Shaik Hussain Arshad Naushad

Space Invaders Design Document

3/23/2020

Youtube Link:

https://youtu.be/_Vz-T4pBeT4

Game History:

Space Invaders is a 1978 arcade game created by Tomohiro Nishikado. It was manufactured and sold by Taito in Japan, and licensed in the United States by the Midway division of Bally. Within the shooter genre, Space Invaders was the first fixed shooter and set the template for the shoot 'em up genre. The goal is to defeat wave after wave of descending aliens with a horizontally moving laser to earn as many points as possible.

Space Invaders is considered one of the most influential video games of all time. It helped expand the video game industry from a novelty to a global industry, and ushered in the golden age of arcade video games. It was the inspiration for numerous video games and game designers across different genres, and has been ported and re-released in various forms. The 1980 Atari VCS version quadrupled sales of the VCS, thereby becoming the first killer app for video game consoles. More broadly, the pixelated enemy alien has become a pop culture icon, often representing video games as a whole.

Goals:

The goal of this project was to recreate the 1978 game in c# using Software Design Patterns.

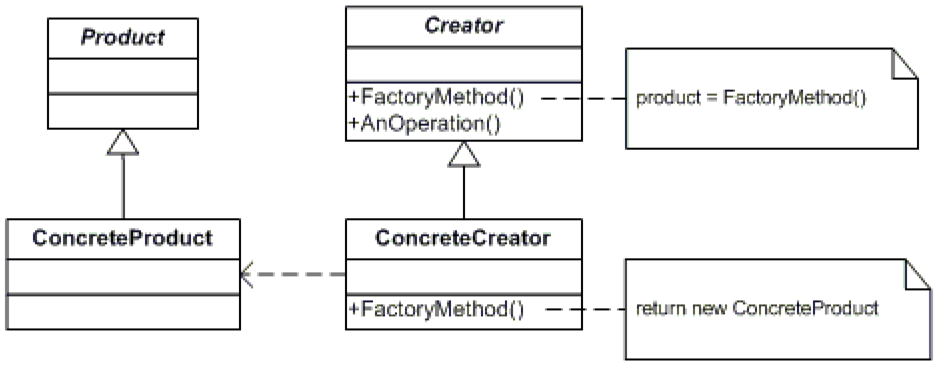
Game Features:

There were many different features I had to create before successfully recreating the game. These features are: an input system to read player inputs on keyboard and perform the appropriate actions. A sprite system used to load in many of the game's graphical features such as Aliens, Player Ship, Shields, Alien Bombs, and Explosions. An animation system using timers had to be created to simulate the aliens moving across the game scene. Finally, a collision system had to be created to check for each of the little collisions that can happen throughout the course of the game. I will break down how I implemented these features and what design patterns I used to reach my solution through illustrations from UML diagrams and code snippets.

Factory Patterns

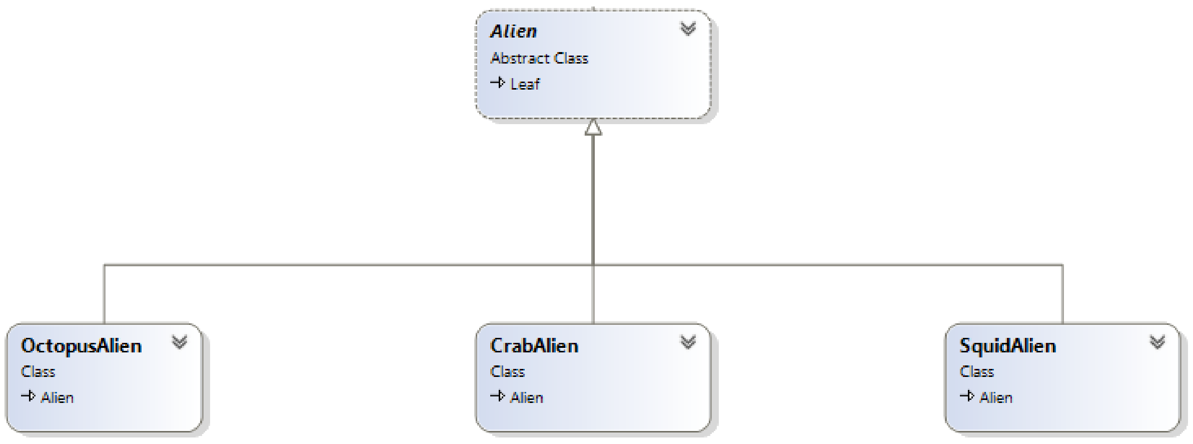
One of the main problems I had was for every game object I Create I had to create a new Sprite, so for 55 sprites I had to create a new game Object for every sprite, even if most of them are same. Factory Design Pattens helps in this case.

This is the UML class Diagram of the Factory Pattern



The factory pattern creates objects without exposing instantiation logic to the client. It is a design pattern used to manufacture objects. It is similar to the concept of making something first in your garage as a prototype and using that prototype to manufacture as many as we want.

For the game we need to generate Alien Sprites of Squid, Crab and Octopus which we derive from these classes. In order to do this, I created an abstract Alien Class which served as base class for the three classes and we can derive these sprites from anywhere from this Alien Factory class.



Within the Alien Factory I created a switch block where I used it to evaluate three cases :

-Alien

-Crab

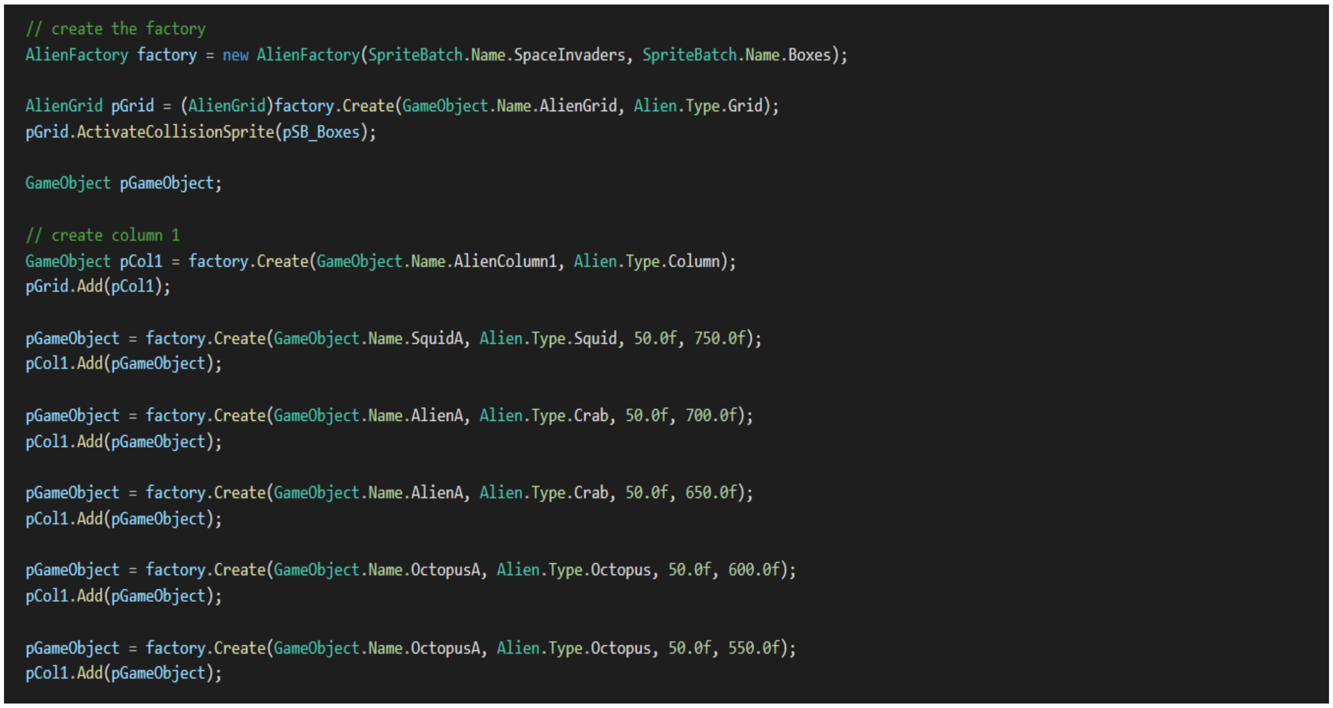
-Octopus

There is also the default case but if the switch statement ever goes to the default it means that something is wrong. Therefore

we only care about implementing three switch statements as we only want our factory to produce three Alien Sprites. At the end of the block I returned the Game Object.

Now using the Create function from our Alien factory we can create as many Game Objects as we want, without explicitly defining all of them.

The code below shows this in action for creating a column of aliens:

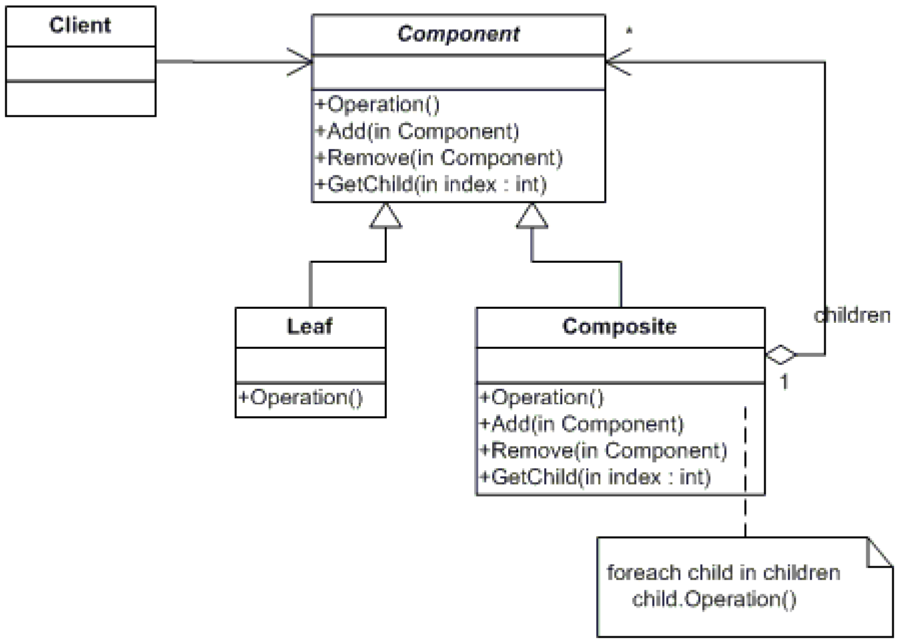


```
// create the factory
AlienFactory factory = new AlienFactory(SpriteBatch.Name.SpaceInvaders, SpriteBatch.Name.Boxes);

AlienGrid pGrid = (AlienGrid)factory.Create(GameObject.Name.AlienGrid, Alien.Type.Grid);
pGrid.ActivateCollisionSprite(pSB_Boxes);

GameObject pGameObject;

// create column 1
GameObject pCol1 = factory.Create(GameObject.Name.AlienColumn1, Alien.Type.Column);
pGrid.Add(pCol1);

pGameObject = factory.Create(GameObject.Name.SquidA, Alien.Type.Squid, 50.0f, 750.0f);
pCol1.Add(pGameObject);

pGameObject = factory.Create(GameObject.Name.AlienA, Alien.Type.Crab, 50.0f, 700.0f);
pCol1.Add(pGameObject);

pGameObject = factory.Create(GameObject.Name.AlienA, Alien.Type.Crab, 50.0f, 650.0f);
pCol1.Add(pGameObject);

pGameObject = factory.Create(GameObject.Name.OctopusA, Alien.Type.Octopus, 50.0f, 600.0f);
pCol1.Add(pGameObject);

pGameObject = factory.Create(GameObject.Name.OctopusA, Alien.Type.Octopus, 50.0f, 550.0f);
pCol1.Add(pGameObject);
```

Composite Pattern:

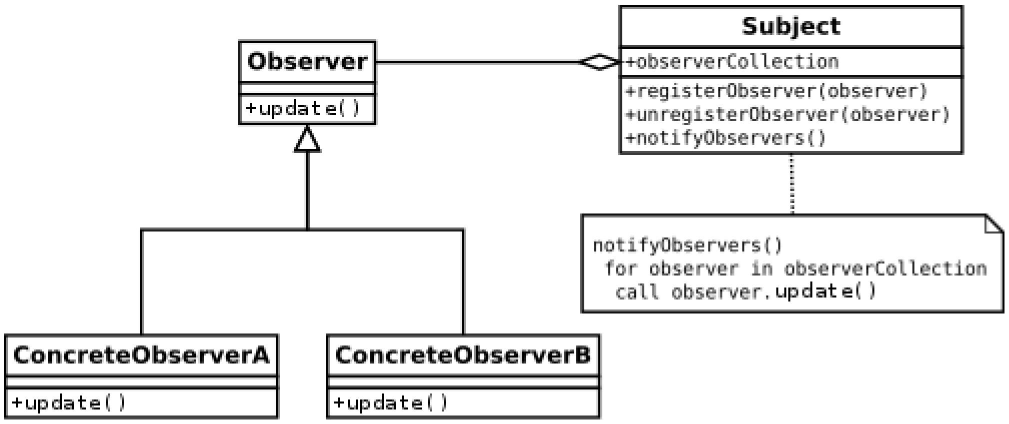
After creating 55 Aliens, the next problem was to group all the aliens into a single grid. We can use the Composite Design pattern to solve this problem. The intent of a **Composite** is to “compose” objects into tree structures to represent part-whole hierarchies. The composite allows the clients to treat individual objects and compositions of objects uniformly. The Composite

defines a unified Component interface for both part Leaf objects and whole Composite objects. Individual Leaf objects implement the Component interface directly, and Composite objects forward requests to their child components.



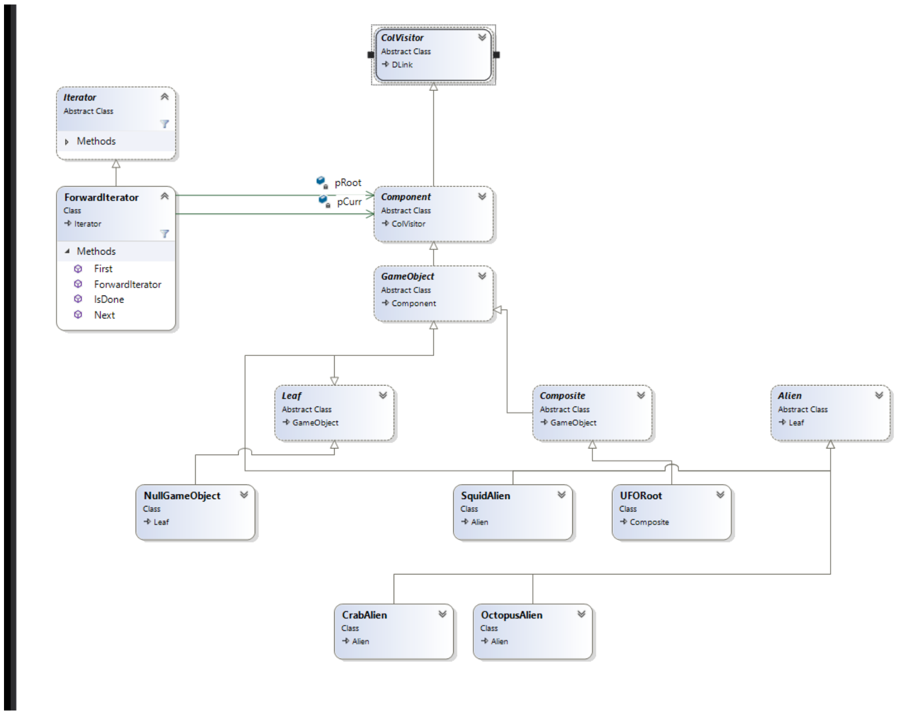
Observer Pattern:

The purpose of the observer pattern is to define a one-to-many dependency between objects so that when an object changes its state, its dependents are notified and updated automatically. The pattern is used when the listener must notify the other objects about certain events but doesn’t know which objects to notify.



Observers are used to Remove the Game Objects after collision and used to execute reactions for example Delete,Add or Multiply

We also use it in GameStats to report that a reaction has occurred and to execute a following procedure.

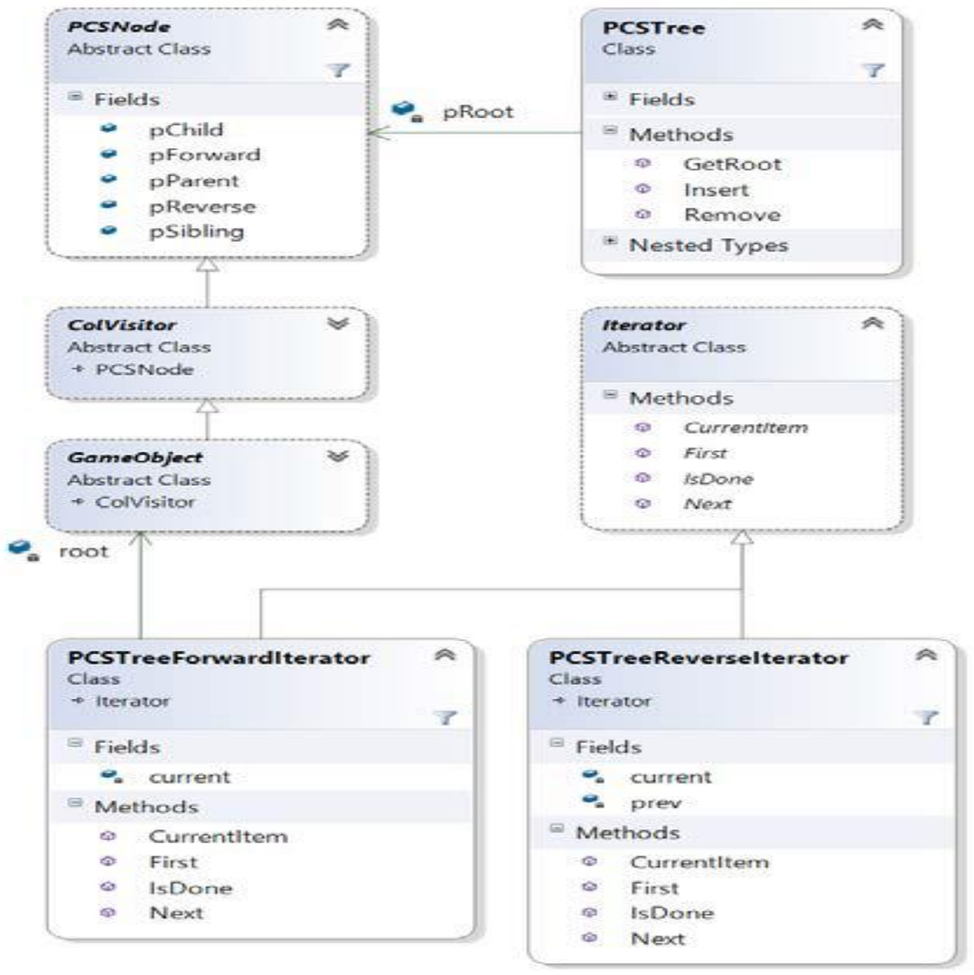


Iterators:

Iterators are used to iterate through the composite tree structure in a consistent and efficient manner. They allow us to control access. It allows us to restructure code. It encapsulates complexity of the storage mechanism. The purpose of the iterator pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

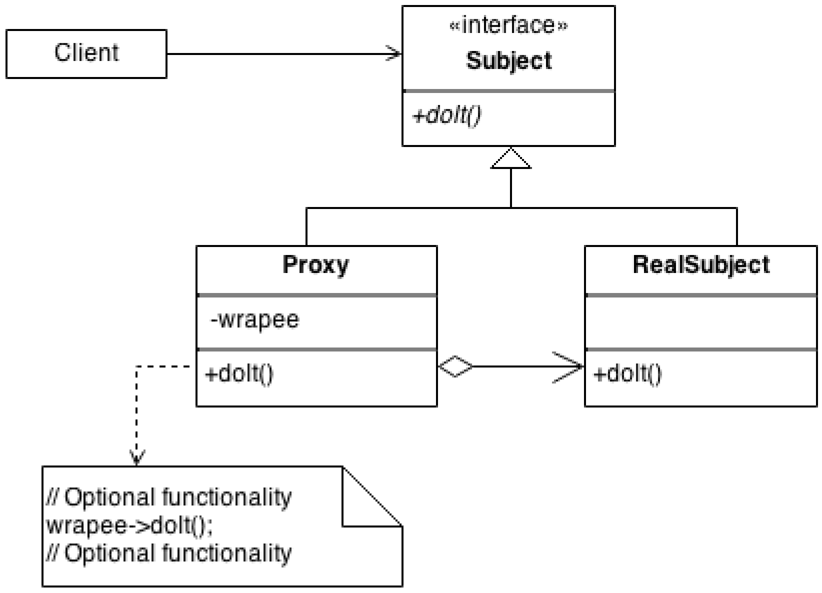
In the project I used the iterator to move down the grid and changing the direction the grid moves when it hits either side of the walls. I also used it to iterate over the collections in the game, for the Alien Grid and the Shield Grid composites

In the code provided above, I'm using a ForwardIterator to do my iterations. The iteration will continue walking through the linked list nodes until it reaches the last node and finishes. As the iterator walks through the list, the Alien Grid jumps down the scene before changing directions and traversing the game scene in the opposite way. As I stated earlier, the iterator pattern also helped immensely to detect collisions between game objects. To do this, I'm always using the Iterator.GetChild() call in all of my Visitor functions



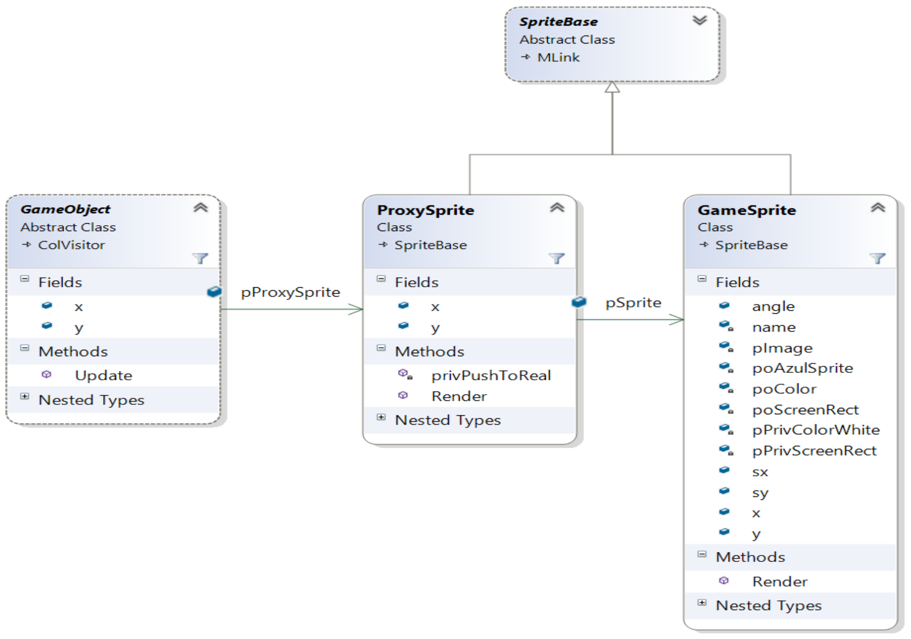
Proxy:

I had the problem of managing the copies of objects which are almost the same. For example I had to group a total of 55 aliens together even though we used just 3 Alien Types. Since the Sprites contain their own data it is not good to create 55 new Sprites so for Optimization purposes we use Proxy Design patterns.



Proxies instance data that is different but they can have different data without overwriting the object.
In my implementation of Space Invaders, the pattern is used to create multiple copies of
a single sprite object that differ only by their X and Y coordinates, providing improved efficiency in both complexity and performance.

Below is the UML diagram for proxy used in the Game



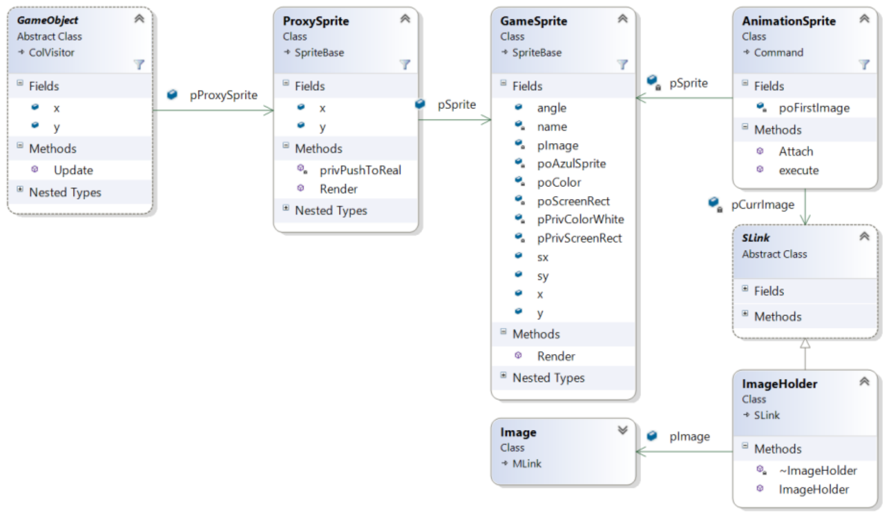
Animation:

After creating the grid the next feature is Animating the aliens. In the game the aliens must simulate movement of the individual aliens as they moved from one side to the screen all the way to another, and moved back once they reach a certain point;

To solve this I used the Animation Pattern.
The aliens Swap direction after hitting the Side Walls They also change the Sprites after every move. I use Animation Sprite Class for this. In the AnimationSprite class, I am overriding the Execute function given by Command and within this Execute I

am swapping between two images: the alien idle image and the alien walk image.

Below is the UML diagram for the Animation Design Pattern:



Strategy:

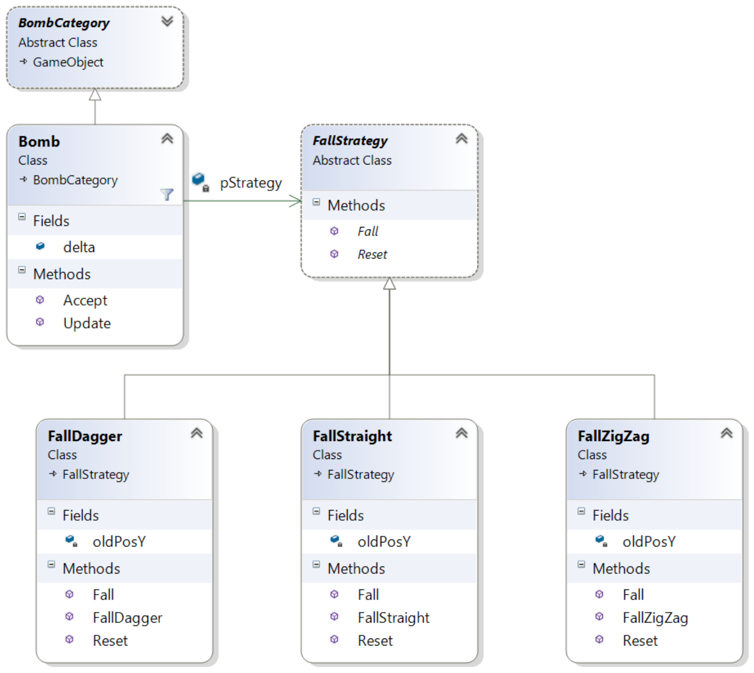
One of the main charecteristics of the game is the Aliens deploy bombs on the ship
If the bombs touch the shield then the Shield blocks slowly start to disappear. For surprise effect the bombs should fall randomly so for this we use Strategy pattern.

The strategy pattern (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

For instance, a class that performs validation on incoming data may use the strategy pattern to select a validation algorithm depending on the type of data, the source of the data, user choice, or other discriminating factors.

In the game we use different types of Bombs. We can use a family of Algorithms and make them interchangeable

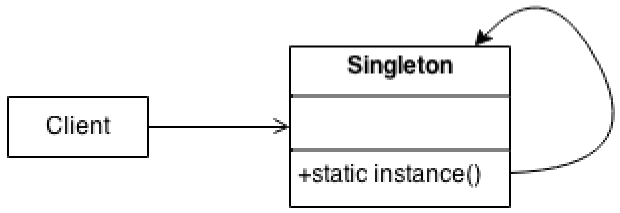
Below is my UML diagram for strategy pattern:



Singleton:

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances. It's simpler to pass an object resource as a reference to the objects that need it, rather than letting objects access the resource globally. The real problem with Singletons is that they give you such a good excuse not to think carefully about the appropriate visibility of an object. Finding the right balance of exposure and protection for an object is critical for maintaining flexibility.

Managers are used as Singletons in the project. The purpose of the manager system was to have a manageable fashion of organizing the various types of data I needed. The benefit of making my managers singleton is that I could reuse them as frequently as needed. This ability allowed me to perform quick functions in the game such as Add, Remove, Create, or Find. Sprites, Textures, Sounds all use Managers to instantiate objects and preform quick functions. I also used Manager to receive data from from GameStats which helped me receive the real time data of the Game



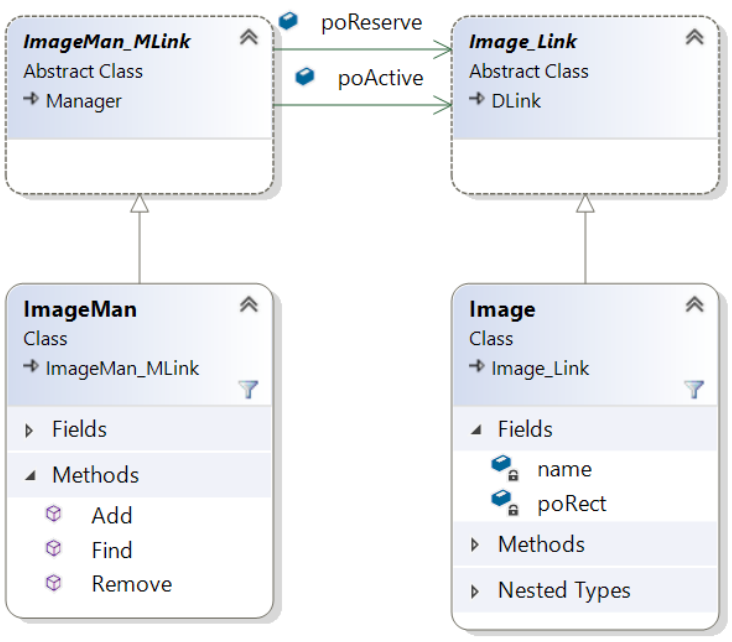
Object Pool:

In the game managing the allocation and deallocation of collections of
different types of objects in an efficient manner. Leveraging the ***new*** keyword and releasing
references to objects throughout the game execution could cause undesired performance
degradation due to changes in the heap and garbage collection activities.

The object pool pattern is a software creational design pattern that uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and destroying them on demand. A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object to the pool rather than destroying it; this can be done manually or automatically.

Object pools are primarily used for performance: in some circumstances, object pools significantly improve performance. Object pools complicate object lifetime, as objects obtained from and returned to a pool are not actually created or destroyed at this time, and thus require care in implementation.

Managers are used to create and pass new nodes to different Game Objects. Every Game Object in the Game uses managers to receive and Execute Objects.



State Pattern:

For the Game to exist we need different scenes for realistic purposes and to choose what we want to do with the game. We also want to know what occurs if we win or if lose. For this State Pattern is used.
The state pattern is used to cycle through the reactions of the Ship and also iterate throught the Scenes. The state pattern helps us to iterate through the behavior of the Ships and on the whole- The Game.

The Player Ship Game Object had 3 distinct states:
• Ship Ready State (First state – starting point)

• Missile Firing State (When missile is shot)

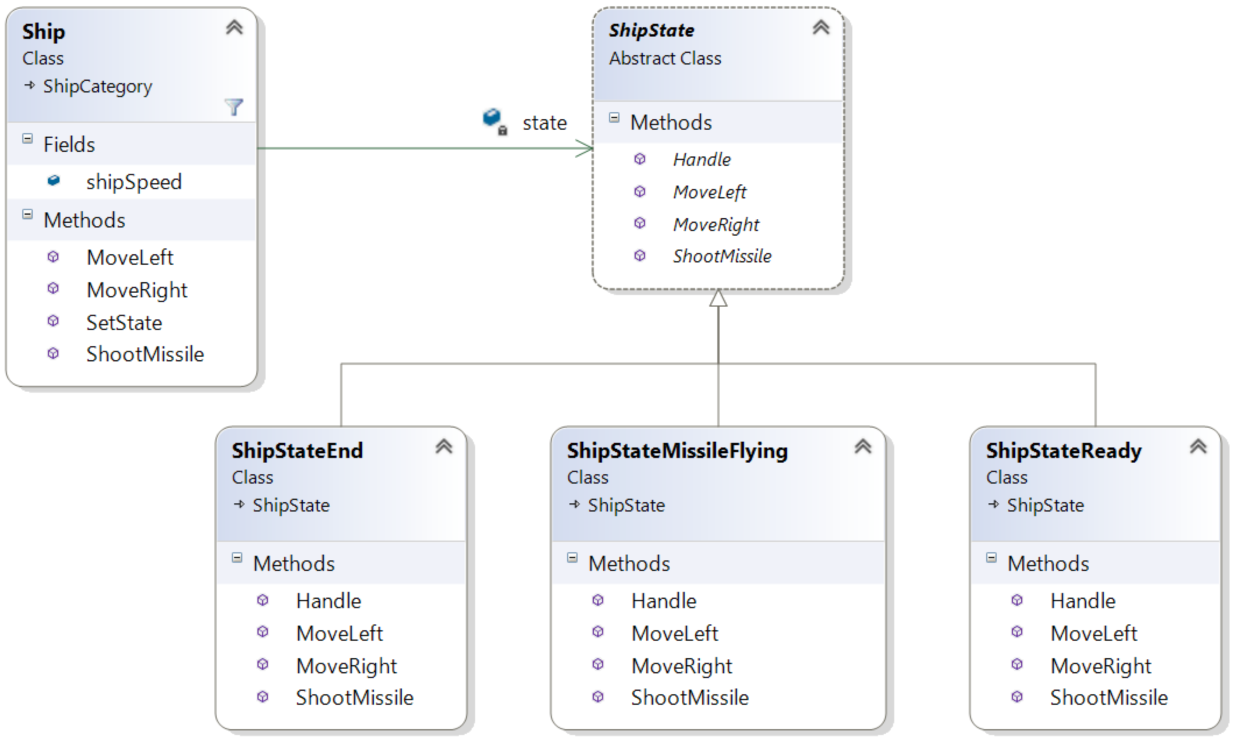
• Ship End State (When Player is hit by bomb)

The Game has 4 states:

- Game Select State
- Game Play State
- Game Over State
- Game Won State

The cycling of these 5 Game States were done by either player inputs or by conditionals regarding the Game's stats being met.

For example, when a player hits "1" on the keyboard they will be taken to GamePlay and if they hit "0" they will be taken back to the Scene Select mode. In order to reach the Game Over or Game Win states certain conditions must be met.. To hit the Game Won state, the player must have cleared all the aliens on the second level. To hit the Game Lost state, the player must have lost all 3 of their lives.



NullObject:

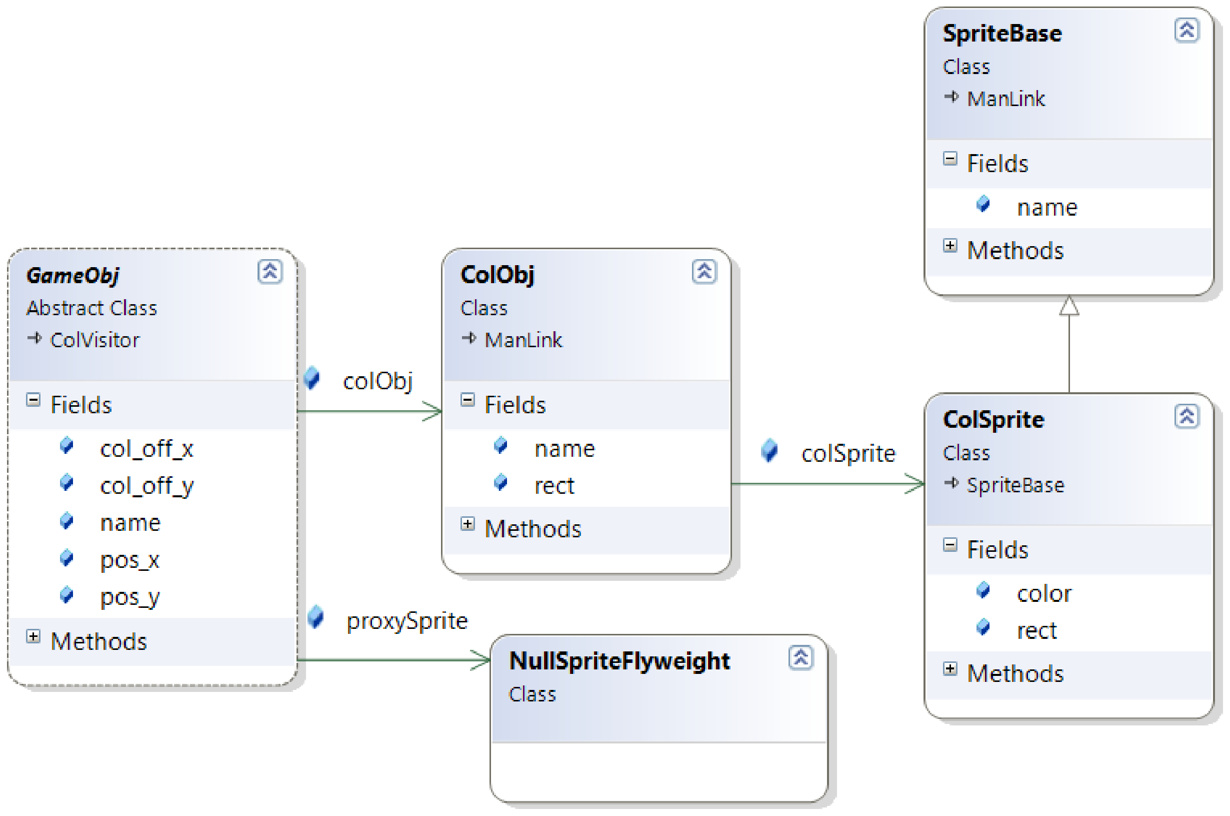
Sometimes the object doesn't have any real work to complete. So we do nothing for no edge case. Null Object help us.

NullObjects basically do nothing and return nothing.

Shaik Hussain Arshad Naushad

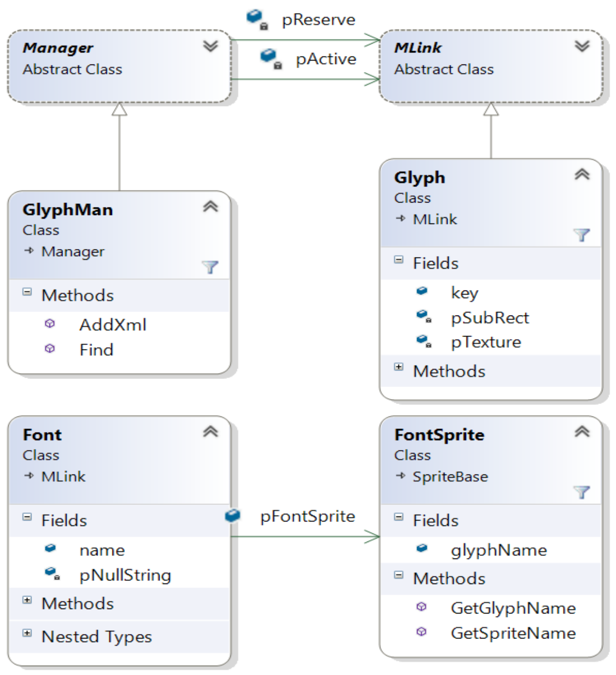
Space Invaders

Below is the UML diagram of Null Object used in my Project



Flyweight:

Flyweight objects helps us in Reusing existing objectint
Objects are shared. We can Reuse shared objects
For example various Fonts are used in the Game like Consolas20 and Consolas36.



Conclusion:

The experience of developing Space Invaders was as illuminating as it was challenging. Even a
game as simple as Space Invaders results in an incredibly complex real-time system and I truly
cannot imagine how I would have been able to complete this project without leveraging the
design patterns that were presented throughout the course.
While I am proud of the work that I have accomplished over the course of the project, there are a number of items that I left incomplete due to time constraints, including:

-2 Player
-Level2