SE 456 Space Invaders Design Document

Winter 2020 - CDM - DePaul University

Mahmoud Salah Thabet Shehabeldin

Introduction:

This document is intended to explain how the Space Invaders game have been implemented using Object Oriented Design Patterns. It will include the encountered challenges while implementing the game along with the detailed design solutions to tackle them. In addition, this document will discuss each design patten used, how it is implemented, and why it is useful. Finally, UML diagrams will be provided to showcase the implementation mechanics.

Design details:

I will list below all the Design Patterns used in the Space Invaders game with detailed analysis of each one.

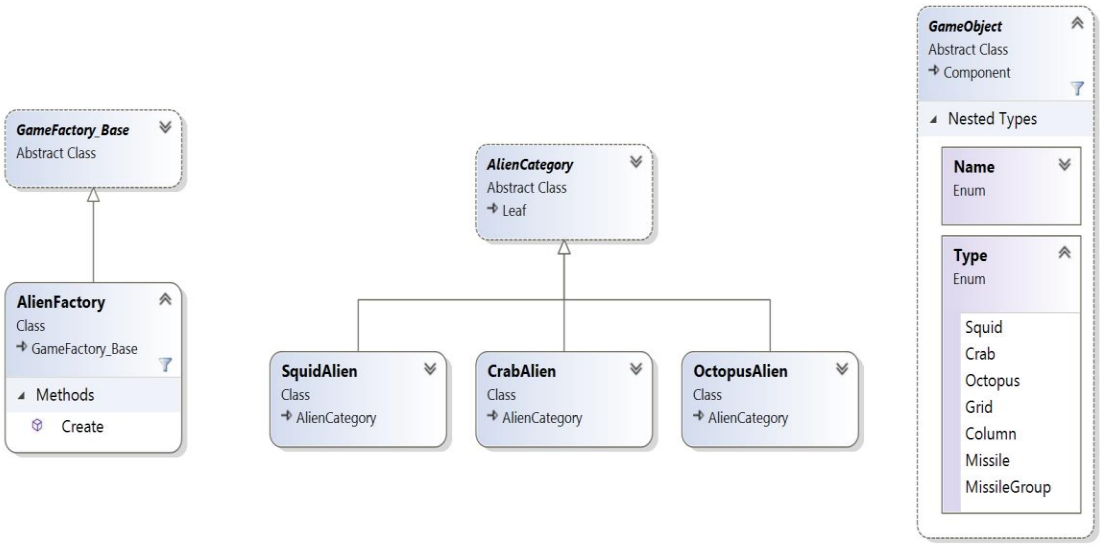
1. **Factory:**

   Problem: I had the issue of creating 55 aliens without worrying about the actual implementation of each alien type (i.e.: Squid, Crab, Octopus). In addition, I needed to create the aliens in the game engine through one standard path.

   Solution: To easily create objects of different types through a mutual interface, I used the Factory pattern.

   Pattern description: Factory pattern is intended to create objects without exposing the instantiation logic to the client (i.e.: The Game Engine in Space Invaders). Additionally, it provides a common interface in order to refer to the newly created objects.

   UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:
   - **Alien Factory:** This is a concrete factory class for creating all types of aliens.
   - **Alien Category:** This is the interface for all the alien types.
   - **Squid Alien, Crab Alien, Octopus Alien:** These are the concrete classes that will be instantiated when calling the Create() method in the Alien Factory class.
   - **Type enumeration in Game Object:** This is an enumeration to list the different types of aliens that can be created.



Object Oriented mechanics:

The concrete Alien Factory class has a Create() method. This method will take the type of alien as an input parameter. Accordingly, it will look up for the requested alien type and instantiates an object of its concrete alien class. For example, if the input alien type is Crab, the method will create a new Crab Alien object and return it to the invoker.

The concrete alien type is abstracted to Alien Category, which is the base class for each concrete alien class. Each concrete class will override the construction of the Alien Category to create the new object with the appropriate attributes.

Pattern usage in the game:

In the Game Engine (i.e.: The client), I am instantiating only one object of the Alien Factory class. Later, I am invoking the Create() method of it multiple times and providing it with the needed attributes (i.e.: position X and position Y) along with the needed alien type (i.e.: Squid, Crab, Octopus). Thus, the game engine will process the returned alien objects to draw them on the screen at their X and Y points.

Pattern benefits:

1. This pattern allows me to use a standard method for creating alien objects without being exposed to the actual implementation of each alien type.
2. In case I needed to add a new alien type (e.g.: Scorpion), it will be significantly easy to implement that change. That is adding new concrete class inherited from the Alien Category abstract class, adding new item to the Alien Type enumeration, and tweaking the Create() method in the Alien Factory class for the new alien type.

2. Singleton:

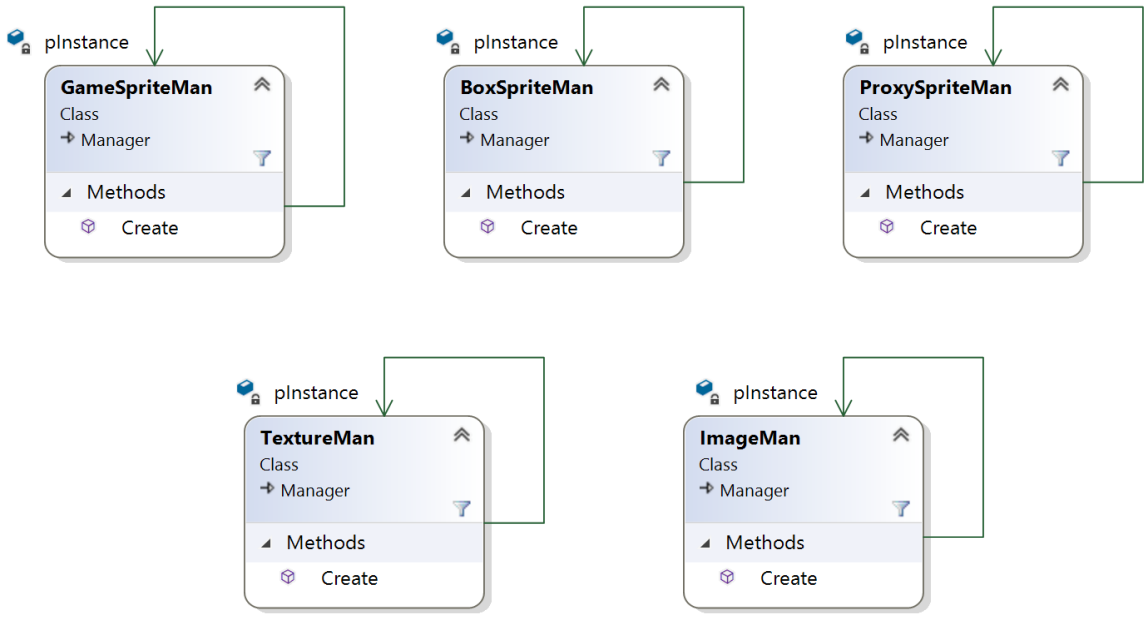
Problem: I had to manage and organize a huge amount of data with easy access from anywhere in the system's components.

Solution: To effortlessly access objects from a global unique point in order to ensure consistency among all the system's components, I used the Singleton pattern.

Pattern description: Singleton pattern is intended Ensure that only one instance of a class is created. In addition, it provides a global point of access to the object without calling the constructor each time it is invoked.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Game Sprite Manager:** This is for managing and accessing the game sprites.
- **Box Sprite Manager:** This is for managing and accessing the collision boxes of sprites.
- **Proxy Sprite Manager:** This is for managing and accessing the proxies of sprites.
- **Texture Manager:** This is for managing and accessing the different textures.
- **Image Manager:** This is for managing and accessing the sliced images into one texture.



Object Oriented mechanics:

A singleton class responsibility is to create only one instance of it and make sure that it doesn't allow any additional instances. Hence, it has a core method named Create() in which the instance is created. Moreover, it has a method named GetInstance() that will get the created instance and make sure it is never created again. The latter can be used anywhere to get the singly created instance.

Pattern usage in the game:

In the Game Engine, I use it for the global managers creation which clients can use to manage the needed data throughout the game execution. Each manager has a number for managing methods. For instance, the Image Manager has a method named Add() that adds images from the texture to the Images Tree. Afterwards, I can simply invoke that Image Manager anywhere to get the needed image using the Find() method.

Pattern benefits:

1. I have only one instance of each manager which I can use it globally without worrying about the construction of it.
2. Managers (i.e. Singletons) are acting like internal APIs that have standard access and outputs.

3. Object Pool:

Problem: I needed to implement the Aliens Grid animation which includes changing the X and Y positions for each Alien and its image when the grid is stepping right or left.

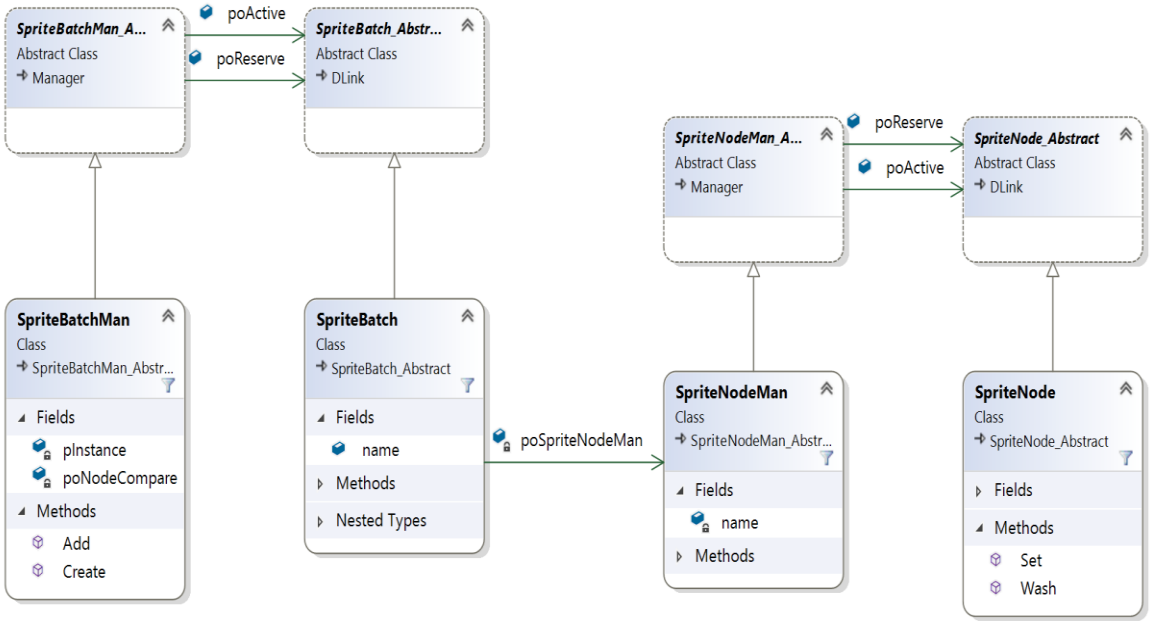
Solution: To avoid recreating new sprite objects with new positions and images on each grid step, which is extremely costly, I used the Object Pools pattern.

Pattern description: The main purpose of the Object Pools is to create a large batch of objects once, then share them for reusing. This technique is significantly fast and not costly as creating new objects each time a new one is needed.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Sprite Batch Manager:** A singleton for managing the sprite batch nodes.
- **Sprite Batch:** A batch node that consists of several sprite node managers.

- **Sprite Node Manager:** A pool for handling the sprite nodes.
- **Sprite Node:** The actual objects created in each pool ready for reusing when needed.



Object Oriented mechanics:

The sprite batch manager is responsible for creating the sprite batches which are used to control the sprite node managers. Hence, each node manager will include an initial number of sprite node that are clean and ready to be reused later.

The first time each sprite node is created, it will be attached to the sprite batch and the sprite node manager. Afterwards, the sprite manager will get the needed nodes through the sprite batch the change its attributes with the new values or clear them when it is marked for deletion.

Pattern usage in the game:

I used the same strategy mainly for the gird animation. First, I attach each alien sprite to the aliens' sprite batch and the aliens' sprite manager. Accordingly, when executing the animation, I fetch each alien sprite, change it position and image, then redraw the in the screen.

Pattern benefits:

1. As the game screen frame updates in millisecond, I believe this most appropriate way to apply the animation without having to improve the performance of expensive objects recreation.
2. This technique could be used for many other game objects, like the bombs animation while dropping.

4. Proxy:

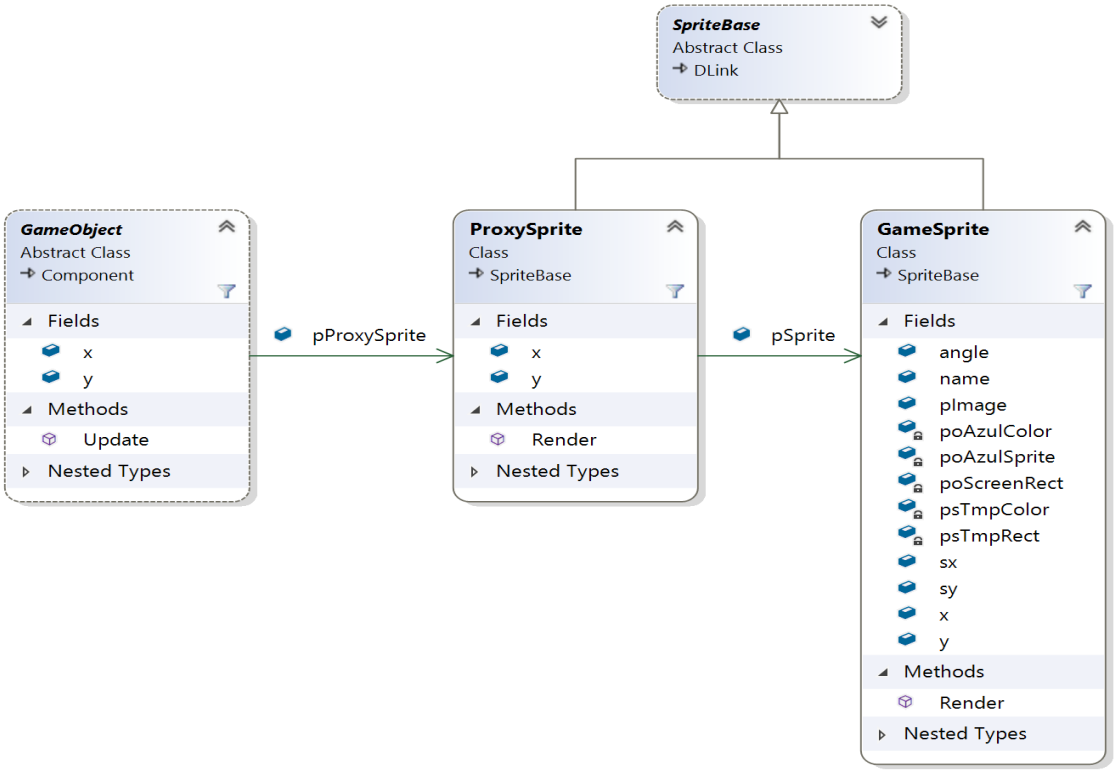
Problem: The aliens objects are heavy as each alien has lots of data values, which will result in an unnecessary extreme memory load while executing the aliens' animation.

Solution: I realized that there is a need of a lightweight interface of the heavy alien objects. Hence, these interfaces will expose the functions of the actual aliens but with lower cost. To implement such beneficial method, I used the Proxy pattern.

Pattern description: The proxies are placeholders that have references to the actual objects. These placeholders include some basic and commonly used properties of the same objects. When the clients need them, the proxies will come into play and will update the values or invoke the required behavior of the real objects.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Game Object:** The object of the game that changes frequently.
- **Proxy Sprite:** The interface of the actual game sprite object with the needed attributes.
- **Game Sprite:** The actual giant object which is referenced by the proxy.



Object Oriented mechanics:

A client object needed to update or invoke specific attributes/behaviors of a heavy object. Accordingly, the client will use the real object's interface (i.e.: proxy), then the proxy will execute the requested changes through the actual object.

Pattern usage in the game:

The image flipping behavior and position changing of the actual alien objects is delegated through the proxies. For instance, the aliens' grid is always moving in both direction (right/left), this requires two steps, updating the x and y values and then render on the screen. Therefore, instead of calling the real subject, the proxy will take this responsibility of doing both action through the reference to the actual object.

Pattern benefits:

1. Proxy can do multiple standard steps consistently applied to the actual objects.
2. If these steps have changed, it will take place into one specific place and affect all the calling clients.

5. Flyweight:

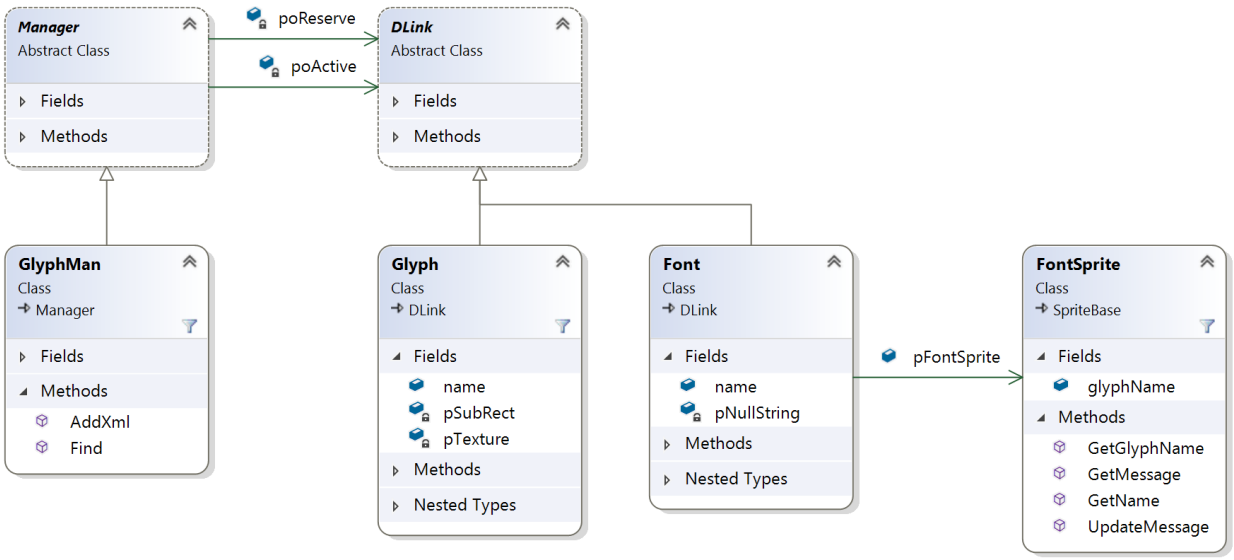
Problem: The font objects for the texts displayed on the screen are numerous and share the same size and representation. Loading each character from the texture while the game is running is really problematic.

Solution: Needed to find a way to load all the characters in a shared place once the game is loaded, then ruse them to draw the texts on the screen. The Flyweight pattern is the prefect match for that case.

Pattern description: The intent of this pattern is to create a shared space that contains a large number of objects which have part of their internal state in common where the other part of state can vary.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Glyph Manager:** A manager to load the huge number of objects.
- **Glyph:** A node for each character in the font texture (the unchangeable part).
- **Font:** A node that references the characters that might be updated later.
- **Front Sprite:** The sprite that used to get or set the Font's text.



Object Oriented mechanics:

The client will use the Flyweight Factory to fetch the required concrete Flyweight object. Therefore, the client can use the available operations to apply the necessary changes to the Flyweight objects.

Pattern usage in the game:

The Glyph Manager (i.e.: Flyweight Factory) loads the characters into one pool from the font texture. Then when specific characters are needed, a Font Manager will be used to create a Font node with the needed characters and link it with a Font Sprite. Later, when the client needed to fetch or update the Font's text, the Font Sprite will be the interface to apply these actions.

Pattern benefits:

1. This technique is significantly easy to use and effective.
2. Allows the reuse of ready-made objects without the need for recreation.

6. Composite:

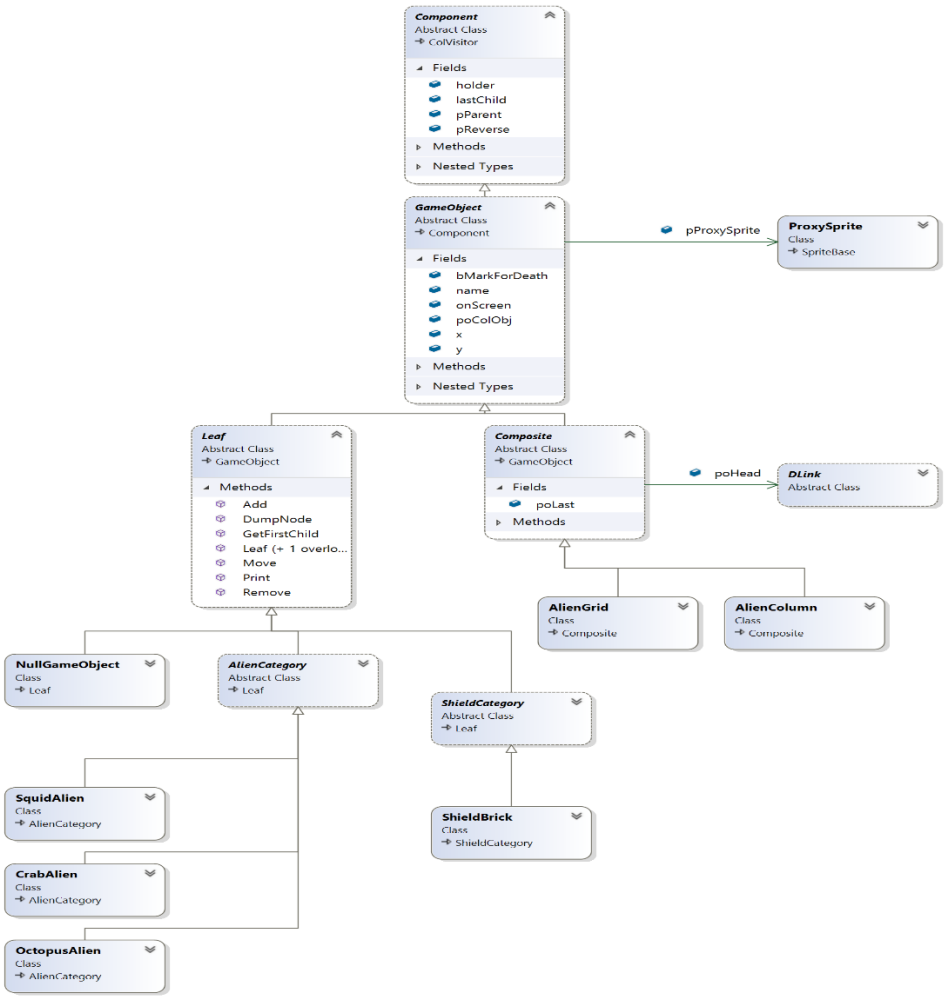
Problem: I have a hierarchy of objects the game objects that are either s composition of other objects or a single object. Both types need to be treated uniformly. For example, the aliens' grid consists of aliens' columns which includes a number of aliens.

Solution: To implement such tree structure correctly in order to be able handle them homogeneously, I used the Composite pattern.

Pattern description: A typical composite pattern consists of three blocks. First, the Leaf objects that are individual components with no children. Second, the composite objects which has many child components. Finally, the component interface which has the methods that requires both the Leaf and the Composite to implement them.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Component:** The components interface that both leaves and composites have to comply with.
- **Alien Grid and Alien Column:** The composite object types that has many children.
- **Shield Bricks, Squid Alien, Crab Alien, etc.:** The individual object types that have no children.



Object Oriented mechanics:

The client will get a reference to the tree structure through the component interface. Then the needed operations will be applied to the target component regardless the fact that it could whether a Leaf or a composite object.

Pattern usage in the game:

I am using the composite structure to control the Aliens' Grid, columns and the Aliens themselves while moving all of them in the same direction (Right or Left) using the same operations without checking if it's a Grid or an Alien.

Pattern benefits:

1. The client doesn't care about the types of the objects that executes the intended process.
2. Easy to scale and/or modify without affecting the clients.

7. Iterator:

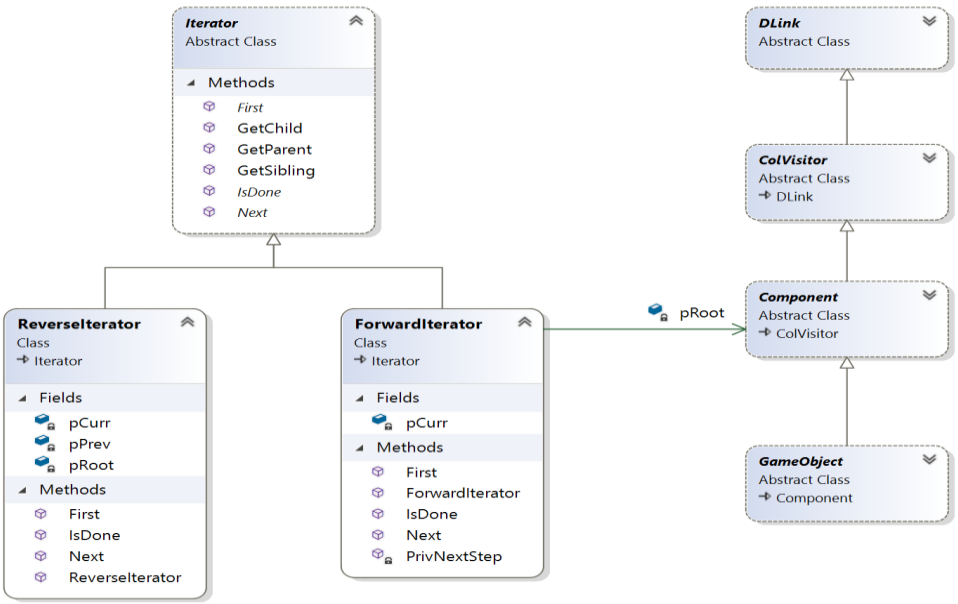
Problem: Sometimes I need switch between multiple iterate method on the Game Objects Trees using an unchanging method.

Solution: The Iterator pattern is the most appropriate resolution for such problem. This is because it provides a standard iteration interface for multiple iteration approaches.

Pattern description: An iterator delivers a way to sequential access to the objects in a collection without exposing its underlying representation.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Iterator:** An abstract of the standard iteration process.
- **Forward and Reverse Iterator:** Concrete containers for the actual implementation of the sequential iteration.
- **Component:** The collection which the iterator in looping on its items (e.g.: Linked List or Trees).



Object Oriented mechanics:

Any iterator always has three operations.

1. First(): Gets the first item in the collection.
2. Next(): Gets the next item of the currently selected item.
3. IsDone(): Returns a Boolean flag to indicate whether the end of the collection is reached or not.

Pattern usage in the game:

I am using this pattern to walk through each item in the Aliens' Grid in order to update the movement, the animation, and the bombs dropping.

Pattern benefits:

1. Changes in the storage mechanism of the items can be applied without affecting the clients.
2. Different iteration procedures can be provided within the same standard interface.

8. Command:

Problem: I needed to trigger a group of timed actions and change that timespan constantly. For instance, the grid has to drop a bomb from the bottom alien randomly within a particular time period.

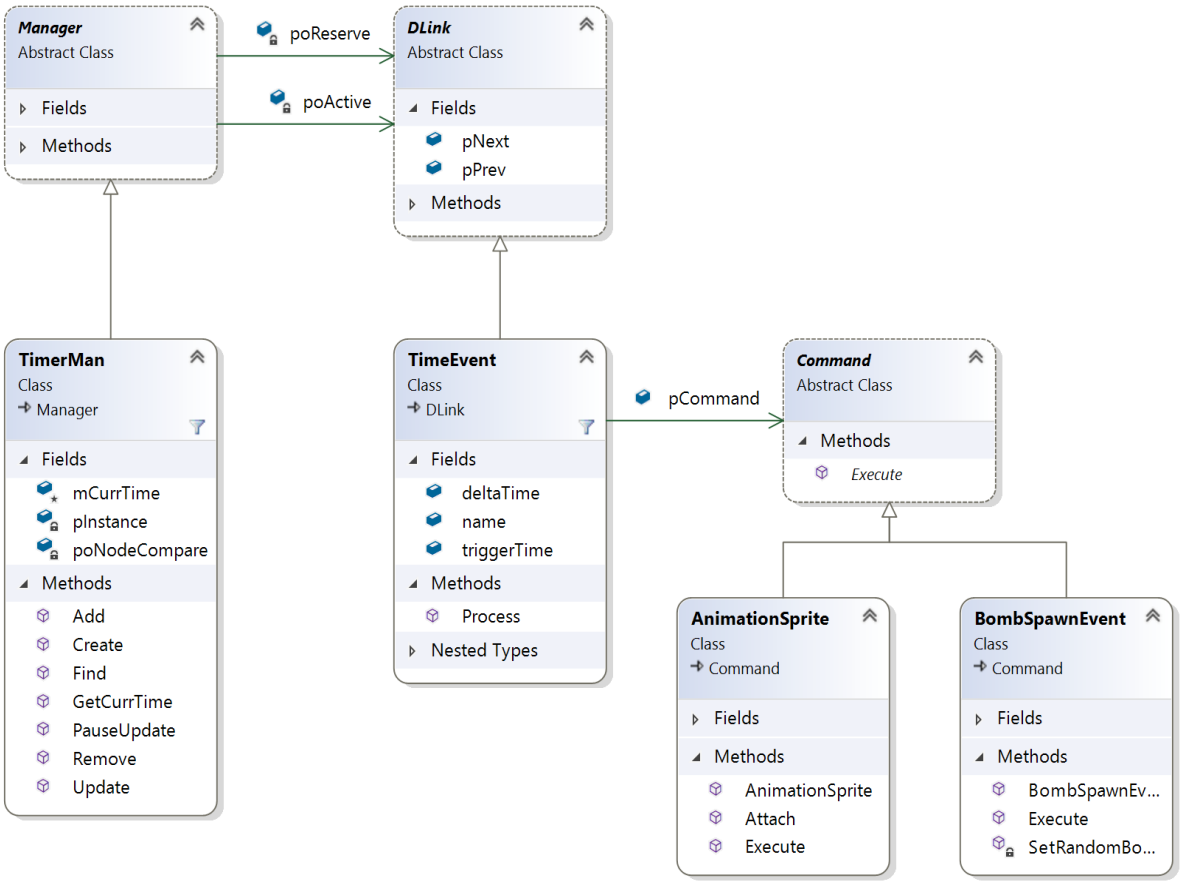
Solution: Implementing Timer Events that can be queued and parameterized as needed. Therefore, I used the Command Pattern.

Pattern description: It encapsulates commands in objects allowing requests triggering without knowing the requested operation or the requesting object. Additionally, it provides the options to queue commands, undo/redo actions and other manipulations.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Timer Manager:** A Manager object that controls the queuing of the time events.

- **Time Event:** The queued events that will invoke the execution operation of each command.
- **Command:** An abstraction to provide standard interface for each concrete command.
- **Animation Sprite and Bomb Spawn:** Concrete commands that will apply the implemented behavior in the Execute() operation when it is called.



Object Oriented mechanics:

The Timer Manager will add, remove, or trigger the events on the queue. When the Process() in the Time Event object method is called, it will call the Execute() method of the associated command to apply the underlying behavior.

Pattern usage in the game:

I am using this pattern to walk through different timed events and execute them when the time matches. Examples of timed events are: Sprites animation, Bombs Spawning, UFO Spawning, Moving Aliens Grid, etc.

Pattern benefits:

1. Encapsulate a request in an object which can be re-instantiated or reused.
2. Allows the parameterization of clients with different requests.
3. Allows saving the requests in a queue.

9. Observer:

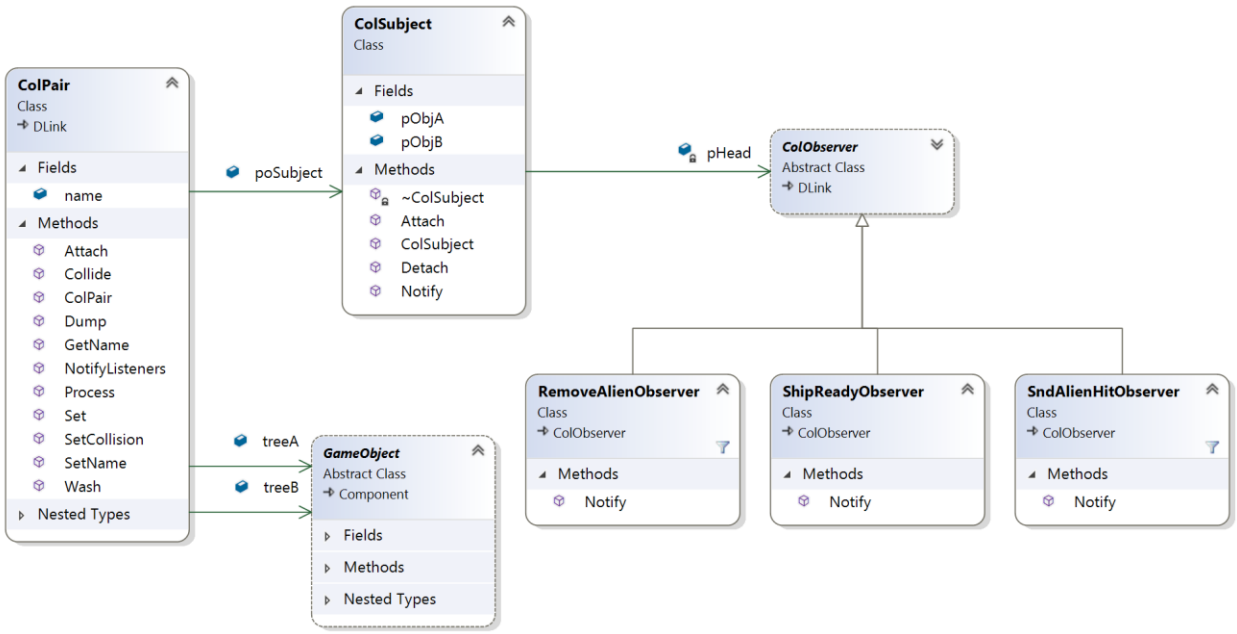
Problem: A sequence of actions have to be executed when an event happens. For instance, when a ship missile hits and alien: The alien should be removed from grid, a splat effect should be displayed, the ship should be able to shoot another missile, a sound effect should be played, and player's score get updated accordingly.

Solution: Build a list of observers that will be notified when the state of the state of the related object changes. I used the Observer pattern to implement this technique.

Pattern description: In the Observer pattern we have to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Collision Pair:** A Manager object that controls the attaching of the Observer to the Subject.
- **Collision Subject:** The subject that will notify the attached observers when changed.
- **Collision Observer:** An abstraction to provide standard interface for each concrete observer.
- **Remove Alien Observer, Ship Ready Observer, etc.:** Concrete observers that will apply the implemented behavior when they get notified by the subject.



Object Oriented mechanics:

The Collision Pair Manager will attach the observers to the Subject. When the Collision Subject changes, it will notify all the related observers by calling the Notify() method in each one.

Pattern usage in the game:

This method is used in the Collision System to apply the list of responses associated with the collision event happened between any pair of Game Objects.

Pattern benefits:

1. Easy to extend, so an observer object will be created and attached to the main object when new response is needed.
2. Allows reusability, the same response can be used multiple times for multiple parents with only different attributes values.

10. Visitor:

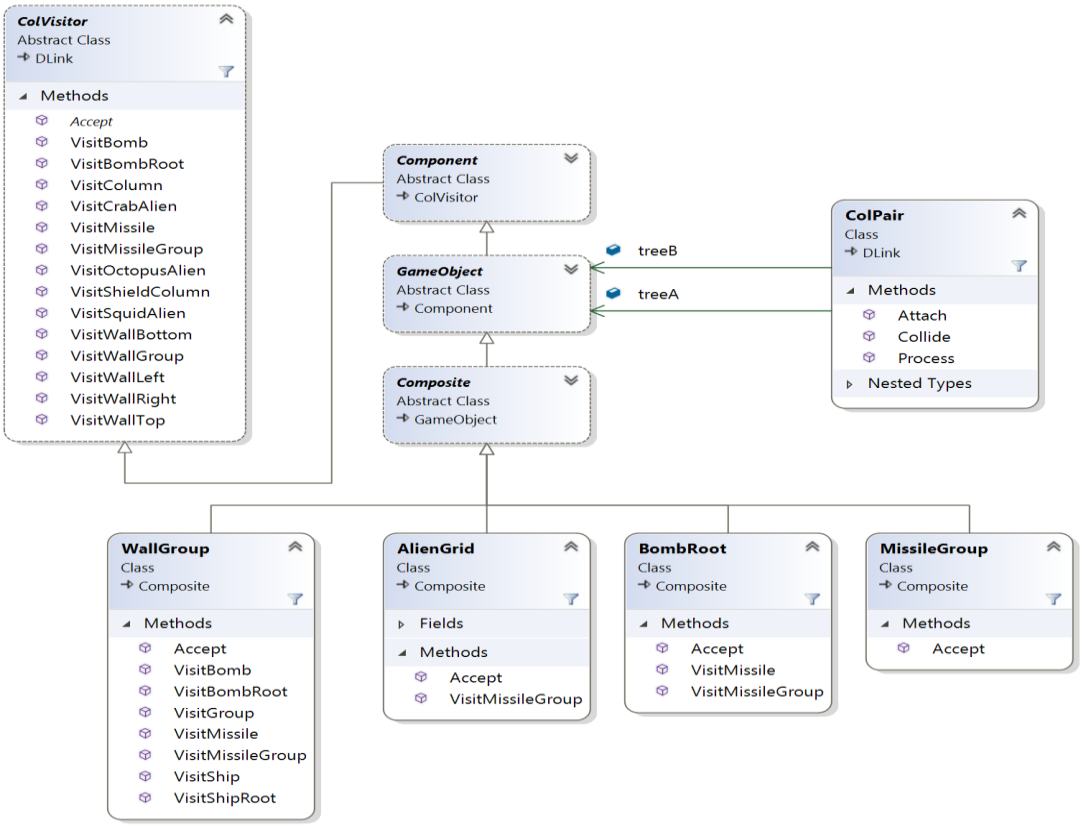
Problem: identifying the collision Game Object pairs and apply the appropriate action accordingly through a clean and scalable way.

Solution: I added a number of standard operations in an object structure that consists of different types, so that different behavior will be applied depending on the object's type. I used the Visitor pattern to implement this technique.

Pattern description: Through the Visitor pattern, we can represent an operation to be performed on the elements of an object structure. In addition, it lets us define a new operation without changing the classes of the elements on which it operates.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Collision Visitor:** An abstraction to provide standard interface for each concrete visitor.
- **Alien Grid, Wall Group, Missile Group, etc.:** Since I am using the double dispatch, the same object can be sometimes the acceptor or the visitor some other times.



Object Oriented mechanics:

When the Accept() method is called in one of the acceptors, this method also has an input parameter which is the visitor object. The appropriate Visit() method will be invoked to execute the appropriate behavior.

Pattern usage in the game:

This method is used in the Collision System to apply the resulting actions in case the two Game Object collide with each other. Hence, each Game Object type has an Accept() and a number of different Visit() methods that handles the various collision cases.

Pattern benefits:

1. The new cases can be directly added to the target object type, by adding a new visitor method, without changing the objects structure.
2. Dynamically detect the operation to execute according to the visitor type.

11. Strategy:

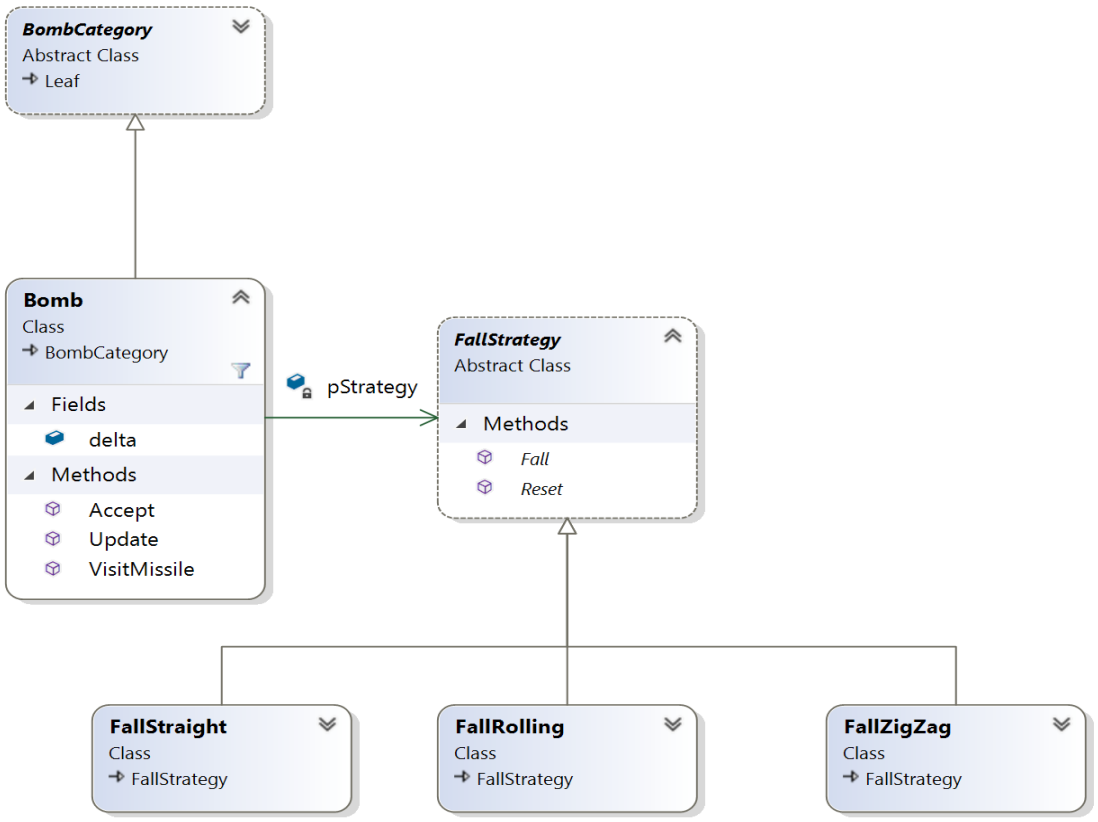
Problem: Each Bomb type has its own behavior of dropping animation.

Solution: I isolate the behavior of each Bomb type and implemented the dropping animation algorithm for each category. The Strategy pattern is used to apply this process.

Pattern description: Using the Strategy technique allows us to define a group of algorithms, encapsulate each one, and make them interchangeable at runtime.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Fall Strategy:** An abstraction to provide standard interface for each concrete strategy.
- **Fall Straight, Fall Rolling, and Fall ZigZag:** Concrete strategies that will apply the implemented behavior when the Fall() or the Reset() methods are called.



Object Oriented mechanics:

The context (i.e.: Bomb), which have reference to the abstract strategy, will call one of the provided algorithms. Consequently, the behavior to execute will be selected according to the type of the concrete strategy.

Pattern usage in the game:

I am using this technique to apply the falling animation according to the bomb type at runtime.

Pattern benefits:

1. It is easy to extend this structure by adding new strategy without affecting the clients.
2. The behavior is automatically selected according to the strategy type.

12. Null Object:

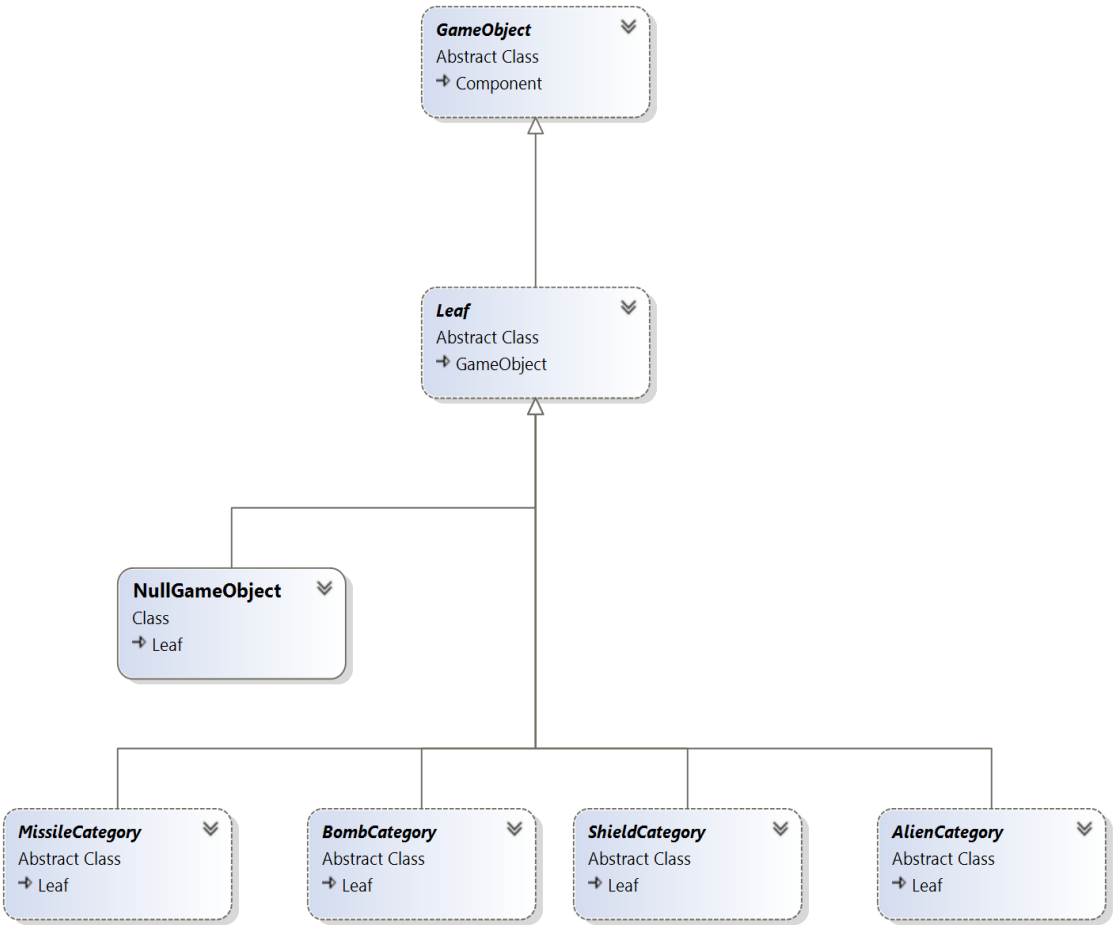
Problem: I needed an intellectual way to handle special cases by doing nothing.

Solution: Have an object type in an object structure that inherits the same interface but does nothing.

Pattern description: It provides an intelligent do-nothing behavior, hiding the details from its collaborators.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Null Game Object:** The Do-Nothing object type.
- **Missile Category, Bomb Category, etc.:** The collaborators with the real behavior implementation.



Object Oriented mechanics:

When the client doesn't have the knowledge of the intended behavior yet, the Null Object type can come into play to handle this special case. Later, this object can be easily replaced by the real object.

Pattern usage in the game:

I am using this technique in all the Managers' constructors to instantiate some of their member references.

Pattern benefits:

1. Can be used the same way as the real object without causing runtime issues.
2. Allows us to avoid using the traditional edge cases validations which sometimes cause bugs and glitches.

13. State:

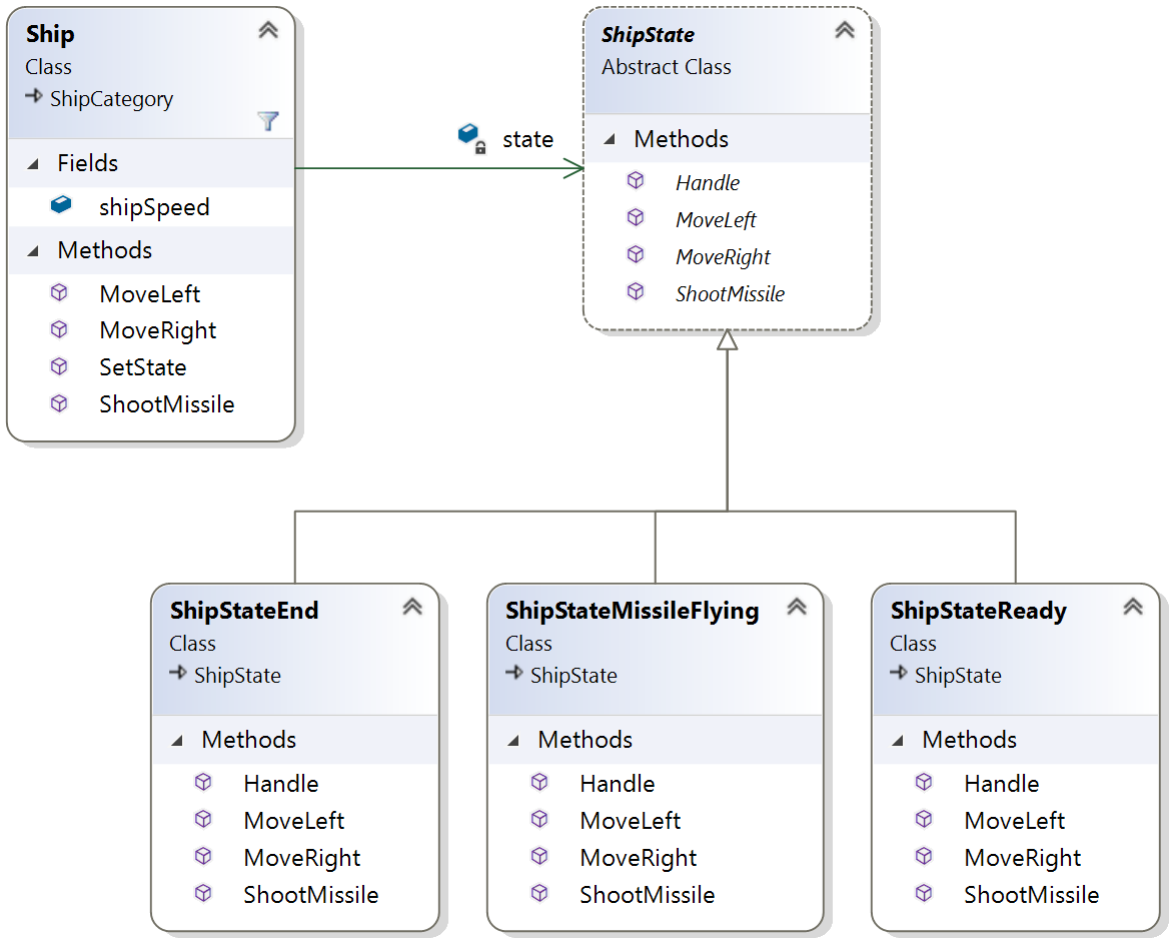
Problem: One of the game's requirements is to control the ship's behavior by not shooting multiple missiles at a time. The missile should hit one of the game objects before the player can launch another missile.

Solution: Create different states of the ship with different implementations and assign the appropriate state to the ship. The state pattern was the best choice for this technique.

Pattern description: This pattern is intended to allow an object to change its state internally according to some circumstances.

UML Diagram: Below is the UML diagram to show the actual implementation into my application. This diagram has the below items:

- **Ship:** The context which changes its internal state from time to time.
- **Ship State:** An abstraction to provide standard interface for each concrete state.
- **Ship State Missile Flying, Ship State Ready, etc.**: The concrete different states that a ship can have at a time.



Object Oriented mechanics:

The context changes its state internally giving some specific conditions. Each state handles the same actions differently.

Pattern usage in the game:

This pattern is used to deal with the player's ship states. For instance, when the ship shoots a missile, its state changes to Missile Flying where the player cannot launch another missile. On the contrary, when the missile hits on of the game objects (e.g.: alien, shield, wall, etc.), the ships state changes to Ready where the player can shoot a new missile.

Pattern benefits:

1. It can be extended easily without affecting the context.
2. It represents a full scaled state in an object class.
3. A clean way to alter an object's state.

Summary:

I realized how important the Design Patterns are because they are solving a lot of Object-Oriented design issues that are common among the majority of the systems under development. Considering the use of these patterns in an application's architecture will significantly facilitate the development and increase the quality of the final product. In addition, these patterns would be precious while scaling, extending, or modifying the system. Finally, I will definitely use the appropriate design patterns on my future software projects.