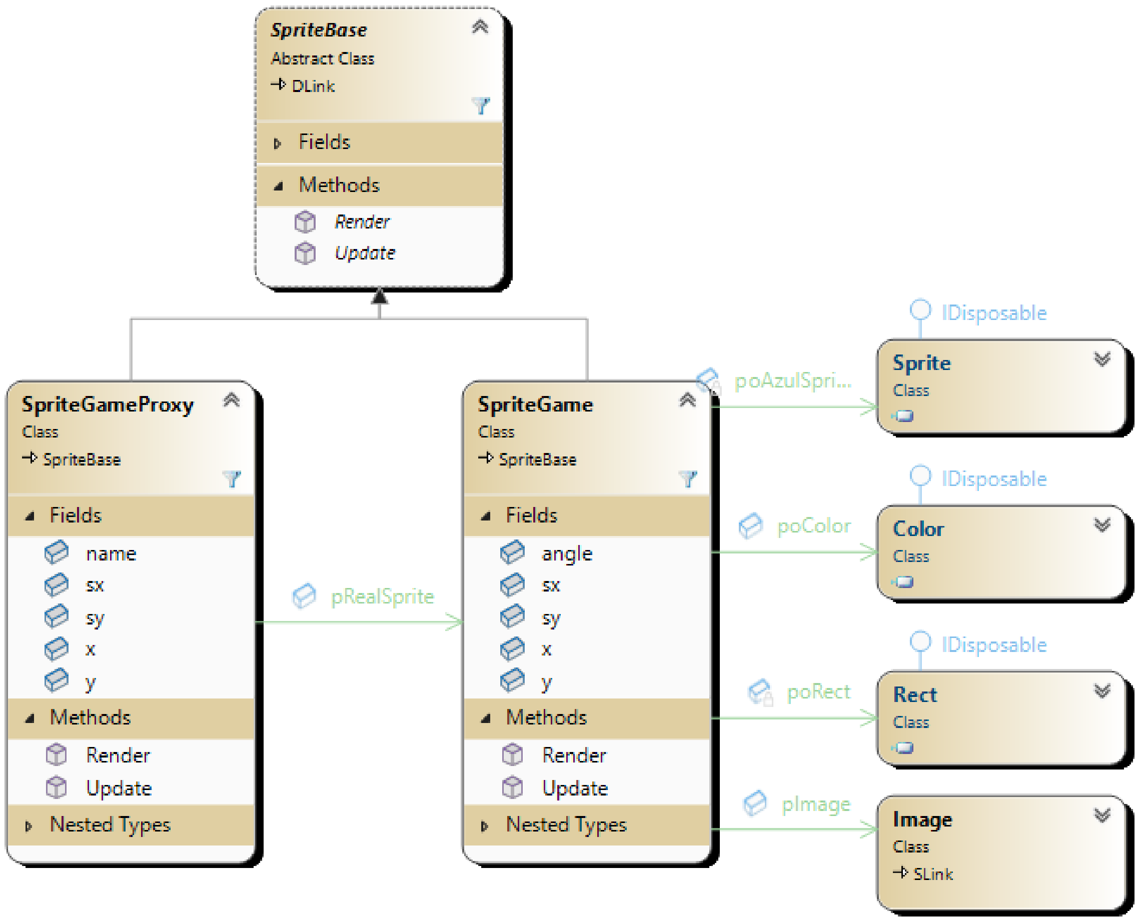
Challenge

Problem:

In the Space Invaders game, we have a large number of similar sprites on screen at once. Specifically, there are 55 alien objects divided into three types: 11 squids, 22 crabs, and 22 octopuses. The only distinguishing feature among these sprites is their position (x, y) on the screen. Rendering and managing these objects individually would be very inefficient. We need a way to reuse the same sprite object while separating the position information to prevent unnecessary duplication.

Solution:

The Proxy Pattern provides an effective solution by creating light weight objects (Proxy) that manage the position (x, y) while referencing a shared heavy weight object (RealSprite). Instead of having 55 individual sprite instances, we create one RealSprite per alien type and assign multiple proxy instances to handle their positions. The proxy serves as an intermediary, forwarding any commands to the shared RealSprite while maintaining some unique data.



Pattern description

This pattern is used to control access to heavy weight object (RealSprite) by creating a light weight surrogates (Proxy).

The pattern allows the numerous sprites which share identical features but differ in their position data (x,y coordinates in our case) to be efficiently represent a Proxy object. The RealSprite contains more data than the Proxy which is common to all Proxy that points to the same RealSprite. Methods that called on the Proxy are forwarded to the RealSprite and the RealSprite executes the actual task.

Each Proxy contains a unique position (x,y) which is pushed to the RealSprite when a method is called on the Proxy. When a method is called, the Proxy “push” its state (updating the x and y coordinates) to the shared RealSprite and when the push is done, the RealSprite becomes to the correct state.

Key Object Oriented mechanics

Delegation plays a central role as each Proxy delegates tasks to the RealSprite through shared base classes, ensuring consistency across Proxy and RealSprite.

Each Proxy maintains minimal unique data and delegates complex tasks to a shared RealSprite object using a pointer. The RealSprite contains significantly more common data (image, color, rect etc.) and executes the actual tasks. This delegation process ensures efficiency and uniform behavior across numerous light weight Proxies.

How it is used in Space Invaders

In space invaders we specifically use this pattern for specifically for managing aliens and the bricks in the shield.

Rather than creating distinct sprite objects for each alien, we created light weight Proxies to represent each alien instance. These Proxies maintain their unique positions while delegating other behaviors to a shared RealSprite. Instead of creating 55 unique RealSprite each with their own image, color, rect, etc., we create 3 unique RealSprite allowing us to animate multiple Proxies pointing the same RealSprite.

Shields in the game contain many bricks inside. Most of the bricks inside the shields are the same. Rather than creating a unique RealSprite for each of the same bricks, creating them as Proxy objects is more efficient.

STATE Pattern

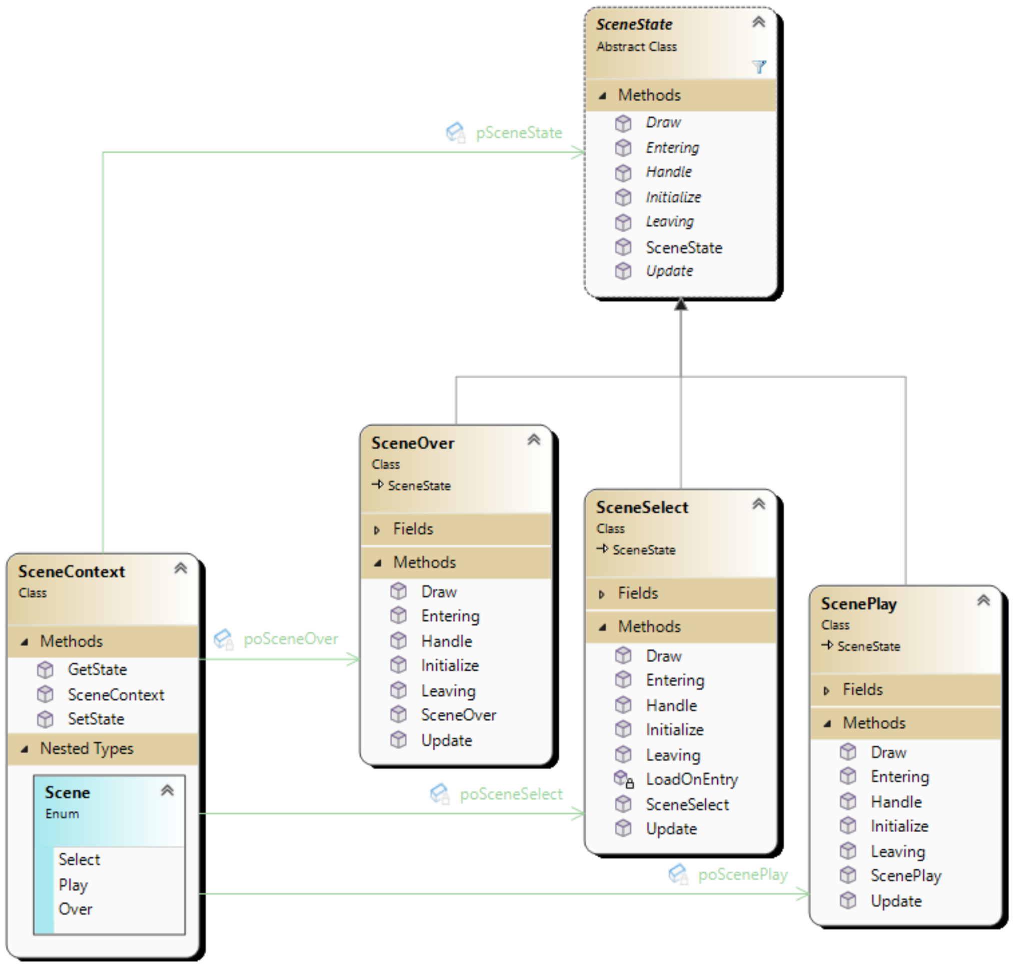
Challenge

Problem:

In the Space Invaders game, we faced the challenge of managing the different behavioral states of various game entities such as ship, missile, scene. For instance, game entities needed distinct behaviors for various states. Implementing these behaviors with long conditionals (if-else statements) would make the code more complicated and more open to bugs

Solution:

To simplify the management of these behaviors, we implemented State pattern. This pattern allows us to encapsulate each distinct behavior within its own state classes, enabling seamless transitions between different states without using conditionals. This pattern reduces the usage of conditionals and allows us to cleanly transition between states while maintaining more clear and modifiable code.



Pattern description

The State pattern allows objects to change their behavior dynamically by transitioning between distinct state classes based on internal conditions and external events. Each behavior state such as the missile being ready or actively flying or the ship's movement states are encapsulated within its specific class. By clearly separating the states, the pattern simplifies code structure and enhances readability

Key Object Oriented mechanics

Each state class encapsulates behavior relevant to a particular state, keeping the state logic independent. For instance, missile firing logic and ship movements are each contained in their respective state classes. The state classes implement a common abstract class (ShipMissileState, ShipMoveState or SceneState) enabling interchangeable behaviors without modifying the main object. This polymorphic behavior allows clean state transitions and simplifies the code.

How it is used in Space Invaders

In our Space Invaders game, the State pattern has been implemented specifically for the Ship, Missiles, and Scene transitions. For the Ship, states such as ShipMissleFlying, and ShipReady manage firing missiles. This ensures the ship responds correctly to player input or collisions based on its current situation. They are managed by ShipMissileState.

For the movement of the ship there are 3 states which are ShipMoveBoth, ShipMoveLeft and ShipMoveRight. When the ship collides with left bumper, only ShipMoveRight state gets active, when the ship collides with right bumper, only ShipMoveLeft state gets active and when the ship does not collide with any of the bumpers ShipMoveBoth state gets active. They are managed by ShipMoveState.

Scene management uses the State pattern to control game flow, transitioning smoothly from the SceneSelect to ScenePlay and eventually to SceneOver based on conditions. SceneSelect is used for the selection of the game. ScenePlay is used for game play and SceneOver is used when the game ends. They are managed by SceneState.

STRATEGY Pattern

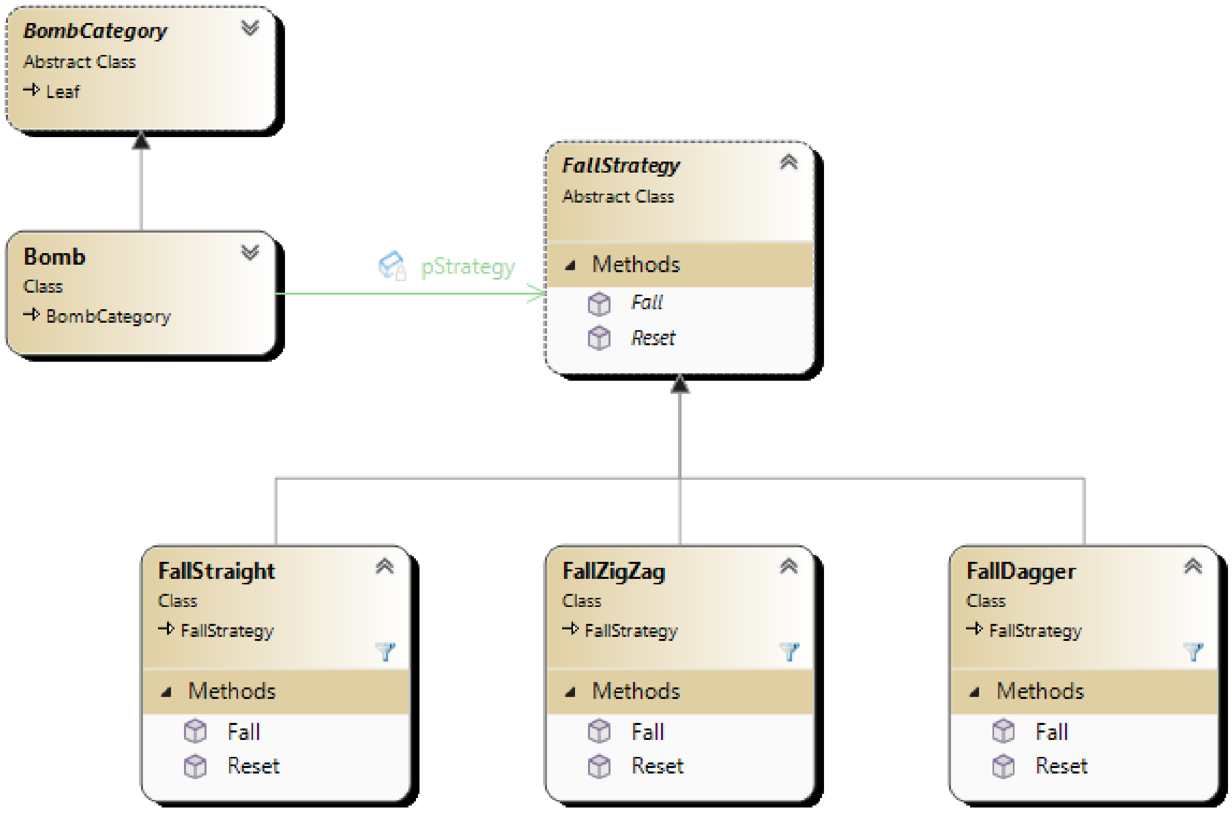
Challenge

Problem:

In Space Invaders game, bombs dropped by the aliens need to behave in different movement patterns. Hardcoding multiple bomb behaviors such as straight fall or zigzag movements to a single bomb class would make the code difficult to maintain and hard to extend with a new one without modifying the core object logic. We need a flexible way to define and switch movement behaviors without altering the bomb objects themselves.

Solution:

We used Strategy pattern the solve this problem. This pattern allows the bomb's movement logic to be defined independently and swapped at runtime. Rather than defining the movement behavior in the Bomb class, we delegate the movement behavior to a different strategy class, enabling us to assign different movement patterns to bombs dynamically while keeping the Bomb class focused on its core properties.



Pattern description

The Strategy pattern enables an object’s behavior to be selected and altered dynamically by delegating specific movement behaviors. This allows the behavior of an object to change dynamically by selecting different strategies at runtime without changing the object’s underlying structure

By using this pattern, we eliminated complex conditional statements from bomb logic, enabled easy addition of new bomb falling behaviors without altering existing code and improved flexibility of the game logic.

Key Object Oriented mechanics

Each specific falling behavior such as FallStraight, FallZigZag or FallDagger is encapsulated within its own strategy class. The FallStrategy abstract class defines abstract functions (Fall and Reset) for all strategy classes, abstracting implementation details from the bomb objects.

Strategies can be dynamically swapped and used interchangeably by the bomb object, enabling different bomb behaviors to be executed without modifying the Bomb class.

How it is used in Space Invaders

Each different bomb has its own falling strategy. The strategies determine how the bomb movement should be. Each strategy class has its own Reset() and Fall() functions, enabling different movement behavior. We use 3 different strategies which are:

FallStraight: The bomb falls in with a straight movement, nothing fancy.

FallDagger: The bomb falls and moves in with zig-zag pattern on y coordinates.

FallZigZag: The bomb falls and moves in with zig-zag pattern on x coordinates.

With the help of Strategy pattern, adding a new movement strategy requires only a new strategy class, leaving the core code untouched.

VISITOR Pattern

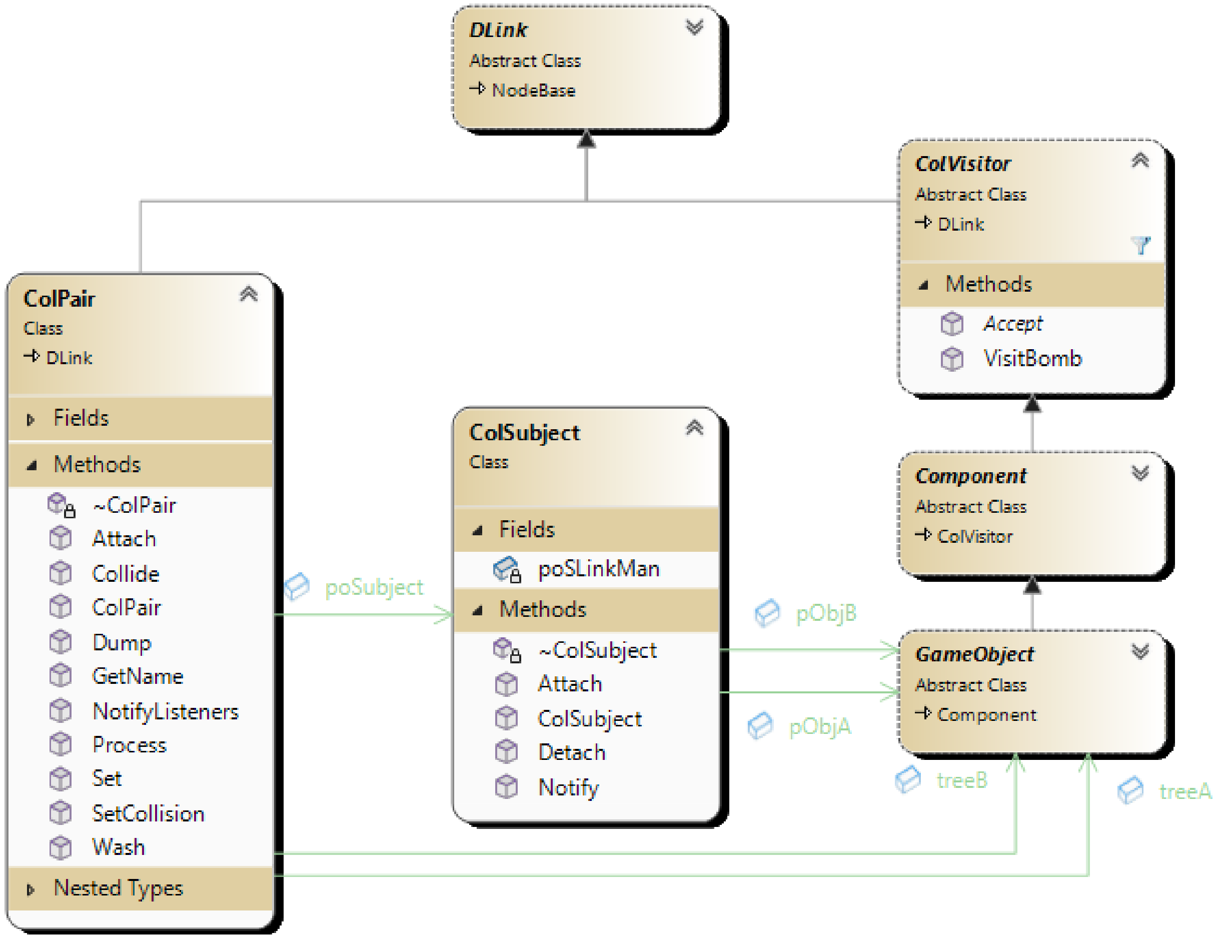
Challenge

Problem:

In the Space Invaders game, we faced a significant challenge in handling collisions between numerous different game objects such as aliens, missiles, ships, walls, bombs, shields and bumpers. We need a structured way to handle the collisions, determine which game object collided with which game object and decide how the collided objects should act after the collisions.

Solution:

Instead of using long if-else statements, we implemented the Visitor pattern. Using this pattern, we efficiently manage collision interactions among diverse game objects. Each game object has its own Accept() function that calls the appropriate Visit() function. These functions allow us to clearly separate the collisions without using any conditionals and how the game objects should react based on the collisions.



Pattern description

Visitor Pattern enables double dispatch, where the behavior depends on both the type of the visitor and the type of the object that is visited. Abstarct base ColVisitor class defines virtual Visit methods for each game object type such as VisitCrab, VisitMissile, VisitShieldBrick etc. Concrete visitor classes inherit from ColVisitor and implement these methods to handle specific interactions. Game objects implement an Accept() function that takes a ColVisitor and calls the appropriate Visit() function on the visitor, passing itself as an argument. The ColPair class operates the collision detection by iterating over pairs of game objects and invoking the Accept() function when a collision is detected.

Key Object Oriented mechanics

The Visitor pattern is effectively implemented through double dispatch. Double dispatch ensures that both objects involved in the collision actively participate in deciding the correct collision response. For example, when a missile and an alien collide, Missile.Accept(Alien) function gets called and it calls Alien.VisitMissle(Missile). This helps to avoid hard-coded type checking. Visitor pattern allows the collision behavior to be defined externally, keeping the game object classes focused on their core responsibilities rather than collision logic.

How it is used in Space Invaders

In the Space Invaders game, every game object has its own Accept() function that calls the Visit() function for the game object that its collided.

For example:

Missile vs Alien: Missile.Accept(Alien) that calls Alien.VisitMissile(Missile),

Bumper vs Ship: Bumper.Accept(Ship) that calls Ship.VisitBumper(Bumper),

Bomb vs Wall: Bomb.Accept(Wall) that calls Wall.VisitBomb(Bomb),

Missile vs Wall: Missile.Accept(Wall) that calls Wall.VisitMissile(Missile),

Bomb vs Shield: Bomb.Accept(Shield) that calls Shield.VisitBomb(Bomb),

Missile vs Shield: Missile.Accept(Shield) that calls Shield.VisitMissile(Missile),

Alien vs Wall: Alien.Accept(Wall) that calls Wall.VisitAlien(Alien)

OBSERVER Pattern

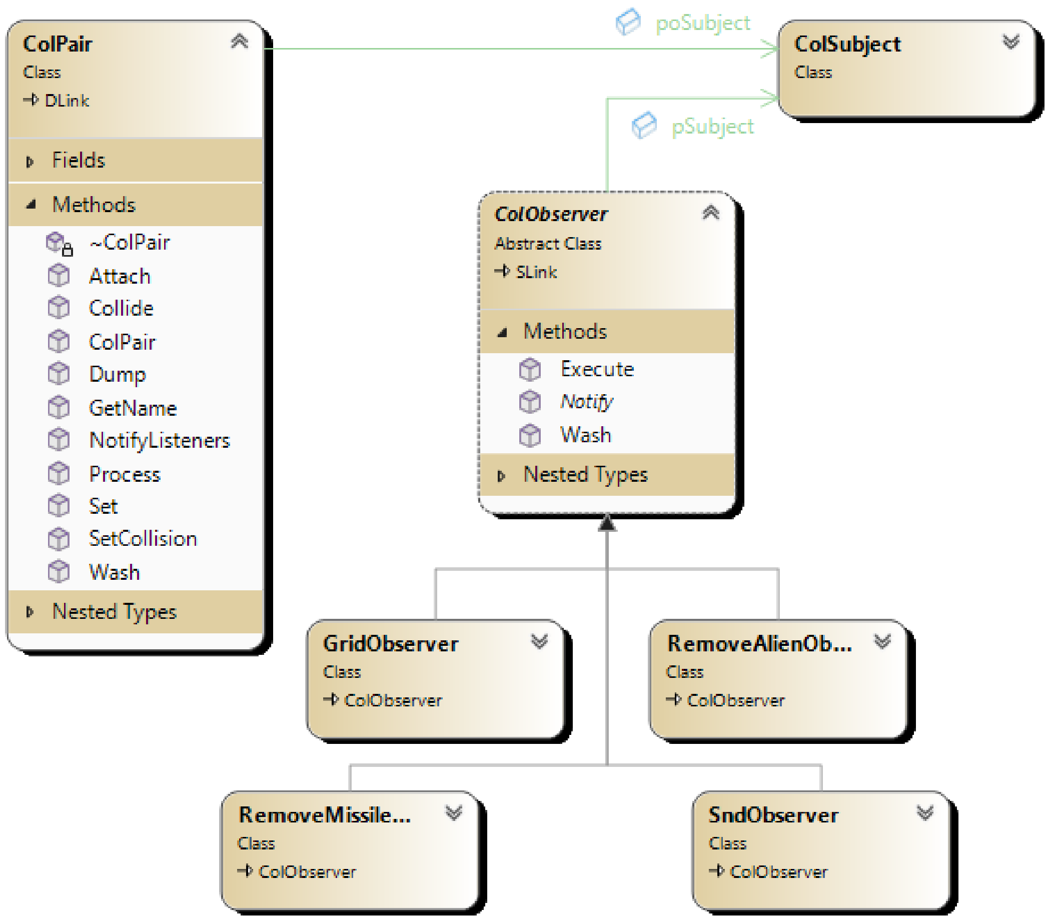
Challenge

Problem:

In the Space Invaders game, multiple objects need to react to events such as collisions (e.g. Missile hitting an Alien). For instance, when a collision occurs, actions like removing an alien, updating the score, playing a sound, or removing a missile must happen in a coordinated way. Events needs to trigger the actions. Handling these actions with direct, hard-coded calls or with conditionals would cause inefficiency and would be very messy. We need a way to notify the systems when an event occurs.

Solution:

To solve this problem, we implemented the Observer pattern. This pattern allows us to notify the according observer classes when an event occurs. Each observer is distinct from each other and notified by the ColSubject (Collision Subject) class.



Pattern description

The Observer pattern allows one-to-many relationship between the subject and its observers. Observers register themselves with the subject and get notified. The ColSubject class holds references to two colliding objects (pObjA and pObjB) and a list of observers. The abstract ColObserver class defines the Notify() function which concrete observers override to implement specific reactions to the events. When a collision is detected the ColSubject calls Notify() function on all attached observers, passing itself to them with collision data. Observers can be dynamically attached or detached from the subject using Attach and Detach methods. This pattern ensures that the subject doesn't need to know the specifics of what each observer does, only that they need to be notified.

Key Object Oriented mechanics

The ColSubject (Collision Subject) maintains a list of observers and provides functions like Attach(), Detach(), and Notify() to the observers. It stores collision data (pObjA and pObjB) for observers to access. The abstract ColObserver defines a virtual Notify() method. Concrete observers such as RemoveAlienObserver, SoundObserver, RemoveMissleObserver etc. override Notify() to implement specific actions. When a collision happens, ColPair.NotifyListeners calls ColSubject.Notify which iterates through the observer list using an Iterator and invokes the correct observers Notify() function.

How it is used in Space Invaders

In space invaders we specifically use this pattern for the collision system.

For example:

Missile vs Wall collusion: RemoveMissileObserver -> removes the missile, ShipReadyObserver-> sets the ship's state to ready so that it can fire a missile.

AlienGrid vs Wall collusion: GridObserver -> moves the grid vertically and changes the direction when it collides with a wall.

Missile vs AlienGrid: RemoveAlienObserver -> removes the alien, RemoveMissileObserver -> removes the missile, UpdateScoreObserver -> Updates the score, AlienExplosionObserver -> animates alien explosion when missile hits an alien. SoundObserver-> plays a sound etc.

COMMAND Pattern

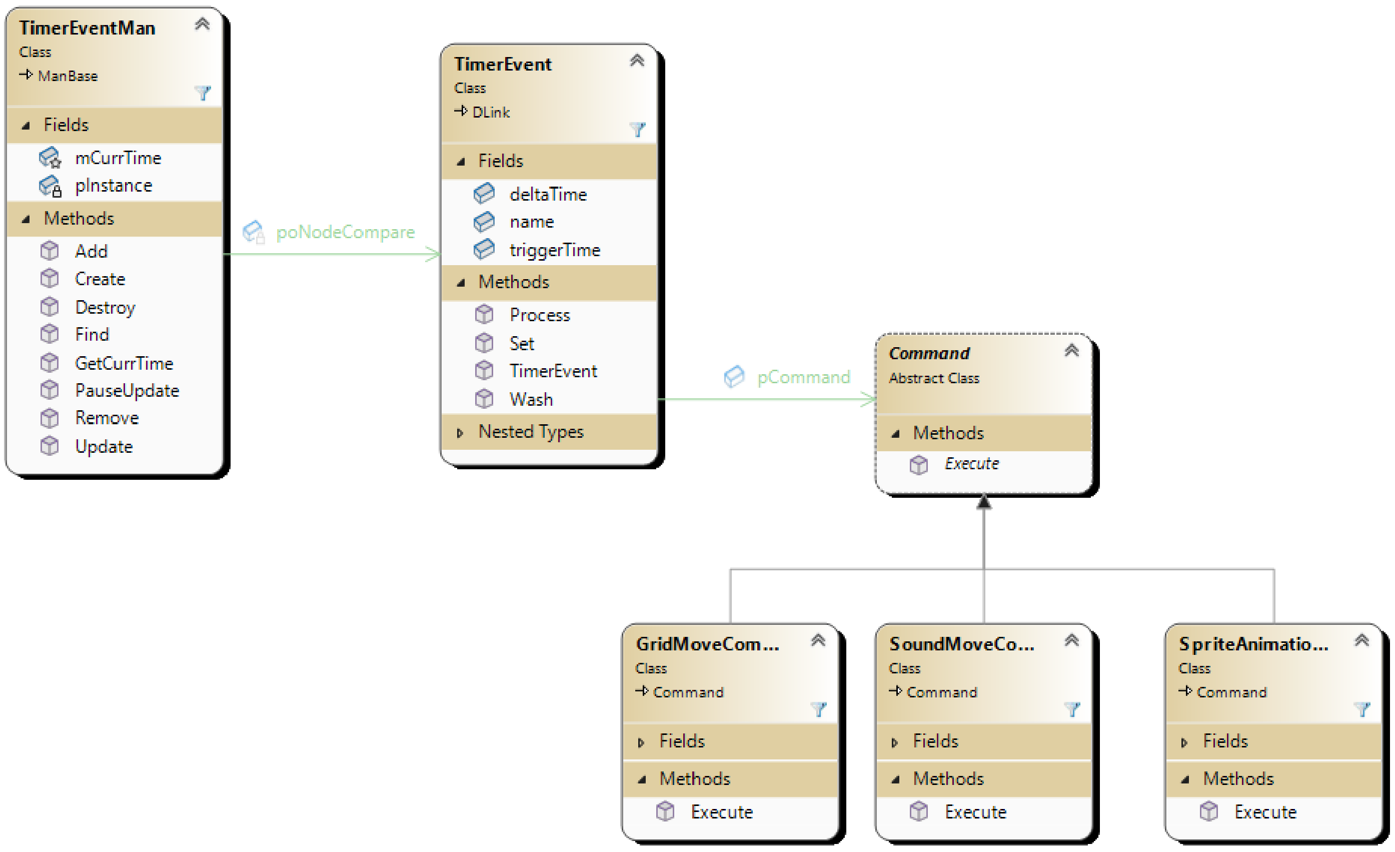
Challenge

Problem:

In the Space Invaders game, various actions such as moving the alien grid, playing sounds, animating sprites or spawning bombs need to be executed at specific time. Hardcoding these actions directly into the game loop would lead to inflexible and difficult to modify or reuse the code. We need a way to execute the actions when the correct time is triggered within a queue structure.

Solution:

The Command pattern addresses this by encapsulating each action as a Command object with an Execute() function. These commands can be queued and triggered as needed. TimerEventMan is a manager that handles the timing and execution of commands while command classes such as GridMoveCommand, SoundMoveCommand etc. define specific behaviors. All the commands are scheduled and queued based on their priority in a TimerEvent system and executed at the right time.



Pattern description

The Command pattern encapsulates requests as objects that can be managed and executed independently. The pattern allows us to store the commands, put them in a queue and execute them at the correct trigger time. The command classes are inherited from the abstract base class. Commands are parameterized during construction allowing them to adapt their behavior to different contexts and they reschedule themselves ensuring continuous execution.

Key Object Oriented mechanics

The Command class is an abstract class defines Execute() function, ensuring all commands share a common Execute() function. Classes like GridMoveCommand, SoundMoveCommand implement Execute() to encapsulate specific actions. For example, GridMoveCommand moves the alien grid and reschedules itself, while SoundMoveCommand plays one of four sounds in sequence and reschedules itself. Commands are added to TimerEventMan with the Add() function which calculates a trigger time based on deltaTimeToTrigger and assigns a priority for ordering.

How it is used in Space Invaders

The Command pattern manages dynamic game behaviors:

The GridMoveCommand moves the alien grid horizontally based on a Drum object's beat. After execution, it reschedules itself with TimerEventMan.Add() with the updated beat time, ensuring continuous movement synchronized with the game speed.

The SoundMoveCommand plays one of four sounds in a loop. It reschedules itself with TimerEventMan.Add() based on the Drum's beat, aligning sound with alien movement.

The SpriteAnimationCommand cycles through a list of images for sprites. It swaps the sprite's image in Execute() function and reschedules itself, creating smooth animations for the aliens.

COMPOSITE Pattern

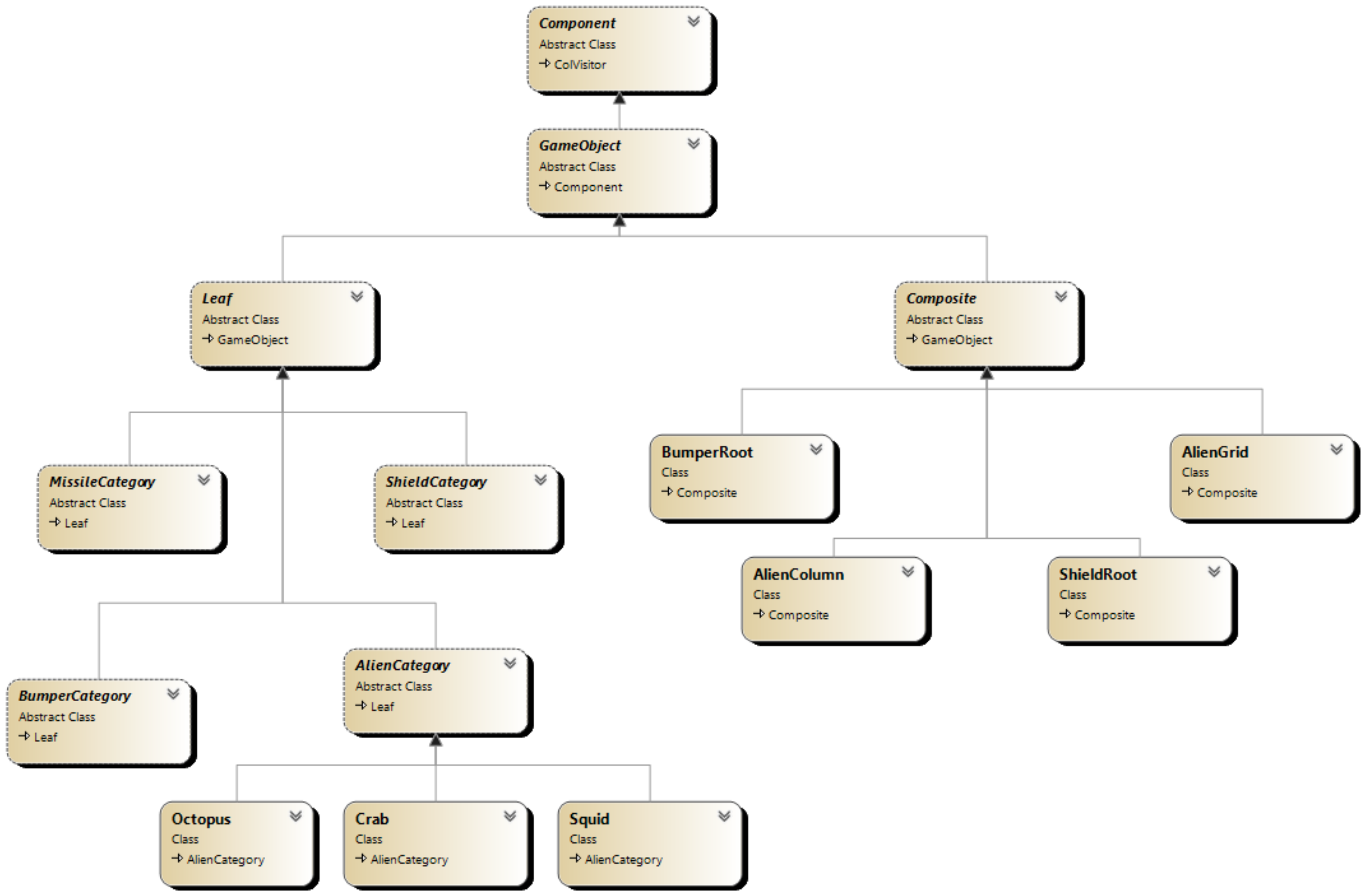
Challenge

Problem:

In the Space Invaders game, game objects such as the alien grid and shields are composed of multiple elements. For example, 55 aliens are organized into 11 columns and a grid, 108 shield bricks grouped into columns and grids etc. Managing these objects separately would complicate operations like updating positions, rendering or checking collisions etc. Without an organized structure, individually updating or rendering for these numerous objects would become repetitive, inefficient and prone to errors. We needed a way to organize and manage groups of objects that behave together in a structed way.

Solution:

To solve this problem, we use the Composite pattern. For example, alien grid itself is a composite that contains 11 columns that are composites and each column has 5 individual aliens (leaf nodes). This pattern allows us to move all the aliens with synchronized behavior by just updating the grid itself. The shields also use the same approach. They are grouped in a hierarchy like the alien grid. Each shield grid has many columns and each column has its own bricks.



Pattern description

The Composite pattern allows us to create a tree-like structure where objects are organized into a hierarchy that can be navigated and manipulated uniformly regardless of whether they are individual elements (leaf) or collections (composite). Using this pattern, we can move the alien grid and when it moves the elements under the hierarchy such as columns and aliens would automatically move with the grid.

Key Object Oriented mechanics

The Component class provides an abstract interface with abstract methods enforced across both leaves and composites. Composite objects encapsulate collections of child components and managing their details internally. Both individual objects (leaves) and collections (composites) share a common base class, allowing us to handle them interchangeably.

How it is used in Space Invaders

The Composite pattern organizes game objects into manageable hierarchies. For example, the AlienGrid (includes 55 aliens and 11 columns) is structured as a Composite with child composites such as AlienColumn and leaves such as Squid, Crab or Octopuse, allowing the entire grid to be moved or rendered as a unit with using GridMoveCommand which updates the root propagates changes down the tree.

Similarly, the shield system with 108 bricks, uses a ShieldGrid (composite) containing ShieldColumn (composite) and ShieldBrick (leaf), making it easier to collisions.

Rather than checking every single node, we can handle interactions more efficiently by using the Composite Pattern.

OBJECT POOLS Pattern

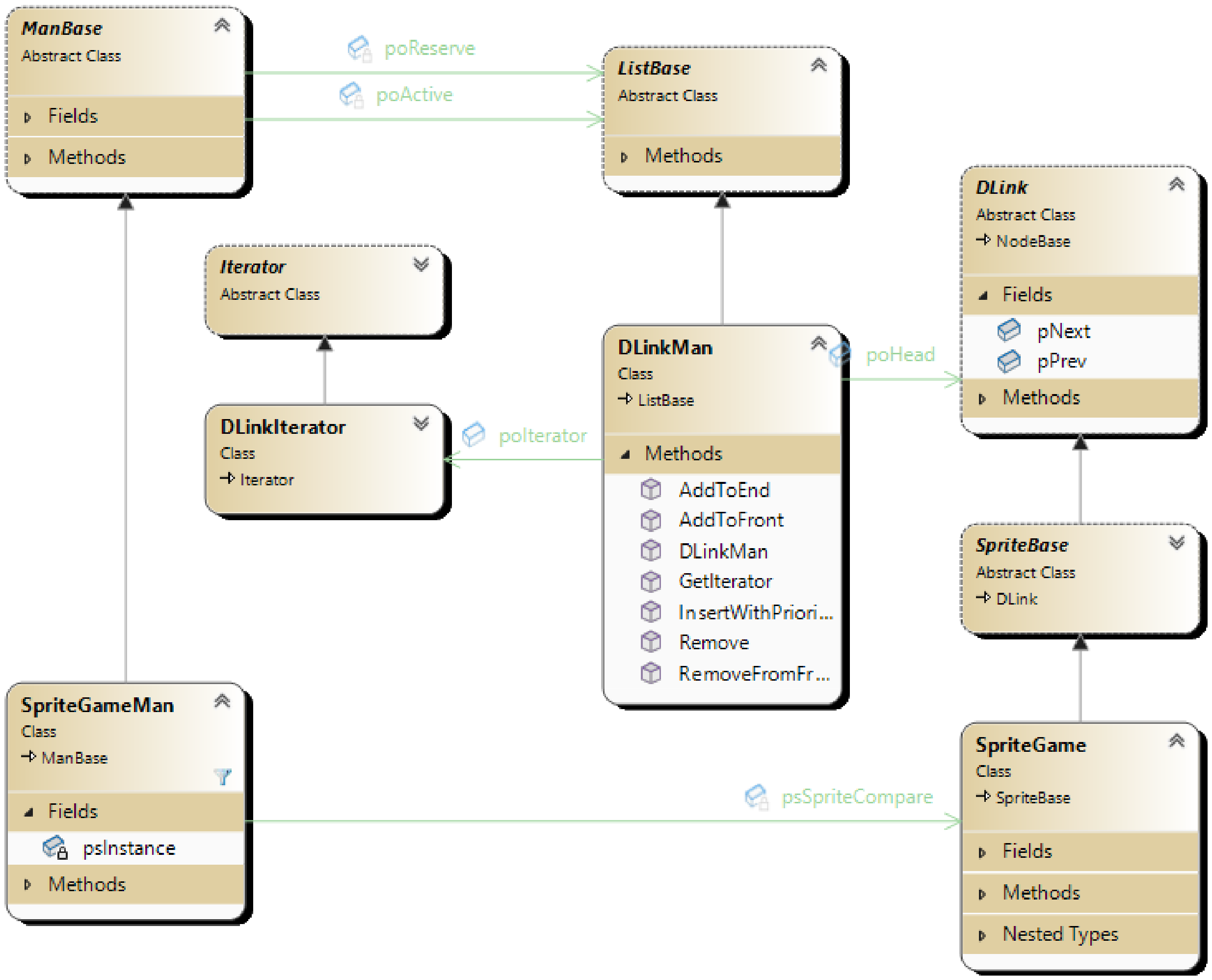
Challenge

Problem:

In the Space Invaders game, frequently creating and destroying game objects such as missiles and bombs etc. would lead to a performance and memory management issue. Continuously creating and destroying these objects during gameplay can lead to significant performance overhead if each instance is dynamically allocated and deallocated. We need a way to manage object creation and deletion efficiently.

Solution:

We implemented Object Pool pattern to our managers by using two lists which are a reserve list and an active list. Reserve list is for reusing the objects and active list is for the objects that are in use. Instead of creating a new one, when we need an object, we just take it out from the reserve list. Instead of deleting an object, we just remove it from the active list and put it to the reserve list and reset its state. Using the object pool, we minimize the object creation and improve the game performance.



Pattern description

The Object Pool pattern allows a system where a fixed set of objects are pre-allocated and maintained in the pool, ready to be borrowed when needed and returned when no longer in use, this helps to avoid the overhead of dynamic memory management and garbage collection. The pattern ensures an efficient way to manage object creation and deletion.

Key Object Oriented mechanics

The singleton managers that are accessed via Instance(), ensures a single pool manages the nodes globally. The Create() function initializes the pool with a base size and growth increment, pre-allocating nodes in to reserve list.

The pool encapsulates the creation and deletion logic, providing controlled access to its reusable objects. Objects are reused multiple times instead of being frequently created and destroyed, this significantly optimizes the performance.

How it is used in Space Invaders

In Space Invaders, we use Object Pooling in various managers such as SpriteGameMan, SpriteBatchMan, ImageMan, GameObjectNodeMan etc. The managers that the Object Pool pattern applied have both lists (active list and reserve list). The managers keep objects in a pool to be used whenever its needed and put back to the pool when we are done with it. This minimizes the performance cost for both creation and deletion of the nodes.

For example, GameObjectNodeMan manages game object nodes, the manager has a reserve list and an active list. When a game object node is needed, we first look at the reserve list, if there is one, we reuse it. If there are no game object node left, we create a new one. When we are done with the node, we simply remove it from the active list and put it back to the reserve list rather than deleting it.

This approach improves the game performance, ensuring smooth gameplay by reducing runtime object instantiation overhead.