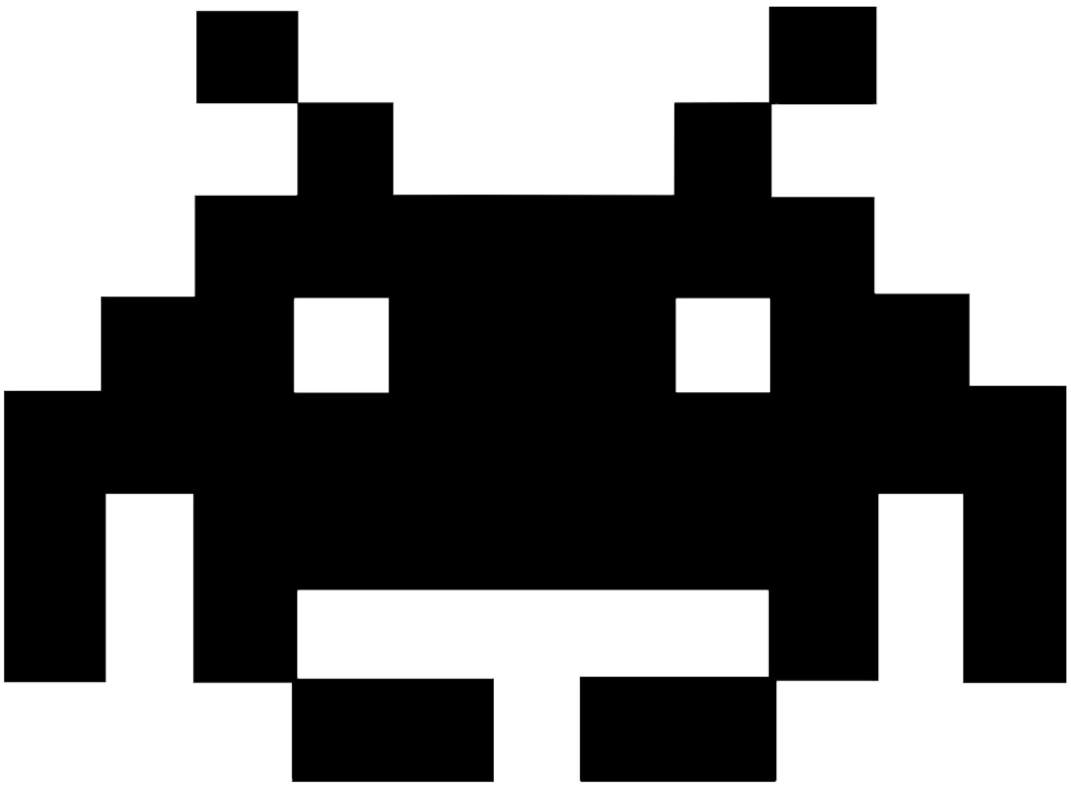
SPACE INVADERS DESIGN DOCUMENT



Submitted by:
ARYAN PATEL

TABLE OF CONTENTS

1. Introduction

1.1 Purpose of the Document

This document is designed to explain the architecture and the primary design patterns associated with the Space Invaders project. The objective of the document is to summarize the software engineering activities, design decisions, and implementation work which constitutes the architecture of the game system. It aims to articulate why specific patterns were chosen, how they contribute to solving design challenges, and their impact on the overall system.

This document is part of the final project handover contents and is important because it is an illustration of the practice in software engineering where documentation is created to promote ease of maintenance, expandability, and inter-team interaction. The document assures that the intended audience is able to comprehend the project's design and mechanics in such a way that subsequent modifications and extensions are facilitated.

1.2 Project Overview

The Space Invaders project is an implementation of the classic arcade-style shooter where the player controls a ship that moves horizontally along the bottom of the screen while shooting missiles to destroy descending alien invaders. The game includes various game states, such as the start menu, active gameplay, and game over screens, and utilizes a state management system for smooth transitions.

Developed using Azul.Engine, a game framework built for efficient real-time rendering and system management, the project follows a data-driven architecture to enhance performance and maintainability. To keep the game modular, flexible, and optimized, various design patterns have been implemented, ensuring smooth functionality while maintaining a consistent 60 FPS gameplay experience.

1.3 Goals and Objective

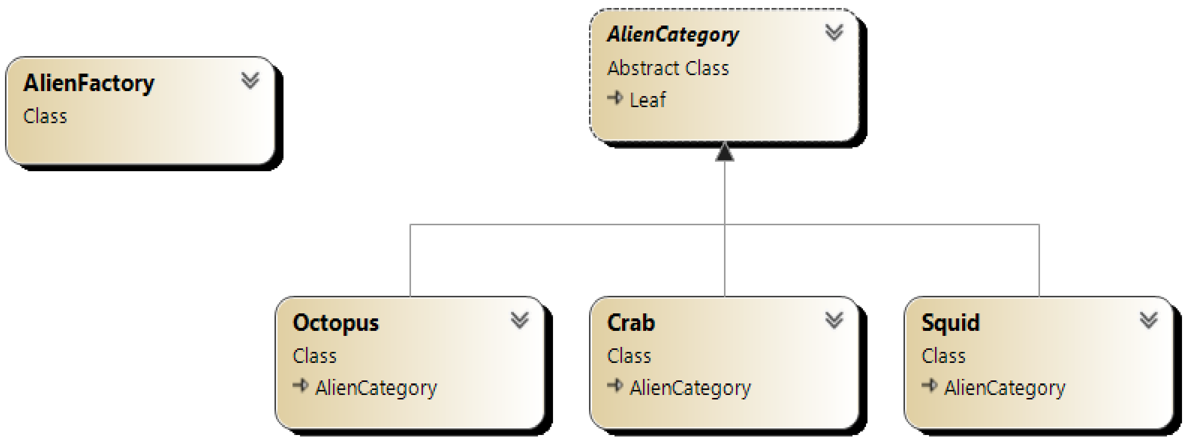
The primary goal of this document is to present the design decisions made during the development of Space Invaders, focusing on how software architecture principles and design patterns enhance the system. This document explains the high-level structure of the game, detailing components such as scene management, collision handling, rendering, and input processing. It also demonstrates the application of multiple design patterns, including Factory, Observer, Command, State, Proxy, Composite, Strategy, Visitor, and Object Pooling, and how they contribute to the flexibility and maintainability of the project.

2. Design Patterns

2.1 Factory Pattern

In Space Invaders, the game needs to spawn a lot of aliens throughout the gameplay, and creating them manually every time would be a hassle. Instead of writing the setup for each alien—assigning sprites, positioning them, and registering them for rendering and collisions—the game uses the Factory Pattern to handle everything in one place. The Alien Factory takes care of creating different types of aliens, like Squid, Crab, and Octopus, making sure they are set up properly before being added to the game.

By using a factory, the game doesn't have to repeat the same setup code all over the place. Instead of worrying about assigning sprites, setting positions, or adding aliens to different managers every time, the game just asks the factory for an alien, and it takes care of the rest. This makes the code cleaner, easier to manage, and simple to update if anything about alien creation needs to change, it only has to be updated in the factory, not everywhere in the game.



The Factory Pattern is a way to create objects without directly calling their constructors. Instead of making objects manually, the factory decides what kind of object to create and returns it**.** This is useful when the exact type of object isn't always known in advance, or when there's a standard process for setting up different objects.

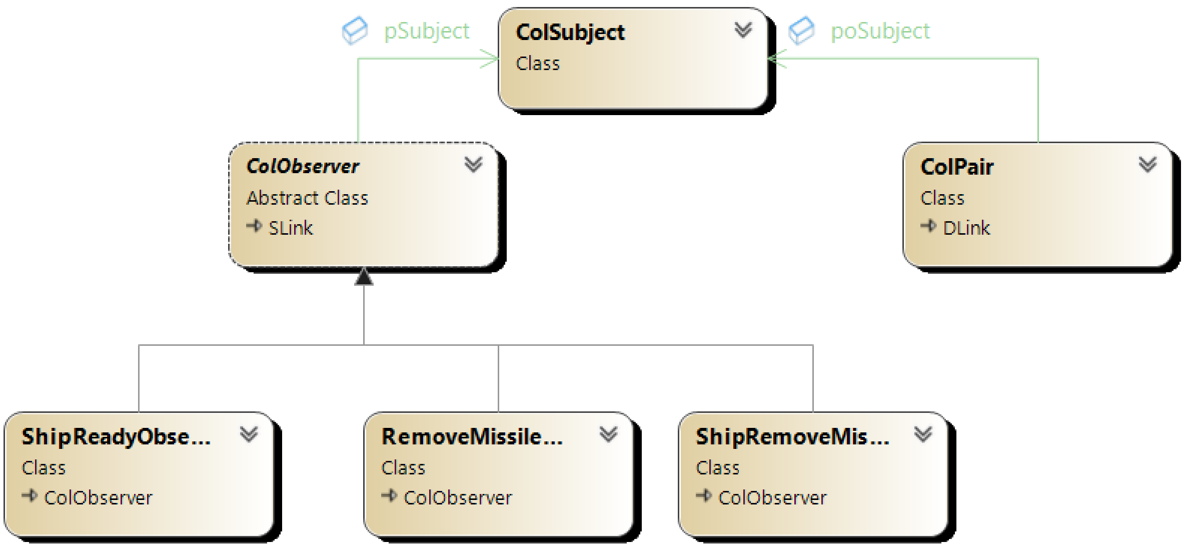
From an object-oriented perspective, using a factory keeps object creation separate from game logic, making the code more organized and easier to maintain. It also allows for polymorphism, meaning the factory can return different types of aliens without the game needing to worry about the details. Plus, decoupling the creation process means that the rest of the game just asks for an alien instead of handling the setup itself.

In Space Invaders, the Factory Pattern is used for both aliens and shields. The Alien Factory takes care of creating Squid, Crab, and Octopus aliens, setting them up with sprites and registering them for rendering and collision detection. The **Shield Factory** does the same for shields, building them out of multiple bricks in the right positions. This way, both aliens and shields are **created consistently and efficiently** without unnecessary repetition in the code.

2.2 Observer Pattern

In Space Invaders, different parts of the game need to respond to events like collisions, player input, and state changes without being directly tied to each other. If every event had to be handled in the same place it was detected, the code would quickly become a tangled mess, making updates and changes much harder. The Observer Pattern fixes this by letting objects subscribe to events and react when they happen, keeping everything modular and easy to manage.

A big challenge in the game is handling collisions. When a missile hits an alien, several things need to happen at once—the alien should be destroyed, the missile removed, the score updated, and maybe an explosion effect triggered. Instead of making the collision system handle all of this, observers listen for collision events and take care of their own tasks. This way, if we need to add a new reaction to a collision (like a sound effect), we can just add another observer instead of modifying existing code.



The Observer Pattern is a behavioral design pattern that establishes a one-to-many relationship between objects. It consists of a subject (which generates events) and multiple observers (which react to those events). The subject keeps track of all registered observers and notifies them whenever a relevant event occurs. This approach decouples event detection from event handling, allowing different parts of the game to react independently without modifying the core logic. Observers can be added or removed dynamically, making the system flexible and easy to extend.

The Observer Pattern also makes handling player input much simpler. Instead of checking for key presses in multiple places, an input manager keeps track of them, and different observers handle things like moving the ship or firing a missile. This makes adding new controls or mechanics easier since everything stays separate and organized.

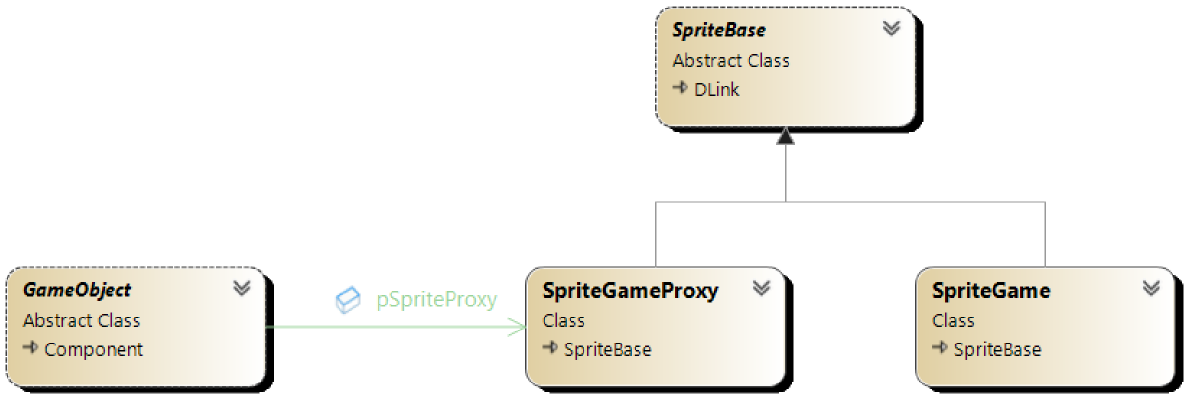
Another key use is in managing game states, like switching the ship's missile state between "ready" and "flying." When a missile is fired, an observer updates the ship's state. If it hits something, another observer resets it so the player can shoot again. This keeps everything in sync without cluttering the ship’s main logic.

Using the Observer Pattern keeps Space Invaders flexible and easy to expand. Instead of constantly tweaking core systems, we just add new observers when needed. This makes the code cleaner, easier to maintain, and more in line with good object-oriented design, where each part of the game only handles what it needs to.

2.3 Proxy Pattern

In the Space Invaders project, we handle a variety of game objects, such as Aliens and Shields, each with their functionality that primarily differences in the screen coordinates. For instance, the game contains 55 aliens divided into 11 Squids, 22 Octopuses, and 22 Crabs and each of them needs rendering and management separately. Direct management of all individual objects simultaneously is not optimal in terms of resource consumption. We need an approach that provides a means for identifying distinct sprite coordinates (x,y) while allowing to management of game sprites.
The Proxy Pattern is used to solve this by separating the actual sprite data from the individual objects representing aliens. Instead of every alien having its own full sprite, they only get a lightweight proxy that just holds position data. The proxy then tells a shared RealSprite where to draw itself on the screen. So, when it's time to render, each proxy updates the position of the RealSprite and lets it handle the actual drawing. That way, instead of having 55 full sprite objects, we just have 3—one for each alien type (Squid, Crab, and Octopus)—and all the proxies just reuse them.

The Proxy Pattern acts as the middleman between game logic and the actual sprite. Instead of creating a full sprite object for every alien, we just make lightweight proxies that store only what's unique like their position(x, y) and scale(sx, sy). Now, every time a proxy needs to update or render, it pushes its data (like position, and scale) to a single shared RealSprite before calling any methods. This way, it looks like each alien has its full sprite. But in reality, they all share the same few heavy-duty sprite objects. One RealSprite for multiple Proxies, All proxies take turns using the same RealSprite, and each time a different proxy needs it, it just updates the RealSprite's position before drawing. Since many proxies share the same RealSprite, every time a proxy needs to render, it pushes its position to the RealSprite first. That way, the RealSprite is always showing the correct state for whoever is using it at that moment. If another proxy needs its next frame, it just updates the RealSprite again before drawing. This keeps things consistent without duplicating big sprite objects.



The whole idea behind the Proxy Pattern is that it acts like a middleman between the game logic and the actual sprite. The proxy only knows its (x, y) position and scale, and whenever it needs to update or render, it pushes this data into the RealSprite. Since multiple proxies can share the same RealSprite, the game only needs to manage a few actual sprites while still making it look like each alien is moving independently. When a proxy needs to draw, it quickly updates the RealSprite's position and tells it to render, so everything stays in sync.

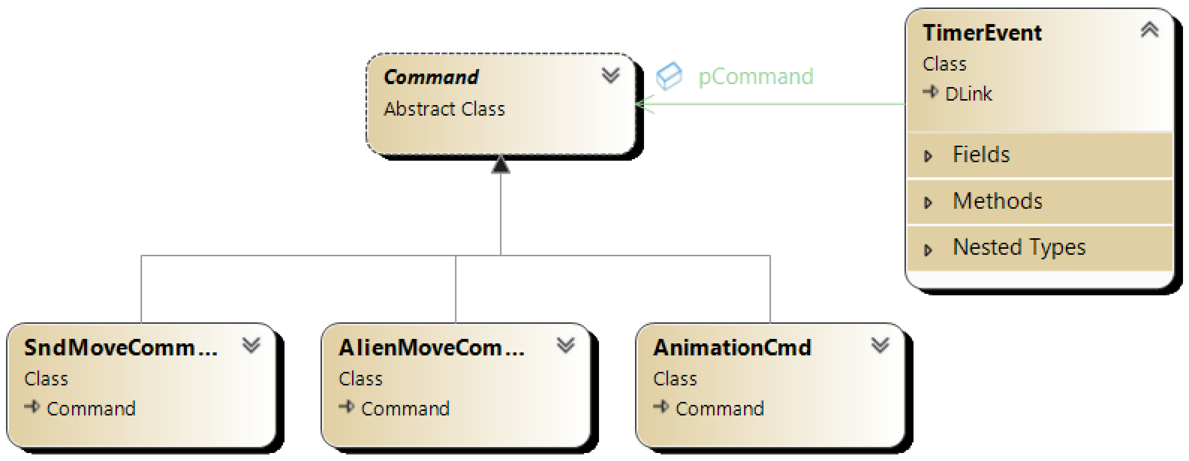
The same approach is used for shields. Shields are made up of multiple bricks, but a lot of those bricks are exactly the same. Instead of giving each one a full sprite, they work the same way as aliens—just a bunch of proxies pointing to a few shared RealSprites. This means the game doesn't waste memory creating separate sprites for every single brick while still allowing each one to be destroyed when hit.

This pattern makes a huge difference in Space Invaders because it keeps things way more optimized. Instead of filling up memory with unnecessary sprite objects, the game only holds onto a few, and the proxies handle all the positioning. It also makes managing animations super easy—since all proxies share the same RealSprite, updating the animation on one automatically applies to all of them. Overall, the Proxy Pattern is what keeps the game running smoothly without overloading it with too many objects.

2.4 Command Pattern

In Space Invaders, handling games actions like shooting, alien Movement, changing the images for animation of alien march, and timed events directly in the game loop would create messy, hard-to-maintain code. Also it has to be in the sorted order, so using global timer each event has to trigger at certain time. The game needs a way to schedule and trigger actions dynamically without cluttering the core update loop.

There comes the Command Pattern. It is used to manage game actions efficiently without cluttering the game loop. Instead of handling everything manually un update functions, the game uses a timer system combined with commands to execute actions at precise moments. The timer manager keeps track of scheduled events and ensures they run exactly when needed. Events like alien movement, missile firing, or animation updates are handled by commands, which are small, reusable objects that encapsulate specific actions. Each event is associated with a command, and when the timer reaches its scheduled time, it executes the command without needing to check conditions manually in every frame.



The timer system in Space Invaders efficiently schedules events using a priority queue, ensuring that actions occur precisely when needed. Instead of scanning a list and sorting it every frame, events are inserted in sorted order based on their trigger time, which means the earliest events are always processed first. This method significantly improves performance, as the system does not need to check every event manually. When the game updates the timer, it processes only the events whose trigger times have been reached, allowing it to skip unnecessary checks for events scheduled later. This approach reduces overhead and ensures that the game loop remains smooth, preventing delays caused by redundant event checks.

The Command Pattern is a behavioral design pattern that encapsulates a request as an object, allowing it to be queued, scheduled, or even undone. In Space Invaders, this means that actions such as playing sounds, animating sprites, or managing collisions are not directly coded into the game loop. Instead, commands act as independent units of behavior that the game can execute at the right moment.

From an object-oriented perspective, Instead of having a single massive update function handling every game event, commands encapsulate these behaviors into separate objects. The Timer Manager holds a queue of events, each linked to a command object, ensuring that actions occur when scheduled. This setup follows encapsulation by keeping command logic separate from the core game systems.

In Space Invaders, the Command Pattern is used extensively for timed actions. Alien movement, sound effects, and animations rely on commands scheduled within the timer system. When an alien needs to move, instead of constantly checking for movement in the main loop, a movement command is scheduled and executed at the right time. Similarly, explosions and missile effects use commands to trigger animations.

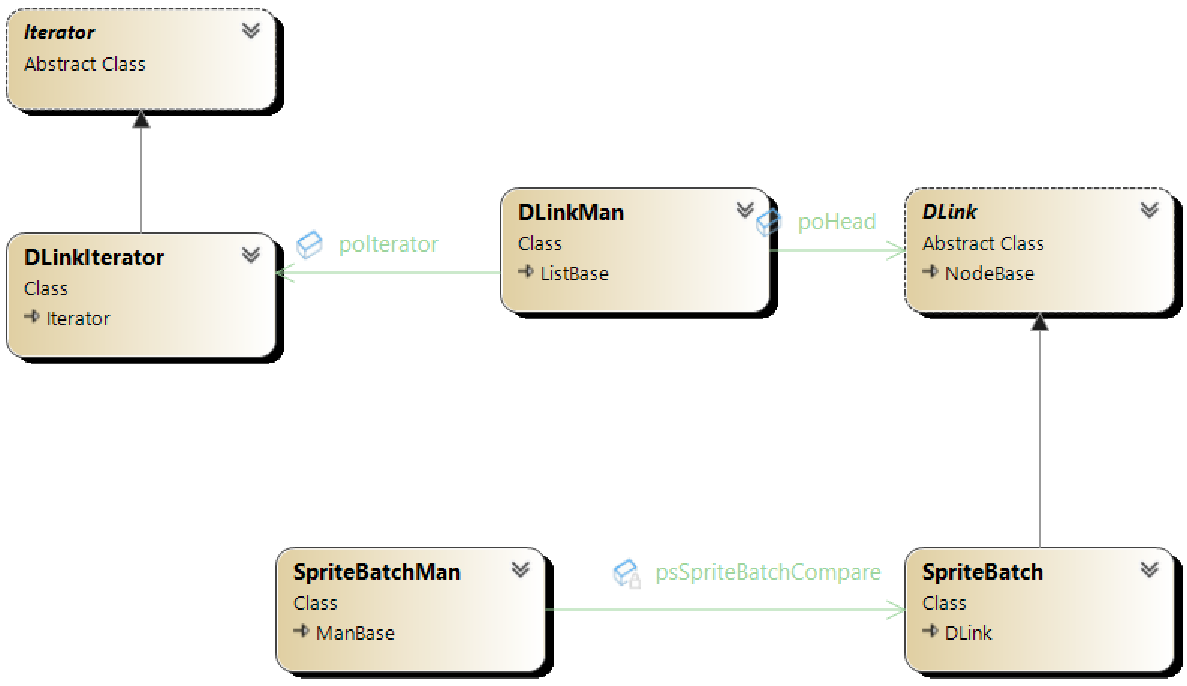
2.5 Iterator Pattern

In *Space Invaders*, various game systems rely on collections of objects that need to be traversed, updated, or processed in a structured way. These collections are managed in different data structures such as linked lists (DLink, SLink) or trees (composites). The challenge arises in how these objects are accessed without making the code tightly coupled to the underlying data structure.

The iterator pattern is all about making it easy to go through a collection of objects without worrying about how they're actually stored. In Space Invaders, different managers hold game objects like aliens, missiles, and shields, and instead of writing separate code to loop through each kind of structure, the iterator handles all of that behind the scenes. This means whether the objects are stored in a linked list, tree, or some other structure, we can still access them in the same way using the iterator's methods like next(), first(), and isDone().

One of the biggest advantages of using an iterator is that it hides the actual structure of the collection. Let's say today we're using a linked list to store our game objects, but tomorrow we decide a tree would be better for performance. If we were manually looping through everything using next pointers, we'd have to rewrite a bunch of code. But with an iterator, as long as the collection provides the right interface, we don't have to change anything in the parts of the game that use it. The iterator just takes care of it.

Another cool thing about iterators is that we can have different types depending on how we need to move through the collection. A basic iterator just goes through everything one by one, but if we're working with a tree structure, we could have a depth-first iterator that moves down before moving across, or a reverse iterator that processes things from bottom to top.



One tricky problem when looping through a collection is when we need to delete an element while iterating. Normally, if you remove an item from a linked list while you're. in the middle of looping through it, you might break the loop because the pointer moves to a deleted object. The solution? The erase() function inside the iterator. This lets us safely delete an object and update the links so the iteration can continue smoothly.

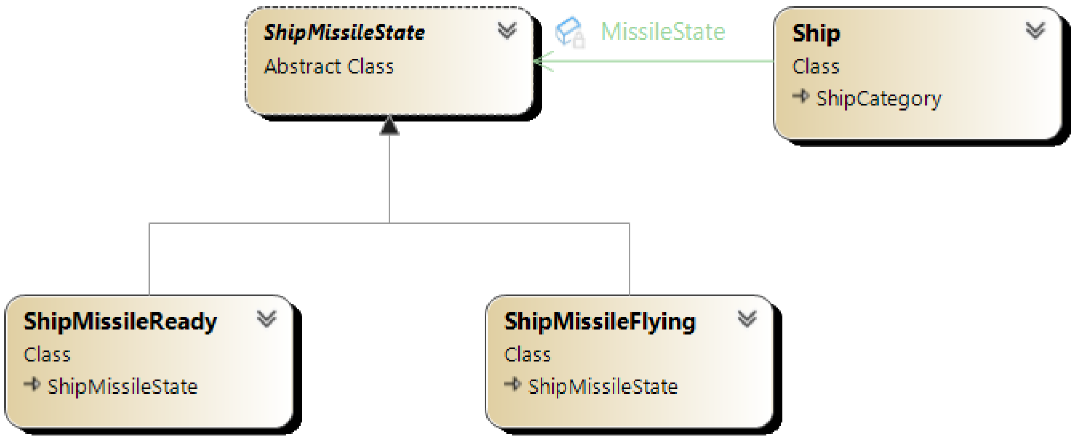
From an object-oriented perspective, the Iterator pattern aligns with the single Responsibility Principle by separating iteration logic from collection management. The Open-Closed Principle is also maintained, as new traversal strategies can be introduced without modifying existing collections. Instead of exposing internal details, each collection provides an iterator that allows controlled access to its elements.

In Space Invaders, iterators are used in multiple places. Managers like SpriteBatchMan and GameObjectNodeMan use iterators to go through all their stored objects instead of manually accessing them. The composite pattern (which organizes things like alien grids and shields) also benefits from iterators since we need a way to move through all elements inside those structures. Using an iterator, we can process every object in the grid without worrying about how it’s actually stored.

2.6 State Pattern

In Space Invaders, various game objects undergo dynamic behavioral changes based on their internal state. A ship can be idle, moving left, moving right, or destroyed. A missile can be ready to fire or already flying. The game itself transitions between menus, active gameplay and game-over screens. Without a proper structure, handling these behvaiors could lead to large conditional blocks (if-else or switch statements) inside game objects, making them harder to maintain, understand, and extend. When an object's state changes, its behavior must adapt accordingly, and managing these transitions manually can become error-prone.

The State pattern addressed these challenges by encapsulating state-specific behaviors into separate classes. Instead of having game objects contain all possible behaviors and manually check which one to execute, each object delegates its behavior to its current state. A context object (such as a ship, missile, or game scene) maintains a reference to its current state and delegates state-dependent actions to it. Each state class implements a common interface and defines specific behavior without modifying its core logic.



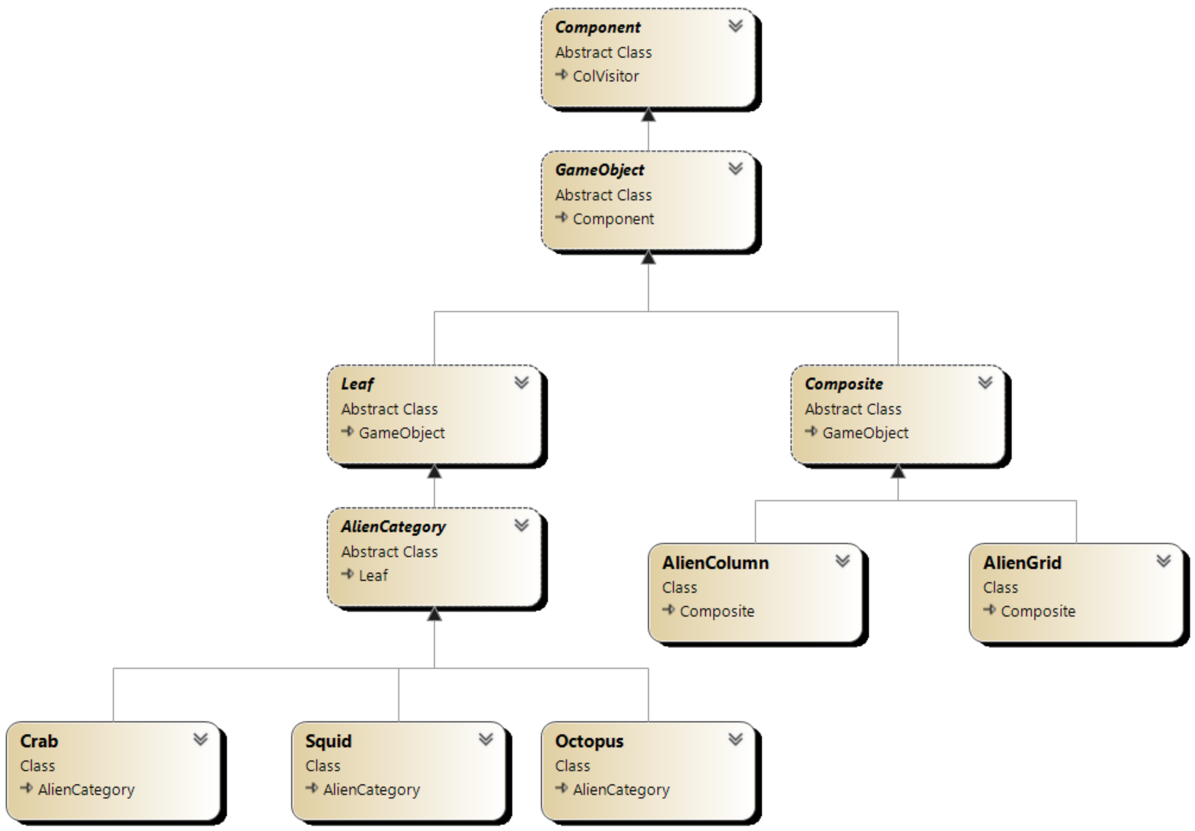
The state pattern is prominently used in Space Invaders to manage various game elements that change behavior over time. One key application is in game scenes, The game transitions between scene Start, scene Play, scene over stats. A scene Manager holds a pointer to the current scene state, such as SelectScene, PlayScene, or GameOverScene. The game loop calls update and draw on the active scene state, allowing each state to handle its behavior independently.

Another crucial use is in ship and missile behavior. A ship can be in different movement states (e.g. ShipMoveLeft,ShipMoveRight) and missile state (e.g. ReadyState, FlyingState). Instead of the ship checking whether it can shoot or move, it delegates actions to its current state, which determines the valid response. If a missile is in the FlyingState, the ships' state prevents another shot until it transitions back to ReadyState.

2.7 Composite Pattern

Managing multiple game objects in Space Invaders presents a challenge due to the hierarchical nature of certain game elements. The game includes walls composed of multiple segments, aliens formations arranged in girds, and bombs that exist withing bomb groups. The issues arise when performing operations such as updating, drawing, and collision detection.

The Composite Pattern addresses these challenges by structuring game objects into tree-like hierarchies, where both individual game objects (leaves) and composite groups (composites) follow a common interface. For example, take the AlienGrid. Aliens are organized into individual columns, with each column containing five aliens (two Octopuses, two Crabs, and one Squid). All these columns are then organized into a Grid. The entire grid consists of 11 columns, each holding multiple aliens. In this hierarchy, the Grid acts as a composite, which holds Columns (also composites), and each column contains individual aliens (leaves). When performing the alien marching movement, we first call the move function on the AlienGrid, which then iterates through all the columns and moves each alien accordingly. Additionally, when a missile hits an alien, the collision detection starts at the Grid level. If a collision is detected, it then checks which column was hit. After identifying the correct column, it further determines which specific alien within that column was impacted.



The Composite Pattern is useful for hierarchical game elements like alien Grid formation and also shields. It allows both single objects and groups of objects the same way. The GameObject class acts as the base interface, with Leaf nodes representing individual objects (like a single alien or single brick) and composite nodes representing groups (like an alienColumn or AlienGrid).

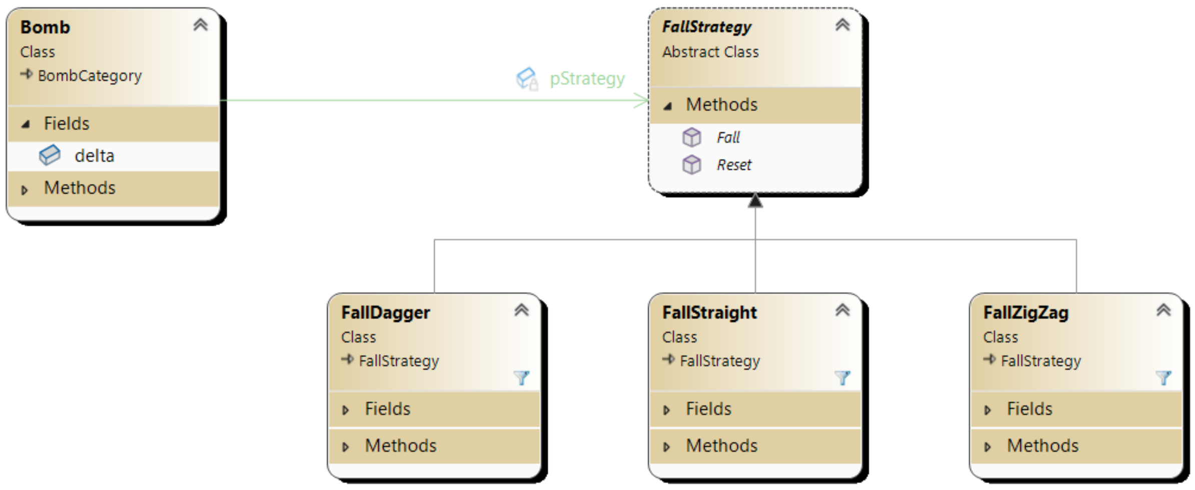
From an object-oriented perspective, this design leverages inheritance and polymorphism to ensure that both leaves and collections of leaves follow the same interface. The AlienGrid doesn’t need to know if it contains individual aliens or entire columns, it just calls the same methods, and each objects take care of its own logic.

In our Game, the composite pattern is used in multiple places. The AlienGrid manages the alien formation, allowing movement and collisions to be handled at the grid or column level rather than at the individual Alien level. The same goes for shields as well.

2.8 Strategy Pattern

In Space Invaders, we have three different types of Bombs which are randomly falling from the aliens. Some bombs fall straight down, while others follow a zigzag or dagger-like path. For that we have to use if or switch statements in a single class with multiple conditions on how to check bomb should fall.

To overcome this problem strategy pattern gives a solution, which allows you to define each behavior in a separate strategy class. The bomb class doesn’t need to know the details of how to fall. Instead, it has a pointer to a FallStrategy object, which defines how the bomb should move. When the Fall() method calls, if bomb is set to fall straight, it uses the FallStraight strategy; if it’s supposed to zigzag, it uses the FallZigZag strategy, and so on. This way, the bomb class stays clean, and if we ever need to add a new falling pattern, we just create a new strategy without touching the existing code.



The Strategy pattern is all about letting an object switch between different behaviors without changing its core logic. Instead of stuffing multiple ways of doing something into a single class, this pattern moves each behavior into its own class.

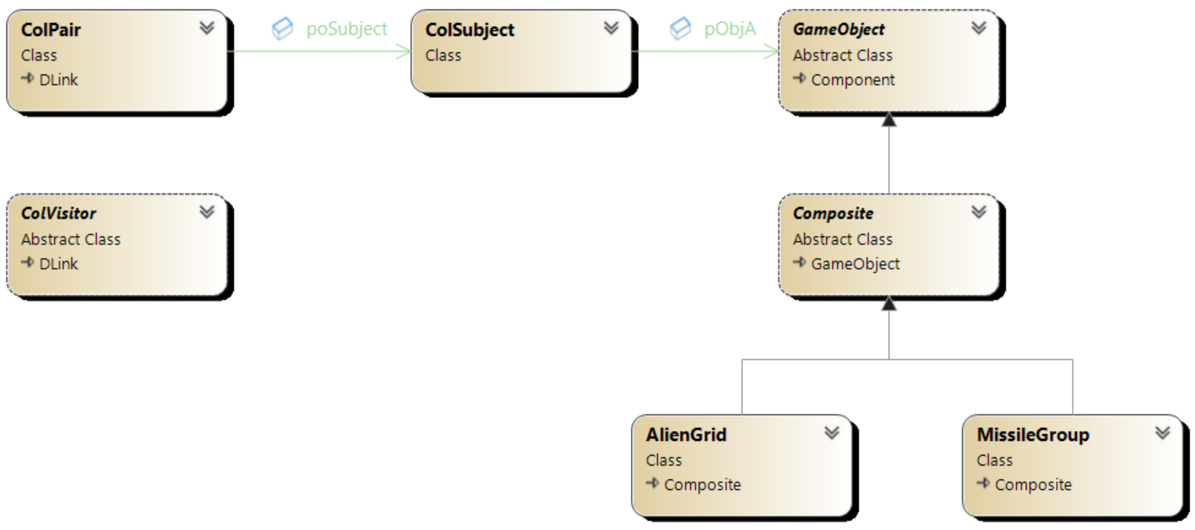
From an object-oriented perspective, this keeps everything modular and follows the Open/Closed principle new behaviors can be added without modifying existing classes. Since all strategies inherit from the same FallStrategy base class, the bomb doesn't care what kind of fall behavior it's using; it just calls Fall(), and the correct strategy handles the movement.

In Space Invaders, this pattern is mainly used for bomb movement.

2.9 Visitor Pattern

In Space Invaders, handling collisions between different game objects is a crucial part of the game logic. The main problem arises when we have to determine what specific objects are involved in a collision and execute the appropriate response. Usually for this kind of problem we will think of polymorphism allows for basic collision between objects based on their base class, but here is the problem with using polymorphism. For example, if a missile collide with another object, the system needs to determine whether the object is an alien, a shield brick or another missile. Again we can't use the switch statement of any conditional logic for every possible combination.

The visitor pattern solves this problem using double dispatch, which allows us to determine exact types of which two objects are collide with each other. Instead of using conditional statements to determine collision types, the pattern introduces an accept method in each game object that takes a visitor as an object. The visitor object then determines what kind of collision by calling a visit method for the appropriate object type. In this manner, the appropriate collision response is called without the need for long chains of conditions. Also, we can encapsulate collision-handle logic within visitor classes, new collision responses can be added without modifying existing game objects.



The Visitor pattern helps manage interactions between different game objects without cluttering their code with collision logic. Instead of each object trying to figure out what it just collided with, the Visitor pattern lets a separate class handle the logic.

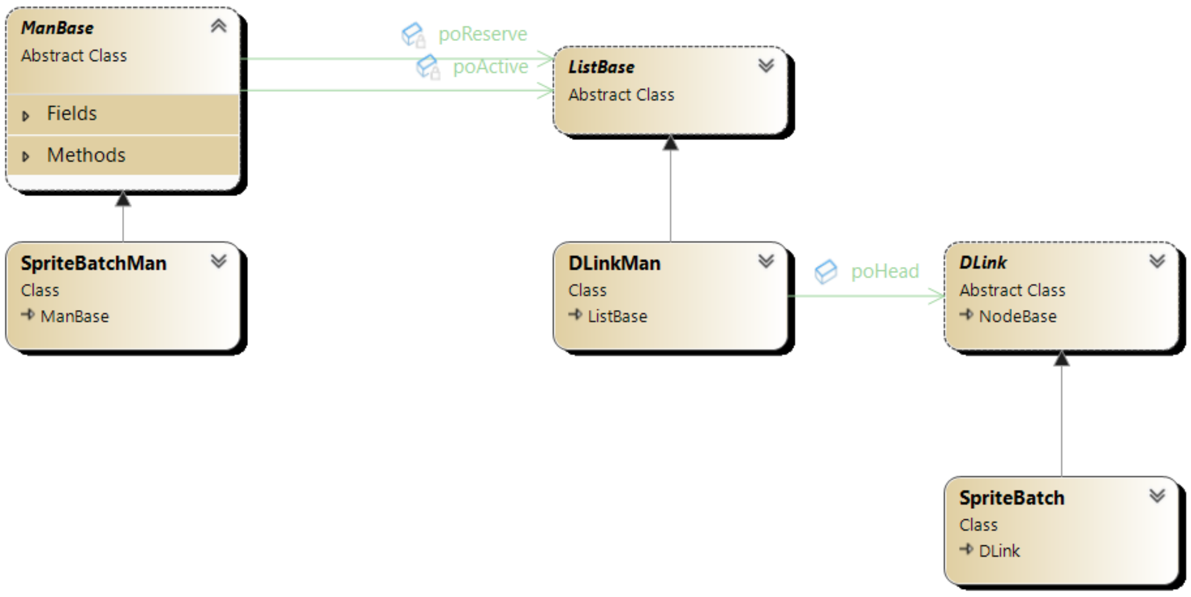
The Visitor pattern keeps collision handling clean and organized by using encapsulation, inheritance, and polymorphism. Each game object has an accept method that takes a Visitor, which then determines what to do based on the object type. Instead of cluttering game objects with collision logic, the Visitor class takes care of it, making it easy to add new collision behaviors without changing existing code. This follows the open/closed principle, allowing new game objects or interactions to be introduced by simply adding new visit methods. Polymorphism ensures the right visit method is called at runtime, keeping the system flexible and scalable while keeping game objects focused on their main tasks.

We are using visitor pattern in various part to detect the collision between objects. When two objects collide, one calls its accept method and passes the other object as a Visitor. The Visitor then figures out what kind of object it's dealing with and calls the correct visit method to handle the collision. For example, if a missile hits an alien, the missile calls visit(Missile), and the Visitor checks that it collided with an alien, triggering the logic to destroy the alien and remove the missile. For bigger collisions, like MissileGroup and AlienGroup, the Visitor helps figure out exactly which alien got hit and applies the right effects, like updating the score and removing the alien. This is especially useful since different types of aliens—Squids, Crabs, and Octopuses—might have slightly different behaviors. The MissileGroup accepts a CollisionVisitor, which then calls visit(Alien) and executes the correct collision response.

2.10 Object Pooling Pattern

In Space Invaders, we are using Images, Boxes, textures, sprites and many more things. Which we are treating as objects. Objects are frequently created and destroyed, such as missiles, aliens, shields. Every time an object is instantiated using new, memory allocation take places, which is costly in terms of cycles and performance. Similarly, when an object is destroyed, memory allocation occurs, which leads to memory fragmentation over time. Repeated allocation and deallocation causes slowdowns, especially when dealing with a large number of objects that have short lifespans.

Object Pooling provides a structured way to manage objects which can be reuse instead of creating and destroying them repeatedly. There are two list manager created when the manager is created an active list and reserve list. Active list in initially holds nothing, while reserve pool have fixed number of nodes created. Whenever an node is needed it is removed from the reserve list and inserted in the front of the active list. Once it's no longer needed in active list instead of deleting it, the game resets it and puts it back in the reserve list. So it can be used later. This allows to keep reference counter always 1 for automatic garbage collection.



A manager class handles this process, keeping track of which objects are available and which are in use. When a new object is requested, the manager first checks the reserve pool. If there's one available, it's reinitialized and returned. If the pool is empty, a new object might be created. When an object is done being used, instead of removing it completely, the manager cleans it up and puts it back in the pool, so it's ready to go next time. This helps prevent lag and keeps memory usage under control.

To make sure objects don't keep any old data from their previous use, the game follows a wash and set system. When an object is returned to the pool, it first goes through a wash step, which clears out any leftover information. Then, when it's needed again, it goes through a set step, which updates it with new values. This makes sure the object behaves like new every time it's reused, without the need to constantly create and destroy new ones.