Santosh Pingali

# Space Invaders

## Design Document

3/24/2019

YouTube Demo:

https://youtu.be/_wM28aI-ovY

# Project Goals:

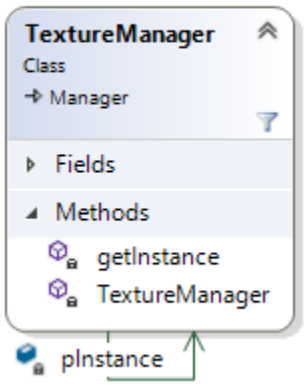
The goal of this project was to recreate the 1978 arcade game *Space Invaders* in the C# programming language using modern software engineering principles and software design patterns. A minimum of 10 patterns were required to be implemented in the game.

# 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.

# Singleton Pattern:

Due to the overall complexity of the game, I needed a centralized manner of managing each of the game's core features. Enter the manager system. The purpose of the manager system was to have a manageable fashion of organizing the various types of data I needed. The key issue presented with this system is that I wanted to have only one instance of each manager. To combat this problem, I designed most of my managers with the Singleton pattern in mind. The UML diagram below provides an illustration of how my TextureManager looks after I made it into a singleton class.



The singleton pattern was the perfect choice for this because its purpose is to ensure that a class only has one instance and provides a global point of access to it. Now, this may seem like a lazy instantiation but that's the point as it isn't used until the first time it is used. 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. As mentioned previously I made a singleton TextureManager class to manage all of the various textures I would be adding in to the project. Within the TextureManager class, I had a private TextureManager constructor to initialize the manager, and a Create function that must be called before the manager is able to be used. Additionally, I have a private getInstance() method to return an instance provided that it isn't null. In doing so, whenever I use the manager to do add or find a texture, I always use an assert to check if an instance exists. If I try to use a manager without there being an instance, the program will not work.

I also found that the singleton pattern wasn't just helpful for the creation of game managers as I also used the design pattern to help me keep track of all the game's important stats. Initially, I made a GameStats class with the intent of keeping track of the alien count, player lives, and player score. Initially, I tried to get this to work without using any design patterns but could not get the accurate results.

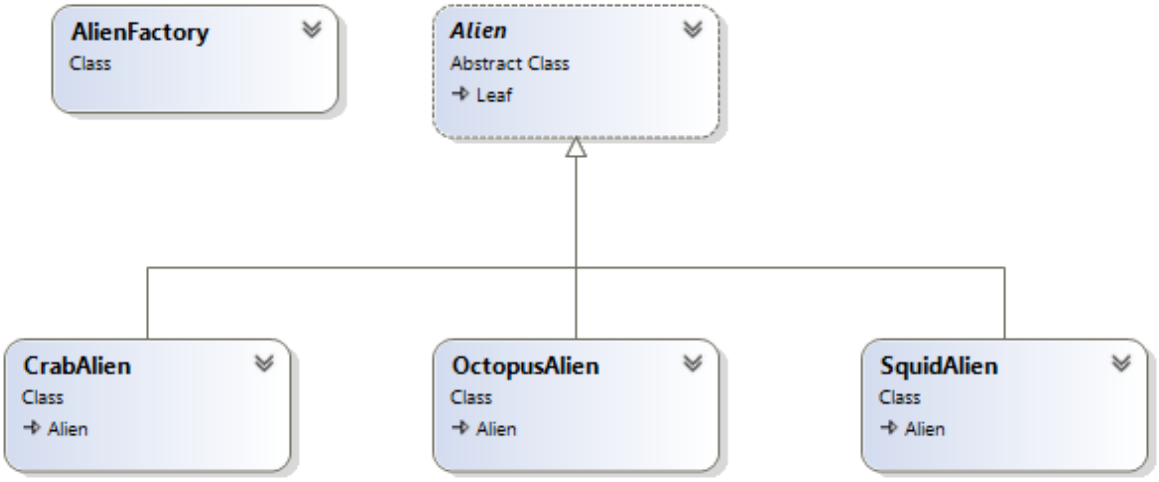
After some experimenting, I realized that the singleton pattern would solve this problem. After implementing the pattern, I could easily add points to the player. To do this I made a public static void function called OnAlienHit with an integer passed in the parameters. Within this function, I updated the score using the instance of the class. The snippet provided below displays how this was done in code.

```
public static void OnAlienHit(int pointsGiven)
{
    GameStats.getInstance().pointsEarned += pointsGiven;
```

## Factory Pattern:

As the name *Space Invader's* would suggest, we need to have aliens in the game to serve as the enemies the player must kill. The goal here is to generate our Alien game sprites and place them in a 5x11 alien grid where the Squid Alien types sit at the top row, the Crab Alien types sit in the 2 middle rows, and the Octopus Alien types sit in the bottom 2 rows. In order to do this, I made an abstract Alien class which served as the base class for the classes I made for the 3 different alien types we need in the game to derive from. Then I used the Factory design pattern

to generate the 55 aliens needed. The UML diagram below illustrates my use of the factory pattern.



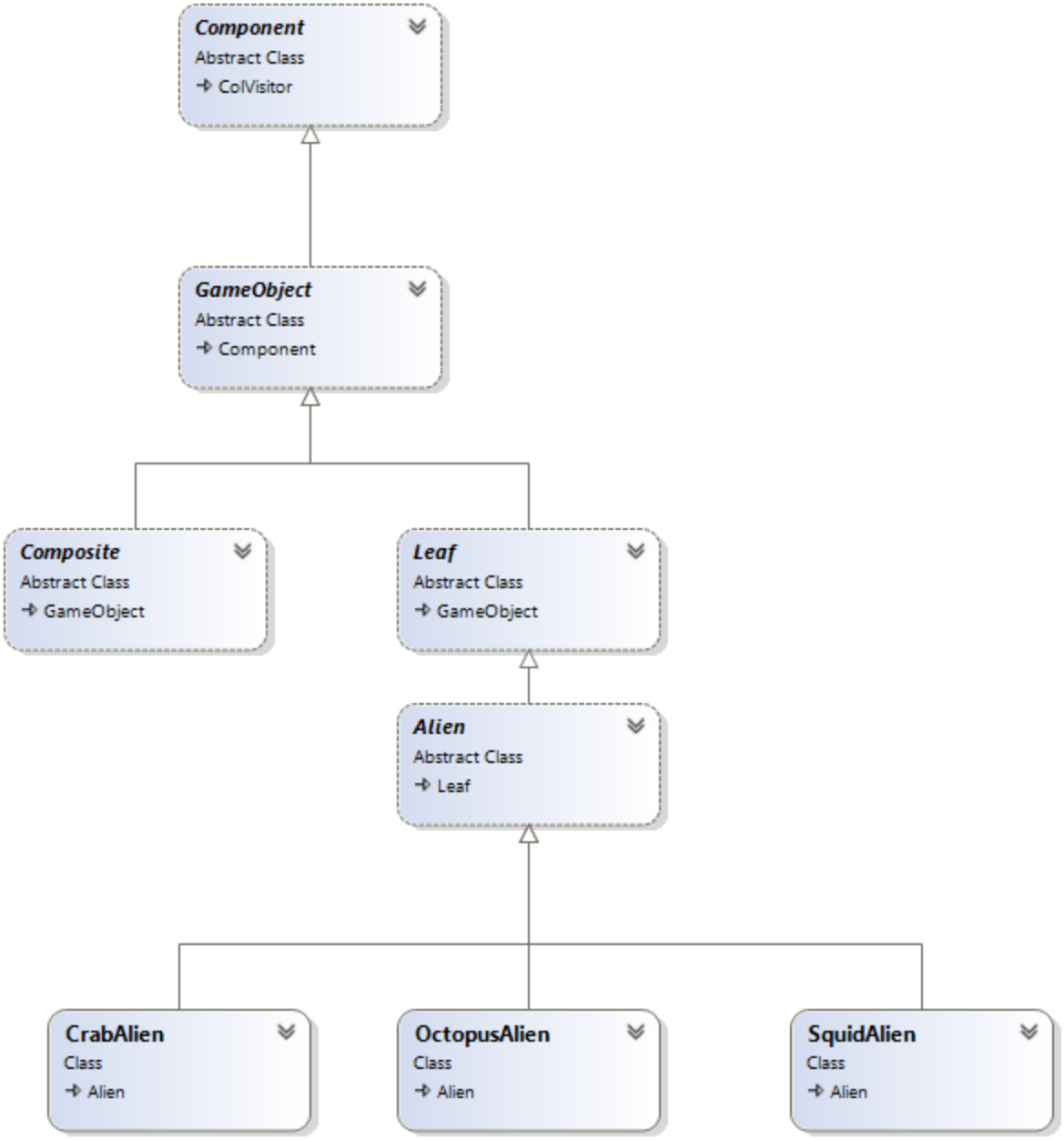

The Factory design pattern creates objects without exposing instantiation logic to the client and refers to the newly created object through a common interface. The factory is a creational design pattern and used to create objects. So, to create my Alien factory, I made an AlienFactory class. Within the class, I had a Create method that uses a switch block to evaluate the various cases in the switch-block. Within the block I had cases for the 3 Alien types (Crab, Octopus, and Squid) and for the Alien Column and Grid. Then, at the end of the method, I returned the GameObject. Once this was done, I went to my Level1 and Level2 game states to create 11 columns, 55 aliens, and the alien grid. I was able to do this by using

calling the Create method discussed earlier. The code snippet below shows this in action for creating one column of aliens.

```
GameObject pCol1 = factory.Create(GameObject.Name.AlienColumn1, Alien.Type.Column);
pGrid.Add(pCol1);

pGameObject = factory.Create(GameObject.Name.SquidA, Alien.Type.Squid, 50.0f, 500.0f);
pCol1.Add(pGameObject);

pGameObject = factory.Create(GameObject.Name.AlienA, Alien.Type.Crab, 50.0f, 450.0f);
pCol1.Add(pGameObject);

pGameObject = factory.Create(GameObject.Name.AlienA, Alien.Type.Crab, 50.0f, 400.0f);
pCol1.Add(pGameObject);

pGameObject = factory.Create(GameObject.Name.OctopusA, Alien.Type.Octopus, 50.0f, 350.0f);
pCol1.Add(pGameObject);

pGameObject = factory.Create(GameObject.Name.OctopusA, Alien.Type.Octopus, 50.0f, 300.0f);
pCol1.Add(pGameObject);
```

## Composite Pattern:

Now that I had created my 55 Aliens, the next problem I faced was to group all these aliens into a single grid. In order to do this, I decided to go with the Composite design pattern. The composite pattern was the perfect solution to this problem because it composes objects into a tree structure to represent whole part hierarchies. The composite allows the clients to treat individual objects and compositions of objects uniformly. This makes it extremely easy to add new components as the client code does not need to be changed. Below you will see my hierarchy with illustrations from the UML diagram.



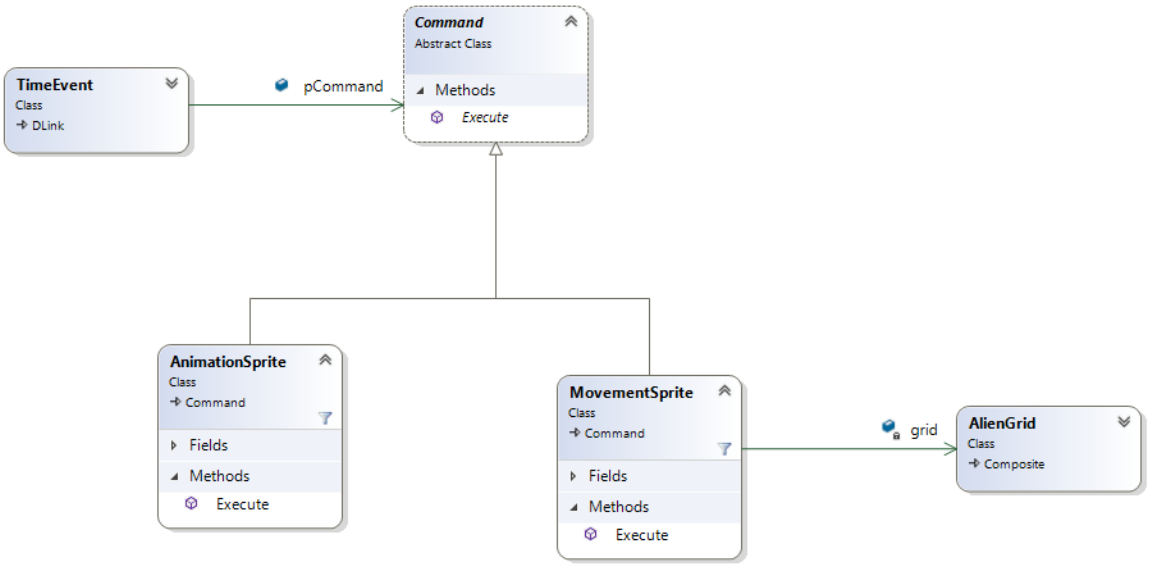

As you can see with the diagram, the class at the top of the tree is the Component class. Within the component class, I have several abstract functions such as Add(), Remove(), and GetFirstChild() that must be overridden by the classes that derive

from it. The class that is directly below Component and derives from it is the GameObject class. Below that I have both Composite and Leaf classes deriving from GameObject. The purpose of the Leaf class is to define behavior for the elements in a composition. Deriving from the Leaf class is the alien class which serves as the base class for the 3 alien classes. Due to the composite pattern's tree-like structure, it became extremely simple for me to add or remove the alien game objects when needed.

## Animations:

After I had created my alien grid, the next feature on my to-do list was animation. The problem I was facing here was fairly self-explanatory; I had to 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; in this case, the aliens must swap directions after they hit one of the side collision walls. I thought of many different techniques to solve this problem, but ultimately, I decided to use the Command design pattern for my solution. The command pattern was the ideal choice here because it encapsulates a request as an object. This means that I can pass the request to objects and support undoable operations. The benefit of using a command is that it makes it much easier to

construct the general components that need to delegate, sequence, or execute method calls. In the context of my code, I created a TimeEvent class which had a delta time and a trigger time. The delta time is responsible for how often the event should run; likewise, the trigger time is the exact moment when the event is triggered. Then in my abstract Command class, I have a signature for the Execute method any class that derives from Command may use. The UML diagram below will provide an illustration as to how this looks in my project.




The two classes that I have deriving from the Command are the AnimationSprite and MovementSprite classes. 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. By doing this,

I was able to simulate an animation as the aliens started to swap between the two images while they moved across the screen. The job of the MovementSprite class was to get all the aliens to move in a single lockstep simultaneously. To do this I had to override the Execute function once again; and inside it I called the MoveGrid() function to move the grid, played the alien marching sound, and added the Time Event back to the TimerManager every time it was executed. By doing this, I was able to get the aliens to slowly march their way across the screen while playing the march noise.

## Iterator Pattern:

Now that I've grouped up all 55 aliens in a grid and got the animations ironed out, I needed to get the aliens to jump down the screen once they hit one of the border side walls. I played around with many different solutions, but the one I ultimately stuck with was the usage of the Iterator design pattern. 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. The pattern provides an abstraction that makes it possible to decouple collection classes and algorithms. The iterator came in very handy when I tried to get the alien grid to move down and when it came to detecting collisions between two different game

objects so that I could remove that exact object after the collision. First, consider the code snippet below, where I am using the MoveGridDown() function in the AlienGrid.cs file to move grid down.

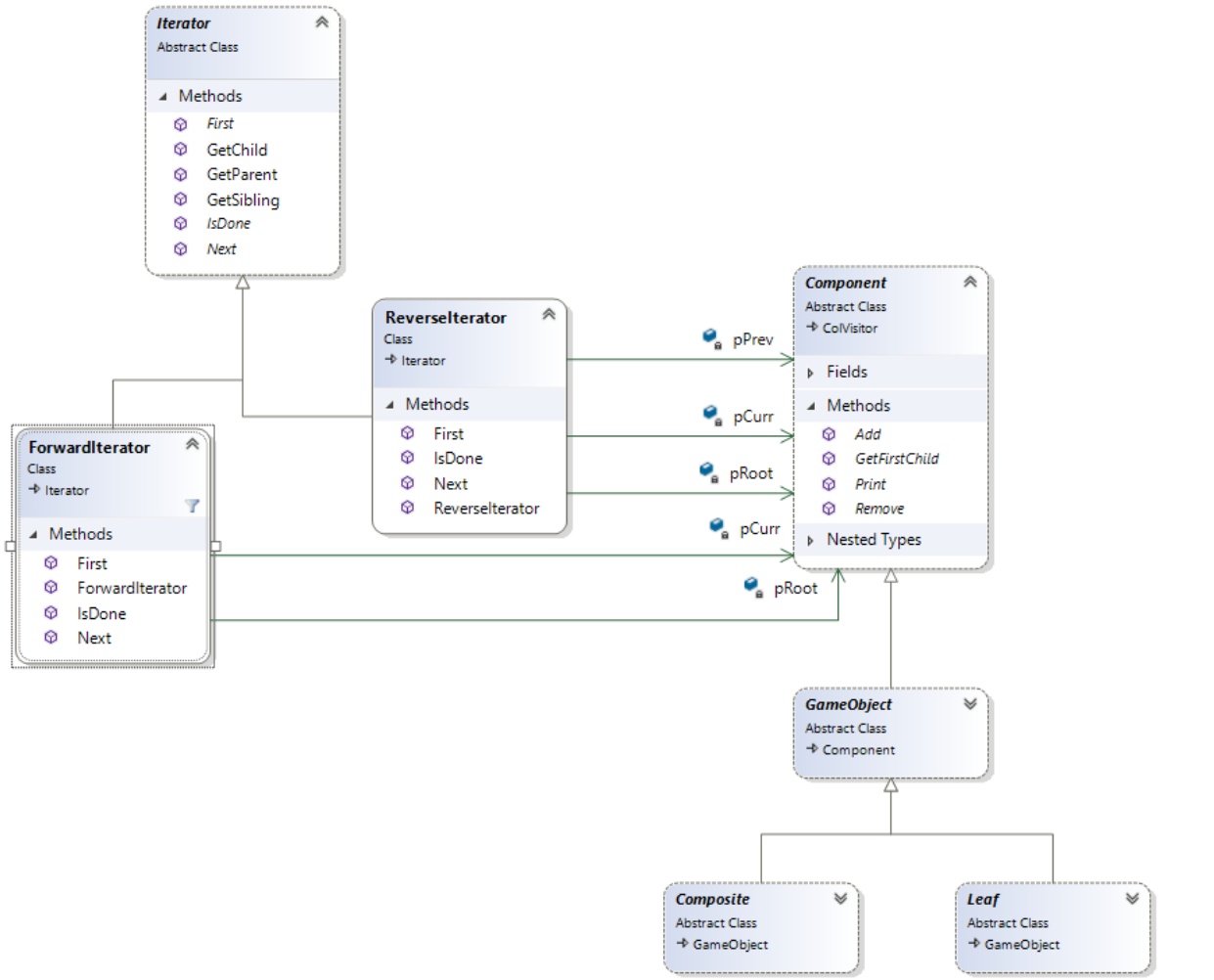
```
2 references
public void MoveGridDown()
{
    ForwardIterator pFor = new ForwardIterator(this);

    Component pNode = pFor.First();
    while(!pFor.IsDone())
    {
        GameObject pGameObject = (GameObject)pNode;
        pGameObject.y -= 25.0f;

        pNode = pFor.Next();
    }
}
```

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. For example, look at the code

snippet below to provide an illustration from BombRoot.cs when the VisitMissile function is overridden.

```
15 references
public override void VisitMissile(Missile missile)
{
    GameObject pGameObject = (GameObject)Iterator.GetChild(this);
    ColPair.Collide(missile, pGameObject);
}
```

Here, with the Iterator, I can find the child node from the list after the collision is detected. The UML diagram below provides an illustration as to how the iterator pattern was implemented for this project.




# Collisions:

Now that all the necessary game objects have been created, a key component I had to get cracking on was one of the most important features of the whole game: Collisions. Collision detection is extremely important because the entire game of *Space Invaders* lives and breathes with the collision between two game objects. There were many different collision pairs I had to create to accurately

recreate the game. To name a few, I created collision pairings for: Ship vs Bomb to play the explosion sound file and take a life away from the player, Alien Grid vs the left and right wall to cause the grid to jump down the game scene and move in the opposite direction, and for Missile vs Alien to kill the appropriate alien and award points to the player depending on what Alien they killed. Collision was such a big system to create that I needed to use two design patterns to get it working properly. The two patterns in question are the Visitor and Observer design patterns.

## Visitor Pattern:

Of the two patterns mentioned above, the Visitor pattern was the first one I implemented when I began creating my collision detection system. The purpose of the visitor pattern is to represent an operation to be performed on the elements of an object structure. I found the visitor pattern to be particularly helpful when I had to change the behavior of classes without having to completely refactor them. In the *Space Invaders* game, the pattern was used for all the Game Objects I created. From the individual Aliens, to Walls, to Player Ship, and to the UFO, just about every game object made use of the pattern because they all collided with at least one other object. First, I created the abstract class

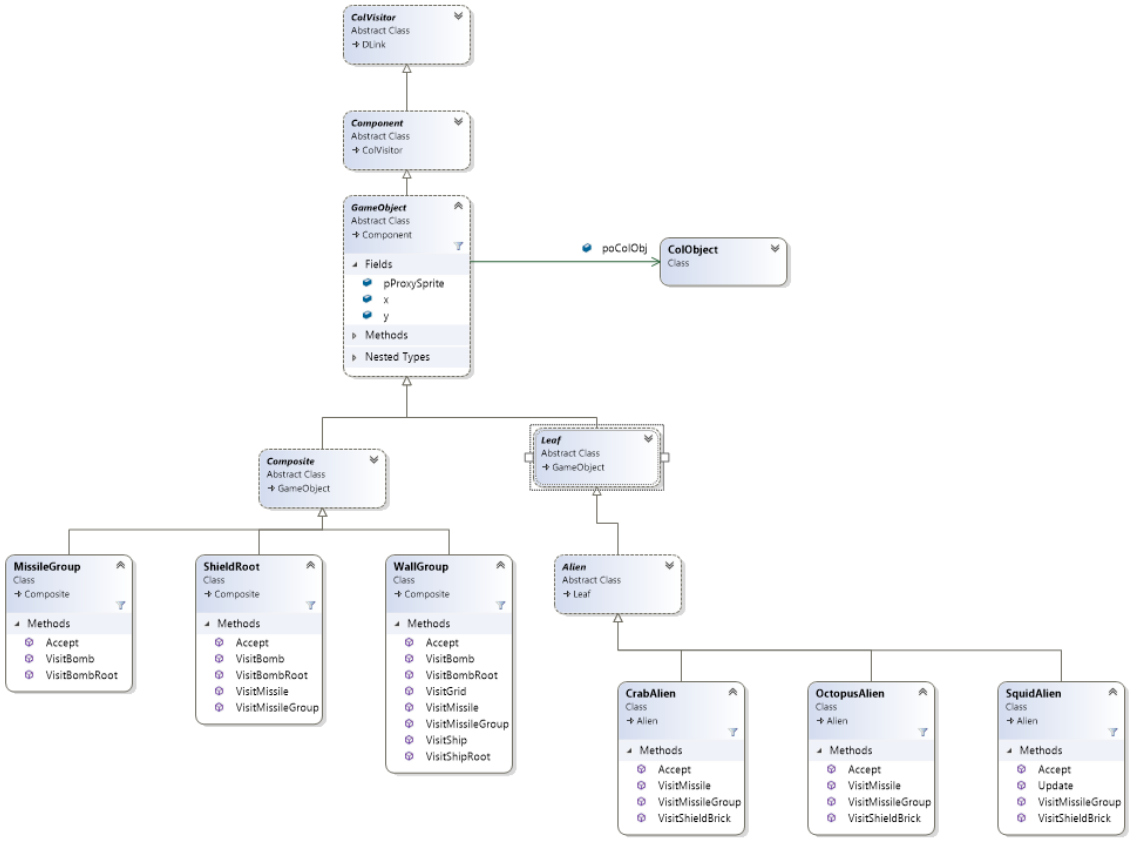
ColVisitor and created virtual void function signatures for every GameObject class. Within the ColVisitor class I also created an Accept function that the Game Objects had to override. For example, In my Bomb class I had to override the Accept function from ColVisitor and inside of the function I made a function call to it's respective Visit function; In this case the function was called VisitBomb(). The code snippet below provides an illustration of how this is done.

```
public override void Accept(ColVisitor other)
{
    other.VisitBomb(this);
}
```

After implementing Accept(), I was then able to override the Visit functions for all of the GameObjects the bomb collided with. The collision events themselves didn't happen within these visit functions as I was mainly setting the collisions and delegating the main work to the observers by notifying the listeners. An illustration of the visitors at work is shown through the code snippet below.

```
public override void VisitMissile(Missile missile)
{
    ColPair pColPair = ColPairManager.GetActiveColPair();
    pColPair.SetCollision(missile, this);
    pColPair.NotifyListeners();
}
```

As shown in the code above, not much of the work post-collision detection is done in the visitor pattern. It merely lets the listeners know that a collision has happened and that the appropriate event must be fired off. I found that simultaneously working on the visitor and observer patterns to be helpful as I kept each Visit function clean and compact with a few lines of code each and delegated all of the hard work to the observer classes. In the UML diagram above, you will see the general flow of my Visitor in action.



As you can see with the diagram, all the GameObjects and their respective group have overridden the Accept functions. Then, they override the visit functions corresponding to the objects they are meant to interact with.

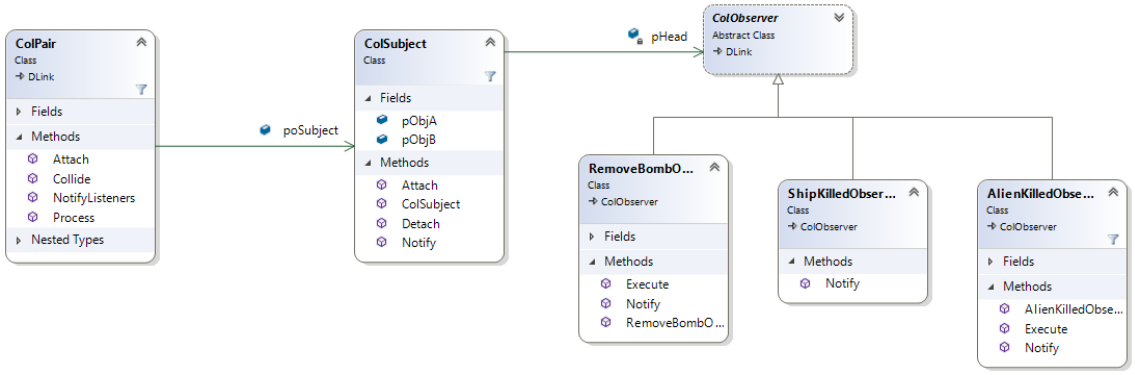
## Observer Pattern:

Now that the Visitor has set up the collision detection and notifies the listeners when said collision happens, it was time to create an observer class for each individual collision event. 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. I learned that the best thing about the observer pattern is that it is very reusable; meaning that I could notify as frequently as I needed to which became extremely helpful for a project of this magnitude. My observers had several distinct tasks to carry out when they were fired off:

- Remove the GameObjects(s) after the collision
- Play the appropriate sound file
- Change the Ship State (in the event the player ship was killed)
- Display a Splat Animation (in the event an Alien or UFO were killed)

- Speed Up Grid (in the event Alien is killed)

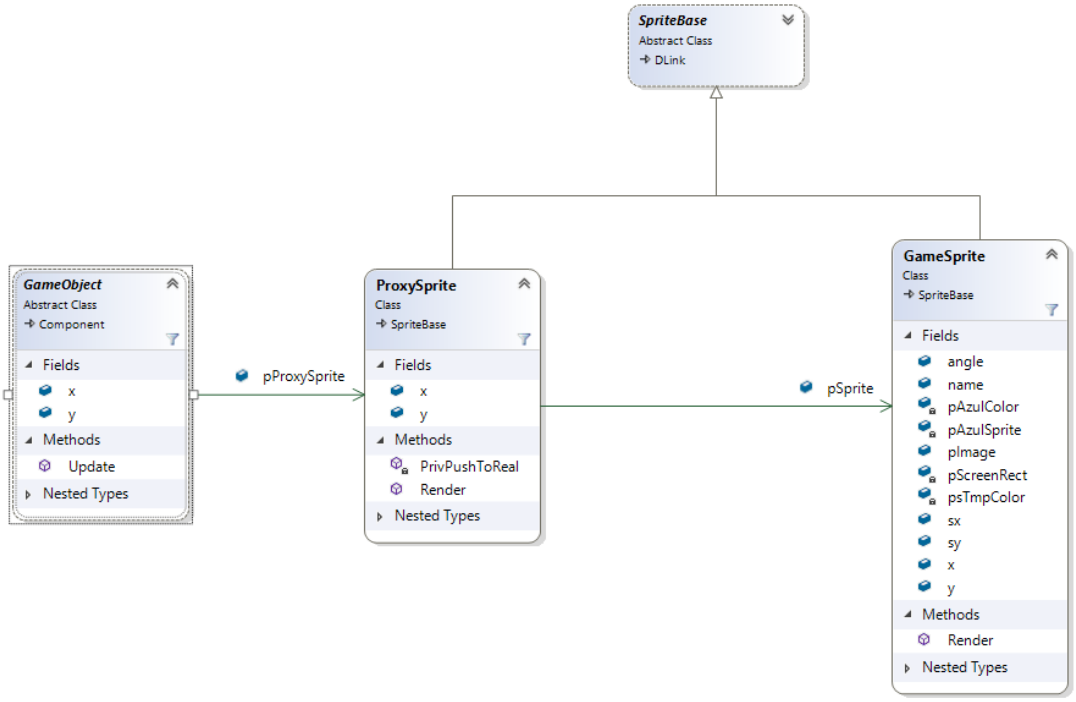
The UML diagram below shows an illustration of my Observer pattern in action.




From the diagram, lets consider how the ShipKilledObserver class works. Since all my Observers worked in conjunction with ColPair, I attached the ShipKilledObserver to the Ship vs Bomb pairing. Then, once the visitor notifies the listener that a collision happened the observer can do its magic. In this case, I have the observer play the explosion sound effect since the player ship had been struck by a bomb. Then, I changed the Ship state from Ship Ready to Ship End to indicate that the Player had been killed and had the x and y coordinates changed to have the new Ship spawn back at it's starting point. The AlienKilledObserver and the RemoveBombObserver work similarly to this as they both remove the GameObject, the alien observer plays the invader killed sound file to indicate that the alien has died and displays a splat where the alien was killed.

# Proxy Pattern:

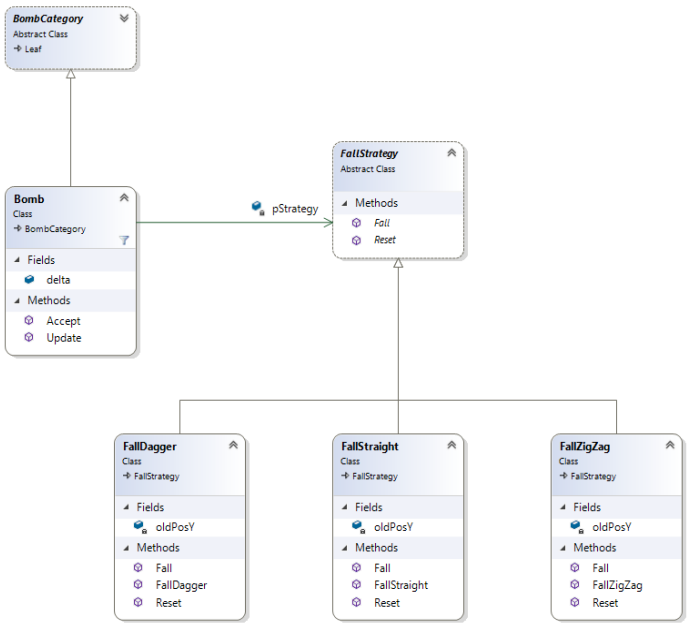
Sprites were used heavily throughout the development of the game. My Alien Grid is one such example of this as I had a total of 55 aliens grouped together but only used 3 different alien sprites for the 3 alien types. But the key problem here is that the sprites contain their own data such as their own x and y coordinates and creating 55 sprites for each individual alien would have made for a very inefficient game. To combat this problem, I used the Proxy design pattern. The purpose of the proxy pattern is to provide a surrogate or placeholder for another object to control the access of it.  The UML diagram below shows my Proxy pattern in action.




Within my application, the GameObject class held a reference to the ProxySprite which held a reference to the GameSprite class. In ProxySprite, I created a function that I called PrivPushToReal() where I set the data of pSprite to the corresponding Proxy Sprite data. After this, when the sprites are rendered, the Proxy Sprite switches with the Game Sprite to draw the various sprites on the screen. The usage of Proxy Sprites was also helpful with drawing the different alien sprites and UFO sprite on the attract scene as I had to modify the x and y coordinates for those sprites.

## Strategy Pattern:

At this point in the development process, I already got most of the components needed to fully recreate *Space Invaders* but a key feature I was still missing was having the alien columns drop random bombs from the lowest alien in each corner. The original game had 3 different bomb types each possessing a different behavior as they were falling to the ground. The different bomb types are the zig zag bombs, the straight bomb, and the rolling shot bomb. In order to accurately capture these behaviors, I decided to make use of the Strategy Design Pattern. The purpose of the pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. Below is my UML diagram to illustrate how I used the strategy pattern with the bombs.



First, I made a BombCategory class to create an enumeration for bomb types to help differentiate from Bomb objects and Bomb Root objects. Then, in the Bomb class, the bomb held a reference to FallStrategy and updated the fall with each passing frame. The abstract FallStrategy class had to function signatures called Fall and Reset that had to be overridden by its three subclasses. The magic of the strategy pattern is at play when the subclasses override the Fall function as they all must behave differently while they are falling. The FallStraight bomb class has no fall behavior so its Fall function was left blank. However, since the FallDagger performs an action where the bomb's dagger rotates from top to bottom, I made a function call to MultiplyScale inside of Fall to change the sx and sy coordinates of the bomb. The code snippet below provides an illustration of this function.

```
public override void Fall(Bomb pBomb)
{
    Debug.Assert(pBomb != null);

    float targetY = oldPosY - 1.0f * pBomb.GetBoundingBoxHeight();

    if(pBomb.y < targetY)
    {
        pBomb.MultiplyScale(1.0f, -1.0f);
        oldPosY = targetY;
    }
}
```

Similar actions were taken for the FallZigZag to create a bomb type where the bomb moves in a zig zag while falling.

## State Pattern:

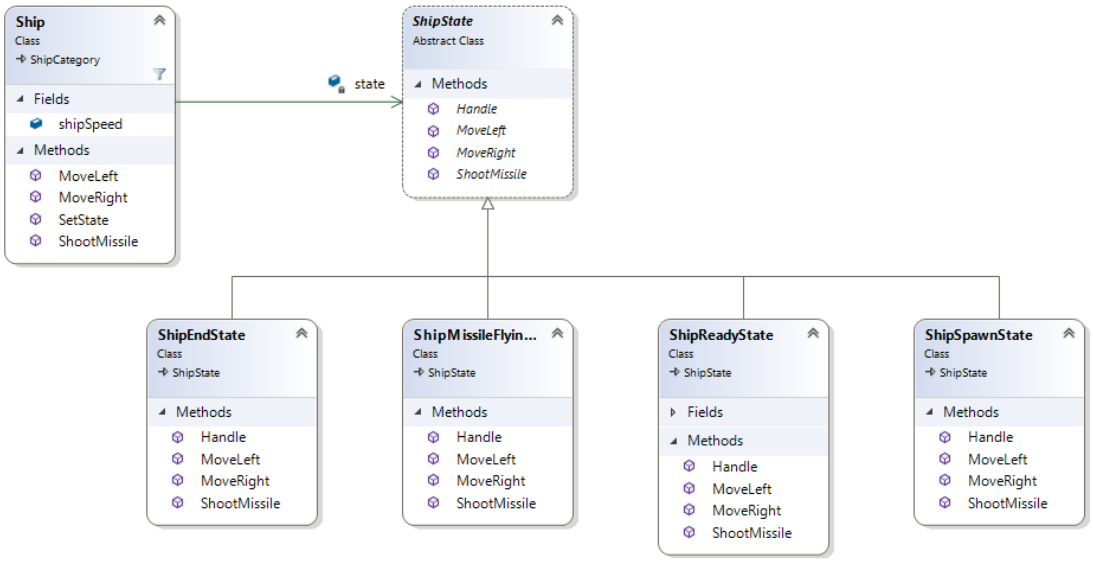
The final pattern I implemented for this project was the state pattern. This pattern helped me solve two key problems: Setting the various Player Ship states and cycling between the various Game States. The state pattern was the ideal choice for these problems because the pattern allowed me to alter the behavior of an object when its internal state changes.

The Player Ship Game Object had 4 distinct states:

- Ship Ready State (First state – starting point)
- Missile Firing State (When missile is shot)
- Ship Spawn State (After missile is shot)
- Ship End State (When Player is hit by bomb)

In regard to the Player Ship, the game loop always transitions around these 4 states. The Ship Ready and End states are set through observers when a collision happens. The Firing and Spawn states are set when the player presses the

spacebar to fire a missile.  Below is the UML diagram for the State pattern I used specifically for the ship states.




Each of the Ship states posses a Handle method. This handle is meant to switch states when the internal state changes. For example, when the Ship is in its ready state, if the player presses spacebar to shoot the missile, the state will switch to Missile Flying and then Spawn State. When spacebar is pressed on Spawn State the missile will respawn and the Ship moves back to ready state with the missile reloaded.

The State pattern was also helpful in allowing me to create the many different Game Scenes needed in the game and provided a quick and easy way to cycle through them in-game.

The Game Scenes had 5 distinct Game States:

- Attract Mode
- Level 1
- Level 2
- Game Won State
- Game Lost 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 level 1 and if they hit "0" they will be taken back to the Attract mode. In order to reach the 2nd Level or either of the Game Over states certain conditions must be met. To get to level 2, the player must be on level 1 and must have cleared the entire alien grid to advance. 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.

# Post-Mortem:

## Final Thoughts:

Overall, I had a lot of fun working on this project throughout the quarter. I believe this was an excellent way to learn about design patterns because I got to see each pattern at work in a real-life application. Working on this project also helped me shore up my knowledge with inheritance as well the linked list data structure. Of course, there were many times where I got frustrated because there were times when the project would break after adding more code which forced me to understand why things were breaking before finding a solution. However, it was extremely gratifying when I was able to fix those issues and get everything working in cohesion.

## Potential Improvements:

While I am incredibly proud of what I accomplished with this project, I still have a lot of things to improve. Obviously, I need to make a few touch ups with features that are in the game but not 100% working properly. Namely the resizing of the collision boxes after an entire column of aliens have been wiped out. Presently, the grid resizes if a column is destroyed, but not instantly which could cause the

aliens to jump down the grid a bit faster. The big feature I'm missing here that I would love to add in later is the 2 Player mode.