Space Invaders

SE456 Design Document

Terry Schmidt

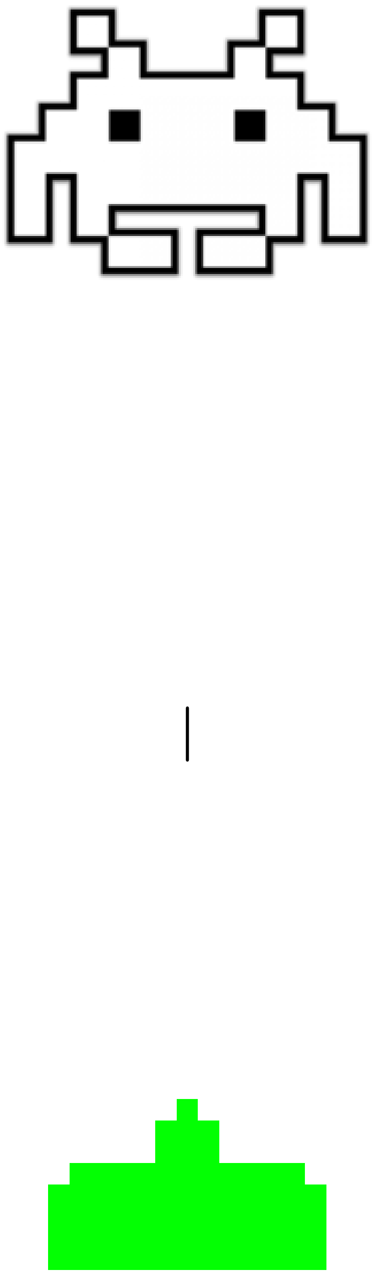


TABLE OF CONTENTS

INTRO

This is a design document for an implementation of the 1978 game Space Invaders made in totality by Terry Schmidt at DePaul University. It was made with the C# programming language in Microsoft's Visual Studio IDE. Object-oriented principles were emphasized and employed, as well as 12 design patterns.

- Singleton
- Command
- Priority Queue
- Factory
- Composite
- Iterator
- Adaptor
- Proxy
- Strategy
- State
- Visitor
- Observer

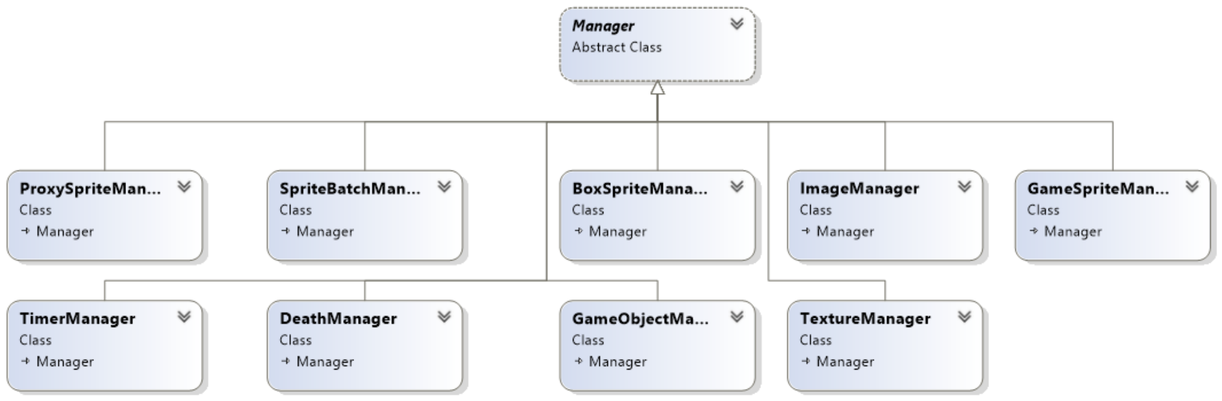
YOUTUBE DEMO:

https://youtu.be/4MnKbrozdvQ

MANAGERS AND THE SINGLETON PATTERN

► **Abstract classes:** Manager.cs, MLink.cs

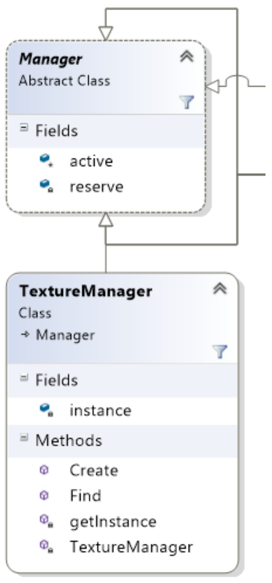
► **Derived classes:** ProxySpriteManager.cs, TimerManager.cs, DeathManager.cs, SpriteBatchManager.cs, BoxSpriteManager.cs, GameObjectManager.cs, ImageManager.cs, GameSpriteManager.cs, TextureManager.cs



The abstract manager class contains two MLink objects, one called active and another called reserve. Each of these is the head of a linked list which contains objects which are either active in the game or inactive (reserve). The derived managers manage their specific kind of objects – TimerManager manages TimerEvents, TextureManager manages Textures and so on.

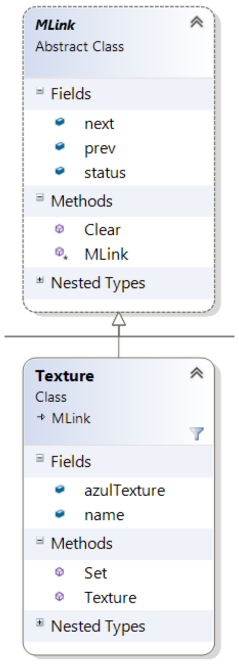
► **Design Patterns:** Singleton, Object Pool

Each of the derived managers are singletons. The singleton pattern solves the problem of somehow ensuring that only one instance of an object is ever made. It also provides a global point of access to that single instance. It accomplishes this by using composition. Each concrete manager holds a single instance of itself. They contain a single static instance of themselves. You can only create the instance by calling the managers static Create(int, int) method. This will check if the instance it holds of itself is null or not. If it is null, then it fully instantiates it and now that instance can be used later on. If Create(int, int) is called and the instance is not null, nothing will happen because the instance was already previously created. Keeping with the idea of singleton, a second instance will not be created because there may be one and only one. When the managers are created, this allows for there to be one and only one point at which to access that managers objects. I can use the manager later in the program by using TextureManager.Find() and by the same token ImageManager.Find and so on for the rest of the managers. This helps to ensure there is no repetition or unnecessary memory allocation in this area. Having tight control over memory allocation in a game setting is important to ensure the smoothest performance possible. Some UML (not an exhaustive diagram of fields and methods):



There are also a few flyweight patterns being used within the managers. The TextureManager is technically making use of a flyweight pattern within the project resources themselves – the texture TGA file. Its combining of all the sprites into a single TGA instead of having many TGA files for many different game sprites is itself a flyweight. GameSprites being managed within the GameSpriteManager are also making use of the flyweight pattern by sharing and reusing the images within them.

The singleton managers are also used to hold linked lists of objects that the game needs. Since all the managers need methods associated with this like Find(), Add(), Remove() and so on, this is implemented in the abstract Manager class. The object pooling is important again to control the amount of dynamic memory allocation happening during the gameplay, which minimizes lag. If a derived manager calls Add(), that uses the abstract managers Add(), and the abstract manager grabs a node from the reserve list and pushes it onto the active list. After that, the derived manager can call node.Set(object specific arguments), and it is **as if** we called new, without paying the penalty of *actually* calling new. But it gets better! Not only are we avoiding the cost of new, we are avoiding the cost of giving the new back (destruction). If an object in the game is done, it gets taken off the active list and sent back onto the reserve list. Avoiding many news and many garbage collections helps ensure the game runs as smooth as it possibly can. Good job, object pool! Some UML:



TIMER AND THE COMMAND PATTERN

▶ **Abstract classes:** Command.cs

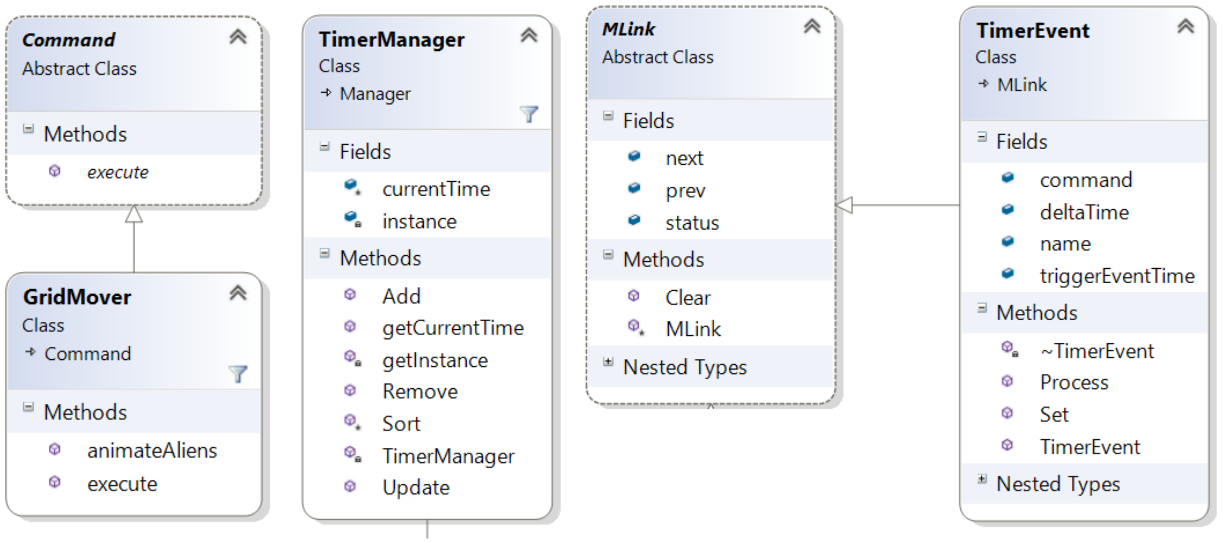
▶ **Concrete classes:** TimerManager.cs, TimerEvent.cs

For this game to function properly, a lot of different events need to be executed on a timer set at a certain interval. The command pattern is where a class derived from an abstract class simply implements the abstract class' only method: execute(). It solves the problem of encapsulating a request in an object. Many things in Space Invaders don't update with every frame. An example of this is the movement of the alien grid. It does not move during each frame update, it moves roughly every half second. To accommodate things like this, we need a timer that can execute a specific action at a specific interval. This is the responsibility of the TimerManager. Its unique field is a float holding the currentTime. It has an Add() function like all derived managers, which takes a command object and a float representing the time interval to repeatedly execute it at. The Add() function then grabs a node off of the reserve list, then Sets() it with the command and the interval it was given, and inserts it into the active list in sorted order. It inserts it in sorted order by walking the active linked list and comparing the intervals to its own interval, and then updating the links accordingly once it finds its place. TimerEvent has 2 floats associated with it, triggerEventTime and deltaTime. With these classes, all that must be done to use this code is to create a class derived from the abstract Command class, provide an implementation for its only required method which is execute(), and then add it on the TimerManager with an interval associated with it.

The TimerManager is an integral part of my implementation of Space Invaders. The launching of bombs by aliens, the UFO event, and the removal of the splat animations are all executed on a timer.

► **Design Patterns:** Command, Priority Queue

Here I'm using the Command pattern. It encapsulates an action, specifically the execute() method, in an object. Then, the actions associated with those objects are saved in a priority queue. The priority queue holds onto the objects in sorted order. In this case, the sorted order is made in relation to each TimerEvent's triggerEventTime. Having the list be sorted is an optimization which helps reduce the amount of code that has to execute during each update. The TimerManager has to walk the active linked list and execute the commands that have a triggerEventTime that is <= currentTime. If the linked list were not using the priority queue pattern, it would need to walk the entire linked list every single update, and the list could be very large in many gaming formats. Since the list is sorted, it just needs to walk the list and execute until it sees a triggerEventTime that is > currentTime, at which point it can break out. This way, the TimerManager is not doing any more work than it has to. Some UML:



ALIEN CREATION AND THE FACTORY PATTERN

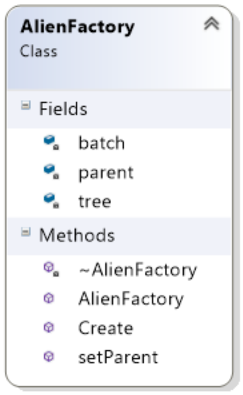
► **Abstract classes:**

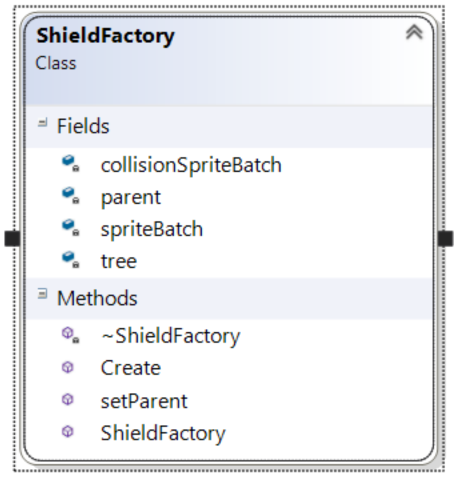
► **Concrete classes:** AlienFactory.cs, Crab.cs, Octopus.cs, Squid.cs, Grid.cs, Column.cs

For a single point to create aliens, I have made AlienFactory.cs. It has a Create() function that takes a type (Crab, Squid, Octopus, Grid) a game object name, and 2 floats for x and y positioning. It then creates the correct type of alien based upon the parameters passed to it by calling new Crab, new Squid, new Octopus, or new Grid. These calls to new are all done up front at the beginning, and not during gameplay.

Factory solves the problem of having a central place for all object creation. It accomplishes this by simply creating objects based on the arguments passed to its Create() function. This is used in the alien creation but also in shield creation as well for the individual bricks.

Some UML:





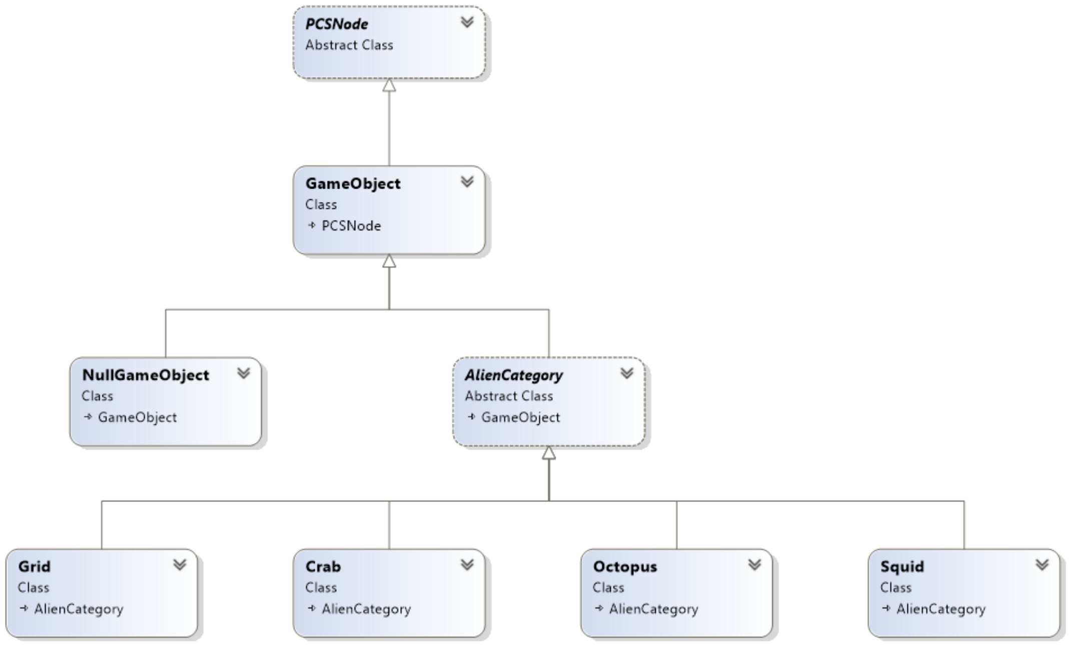
► **Design Patterns:** Factory

The factory is used to have a central place for object creation.

ALIEN GRID MOVEMENT AND ANIMATION

► **Abstract classes:** PCSNode.cs, AlienCategory.cs, Command.cs

► **Concrete classes:** GameObject.cs, Grid.cs, Crab.cs, Squid.cs, Octopus.cs, NullGameObject.cs, GridMover.cs



The PCSTree is a Parent, Child, Sibling tree. This structure is needed so that we may treat the aliens as a group uniformly. This is the way to make moving the alien grid move and making the individual aliens animate back and forth into something relatively easy instead of more complex if dealing with each alien in isolation.

To move and animate the grid, I created a class called GridMover.cs, which is derived from Command.cs and therefore has an execute() method. Inside this method, I get a reference to the alien grid by using GameObjectManager.Find(Grid). I can then call the grid's MoveGrid() method, which recursively moves each alien in the tree. Lastly, the aliens are animated in GridMover.animateAliens(). AnimateAliens() animates the real sprites underlying all the proxy sprites. It checks if the crabs are currently on the first crab frame, and if so, changes them to the second crab frame. If the crabs are on the second crab frame, it flips them back to the first, and so on for the rest of the aliens.

► **Design Patterns:**

In this section we make use of the composite pattern – we compose objects into a tree structure to represent a part-whole hierarchy. This allows the aliens to treat individual objects and compositions of objects uniformly.

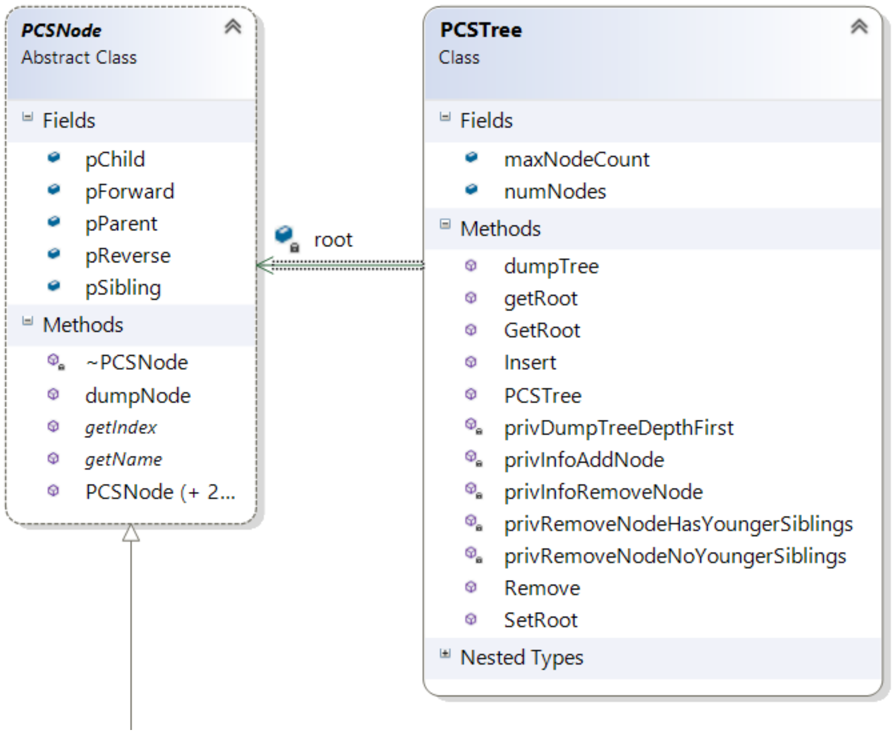
In this section, we also make use of the null object pattern. Rather simple, the null object is there to do nothing. It is an object designated to be there in the case of a lack of an object of a given type.

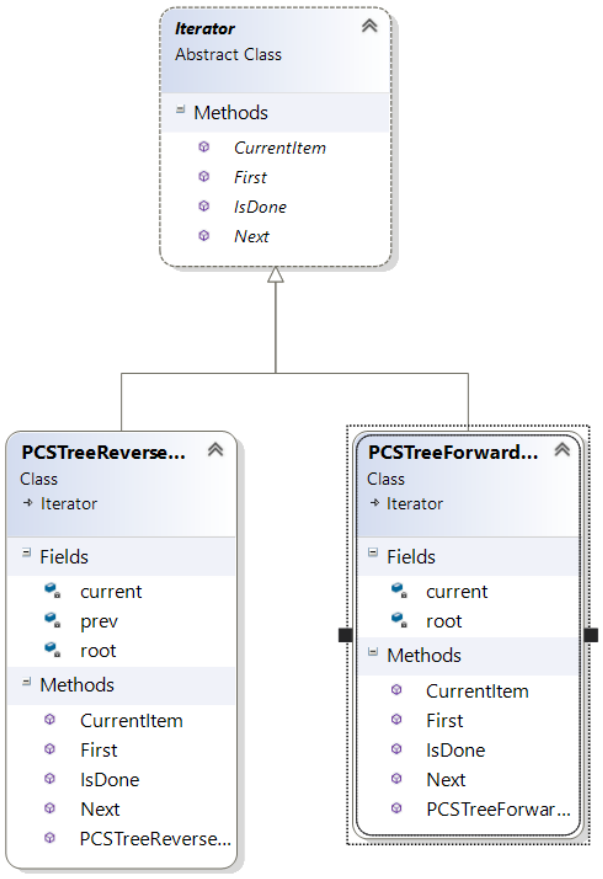
ALIENS AND THE COMPOSITE AND ITERATOR PATTERNS

► **Abstract classes:**

► **Concrete classes:** PCSNode.cs, PCSTree.cs, PCSTreeForwardIterator.cs, PCSTreeIterator.cs, PCSTreeReverseIterator.cs

There are a lot of operations that have to be done every frame for Space Invaders. I have to update object positions, look for collisions, and draw everything. Some of these operations need to be carried out sequentially. The purpose of the batches and trees is to perform some of these operations on either a part or the entire group. Object positions are determined by other objects in the tree. Grid derives its positions based on the positions of the columns for example. When an operation is called on the composite it can call that operation on all of its children, and some or all of those children could themselves be composites. Composite iterates through its structure as each operation on a composite object then calls the same operation on all of its children which is a way of doing depth-first iteration.





► **Design Patterns:**

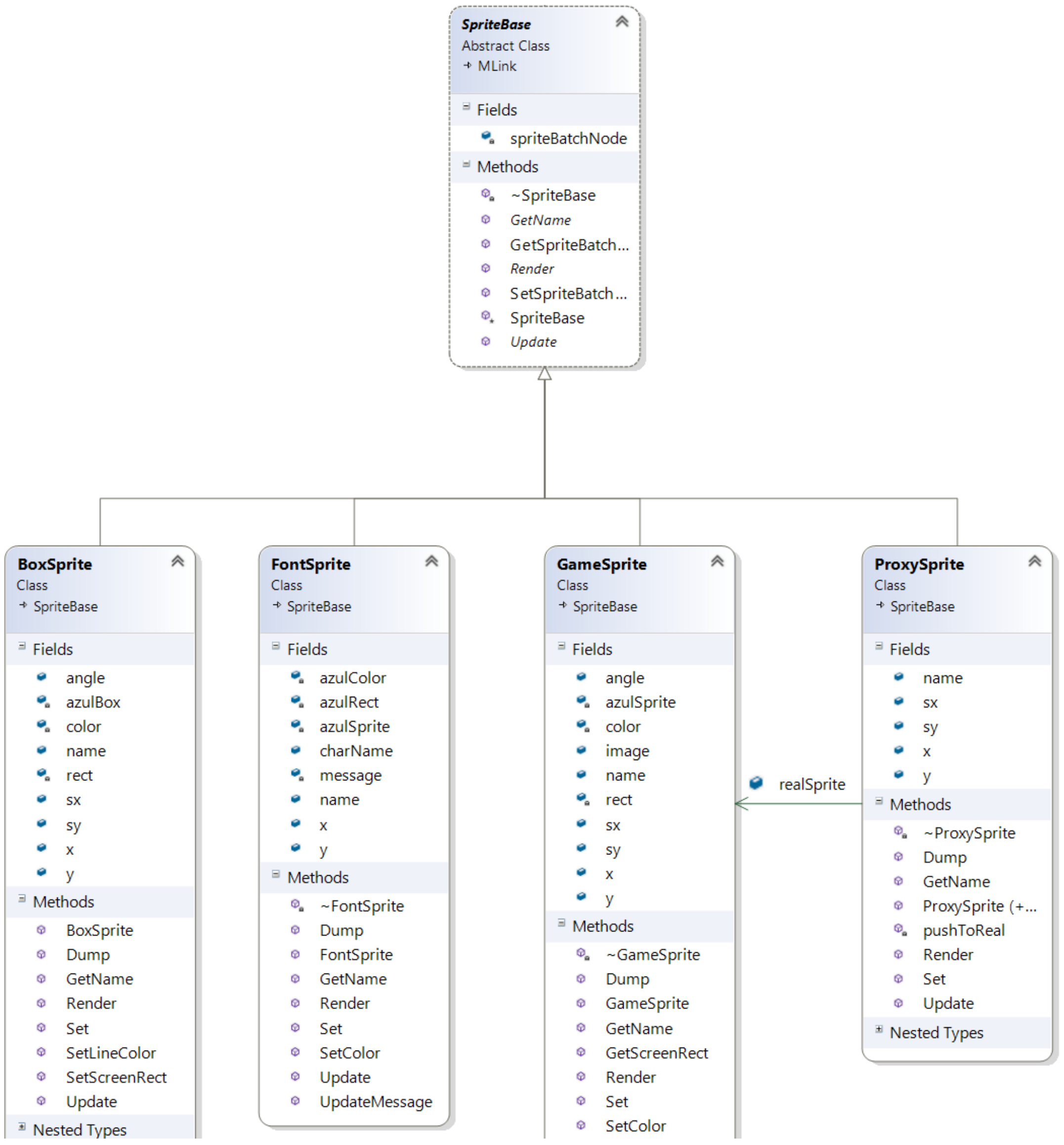
For the alien structure, a PCS tree is used. The composite pattern is used and this is a tree structure of simple and composite objects. It solves the problem of composing objects into a tree structure in order to represent part-whole hierarchies. Clients can treat individual objects and compositions of objects uniformly. It accomplishes this by using composition. The alien grid and also the shields are uses of the composite pattern.

The iterator pattern is used in order to solve the problem of having a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It accomplishes this by using inheritance. The abstract Iterator class has First(), Next(), IsDone() and CurrentItem() methods that the concrete iterators must implement. This is used for the sprite batch manager to draw all the sprites, as well as for the game object manager to update all of the objects in the game.

SPRITES, PROXY SPRITES, SPRITE BATCHES AND THE PROXY PATTERN

▶ **Abstract classes:** PCSNode.cs, AlienCategory.cs

▶ **Concrete classes:** GameObject.cs, Grid.cs, Crab.cs, Squid.cs, Octopus.cs, NullGameObject.cs



▶ **Design Patterns:** Adaptor, Proxy

In this section, I use the Adaptor pattern by using a new custom Sprite class which wraps the primitive Azul.Sprite class, and then adds functionality I want to it. This "adapts" the original Azul.Sprite class to the type that I want it to be, which is GameSprite, instead of its original Azul.Sprite.

In this section, I use the Proxy pattern. Proxy pattern has a few specific fields for itself, and uses some other class as the rest of its fields for everything else it has in common with that class. It solves the problem of reusing resources when possible, and eliminates code duplication also. It accomplishes this with composition, since the proxy has a field in it which is a real object. Instead of having many sprites which all have their own name, Azul.Sprite, Image, Color, Rectangle, x, y, sx, sy, and angle fields, I use a ProxySprite which can point to a real GameSprite object which it shares characteristics of, but has its own fields that it does not share with the real GameSprite – namely, the x and y coordinates. So we can have essentially, a carbon copy of a GameSprite with a ProxySprite, and the ProxySprite can control its own specific (x, y) fields while sharing the rest of the real GameSprite fields.

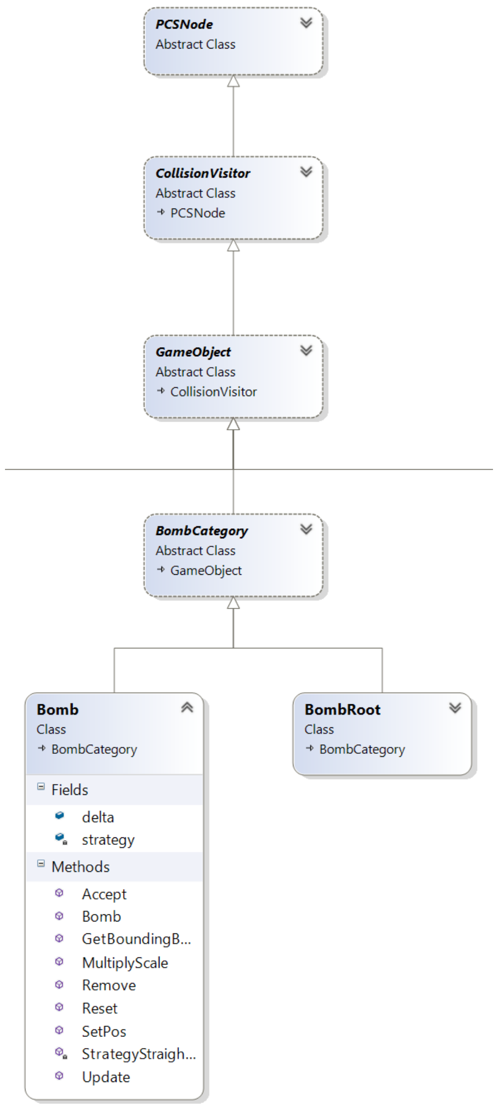
BOMBS AND THE STRATEGY PATTERN

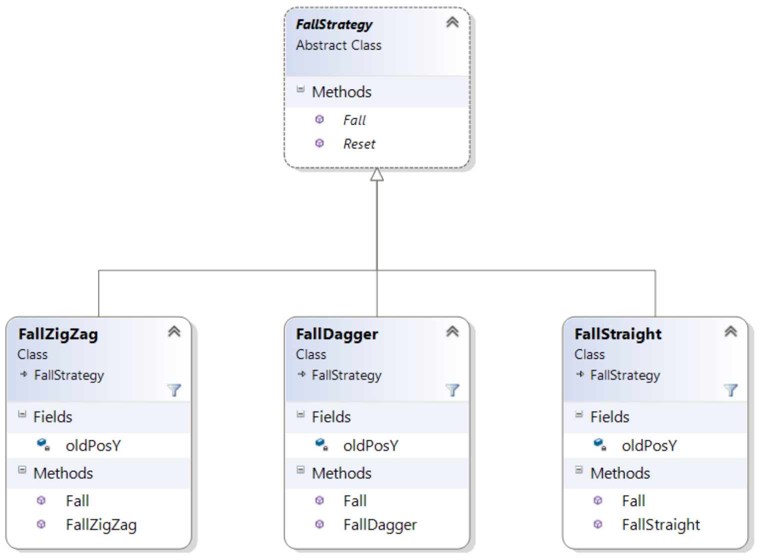
► **Abstract classes**: BombCategory.cs, FallStrategy.cs

► **Concrete classes**: Bomb.cs, BombRoot.cs, FallDagger.cs, FallStraight.cs, FallZigZag.cs

Bombs in Space Invaders can fall in three different ways. Straight (no movement), zig zagging, or dagger. To accommodate these different ways of falling for different bombs, the strategy pattern comes in handy. Every bomb has a private field called strategy which holds its personal FallStrategy object. That object can be straight, zigzag, or dagger. Inside each bomb’s override Update() function, I use that bomb’s specific strategy.Fall() function. This tells individual bombs what way they should fall. FallDagger, FallZigZag, and FallStraight all inherit from the generic FallStrategy class.

The actual dropping of the bombs is happening in a class called BombDropper.cs, which inherits from Command, and therefore gets put on the Timer over and over throughout the duration of the game. How this works is first I generate a random number. Based on that random number, I pick out the a column from the grid with that index, and check if it is null. If it’s not null, that means the column is still available to drop bombs in the game – meaning, that the player has not shot the entire column out yet. So now that it has found a valid column pseudorandomly, it can access that columns x and y coordinates and place a bomb appropriately based upon that, at which point it begins to Fall() during each game Update(), toward the player ship. Also, there is a small percentage chance that the UFO can also drop its own special bomb. Watch out!





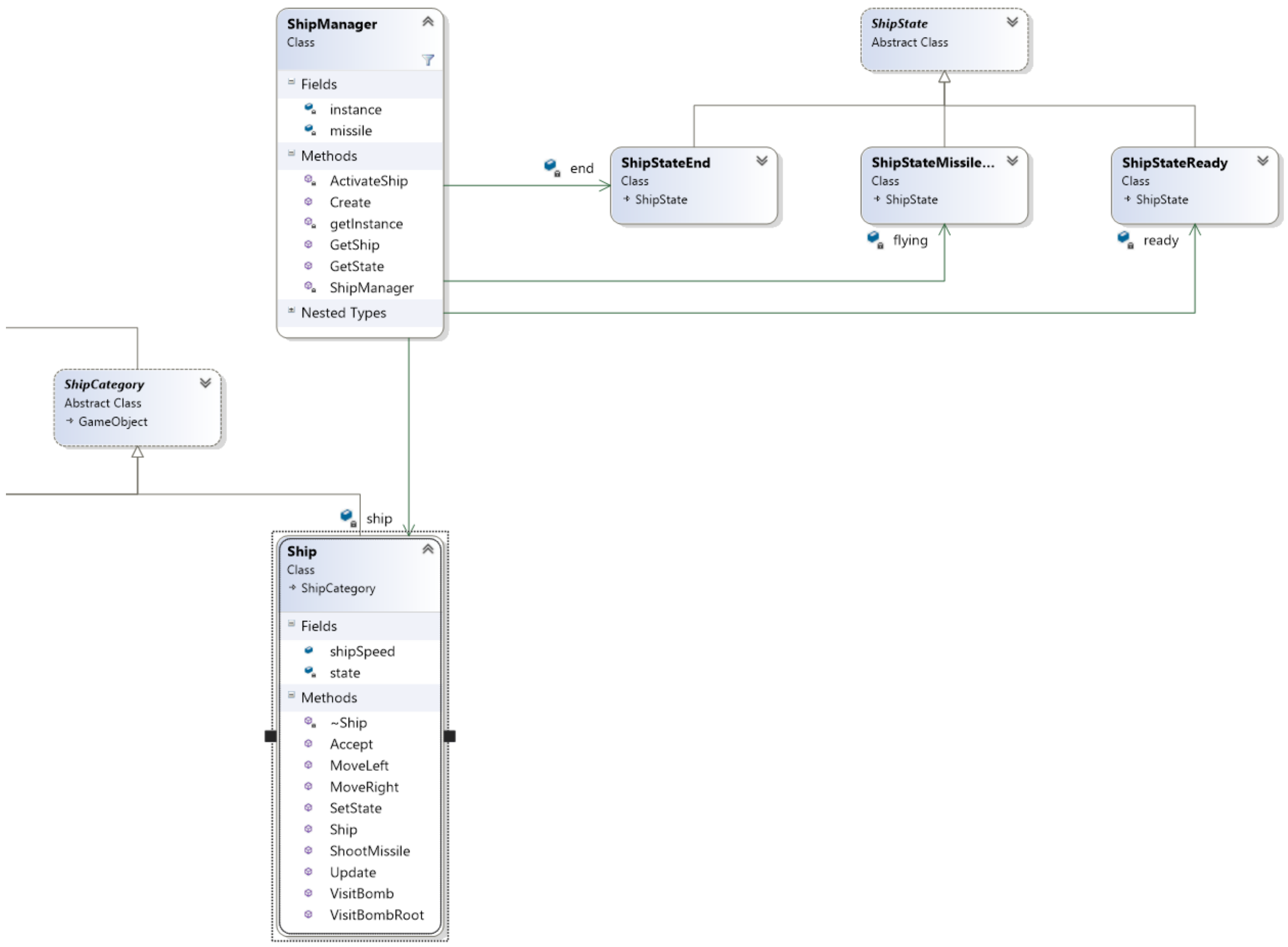
► **Design Patterns:**

Using the strategy pattern here was useful because it solved the problem of having an algorithm that varies independently of who uses it. In order to accomplish that, it uses both composition and inheritance. Bomb.cs contains a FallStrategy object, which gets set in the Bomb constructor. When a bomb is created, a FallStrategy is passed to the bomb constructor, and that FallStrategy object tells the Bomb how to Fall() during each game Update().

SHIP, SHIP MOVEMENT, AND THE STATE PATTERN

► **Abstract classes:** InputObserver.cs, ShipState.cs

► **Concrete classes:** Movement observers, ShipStateEnd, ShipStateMissileFlying, ShipStateReady, Ship, ShipManager



Movement of the ship is occurring inside of the InputManager. The InputManager's Update() function is getting called every frame, and it checks the keyboard for input. When it detects relevant input (arrow keys, space bar, etc.) it can then do something specific based upon that. When the spacebar is pressed, the InputManager notifies its subject about it and as a result, all of the observers watching for that pressing of the spacebar event get notified that the event they are waiting for did indeed occur. Those observers can then snap into action, and provide functionality after the fact. In my projet, there are 3 observers attached to the input subject. The MoveLeftObserver, MoveRightObserver, and ShootObserver. These are the observers who are watching and waiting to be notified of their specific event, who then execute some code knowing about it.

So now we see that we are taking some action based upon some event (a keyboard press). But what if the game rules dictate that that action can't currently take place? Example: if the ship fires it's missile, and the missile is still in flight, the user cannot fire another, no matter how many times they furiously hit the space bar asking for one. But we just made our ShootObserver launch a missile every time the user presses spacebar didn't we? State pattern to the rescue!

The Ship can be in different states – ready, end, or missileflying. Each of these states have their own specific implementation of MoveLeft, MoveRight, and Shoot(). Ship has its personal ShipState object, which changes throughout the game, and we use each ShipState's implementations of these functions when it is appropriate according to the game rules. The state pattern just saved me from a ridiculous amount of if-statements, and besides that, it's my favorite pattern.

► **Design Patterns:**

For the ship and its movement, I used the state pattern. The state pattern comes in handy here because it allows for the ship to have different "states". It has a Ready state, end state, and a missile flying state. Each of these concrete states has its own implementation of the abstract methods in ShipState.cs which are MoveLeft(), MoveRight(), and ShootMissile(). The ship has a single current state, which might be any of those 3 states at any point in time, based upon the game events. This solves the problem of needing different implementations of methods at different points, or based on different events that have occurred in the game. It accomplishes this goal by using inheritance. There is an abstract ShipState class which the derived states inherit from and implement its methods.

COLLISIONS AND THE VISITOR AND OBSERVER PATTERNS

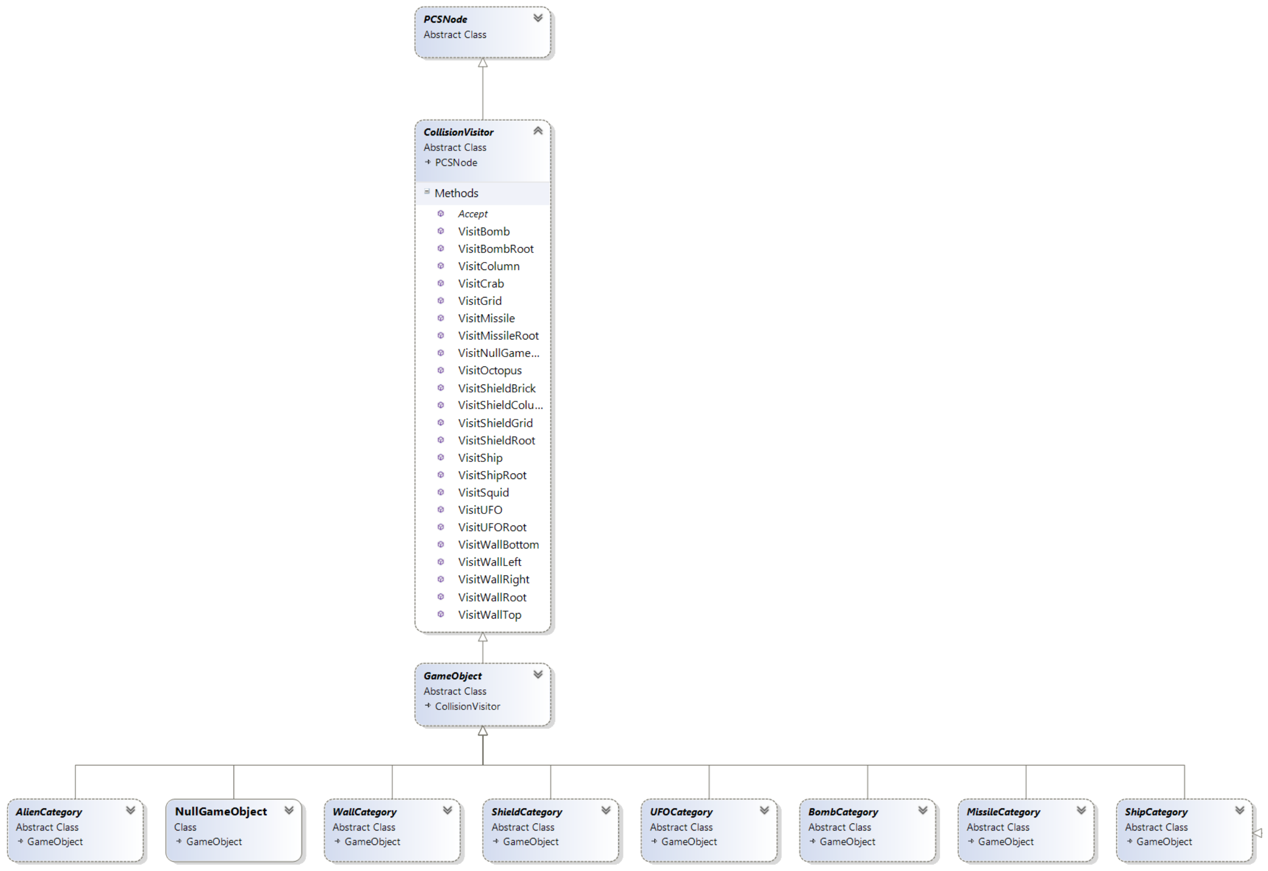
► **Abstract classes:** CollisionVisitor.cs, CollisionObserver.cs

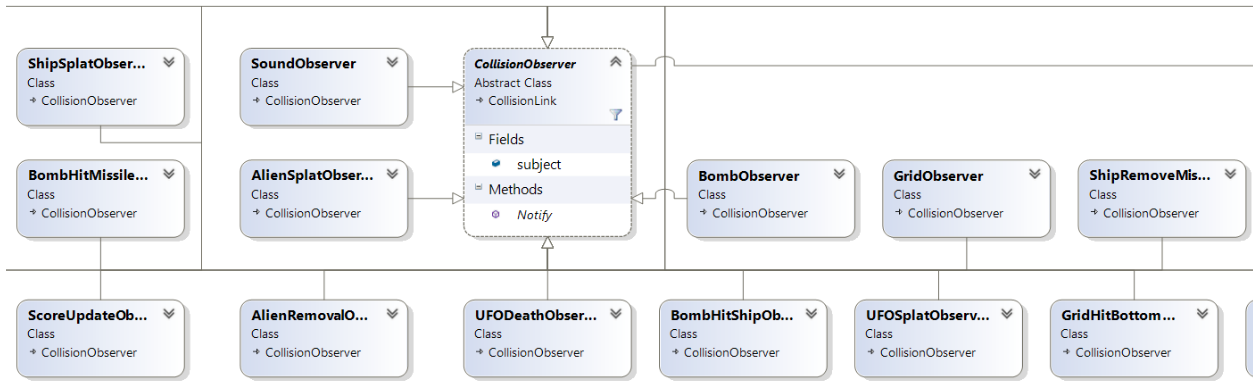
► **Concrete classes:** Way too many to list.

Here we are, the real meat of Space Invaders is right here. You've arrived! Here is where it is determined which objects collided, if any and also, what might need to be done as a result. The Visitor patterns very important main responsibility is to give a way to determine which two specific objects collided after a collision. Just knowing that two game objects collided isn't enough, I need to know what kinds. Was it a missile vs a bomb? Did the missile hit an alien? Did a bomb hit the ship? This is all the domain of the Visitor pattern. The observer pattern, working side by side with its Visitor pattern friend, is concerned with figuring out what needs to be done as a result of those specific collisions.

The abstract class CollisionVisitor is the class that defines the many different Visit() functions. VisitShip(), VisitBomb(), VisitWallRoot(), it goes on and on. All of the derived GameObject classes, Alien, Wall, Shield and so on can then override only the specific Visit functions that it needs to, based upon the game rules. For example, WallLeft is never going to collide with WallRight. Neither of them need to implement a Visit function for the other because based upon the game rules, it's impossible. So only a small subsect of all the possible Visit______() functions need to actually get implemented. And in most cases, the implementations of those functions are very simple: they simply call the notify() function of their observers. This is where Visitor passes the work to his best buddy Observer. There are many, many observers in this project which are watching a collision pair and when notify() of their collision by the Visitor pattern, the observer's execute() function is called. For example, I have an observer attached to the collision pair AlienGrid/MissileRoot. When the two collide, the observer removes the alien from the grid. Another observer places a splat image at that spot. Another observer adds points to the players score. And so on.

So there are many different observers doing many different things based upon these collisions it is notified about by the Visitor pattern. Here are some of the more special and unique observers: NewWaveObserver.cs is an observer watching the collisions between the missile and the aliens. When a missile hits an alien, it checks if it is the last alien in the grid (the 55th alien) and if so, that observer sets in motion the required steps to load a new wave of 55 aliens. GridHitBottomObserver.cs is watching to see if the grid hits the bottom wall. If so, it's an immediate game over. It pauses all the timers and places the Game Over message on the screen.





► **Design Patterns:**

I use the Visitor pattern in association with the collision system. Visitor pattern declares a visit function for each class in the structure. The function names are indicative of the class that sends the Visit request to the visitor. This allows the visitor to determine the concrete class of the element being visited. The visitor pattern solves the problem of defining a new operation to a class. It helps Space Invaders conclude which two specific objects collided at a given point. It accomplishes this by using double dispatch.

I also used the observer pattern in association with the collision system. The observer pattern is noticeable by being an object that gets attached to and observes some event. When the event occurs, it’s notify() function is called in the Visitor pattern, and it then executes its to-do list. This solves the problem of having a way to notify any number of objects that an event took place, and then those observers execute their code in association with that event. It accomplishes this goal by using inheritance and aggregation. There is inheritance between abstract observer and the concrete observers, and another class has references to those observers and can call their notify() function when an event takes place. Collision observers are attached to the collision pairs and when it is determined by the Visitor pattern that a collision happened between one of the pairs, the observers for that pair have their notify() functions called and something specific is done in that function as a result.

FINAL DISCUSSION: WHY IS THIS A GOOD DESIGN?

- Loosely coupled classes doing specific things
- Adheres to SOLID object-oriented principles
- Reuse of resources
- Avoid paying the penalties of memory allocation and deallocation during run time

POST-MORTEM: IMPROVEMENTS

► 2 player

Make 2 player functionality. For this, there would need to be a scene manager which would be holding references to each players Grid and Shields.

► Tunneling

Remove or minimize tunneling. I noticed in the game play videos of the original space invaders, that the missile is quite fast in the game. Similarly, my missile is somewhat fast and as a result there’s tunneling.

► Shield Regeneration

Regenerate shields for each new level.

► Shield erasing for all aliens

Have all the aliens erase the shields.