Space Invaders

Carlo Caunca

SE456

Winter 2016 Q1

Table of Contents

Objective

To recreate the classic arcade shooter game Space Invaders using modern software development design patterns.

Design Patterns

Although this game may seem simple, there are many aspects to it that require much thought in order to be implemented intelligently and efficiently. There were many design patterns used throughout this game.

I originally wrote this design document combining Implementation with the Design of it. Many of these Design Patterns will make mention of Implementation that will be explained further below.

Singleton

We have a problem of general management of each component of this game, from images and sounds, to textures and fonts we need an efficient way to refer to the single instances of each of these managers in order for them to interact with one another. In order to do so, I created, for instance, an Image manager to manage all images used in this game with the Singleton pattern. The Singleton pattern ensures a class has only one instance and provides a global point of access to it. This problem is solved by taking advantage of this pattern's lazy instantiation, allowing the Manager class to not be initialized until the first time it is used. In Code Snippet 1, by making the constructor and **GetInstance**() methods private, enforces client code to call the public static method Create in order to create an instance of the Manager class. Once Create is called, it can be used as shown in Code Snippet 2, by first calling the static GetInstance() method before using it as needed. This is used in Space Invaders for the following management systems: collision pairs, delayed game objects (used to properly delete a game object once and only once), fonts, game states, game objects, images, input, proxy sprites, score, ship, sounds, sprite batches, sprite boxes, sprites, textures and timer events. Virtually every system in the Space Invaders game is managed by a Singleton manager class.

These managers are designed with the Singleton design pattern to not allow multiple manager instances at a given time since we want one and only one manager managing assets throughout the game. The way it has been implemented is a bit of a variation of the textbook Singleton design pattern:

Code Snippet 1

```
public class ImageManager : Manager
{
    private static ImageManager pInstance;
    private ImageManager(int reserveSize = 5, int reserveIncrement = 2) :
        base(reserveSize, reserveIncrement)
    {
    }
    public static void Create(int reserveSize = 3, int reserveIncrement = 1)
    {
        Debug.Assert(reserveSize > 0);
        Debug.Assert(reserveIncrement > 0);

        if (pInstance == null)
        {
            pInstance = new ImageManager(reserveSize, reserveIncrement);

            Image image = ImageManager.Add(ImageName.Null, TextureName.Null, 0, 0, 1, 1);
            Debug.Assert(image != null);
        }
    }
    private static ImageManager GetInstance()
    {
        Debug.Assert(pInstance != null);
        return pInstance;
    }
```

There is still the private instance class variable and the GetInstance() static method, however the GetInstance() method only returns a non-null instance in order to reinforce users of the manager to call Create() before using the manager. As can be seen in the code snippet above, to stay true to the Singleton

design pattern each Manager class has a private static instance variable to hold the single instance for that given class. And when Create is called there is a null check for the instance variable and if that instance is null, then we call the private constructor which creates a new manager instance variable. However if the instance variable is not null we simply do nothing upon calling Create().

Code Snippet 2

```
public static Texture Add(TextureName texName, string assetName)
{
    TextureManager texMan = TextureManager.GetInstance();
    Texture tex = (Texture)texMan.BaseAdd();
    Debug.Assert(tex != null);
    tex.Set(texName, assetName);
    return tex;
}
```

As can be seen above in each public static manager method, we first call GetInstance() before we do anything, and within GetInstance() if no instance is available we Assert, hence we reinforce clients using the manager to call Create() before being able to use the manager class and therefore reinforcing the singleton design pattern.

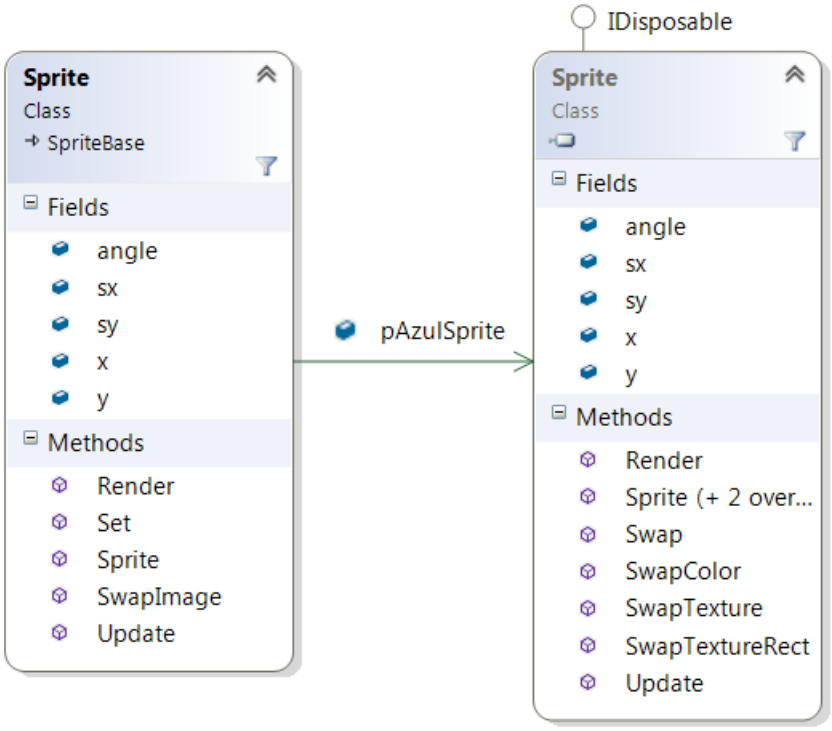
So in the beginning of the LoadContent() function in Game.cs we initialize each Manager by calling the static Create() method and specify the size of the reserve pool we wish to create and the number of nodes we want to generate once we run out of nodes on the reserve pool. Once each Manager's Create() method has been called, any client of that particular Manager can call any public static method of that manager to manage that specific asset.

Adapter

In order to separate and abstract the Azul data types from the rest of our custom game types, I used the Adapter pattern.

The Adapter pattern converts the interface of a class into another interface clients expect. This pattern lets classes work together that couldn't otherwise because of incompatible interfaces. This is accomplished by aggregation and creating a wrapper class that acts as almost an interface into the class that we desire to interact with. Depending on what purposes we intend for this class we build our wrapper class accordingly.

Adapter



In Space Invaders, I created my own Sprite class that holds a reference to an Azul.Sprite and as can be seen above my Sprite class has nearly the same properties and methods as the Azul.Sprite class. There are a few exceptions with some of the methods, where my Set() function calls the Azul.Sprite's Swap() function and my SwapImage() function calls the Azul.Sprite's SwapTexture() and SwapTextureRect() functions. My Sprite class acts as the Adapter to the Azul.Sprite's Adaptee role in the Adapter pattern.

Iterator

As we were instructed to do so early on in this project we were told that we could not use built in C# containers like lists or arrays, so we need to create our own. In doing so, the main data structures that I've built myself and use throughout this game are doubly linked and singly linked lists which are used mainly in our Managers (as discussed above). Now in order to iterate through these linked list structures I take advantage of the Iterator pattern.

The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation and places responsibility where it should be. This is done simply by delegation. An abstract Iterator class can simply take your collection and create an iterator (via the createIterator() method) from it whilst using its basic iterator interface. The iterator interface has the following essential methods, hasNext(), next() and remove(). However, the most important in creating and using an iterator to iterate through collections are hasNext(), next() and createIterator().

In Space Invaders, I am using the Iterator pattern when traversing a GameObject PCSTree collection to find a particular GameObject:

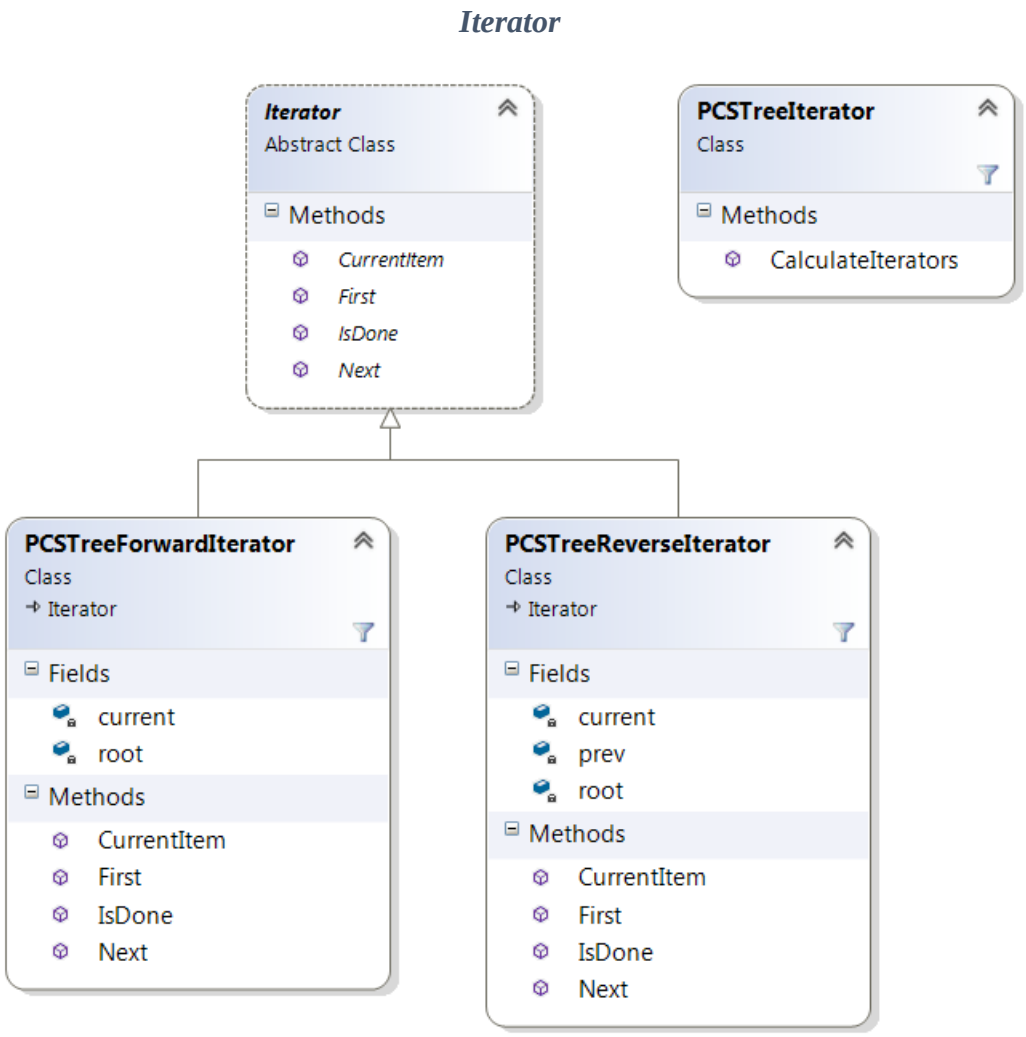
Code Snippet 3

```
public static GameObject Find(GameObjectName goName, int index=0)
{
    GameObjectManager goMan = GameObjectManager.GetInstance();
    GameObjectNode pRoot = (GameObjectNode)goMan.pActive;
    GameObject pGameObj = null;

    bool found = false;
    while (pRoot != null && found == false)
    {
        PCSTreeForwardIterator iter = new PCSTreeForwardIterator(pRoot.pGameObject);
        pGameObj = (GameObject)iter.First();

        while (!iter.IsDone())
        {
            if ((pGameObj.gameObjectName == goName) && (pGameObj.index == index))
            {
                found = true;
                break;
            }
            pGameObj = (GameObject)iter.Next();
        }
        pRoot = (GameObjectNode)pRoot.pDNext;
    }
    return pGameObj;
}
```

Using the PCSTree class as the basis collection to create a PCSTreeForwardIterator, it's usage is similar to the classic iterator. Instead of using a while loop and setting the current pointer to the first link in the list, then at the end of the while loop, setting curr to curr.next (as seen above in Code Snippet 3) I create the iterator first, select the first item, and use the iterator.IsDone() as the conditional in the while loop. Then at the end of the while loop I use the iterator's Next() method to get the next item in the linked list:



However, instead of hasNext() I am using IsDone() which operates essentially the same way. This allows me to create an iterator on any collection without having to know its implementation.

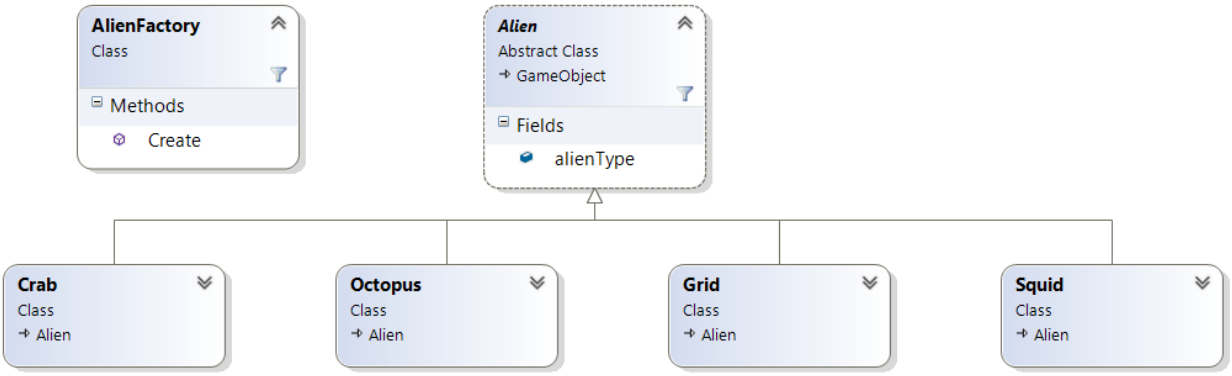
These design patterns will be evident in the implementation below.

Factory

We need to create alien sprites to be used and placed in a 5x11 grid with specific sprites populating each of the 5 rows in a particular order and in specific locations. In order to solve this problem I decided create an abstract Alien class to represent the general alien sprite on screen along with the Factory pattern to produce these aliens. The Factory pattern solves the design problem developers face with lumping object creation with object instantiation. Oftentimes, developers instantiate objects before knowing what they want and how they intend on using these objects. This creational pattern quells this predicament by keeping objects cohesive, decoupled and testable. The Factory method design pattern handles this problem by defining a separate method for creating objects; and the (Crab, Squid, Octopus and Grid classes as seen in Code Snippet 4) subclasses can then override to specify the derived type of product that will be created. By using inheritance, these subclasses derive from the Alien abstract base class to be used by the AlienFactory Create() method (seen below in Code Snippet 4) and defers instantiation to the subclasses to create the corresponding Alien object (Factory pattern product). As it is used in Space

Invaders, we pass in the AlienType we desire to create along with its x and y coordinates where we want them to be placed and a corresponding alien of that type and that coordinate is created.

Factory 1



The Alien Factory class is designed to follow the Factory design pattern. The products in this case correspond to the three main NPC types, a squid, an octopus and a crab. Within the Alien Factory class is a Create() method which takes as a parameter an AlienType (product type) that we want to generate:

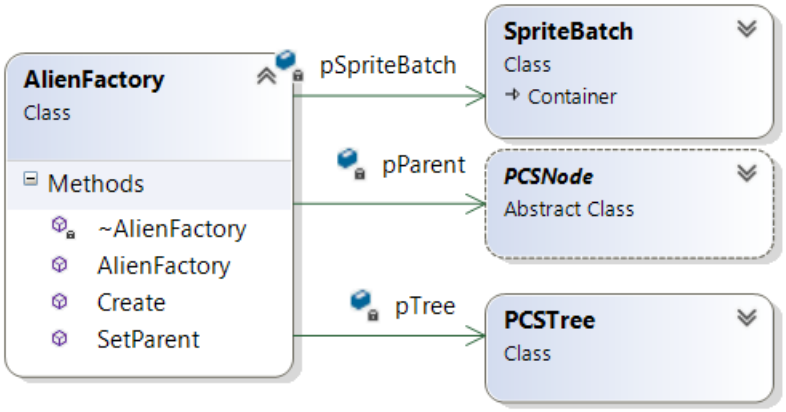
Code Snippet 4

```
public Alien Create(AlienType alienType, float x, float y)
{
    Alien pAlien = null;
    switch(alienType)
    {
        case AlienType.Crab :
            pAlien = new Crab(GameObjectName.Crab, SpriteBaseName.Crab, x, y);
            break;
        case AlienType.Squid :
            pAlien = new Squid(GameObjectName.Squid, SpriteBaseName.Squid, x, y);
            break;
        case AlienType.Octopus :
            pAlien = new Octopus(GameObjectName.Octopus, SpriteBaseName.Octopus, x, y);
            break;
        case AlienType.Hierarchy :
            pAlien = new Grid();
            break;
        default :
            Debug.Assert(pAlien != null);
            break;
    }
    GameObjectManager.Add(pAlien);
    this.pTree.Insert(pAlien, this.pParent);
    this.pSpriteBatch.Attach(pAlien.pSprite);
    return pAlien;
```

As can be seen above the Create() method returns an Alien object, which is an abstract base type that is designed specifically for the PCS Tree design pattern that will be described later. This also helps in categorizing and organizing our NPC types and will deem valuable in other areas of the game from a

design perspective. This Create() method corresponds to the concrete factory in the factory pattern and each object constructor that is called within the switch statement, the Crab, Squid, Octopus and Grid Alien objects represent the concrete product in the factory pattern. Once the Alien object (product) is created, it is added to the Game Object Manager, inserted into the associated PCSTree and then added to the associated Sprite Batch:

Factory 2



Each AlienFactory instance has a PCSNode, SpriteBatch and PCSTree associated with it, in this section we will focus on the sprite batch associated with object.

Code Snippet 5

```
//--------------------------------------------------------------------------
// Create GameObjects via AlienFactory
//--------------------------------------------------------------------------
PCSTree pcsTree = new PCSTree();
AlienFactory alienFactory = new AlienFactory(SpriteBatchName.Aliens, pcsTree);
Alien pGrid = alienFactory.Create(AlienType.Hierarchy, 0.0f, 0.0f);
alienFactory.SetParent(pGrid);
for (int i = 0; i < 11; i++)
{
    int additive = _horizontalSpace * i;
    alienFactory.Create(AlienType.Squid, 57.0f + additive, 800.0f);
    alienFactory.Create(AlienType.Crab, 51.5f + additive, 725.0f);
    alienFactory.Create(AlienType.Crab, 51.5f + additive, 650.0f);
    alienFactory.Create(AlienType.Octopus, 50.0f + additive, 575.0f);
    alienFactory.Create(AlienType.Octopus, 50.0f + additive, 500.0f);
}
```

Within the LoadContent() method in Game.cs, first a AlienFactory object is created by specifying a SpriteBatchName which will associate this factory instance with a sprite batch. In the code snippet above, the for loop will call the AlienFactory's Create() method to create each of the alien types (products) that we want to draw (by adding the created alien in the factory's Create() method to its associated sprite batch) to the screen. Once this for loop completes, the Alien Factory will have created (added them to sprite batch) the 5x11 grid of aliens that will be generated onto the screen.

PCS Tree

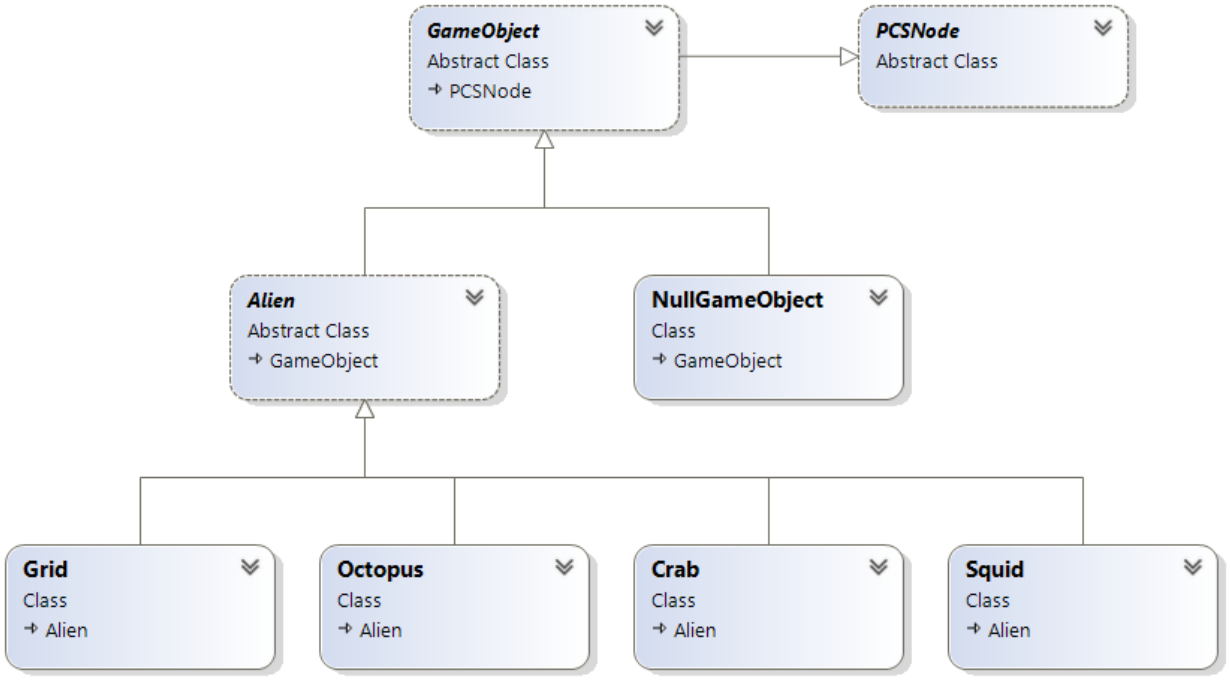
We have the problem of moving the alien sprites together in unison across the screen. In order to solve this problem I used the Composite pattern or in this class the PCS Tree class. The PCS Tree is an alternate method to the Composite design pattern which allows you to compose objects into tree structures to represent part-whole hierarchies. This enables clients to treat individual objects and compositions of objects uniformly. This pattern accomplishes the job multiple inheritance in a manner that creates a hierarchy of objects (as seen in PCS Tree) all inheriting from the PCSNode abstract class which is concrete in the leaves of our hierarchy tree (Grid, Octopus, Crab and Squid are all concrete subclasses that implement the PCSNode). However each subclass deriving from PCSNode is itself treated like a PCSNode, so when we construct this tree we set the Grid as the parent node and insert each Octopus, Crab and Squid into the tree as children of the Grid. So when we can Grid.MoveTree() is called on the Grid GameObject we iterate through its children and move them as well.

```
private void MoveTree(GameObject pNode)
{
    PCSNode pChild = null;
    pNode.x += this.movementXDirection;
    pNode.y += this.movementYDirection;
    pNode.Update();
    if (pNode.pChild == null)
    {
        // base case
    }
    else
    {
        pChild = pNode.pChild;
        while (pChild != null)
        {
            MoveTree((GameObject)pChild);
            pChild = pChild.pSibling;
        }
    }
}
```

This is used in Space Invaders not only for Grid movement, but it's also used heavily in the CollisionSystem, where we have the Grid as the parent, each column of aliens is a child of the grid and also a parent of each of the alien objects that make up each column. If another GameObject collides with the Grid, then we traverse into the Grid's children and check if there's a collision with any of the columns, then if there is we traverse into the column's children reaching the alien objects and if there's a collision there then we handle it accordingly.

The PCS Tree design pattern helps in creating a game object hierarchy to lay the groundwork for the Grid class that will be used to group the 5 rows of 11 aliens. Given the PCSTree.cs and PCSNode.cs files from class along with the code snippets above in the Factory section (*Code Snippet 4* and *Code Snippet 5*), The PCS Tree was implemented by first creating an instance of the PCSTree and passing it the AlienFactory constructor, which simply sets that pParent to the PCSTree that was passed in as a parameter to the constructor.

PCS Tree



The GameObject abstract class inherits from the abstract PCSNode class, in turn treating all GameObject derived classes as PCSNodes. Due to this fact, when AlienFactory's Create() method is called (refer to *Code Snippet 4*) any class that derives from Game Object can be added to the AlienFactory's PCSTree. So after the AlienFactory is instantiated, the grid alien object is created as an AlienType.Hierarchy (this is a GameObject that doesn't get drawn to the screen) type of alien, and then the newly created grid is set as the parent of the AlienFactory's PCSTree. When Create() is called in AlienFactory and the specific AlienType is created and inserted into the AlienFactory's PCSTree, all of these PCSNodes that are inserted are inserted into the PCSTree with the grid object as parent. Therefore, all of these Alien objects that are generated by the AlienFactory in this for loop are added to the PCSTree with the grid as the parent node.

With the AlienFactory now holding a PCSTree with the grid as the parent node and all aliens as its children nodes, in the Update() method in Game.cs (*Code Snippet 6*) this grid of aliens will all move by making a single call to the Grid object's Move() method within the TimerManager (this will be explained later) Update() method (*Code Snippet 7*).

Code Snippet 6

```
public override void Update()
{
    Grid pGrid = (Grid)GameObjectManager.Find(GameObjectName.Grid);
    TimerManager.Update(this.GetTime(), pGrid);
    GameObjectManager.Update();
}
```

Code Snippet 7

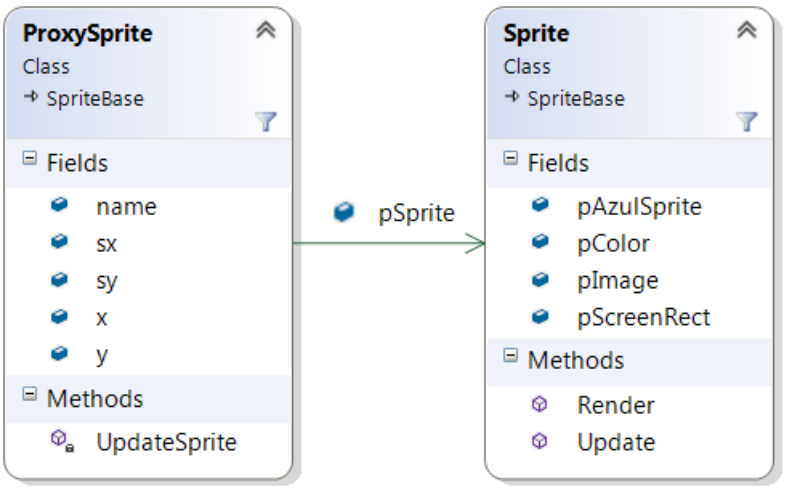
```
public static void Update(float currentTime, Grid pGrid)
{
    TimerManager timerMan = TimerManager.GetInstance();
    timerMan.currentTime = currentTime;
    DLink curr = timerMan.pActive;
    TimerEvent te = (TimerEvent)curr;
    while (curr != null)
    {
        if (timerMan.currentTime >= te.triggerTime)
        {
            te.Process(currentTime);
            pGrid.Move();
            timerMan.BaseRemove(te);
            Add(te.name, te.triggerTime + te.deltaTime, te.deltaTime, te.command);
        }
        curr = curr.pDNext;
    }
}
```

In order to get the grid of aliens to animate and march the grid move method call is made with the TimerManager's Update method on the Grid object that is passed in from the Game.cs Update method.

Proxy

We are populating a grid of 55 aliens, 1 row of 11 Squids, 2 rows of 22 Crabs and then 2 rows of 22 Octopi, that's a lot of images and textures we're using that we can simplify and make more efficient. However seeing that each Squid, each Crab and each Octopi all share the same image and texture, we decided to use the Proxy pattern to solve this problem. The Proxy pattern breaks an object into 2 parts, a commonly used portion and a unique portion and copies the commonly used portion to a real class where it is used. Data is copied, in our case, in the form of the Sprite class which is then used in a real class, the ProxySprite class. The data that is unique to each Proxy are x, y, sx, and sy values for the different locations on the screen that they will be placed along with scaling factors. The ProxySprite class holds the unique small data and holds a reference to a larger Sprite object where it delegates rendering and updating of the actual game sprite to the Sprite class. In Space Invaders, when we want to animate each alien we don't have to modify 55 game sprites, we simply modify 3 unique sprites which are reference by 55 different Proxy sprites which in turn animates all 55 sprites on the screen.

Although 55 alien sprites may not seem like a lot, but when creating, managing and animating them constantly frame by frame throughout the duration of this game, it can all add up. To alleviate this the proxy design pattern has come in handy. The major difference that distinguishes between each of the NPC sprites are their position on the screen which is determined by their x and y position. So a ProxySprite class has been created with these position fields as properties:



Proxy

So instead of creating actual sprite objects, we create ProxySprite objects that each hold a reference to an actual Sprite object. In designing it this way, for instance for a row of Squid sprites, there is one Squid sprite object that is created:

Code Snippet 8

```
SpriteManager.Add(SpriteBaseName.Squid, ImageName.Squid1A, 100.0f, 800.0f, 50.0f, 50.0f);
SpriteManager.Add(SpriteBaseName.Crab, ImageName.Alien1A, 100.0f, 750.0f, 50.0f, 50.0f);
SpriteManager.Add(SpriteBaseName.Octopus, ImageName.Squid3A, 100.0f, 700.0f, 50.0f, 50.0f);
```

and within the GameObject class instead of holding a reference to a Sprite object we instead hold a reference to a ProxySprite object. Within the GameObject Set() method a new ProxySprite object is created.

Code Snippet 9

```
public void Set(GameObjectName goName, GameObjectType goType, SpriteBaseName spriteName, float x, float y)
{
    this.gameObjectName = goName;
    this.gameObjectType = goType;
    this.pSprite = new ProxySprite(spriteName);
    this.pSprite.x = x;
    this.pSprite.y = y;
    this.x = x;
    this.y = y;
}
```

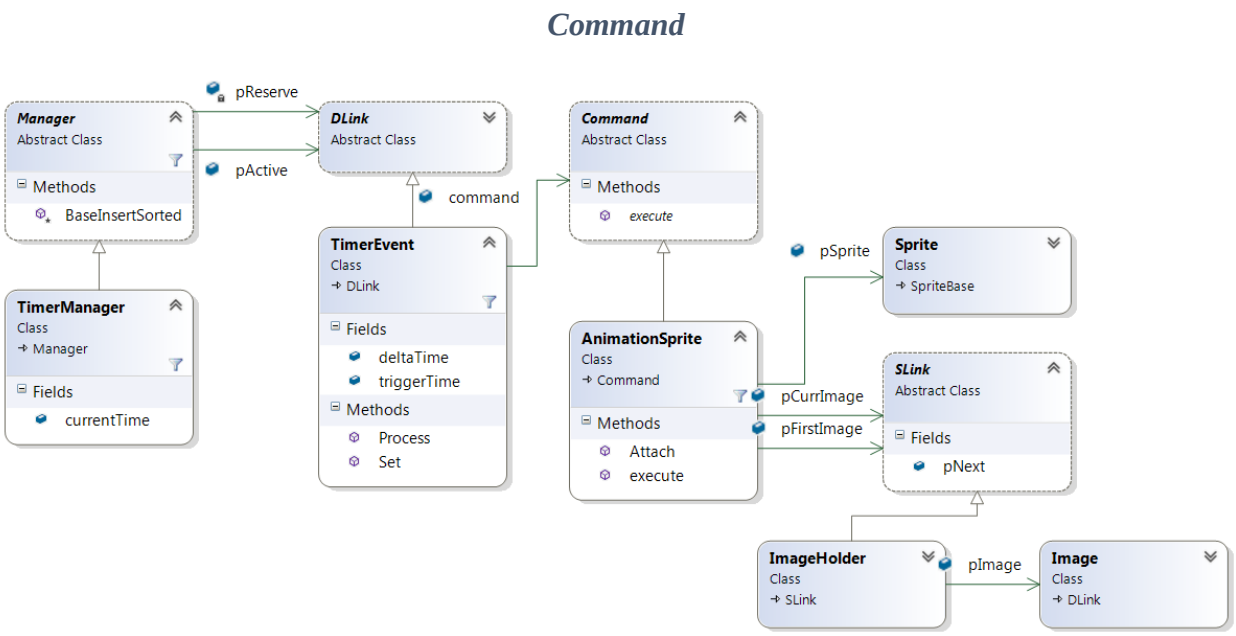
So finally, when all of these GameObject derived classes, the Crab, Octopus and Squid, as pictured above, call their Update method, it will update the ProxySprite's x and y fields accordingly. This will allow the game engine to create 3 unique Sprite objects that contain an Image, an Azul rectangle and an Azul Sprite, and then 55 unique ProxySprite object that only contain a reference to a Sprite object, and an x and y value specifying that ProxySprites unique position. This allows the Sprites to be efficient and lightweight when used within the game.

Command

We have a problem of animating the game sprites to switch between 2 images in a timely fashion, to simulate the aliens marching. A sequence of events need to take place in a specific order based on time,

so in order to solve this problem I decided to use the Command pattern. The Command pattern makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters. This is accomplished by creating TimerEvents which hold a deltaTime and a triggerTime that signify the time between each TimerEvent and the time that TimerEvent is triggered respectively. In addition each TimerEvent holds a reference to a Command object and delegates the execution of that TimerEvent to any derived class of the Command object. That way a number of TimerEvents can occur based on the timekeeping of the TimerManager and fire off a number of implemented Command objects to do a range of things from animating game sprites to playing sounds. In Space Invaders, I am using the Command pattern for animation as well as for playing the marching sound based on a march speed that is changed with each removal of an alien from the alien grid.

The command pattern was used in conjunction with the AnimationSprite and TimerManager with its TimerEvents in order to create and manage the animation system:



The TimerManager uses the typical Manager design pattern described earlier, but with a slight variation, a BaseInsertedSort has been added in order to maintain a sorted list of TimerEvents. The TimerManager will also hold the current time within the game, which will serve very useful in creating and managing TimerEvents. These TimerEvents each have a reference to an abstract Command class to be implemented by the AnimationSprite class. They also have a trigger time and a delta time corresponding to the time the game engine will begin the animation and the time in between these animation events respectively. These AnimationSprites were designed to hold a singly-linked list of Images to animate or flip (by using the Sprite SwapImage method within the AnimationSprite's execute() method) between via an ImageHolder object. This ImageHolder has a single property, a reference to an Image object.

Code Snippet 10

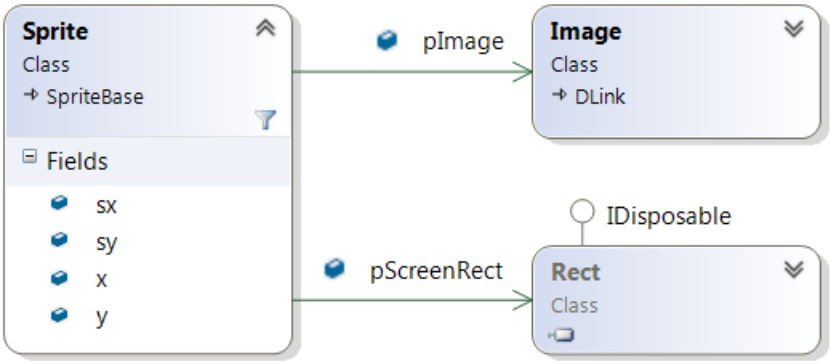
```
private void InitializeTimerManager()
{
    AnimationSprite animSpriteOctopi = new AnimationSprite(SpriteBaseName.Octopus);
    animSpriteOctopi.Attach(ImageName.Squid3A);
    animSpriteOctopi.Attach(ImageName.Squid4A);
    AnimationSprite animSpriteSquids = new AnimationSprite(SpriteBaseName.Squid);
    animSpriteSquids.Attach(ImageName.Squid1A);
    animSpriteSquids.Attach(ImageName.Squid2A);
    AnimationSprite animSpriteCrabs = new AnimationSprite(SpriteBaseName.Crab);
    animSpriteCrabs.Attach(ImageName.Alien1A);
    animSpriteCrabs.Attach(ImageName.Alien2A);
    TimerEvent timerEventSquids = TimerManager.Add(TimerEventName.AnimateSquids, 0.75f, 0.75f, animSpriteSquids);
    TimerEvent timerEventCrabs = TimerManager.Add(TimerEventName.AnimateCrabs, 0.75f, 0.75f, animSpriteCrabs);
    TimerEvent timerEventOctopi = TimerManager.Add(TimerEventName.AnimateOcotpi, 0.75f, 0.75f, animSpriteOctopi);
}
```

Within Game.cs in LoadContent() the game engine initializes the TimerManager, by creating three AnimationSprites, one for each unique Sprite object that we want to animate, and then creating three TimerEvents associated with these AnimationSprites. By tying this all together with the Manager InsertedSort method, allows the game engine to continuously add another TimerEvent via the TimerManager to be processed frame by frame based on the given TriggerEvent's trigger time relative to the game's current time (held in the TimerManager). This is all seen in the TimerManager's Update() method (*Code Snippet 7)*.

Flyweight

Seeing as performance and memory management is a common theme with this game, and since we are primarily working with game sprites throughout the entire duration of this game, we need to make them as lightweight as possible. In order to solve this problem, I have each game sprite hold a reference to an image which is shared across all game sprites and I decided to use the Flyweight pattern to solve this problem. A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects. It is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. This pattern helps to reduce the number of object instances at runtime, saving memory. It also centralizes state for many "virtual" objects into a single location. By holding a reference to a shared data object, allows all of our Octopi game sprites to share the same image while also having their own unique position, scale and a reference to an Azul.Rect representing their screen rectangle position. In the diagram below, the Image is the Flyweight, the shared data across all sprite objects. In Space Invaders, this pattern is used in the Sprite system as described above when we want to introduce a new sprite to our game and say for instance we want to create an army of 100 of these new sprites. We simply create 1 new Image and 100 sprites that all refer to the same image. So when we spawn this army, they will all have their own unique sprite objects associated with them, but they will all hold a reference to the same Image.

Flyweight

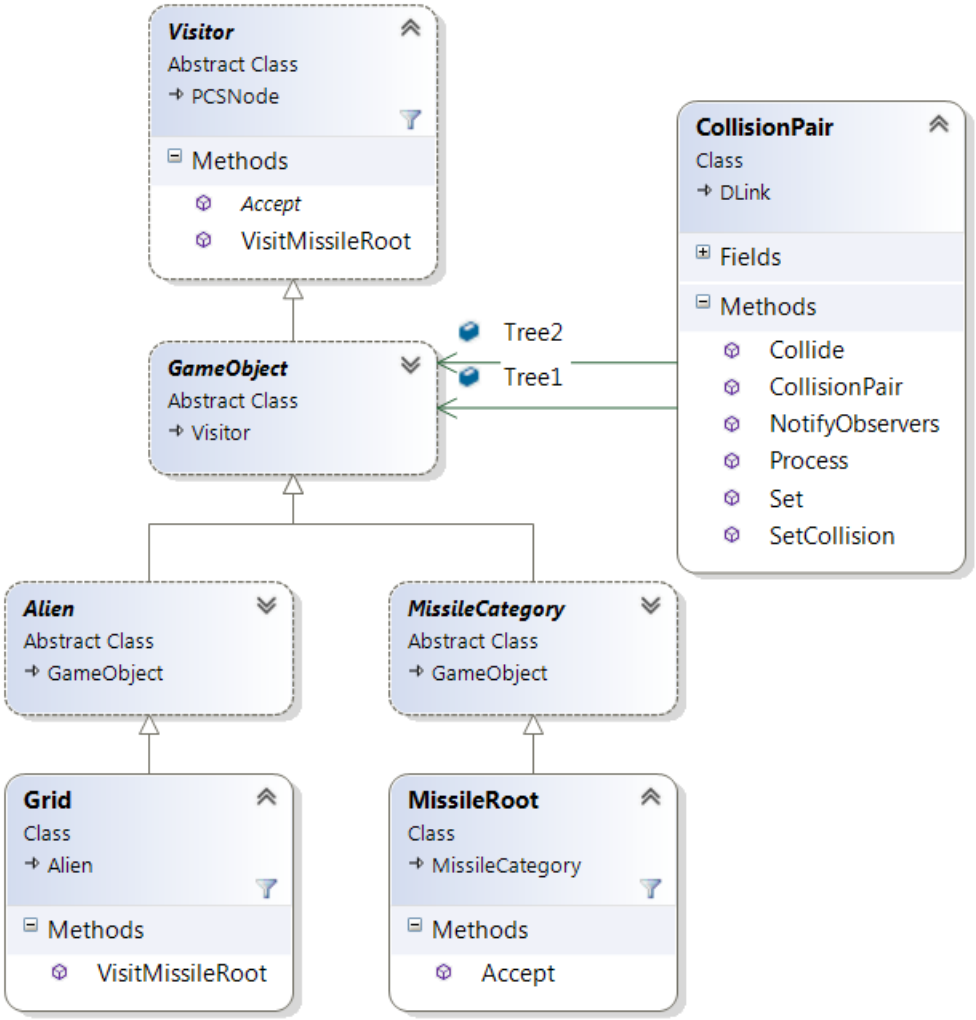


Visitor

For the large-scale problem of the collision system in creating a succinct way of discriminating between objects that take part in a collision I decided to use the Visitor pattern. The Visitor Pattern is a way of separating an algorithm from an object structure on which it operates. This allows me to add new virtual functions to a family of classes without modifying the classes themselves. I just created several visitor classes that implement all of the appropriate specializations of the virtual function. The Visitor takes the instance reference as input and implements the goal through double dispatch.

In Space Invaders, I created the abstract Visitor class as a base class of the GameObject class

Visitor



The example above just shows a small portion of the vast Visitor pattern in play with the entire collision system. Based on the game rules that I have established, I allow for a collision pair to exist between the Grid GameObject and a MissileRoot object. So essentially when a Grid and a MissileRoot collide as per the Collide method in the created CollisionPair, we have a 2 GameObject trees that collide. Based on how the CollisionPair is created:

```
CollisionPair cpMissilevGrid = CollisionPairManager.Add(CollisionPairName.MissilevGrid, pMissileRoot, pGridRoot);
```

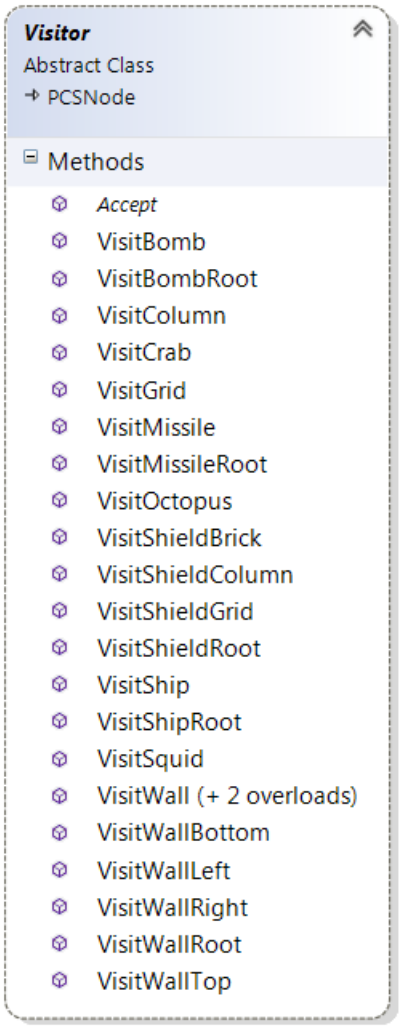
we establish our resulting Visitor pattern A.Accept(B) format with MissileRoot being A and Grid being B. So when we have a collision between the two GameObjects, MissileRoot will call its Accept function passing in the Grid GameObject:

```
public override void Accept(Visitor other)
{
    other.VisitMissileRoot(this);
}
```

From there, since GameObject is derived from Visitor, we can call its VisitMissileRoot function via double dispatch:

```
public override void VisitMissileRoot(MissileRoot pMissileRoot)
{
    CollisionPair.Collide(pMissileRoot, (GameObject)this.pChild);
}
```

This VisitMissileRoot function in my Grid object will them call collide from CollisionPair again, but with the Grid's child GameObject which in this case is the first Column of Aliens. This process continues until either there is no more collision between CollisionObjects or we end up with a Visit function that handles the collision accordingly using the Observer pattern, which I'll cover in the next section.



As mentioned before, this was only a small taste of the Visitor pattern shown in order to focus in on the basics of the Visitor pattern. The way it is used in Space Invaders is for the entire Collision System, as seen to the left. When we established our game rules in class we determined what collisions were acceptable and created our abstract Visitor class to accommodate all of these collision rules as virtual functions that are implemented in the appropriate concrete GameObject classes. If a specific function is not implemented it will be called in the abstract Visitor class and an assert will be thrown stating that the specific function was not implemented.

Observer

Now that we have a collision between two GameObjects and want to take action on a specific collision, what do we do next? More importantly, how do we go about accomplishing this variable laundry list of tasks upon each specific collision? I decided to use the Observer pattern to help with this situation.

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. These dependents, or observers are all observing a specific Subject type, this Subject represents the one object that's state change the observers care about. This pattern delegates the actions to be taken upon state change to a list of observers. In turn, these observers all inherit from an abstract Observer class and implement the Update() function to take any number of actions or customized tasks, taking care of the delegated task. Each custom Observer class helps to accomplish the tasks of the game behind the scenes.

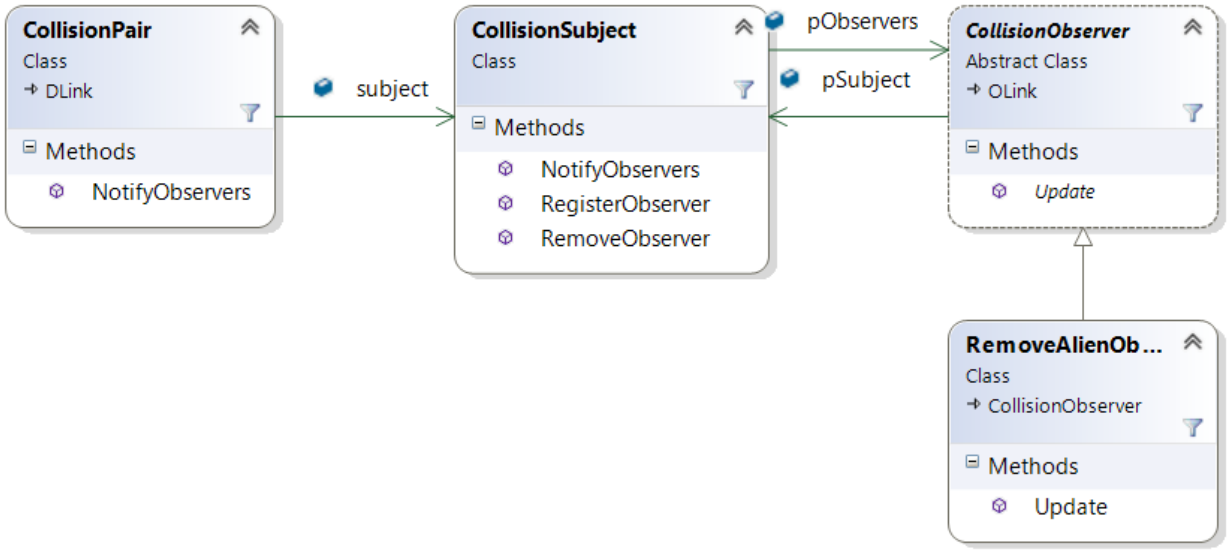
In Space Invaders, continuing the discussion of the collision system with the CollisionPairs, in the code snippet of Crab.cs below, Crab calls the VisitMissile() function which signifies a successful collision as per our game rules:

Code Snippet 11

```
public override void VisitMissile(Missile pMissile)
{   // we have a hit
    Debug.WriteLine("Hit crab!");
    CollisionPair collisionPair = CollisionPairManager.GetActiveCollisionPair();
    collisionPair.SetCollision(pMissile, this);
    collisionPair.NotifyObservers();
}
```

As can be seen above once we have our success use case, we call NotifyObservers() from our collision pair object:

Observer



This will notify all observers that are registered with this CollisionPair's Subject. Each CollisionSubject holds a reference to a linked list of Observers, and these Observers have a reference to the CollisionSubject. This design decision was made in order to have a quick reference to the two GameObjects involved in the collision without having to pass them as parameters to the Observers. These Observers that we create are registered to the CollisionPair when the CollisionPairs are created as seen below:

Code Snippet 12

```
CollisionPair cpMissilevGrid = CollisionPairManager.Add(CollisionPairName.MissilevGrid, pMissileRoot, pGridRoot);
Debug.Assert(cpMissilevGrid != null);
cpMissilevGrid.subject.RegisterObserver(new RemoveMissileObserver());
cpMissilevGrid.subject.RegisterObserver(new RemoveAlienObserver());
cpMissilevGrid.subject.RegisterObserver(new ShipReadyObserver());
cpMissilevGrid.subject.RegisterObserver(new AlienDeathSoundObserver());
cpMissilevGrid.subject.RegisterObserver(new SplatObserver());
cpMissilevGrid.subject.RegisterObserver(new UpdateScoreObserver());
```

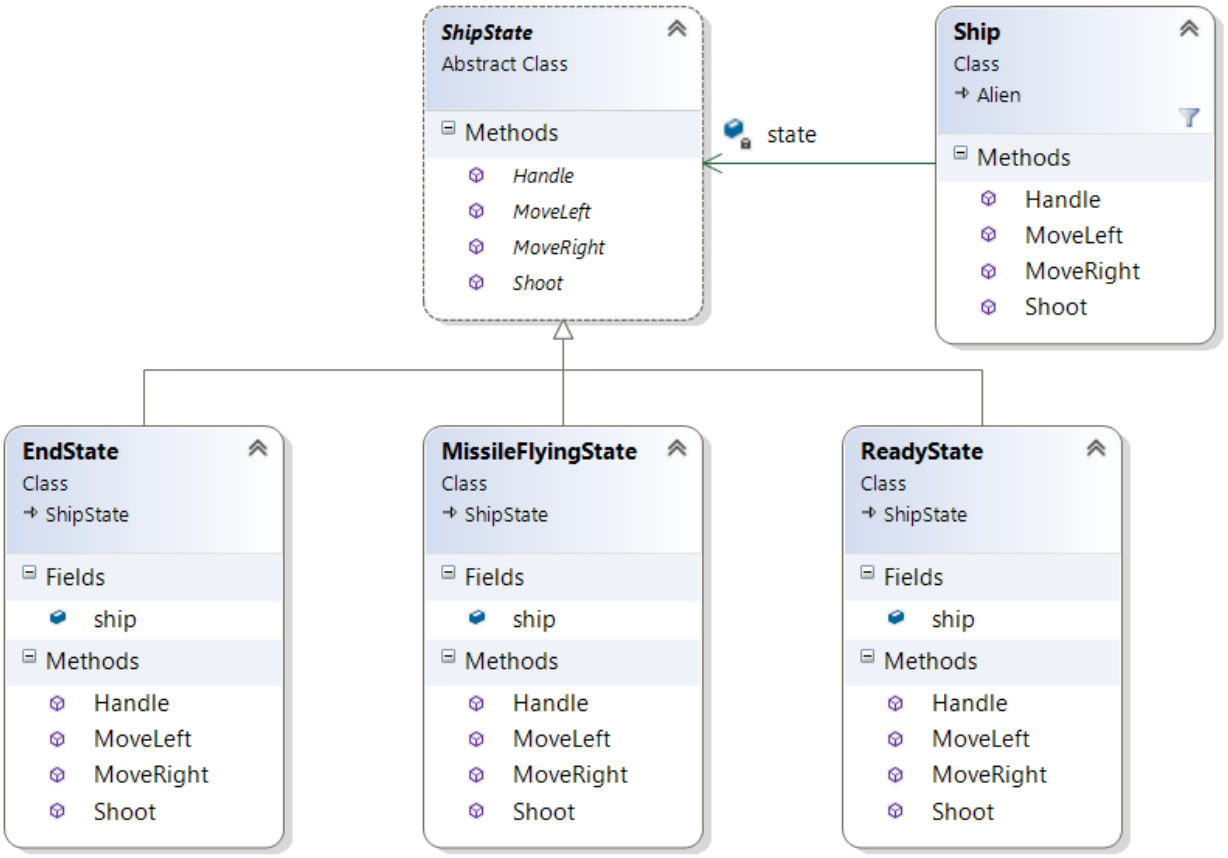
So this MissilevGrid collision pair has the above list of Observers registered and they will all run their Update() functions when the CollisionPair's NotifyObservers() function is called (in Code Snippet 11).

State

One of the major problems we are faced with in this game is handling input from the user in order to control the user ship. How do we manage the user ship's state at any given time? I decided to use the State pattern in order to change and manage the state of the user ship.

The State pattern allows an object to alter its own behavior when its internal state changes. Unlike a procedural state machine, the State Pattern represents each state as its own class. This is accomplished by creating with inheritance and aggregation. Given a class that we want to track the state of, we create a State class property. This abstract State class contains the types of actions or transitions that the object can take to transition between states. Each state of that object then derives from the state class and implements these abstract functions:

State



So in order for the Ship to transition between each state we simply refer to the State property in the Ship class and call each method accordingly:

Code Snippet 13

```
public void Handle()
{
    this.state.Handle(this);
}
public void MoveLeft()
{
    this.state.MoveLeft(this);
}
public void MoveRight()
{
    this.state.MoveRight(this);
}
public void Shoot()
{
    this.state.Shoot(this);
}
```

And depending on what the current state is of the ship, it will take the appropriate action if possible:

Code Snippet 14

```
public class ReadyState : ShipState
{
    public Ship ship;
    public override void Handle(Ship pShip)
    {
        pShip.SetState(ShipStateEnum.MissileFlying);
    }
    public override void MoveRight(Ship pShip)
    {
        pShip.x += pShip.speed;
    }

    public override void MoveLeft(Ship pShip)
    {
        pShip.x -= pShip.speed;
    }

    public override void Shoot(Ship pShip)
    {
        Missile pMissile = ShipManager.ActivateMissile();
        pMissile.SetPosition(pShip.x, pShip.y + 20);
        pMissile.SetActive(true);
        SoundManager.PlaySound(SoundManager.Find(SoundName.invaderKilled));
        this.Handle(pShip);
    }
```

As can be seen above in the ReadyState, the ship can transition into the MissileFlying state after executing the Shoot() function, which in turn calls the Handle() function which set's the ship's state to the corresponding MissileFlying state. So within each specific state, I can handle the ship's actions and states accordingly.

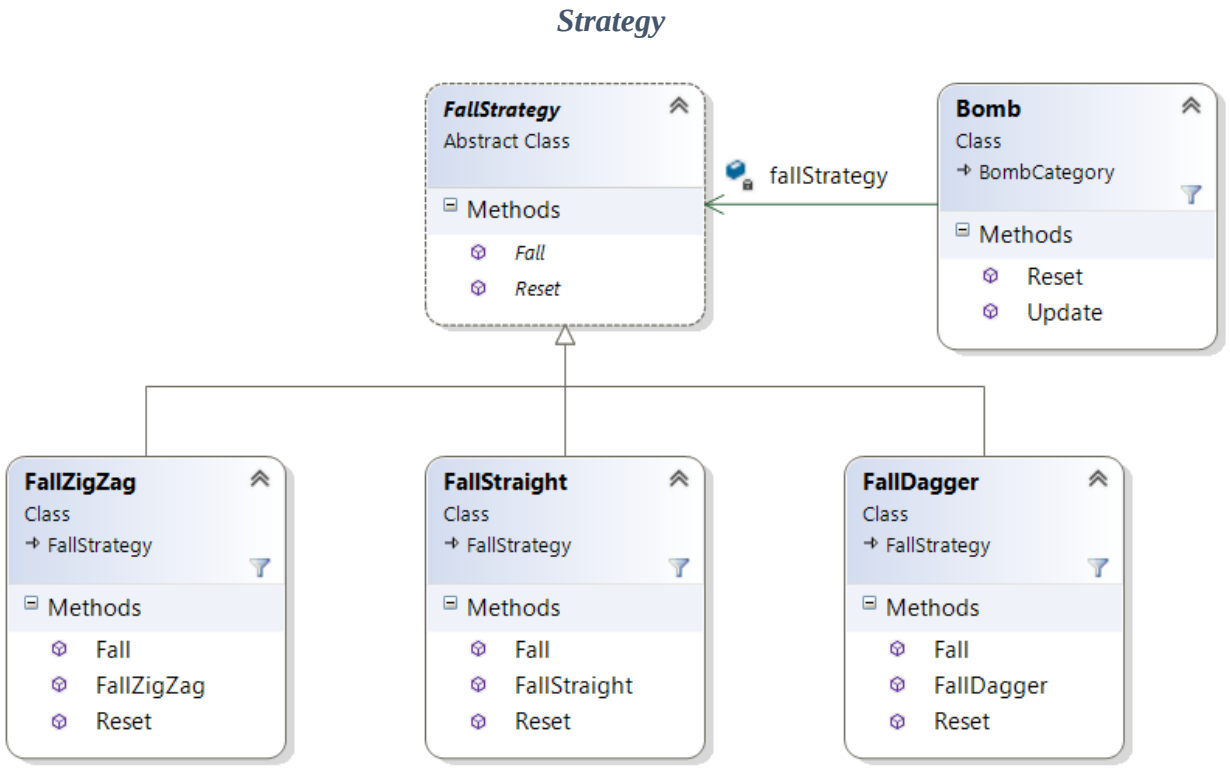
In Space Invaders, along with the ship, the State pattern is also used similarly in the managing the game's state. The game starts off on the Instructions screen and waits from input from the player, then transitions into the Active state and while there if the player runs out of lives then the game transitions into the Game Over state.

Strategy

As can be seen in the original Space Invaders game, we can see the aliens drop several different types of bombs at any given time, a straight dropping bomb, a zig zag bomb and a cross bomb. All of these are alien type bombs however the way each of them behaves is slightly different. I decided to use the Strategy pattern in order to implement the different behaviors of each of these alien bombs.

The Strategy pattern defines a family of algorithms, encapsulates each one and makes them interchangeable. It also lets the algorithm vary independently from clients that use it. The way this pattern accomplishes this is by means of delegation and inheritance. The Strategy class is abstract and contains an abstract method(s) that define a specific behavior(s) that will vary amongst this type. Each behavior then is derived from this abstract Strategy class and they will each have to implement that specific behavior accordingly.

In Space Invaders, I created an abstract FallStrategy class with a Fall behavior to be implemented by the three different FallStrategy derived classes: FallStraight, FallZigZag and FallDagger:



My Bomb class has a FallStrategy property that is set via its constructor.

Code Snippet 15

```
public Bomb(GameObjectName goName, SpriteBaseName sName, FallStrategy strat, float x, float y, int idx)
    : base(goName, sName, BombType.Bomb, idx)
{
    this.x = x;
    this.y = y;
    this.speed = 5.0f;
    Debug.Assert(strat != null);
    this.fallStrategy = strat;
    this.fallStrategy.Reset(this.y);
    this.pCollisionObject.pCollisionSpriteBox.pLineColor = ColorFactory.Create(ColorName.Orange).pAzulColor;
}
```

And in the Bomb's Update() function it calls its corresponding FallStrategy's Fall() method.

Code Snippet 16

```
public override void Update()
{
    base.Update();
    this.y -= this.speed;
    this.fallStrategy.Fall(this);
}
```

So whenever I want to create a Bomb I simply designate the type of strategy I want to use and pass it into the Bomb's constructor (as seen below).

Code Snippet 17

```
switch (pGrid.pRandom.Next(0, 3))
{
    case 0 :
        sName = SpriteBaseName.BombStraight;
        pStrat = new FallStraight();
        break;
    case 1 :
        sName = SpriteBaseName.BombDagger;
        pStrat = new FallDagger();
        break;
    case 2 :
    default :
        sName = SpriteBaseName.BombZigZag;
        pStrat = new FallZigZag();
        break;
}
Bomb pBomb = new Bomb(GameObjectName.Bomb, sName, pStrat, pColumn.x, pColumn.pCollisionObject.pCollisionRect.minY, 0);
```

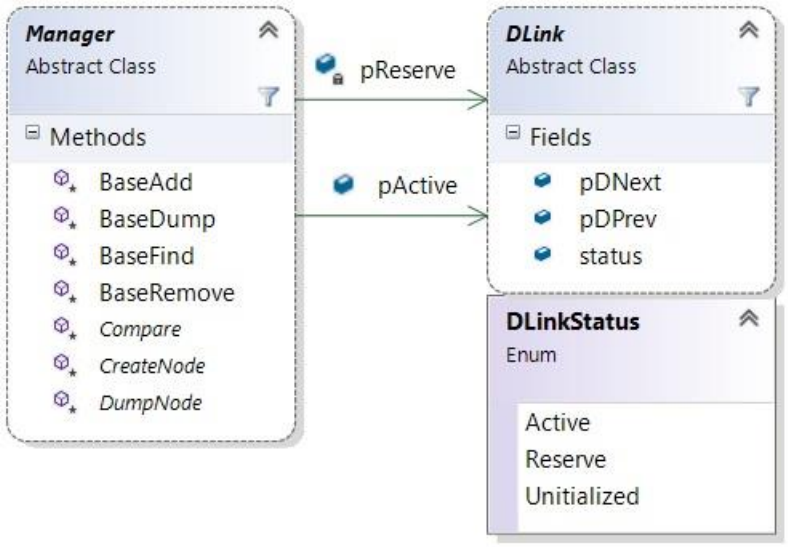
Implementation

In order to recreate this classic arcade game it is necessary to break down this game into several different areas.

Manager

But before we delve into each aspect we have created a way to manage assets as well as manage the amount of memory used in the game. The abstract Manager class has been created along with the abstract DLink class along with the Singleton design pattern to implement this Manager pattern:

Manager



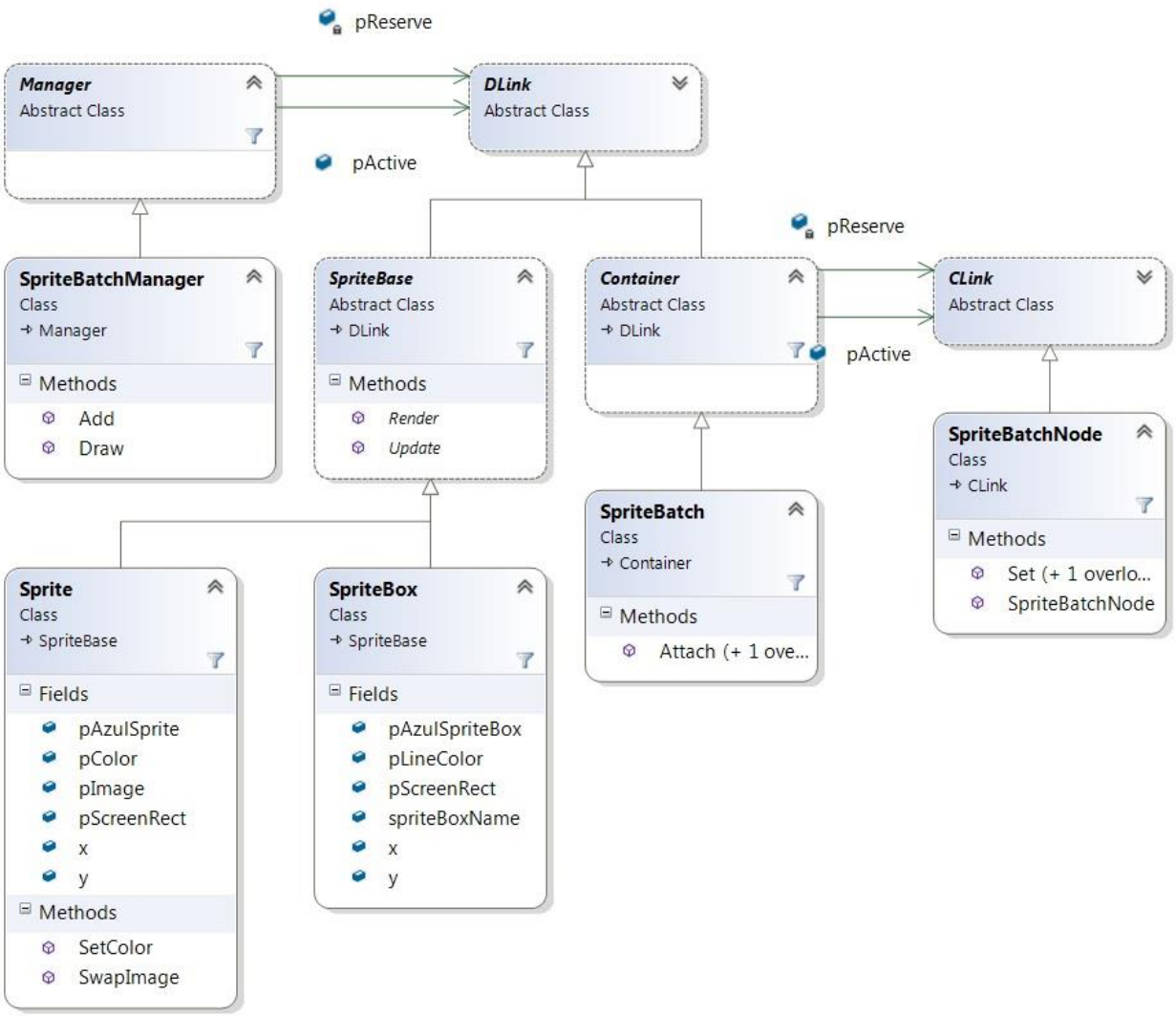
Each Manager holds two references to a DLink class that represents a doubly linked list data structure which is evident in the pDNext and pDPrev fields in the DLink abstract class. When the game first loads (in the LoadContent() function in Game.cs), we allocate all of the memory we plan to use upfront to minimize the amount of expensive calls to "new" throughout the duration of the game. The doubly-linked lists represent the reserve and active pool that each manager will utilize to manage all active and

reserve images, textures, sprites, etc that are going to be used in the game. First, we allocate our desired number of assets that we intend to use in the reserve pool, then when the asset is being used in the game we move a DLink object from reserve to active. If ever we no longer want to use a specific asset at a given moment in the game we can simply move the asset from the active to the reserve pool.

Sprite Batch

One of the main components of this game is populating it with Non-Player Character (NPC) sprites (the aliens and UFO ship you see and shoot in the Space Invaders game). Upon starting the game you see a 5x11 grid of aliens moving together across the screen. Each of these sprites has a specific texture and image associated with it. In our case we are using the same texture file for every sprite that will be used in the game to save on memory. In order to efficiently create and manage our sprites on the screen we designed the sprite batch and its corresponding manager:

Sprite Batch



The Sprite Batch Manager follows the similar Manager design described earlier with a new method Draw() and another Container class involved. The Draw() method will simply loop through our manager's active pool and call the corresponding sprite base object's render method which will in essence

draw that particular sprite base to the screen. A sprite base currently consists of either an Azul Sprite or an Azul SpriteBox, which will later be used for collision detection. The Container class holds a specific grouping (in this case we call it a Sprite Batch) of sprites that we want to draw onto the screen. Unlike the doubly-linked list data structure we use with the manager design, we use a singly-linked list (in this case we call it a Sprite Batch Node) with this Container class since all we need to do is loop through each container and call each sprite base object's Render() method and move on to the next sprite base object in the list. The Sprite Batch Manager holds a doubly-linked list of Sprite Batches which consists of a singly-linked list of Sprite Batch Nodes to specify the sprite base objects that we want to render to the screen. The reason it is designed this way is to allow for quicker and easier drawing of multiple sprites at a time. For instance if we want to create a Sprite Batch for all squid type aliens and another for all octopi type aliens, we can add two sprite batches, but if we only want to draw just the squids to the screen, we can just add the squid sprite batch. As seen above this adding is done with the Sprite Batch Manager Add() method. Within each sprite batch is a sprite batch node that holds each sprite that belongs to a given sprite batch. This can be done with the Sprite Batch Attach() method (as it is currently designed this is done in the Alien Factory class which will be described later).

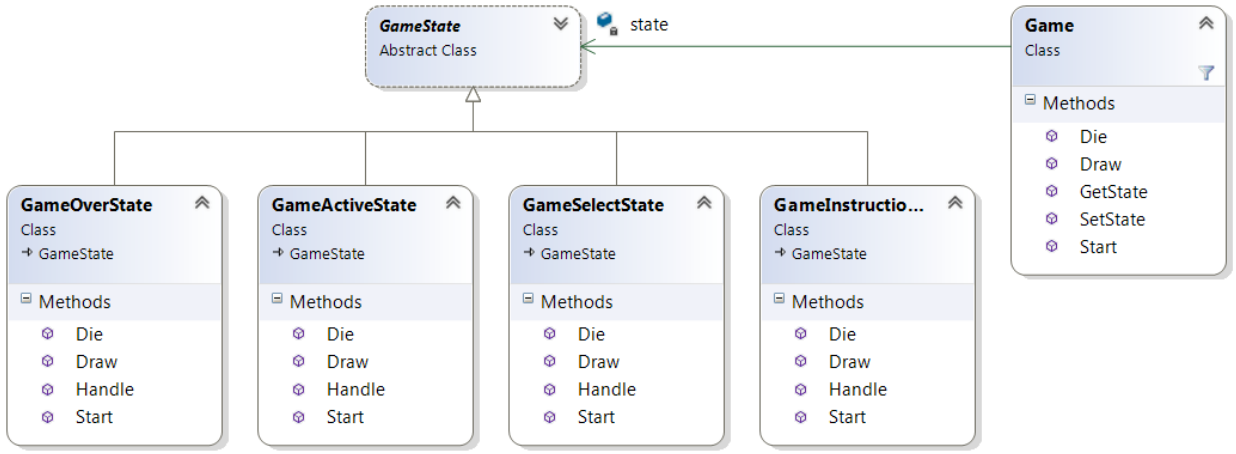
Collision System

As mentioned previously in Visitor this is the meat and potatoes of the Collision System that I have designed. The reason I've used the Visitor pattern is to allow me to discriminate between the two game objects that are taking part in the collision. However when we determine what collision we have, we next need to decide what plan of action we want to take. And this is done using the Observer pattern mixed in with the Command pattern. As described above with the Observer, we simply notify all interested parties that we a specific collision has occurred. The CollisionSubject is object that all interested Observers will listen to or "observe" to await for further instructions. For these Observers to become "interested" I created the RegisterObserver function on the CollisionSubject to hold a linked list of Observers to be notified. This CollisionPair that is involved in the collision holds a reference to a CollisionSubject and calls its NotifyObservers function to notify all observers. The key concept that I learned through trial and error was when to use the Observer pattern over the Command pattern. Unless sorted, Observers are not aware of each other and by default are not guaranteed to run their update function in a specific order. If you are concerned with the sequence of events that occur then it's best to use the Command pattern. In this way I designed the TimerManager with TimerEvents using the command pattern to create generic events that will execute based on a specified trigger time and repeat again on a delta time relative to the game's current time.

Game State

I decided to use the State pattern to design the Game states and managing and directing between different game screens.

Game State

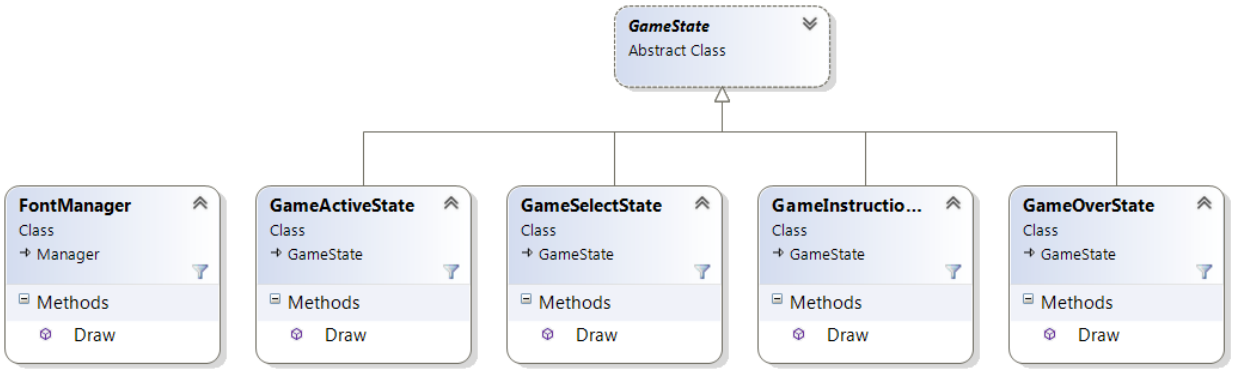


As shown above, I created an abstract GameState class that each of the individual states of the game derive from. This inheritance allows for me to be able to transition between states by executing the derived methods per state. I created a Game class (seen on the right, above) that holds a reference to an abstract state, which must be implemented in one of the derived concrete classes. So at any one point in time the Game class, which I created to remove all of the LoadContent logic from the SpaceInvaders Azul.Game Game.cs class, will point to a particular state. And in order to transition between states, you need to call the Game class's SetState() function and then call that particular state's derived method. These specific states were designed to deal specifically with that given state. For instance in the Game OverState, the Die method is empty since you cannot die in the GameOverState, but in its Start method there's functionality for restarting the game and then a call to the Handle function which changes its state to be implemented by that corresponding next state. This was an easy choice, as seen with Ship controller using the State pattern this was a no-brainer.

Font System

Similarly to the GameState, I created a Draw method that uses the FontManager to display the appropriate HUD (Heads Up Display) elements per Game screen. For instance, the GameInstructionsState's Draw() function draws the alien score key that is displayed on the first screen when the game is first launched. I decided to add another derived method that would be used by another system in order to connect the Game's states with the FontManager in order to determine when to display the appropriate text to each screen.

Font System



So when I hit my Draw() loop, I can just call FontManager.Draw() and the appropriate Game state's Draw() method is called to display the appropriate text to the screen.

Code Snippet 18

```
//--------------------------------------------------
// Game::Draw()
//      This function is called once per frame
//      Use this for draw graphics to the screen.
//      Only do rendering here
//--------------------------------------------------
public override void Draw()
{
    SpriteBatchManager.Draw();
    FontManager.Draw();
}
```

YouTube

https://www.youtube.com/watch?v=Vs5Q42XaqZU

Post-Mortem

Improvements

There are so very many improvements that I can make in this game, from performance, to memory management, code cleanup, adding 2 players, adding dynamic collision boxes and only allowing 1 alien bomb per column at a time just like the 1 active missile for the user ship, implement the ship left and right observers using the collision system, just to name a few. First and foremost, I would like to tackle and redesign the TimerManager and the way TimerEvents are created, since I created a hairy mess and I can easily speed up performance by tweaking the TimerEvents instead of having that if statement in my TimerManager Update. I only used Iterators in a few places as well, when I could have used them all throughout my game. Another improvement that I stubbed out and attempted to do was the FontManager typing, I was trying to use TimerEvents to spell out an entire word and draw 1 character a second, but the TimerEvents can occur so fast that it's difficult to debug and make really smooth. There are so many areas of improvement that I can foresee. There were some things that I was confused on early on and developed iteratively until I got it to work for my purposes then moved on. Granted this created problems

and issues for me down the road, I learned the hard way and in the process of learning, I started to think of systems I built beforehand in different ways too. And now, in hindsight there are a couple of things I would try again.

Comments

Simply put, I created one heck of a nightmare for myself. I made mention in one of my YouTube recordings, not the final one posted (sadly, however probably the best of the ones I recorded), that the largest obstacle for me was the TimerManager fiasco. I was working on creating a State pattern to control the Game's state from Instruction screen to Select screen to active Game Screen to Game Over screen and back to Instruction screen again when I ran into this issue. I learned the hard way that I need to include the Current game time when setting the TimerEvent trigger times. Along with implementing the UFO into a manager as well as getting the timing of timer events was also very difficult. I learned through trial by fire when use an Observer over a Command when handling the "what's next?" after collisions.

As I had mentioned on the video, I really enjoyed this class, granted it was very difficult and very time consuming, it was very rewarding and I feel like I learned a lot. Over the past 3 and half years that I've been working as a Software Developer, I've formed some poor habits and haven't until recently started actually diving into design patterns. I really think this has helped me to think and tackle my design problems in a different way. In hindsight though, and as mentioned on my Final exam, I don't think I'm built for the whole work and school full time thing, it's either one or the other. Early on in graduate degree track the intro courses weren't as time consuming and were manageable while working, but the more I progress, naturally the more difficult it gets. This course has challenged me immensely and has led me to reevaluate the next course of action for my life.