Corcoran, James

Space Invaders Design Project

Video Link: [Space Invaders Demo]

The design project of a real-time systems is quite an eye opener. Prior to this project I had no knowledge of these design patterns I will be discussing. But first, we need to elaborate on some concepts that help keep these designs in focus. Some information is needed, for my own well-being primarily, when I walk through the various tools and techniques we used on the Space Invaders project. These concepts are inheritance, association, and encapsulation.

Object-Oriented Principles

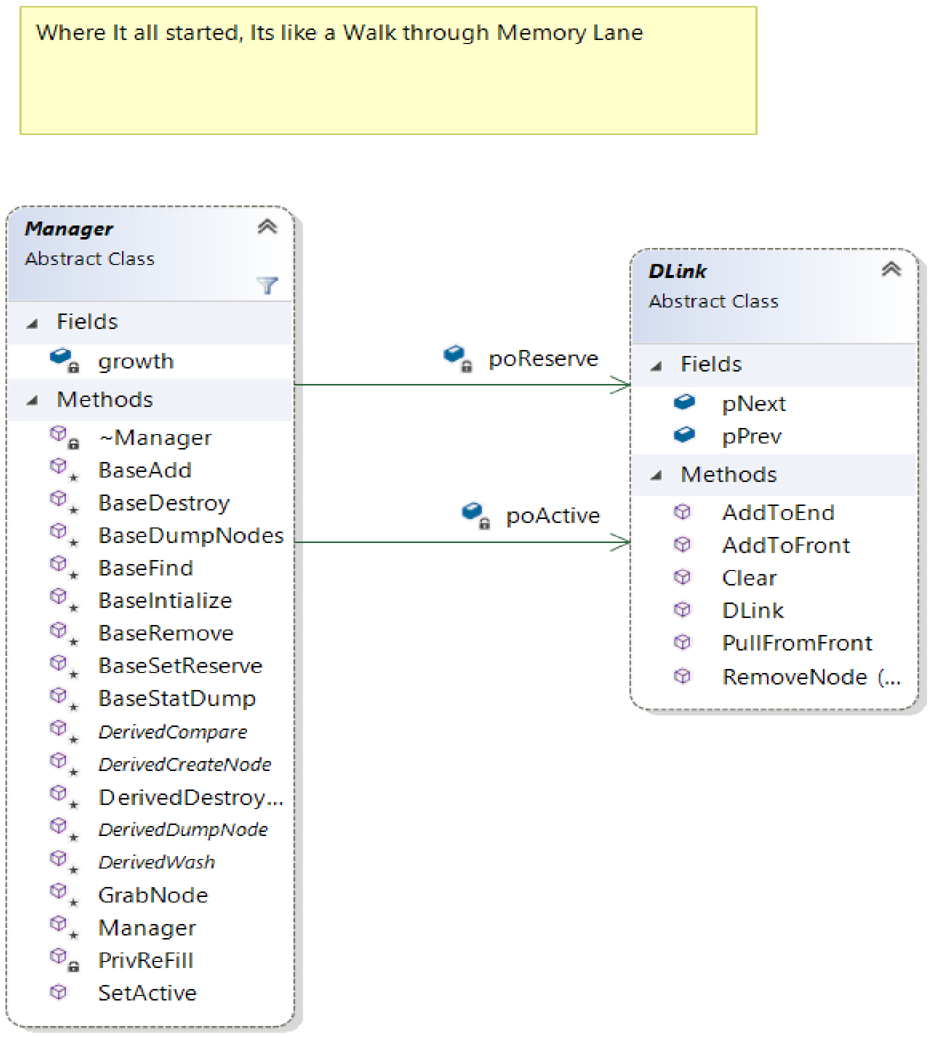
Encapsulation is one of those Object-Oriented principles you know about, but do you utilize it as much as you should? Primarily this is solely speaking from my own personal experiences. When I'm writing code, I make everything public and the scope is so broad I can barely even tell who should and shouldn't use certain objects and data. Almost everything is public, but as I progressed through this design project I could just look at some classes, data and say, "hey that should be private", or "that data should be protected because only the derived classes should know about it". I think it's important to remember encapsulation because in the past I would make something a global variable just so I could use it all over the place. But, this can cause quite a lot of work when your debugging how that variable was messed with.

Inheritance plays an entirely huge role in this design project, and it's an important part of delegating information and work load of classes. Inheritance defines an "is a" relationship between classes. Say you have a dog class that eats, plays, and sleeps. Then you have a cat class that eats, plays, and sleeps. Well already we can see the same methods and code in two different objects, what if we make another class called a Pet class that eats, plays and sleeps. Now we can derive from the pet class and even override/ implement some variety to the dog and cat classes, if we need to. A cat sleeps much longer than a dog so maybe it requires a slightly different implementation. We could give a keyword "virtual" in the base class method Sleep so that its derived classes can overwrite the inherited method if they need to implement it differently. From here we can run into more opportunities like, maybe the Pet class can serve as an abstract base class that tells its derived classes you need to implement the eat, play and sleep methods. What if we add a Spider as a pet, now we see some importance of an abstract class because now we have multiple pets, but do they all share the same properties? A spider has eight legs, dogs and cats only have four. They all need to implement the three previous methods, but now their specific properties are different. I like to think of inheritance as a management tool; which helps us from writing redundant code and gives our system more structure. The use of the cat, dog, and spider inheriting from the pet class example helps us think about a hierarchy between the objects as well as a little concept called D.R.Y (Don't Repeat Yourself). These ideas are important in Object Oriented design because now we have a managed system that has allot more organization to it.

Another concept that plays an important part in organizing a project is the associations between objects with a "has a" relationship. We will see continuously that these design patterns will have compositions and aggregations throughout. Association basically says that if one object has another object they both can exist independently without each other. A pet owner class can have a dog class, in this example let's say the pet owner can only have one pet object, and if the pet owner was to add another pet object then the dog would still exist even though it's not attached to the pet owner object. The pet owner object doesn't create the pet object it receives(attaches) the pet from, let's just say in this scenario, the pet store. Next, we will look at a stronger form of aggregation known as composition. Instead of the pet owner example let's move to another example called the house class. A house class can be comprised of many different objects like windows, doors, and people objects. Now the people objects are an aggregation they can move to another house and both, they and the house object, will still exist. But if we were to destroy the house object the door and the windows would be destroyed too. The windows and the door objects cannot exist without the house object. This is the essential difference between these associations; composition is a strongly tied association between objects and an aggregation is a looser tie between objects.

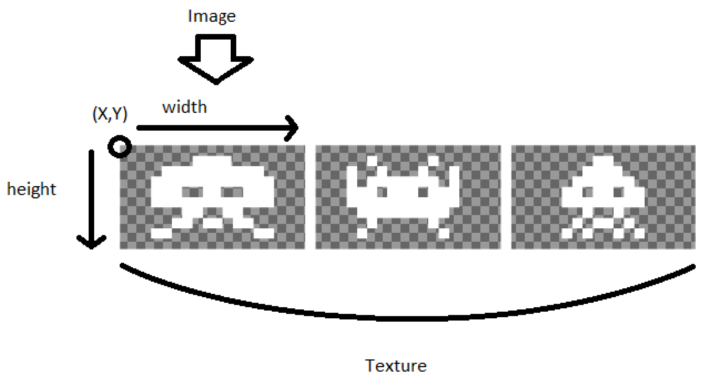
Object Pool Design Pattern

Now that we have discussed some core concepts that we will be referring to a lot in our design patterns we will start with our basic setup of managing all the data in our space invaders design project. As we start this project we see that we will need a basic structure of holding all of our data and some object to manage them, called an object pool. Enter now is the double linked list and the manager class. In a real-time systems architecture, efficiency and speed are to very sought-after characteristics. Our double linked list, comprised of DLinks, data structure, has a starting node that is connected to consecutive nodes by referencing them as previous and next, two links per node. When we add or pull a node from the list we always grab them from the front to make it as fast as possible. We want the worst-case scenario for time complexity, big O notation, to be constant time. For example, if we have an operation that needs to be performed we want it to be done in exactly one step, O(1). When we go back to our double linked list we don't care what order or how many are in the list. We simply want whatever is there, specifically at the front because that's the quickest point of access.



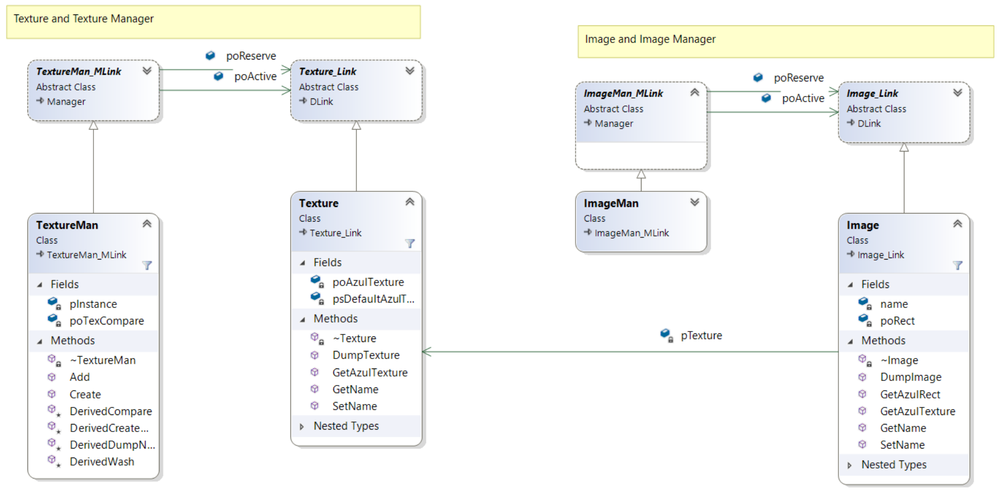
We also need to find an effective way to remove a node from any given position in the double linked list. This can be a little trickier to illustrate, but if we know that a DLink is in a given list we just need a couple conditional statements to connect its previous and next nodes, if it has them, so we don't lose the integrity of the list. One other important part of a list, that holds references to each other, is if we remove, pull or add from one list to another we need to make sure that the references to its next and previous are re-set. Otherwise you will learn very quickly how to debug, and what could happen if a DLink brings its connected references to another list. The DLinks can only know so much about themselves and how they function. We need another class to help organize and hold these double linked lists. Out of this requirement we get the manager class. The manager class creates and holds, composition, two double linked lists that store and recycle the DLinks. When we use the word recycle its becomes apparent that we don't want to keep allocating and de-allocating resources throughout our system. We shouldn't be calling for "new" objects then letting garbage collection get rid of them when we are done using them. Our object pool holds onto the allocated resources and doesn't let go until the termination of the manager or the project. Instead we use methods that switch our links from these active and reserve lists, and clears their references (and fields) so old data isn't re-used or copied over. As we progress and our project grows we will see needs that causes us to go back and refactor our old code to make it more useful for later use.

Our project is starting to manifest itself and now we can start managing objects that will be used in our Space Invaders project. We started to work with the actual Azul game and how it functions. The screen serves as an X and Y coordinate map that can project "sprites" onto the screen in different locations. It constantly goes through a loop of updating their X and Y coordinates and rendering(drawing) these "sprites" to give the appearance of movement. A question that might be raised is what is a sprite and what is it composed of? Our sprites serve as boxes on the screen that have an image drawn over them. This image comes from a texture file, that has various images on it. We must extract the images from these files via an (x,y) location on the texture and the width and height to associate to the image. Here is a simple picture to represent three alien images in the texture file, "aliens14x14.tga".



We use the top left corner of a certain image to determine the starting location. Let’s say the X,Y are at positions 0,0 respectively. We determine that the width is 125 and the height is 80. So now the Azul game system knows to look at point 0,0 and create a box with the parameters 125 and 80 and that will be the image we use.

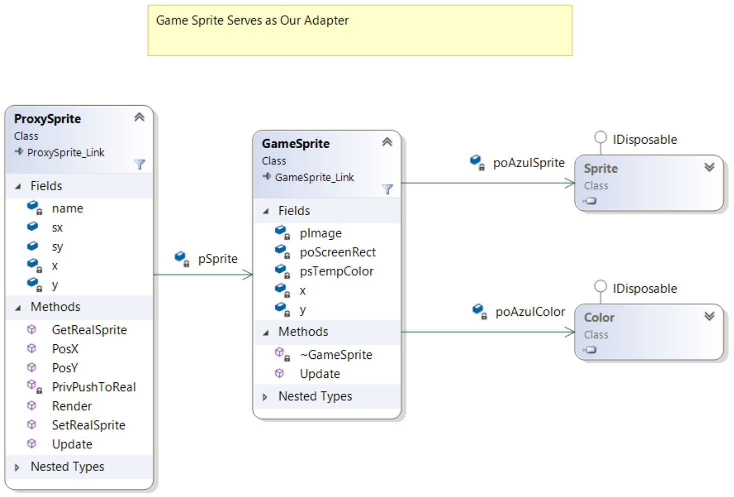
But, before we get ahead of ourselves we need to hold onto all this information. This is where our manager and double linked lists will come into play. Let’s say we need 3 different texture files for all of our images and we need several images from each texture file. We need to hold onto all this data by referencing these objects in our DLinks and managers. If we remember our DLinks are holding references to objects and not actually the objects themselves. This saves us a lot of memory storage in these lists. So now we can start to figure out we need a texture manager that holds the three different texture references and we need an image manager that holds the various image references. Our manager class is starting to get a lot bigger trying to hold all these different objects, right? Wrong!!! if we use inheritance and make the manager class abstract we can derive specific managers to work with these specific objects. Our base manager class only knows about DLinks. So, if we make textures and images derive from DLink we can re-use all of our older code and add a few different implementations to some of the classes if need be. We can also turn DLink into an abstract class too, since we are only really using it in a “How To” fashion. I call these abstract classes “How To” because we don’t necessarily construct them, but their parameters and methods are implemented in the concrete base classes.



The image manager will have a restriction and will always needed to be created after the texture manager because the images come from the textures. Since the Azul game system uses all these primitive data types like Azul Texture and an Azul Rectangle to create our images. We need to “Wrap” these primitive data types with our new image and texture objects that derive from DLink. Our objects hold a lot more information now; our texture object has a composition type relationship with its primitive type Azul Texture, a next and previous reference, inherited from the DLink class, and now an Enum called “name”. The Enums are used so we can locate and differentiate our objects without having to worry about dealing with different string types with varying size and spelling (makes the names consistent). When we get our different texture files wrapped and added to a texture manager class we use those textures to extract the images we want and then draw them on our Azul Rectangles. This will serve us later when we need to draw these images onto our Sprites.

Adaptor Pattern

Our Azul game system uses sprites to render images on the screen. Our Adaptor pattern serves as a middle layer concept where we use encapsulation to store data from our end of the system and send that data to our Azul.Game system. They need to know what texture they use and what image from that texture will be written onto their own Azul Rectangle. Our Sprites have a composition type relationship with it Azul rectangle because if we get rid of the Azul Sprite we lose its rectangle representation too, obviously. It also has an aggregate type relationship with its image associated to it because if we get rid of the Sprite the image still exists and it might be even used on another sprite! We can see the ties of our past concepts to help



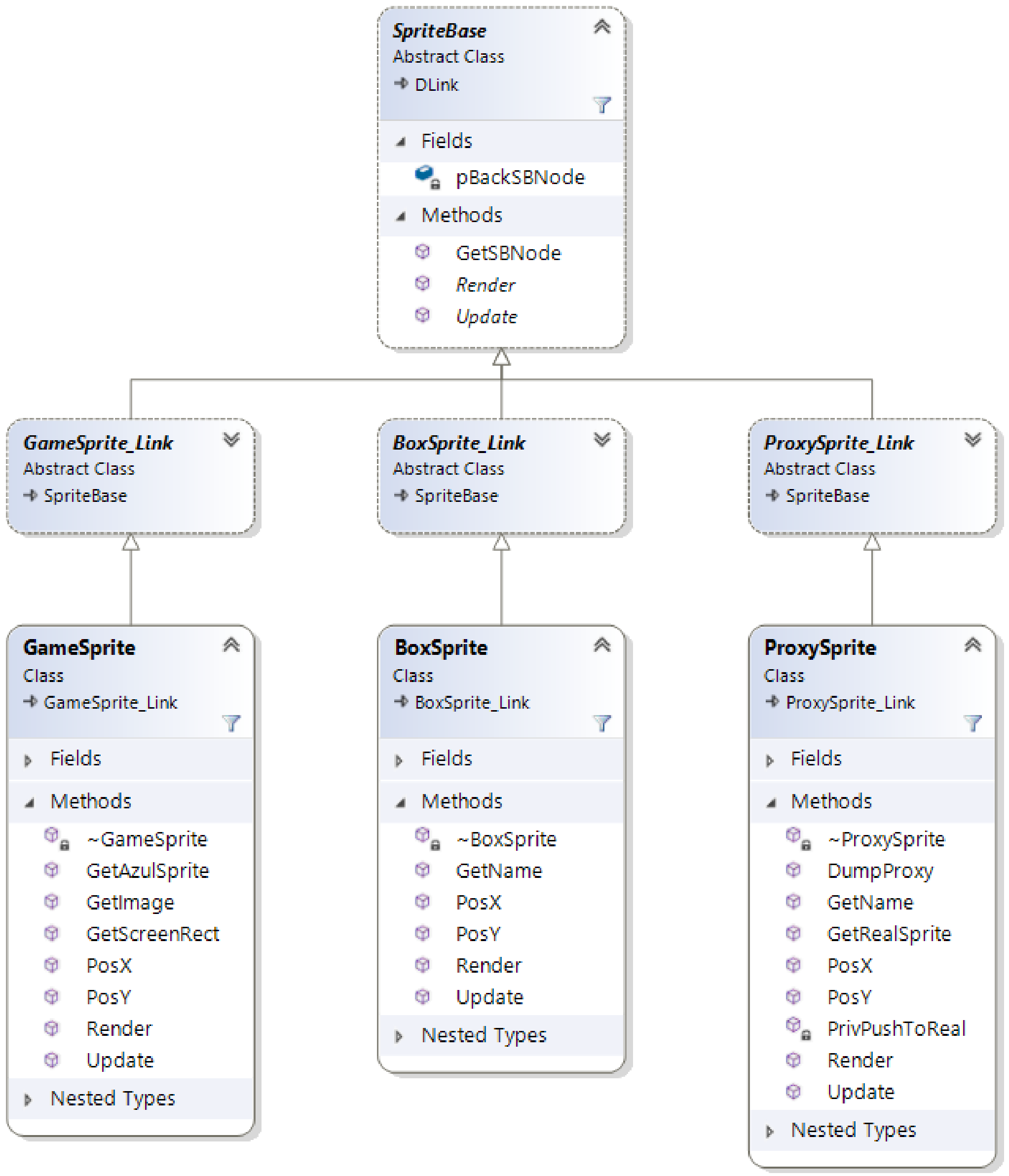
organize our work later on. The Sprite though is the actual representation of our images on the screen, and there is some information the Azul Game Systems needs to know that the Azul Sprite has. Just like the texture and images our system needs to know where we want to the Sprite to be and of what size. The Sprite needs to hold these fields so that we can project them on the screen and when we want them to move we need to change the coordinates and update them in the system. We can see how the adaptor pattern converts our information and hands it off to another part of our system so it's something that it can work with. We give a method a point to an object, it takes that object in grabs information from that location and then sends it to the Azul.Game. Another similar object I forgot to mention is another form of a sprite called a sprite box. It is very similar to a sprite, but it doesn't need an image to be associated with it. It's part of the Azul game system and all it does is create a box with a certain size and color. I'm not too concerned with having to explain how you can change the color and size of sprites. But, you can do that in this game system and it might be more relevant later, so we will table it for now.

Singleton Design Pattern

As our system grows we see that we need more managers to hold the references to these new sprites, the sprite boxes, and previously stated objects (texture and image). But, do we need more than one manager for each of their corresponding objects that they manage? How do we access just one object throughout all of our programs? This is where we see the usefulness of our my very first design pattern, the Singleton(Oh the memories)! A Singleton design pattern is a clever concept, where its instantiated once and is persistent throughout the system like a global variable. In our implementation, the first time the class' static method is called is when it becomes instantiated, I thought a very interesting little addition was adding a private get Instance method that returns the instance of the object. I think that is a very cool safety check that limits the access of the instance and forces a person to call the static create method first, in the specified singleton class. The use of static methods makes them class methods so we can use them anywhere in our system, and they all pertain to that one instance. Now that we have a basic idea of the design pattern we can see how it helps with our managers. We only need one texture manager, one image manager, and so on. We will see at some points where it won't be useful for all our managers. But, as of right now this keeps us from repeating ourselves and helps us to debug our code better, unlike global variables where they can be altered and change anywhere.

Template Design Pattern

As I was saying earlier, the sprite box performs very similarly to a sprite so we can see that maybe we can create a base class that they both derive from. All it asks us to do is implement an Update and Render method so we can change/update their coordinates and then render/draw their rectangles onto the screen. Plus, if we are trying to possibly manage two different objects in a manager we should maybe refer to them only by a base class so we can box them and store them together. This is how helpful the Template Design Pattern can be. What do they all share? They all need a location to be drawn, and to be drawn. All of our sprites have those properties so let's group them together.

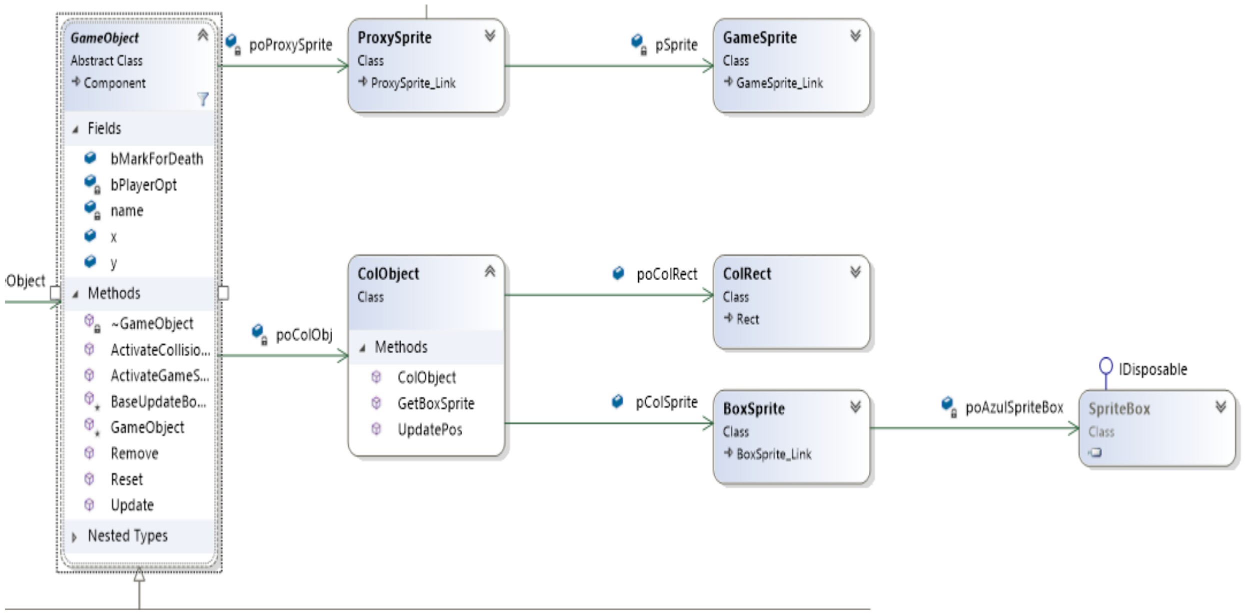


We already started to extract a sprite base earlier so let's make a manager that stores these sprite bases as nodes. These nodes have an aggregate sprite base associated with them and we don’t necessarily care if it’s a sprite or a sprite box.

At this point we can start creating the sprite and sprite boxes, but maybe instead of having each and every Sprite having to render themselves individually, what if we give them to a manager that does it for all the sprites it owns. This is where we see the sprite batch concept starts to evolve. The idea is simple in retrospect; we attach a manager of sprites and sprite boxes, each are upcasted to sprite base and the reference is held by a node. These nodes are grouped together in a manager and attached to a “batch.” This batch now has a composition of a sprite base node manager, which can then be turned around and added into a batch manager. This manager can then loop through and call the draw method on all the nodes in the batches it was given, and now we just populated our screen with all of the different types of sprite by converting them to one general class. There is a subtle difference between the sprite base node manager we discussed and most managers before it. The subtlety is that it can’t have a singleton design pattern because we might have more than one of these managers. The sprite base node has an aggregate association with sprite base, but sprite base can be a sprite or a sprite box. A sprite batch has a node manager associated to it so this means we can have a few different managers. Therefore, we should Un-Singleton the sprite base node manager so we can create a few instances of it if we need to.

Proxy Design Pattern

So far, we have been able to create GameSprites and draw them onto the screen, but our limitation is to create a new GameSprite every time and it’s another Update we must worry about, when moving them. The new Enum names, alone, we must create can be a nuisance. We already created a GameSprite called Crab, but is there a way we can populate the entire screen with them without having to create hundreds of Crab Sprites and naming them something like Crab1, Crab 2, Crab3…..Crab100? Well what is the requirements for a GameSprite to be placed on the screen, just a position x and position y? This is something that we can extract quite easily, and now we have a Proxy Sprite that derives from SpriteBase. A Proxy is defined as something that represents or substitutes another. Our ProxySprite will serve as a surrogate for a GameSprite and it represents it on the screen via the (x, y) location on the screen. All the ProxySprite needs is to have a reference to whatever it's going to substitute, aka the GameSprite. We can go one layer further and create GameObjects that can have a lot of different versatility, which we will see much later with the Composite design pattern.

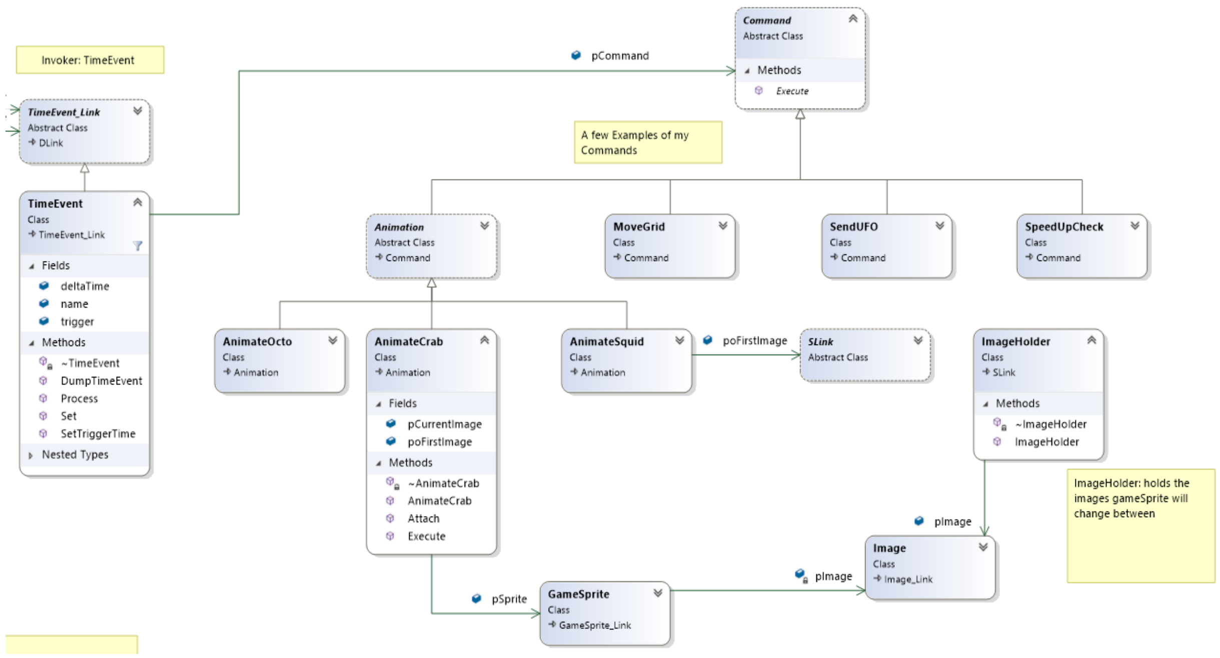


But, for now, instead of a GameObject referencing the GameSprite we can use the Proxy Sprite as a surrogate between the two objects. The Proxy holds the location, on the screen, of the GameSprite and the GameObject will be doing the Update on the ProxySprite that holds a reference to the various GameSprites we need to populate the screen. We see a long line of delegating work that flows all the way back to the Textures and Images we need. Speaking of

Images, we were initially able to swap textures and images to animate our sprites, but is there some way we can even delegate this concept too?

Command Design Pattern

One of my favorite design patterns we used to solve our animation problem was the command pattern. The command pattern is very simplistic in nature, but when utilized properly like in our time event manager it becomes a very powerful tool. The command pattern is an abstract class that has a contract with its derived class and all it asks is for them to implement an execute method. This method can really do anything. In our project, I make an invoker, a TimeEvent object, that has a Command, and when the Client, Time Manager, hits a certain time it tells any TimeEvent that has a trigger at or before that time to Process their command object via the Execute method.



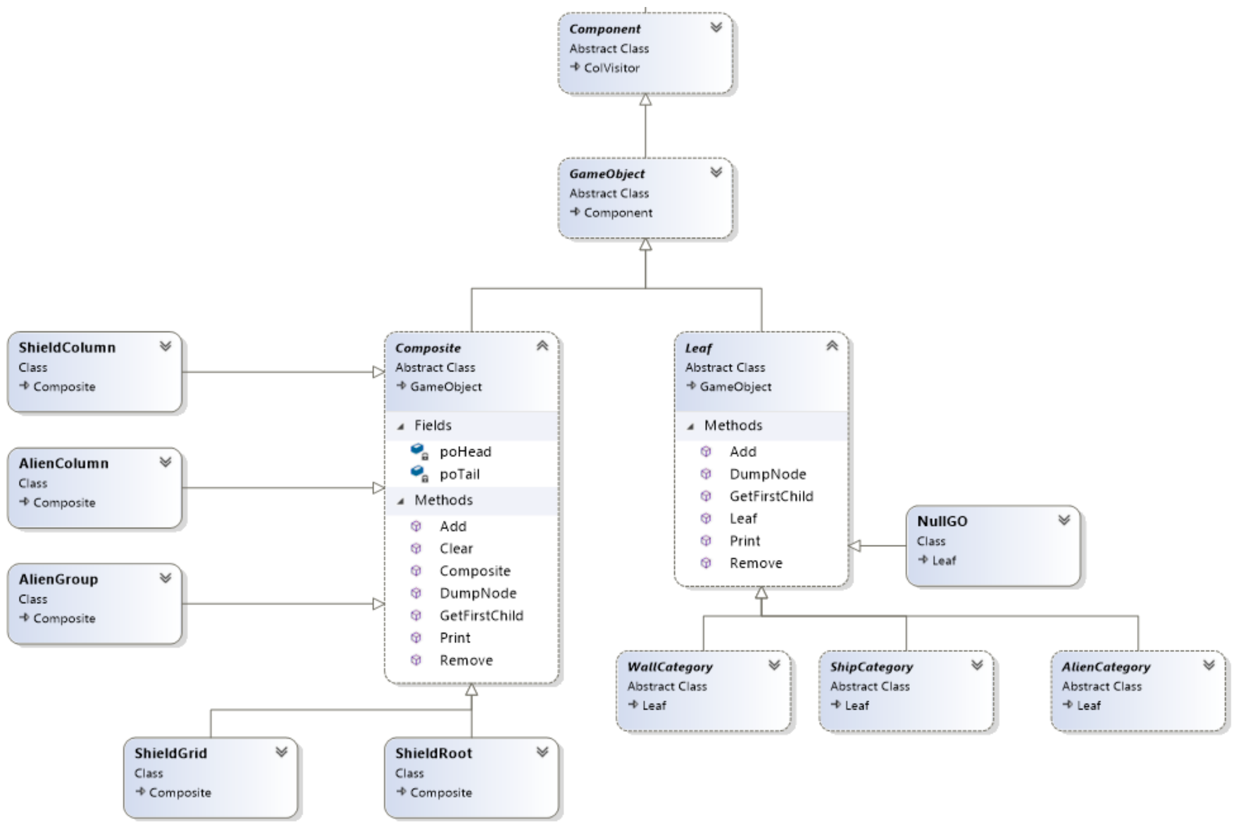
Now in one example we have a Command called AnimateCrab, this command has a receiver GameSprite and a list of images that will be associated with it. When the Execute method is called it switches the current Image of the GameSprite with the next image, and if it’s at the end of the list it goes back to the first image in the list. A very important side note in this method is that it adds the same Time Event back into the time manager. If we didn’t add the event back in it would occur only once and never happen again. We can also see that without a singleton pattern this would be difficult to implement efficiently. This design pattern eliminates a lot of redundant code for manually swapping the images of our sprites. All we do now is put it on a Timer and say, “hey every second process this command”, which swaps the image for us. Not only does it swap it for one sprite, but all GameSprites associated with the Crab since our Proxy Sprites all share the same Game Sprite! Now we can give an appearance of all the aliens marching and moving together. But, we have little hick-up with this timer manager. If we implement it as is, where our derived DLinks, Time Event, are added at the front we have a problem. We might efficiently insert into the list, but when it comes to processing these Time Events. Our manager must go through the entire list and check all the trigger times to see if they need to process them. To make this manager more efficient what if we have it sorted so the lowest trigger times are at the front so our manager doesn’t have to walk through the entire list. If we insert according to the trigger time, we can increase the performance of our timer.

Factory Design Pattern

The project keeps improving, but we are constantly looking at it trying to make it more efficient. We start looking at our Game Objects a lot more and instead of using the proxy pattern directly to project our sprites we use the Game Objects. These become especially important because they link a lot our design patterns together. One of these designs is the Factory pattern. We use it to create our concrete Game Objects like the Aliens; Crab, Octopus, and Squid. We give the factory an Enum in this scenario and from that it does a switch statement to create the specified object. This object is then added to our Sprite Batch(renderer), Game Object manager(updater), and our…… Composite. Before we get ahead ourselves talking about the Composite, let’s keep the focus on the factory. The factory is created in our program and it will make our Game Objects for us, not to mention we don’t even need to worry about the instantiation of our Game objects after we implement a factory. We can essentially give it a variable like an enumerable and we can use that enumerable to give us a variety of different concrete game objects that can differ in subtle ways. This gives us the ability to create a large or small number of objects from a family hierarchy like the Game Objects, and put them in specified locations for use throughout our system, including where they go on our screen. We must keep in mind this very last part of the factory because this plays a vital role in our next design pattern the Composite.

Composite Design Pattern

The Composite design pattern functions like a tree; a composite is part of an abstract class called component. The component holds composites and leaves, and a composite is a group of leaves and even more composites if need be. This is important because if we tell a composite to do something it will delegate that information to all of its leaves in a recursive manner. In our scenario it gives some uniformity to a grouping/ collection of objects in the manner of movement. All we need to do is make our Game Objects derive from the component class, and the composite and leaf classes will derive from the Game Object. This makes our Game Object inherit certain variables that are specific to the component pattern and the composite and leaves inherit certain variables that are unique to the Game Object (Implementation inheritance). (I had to condense my UML because this could be a long picture.)



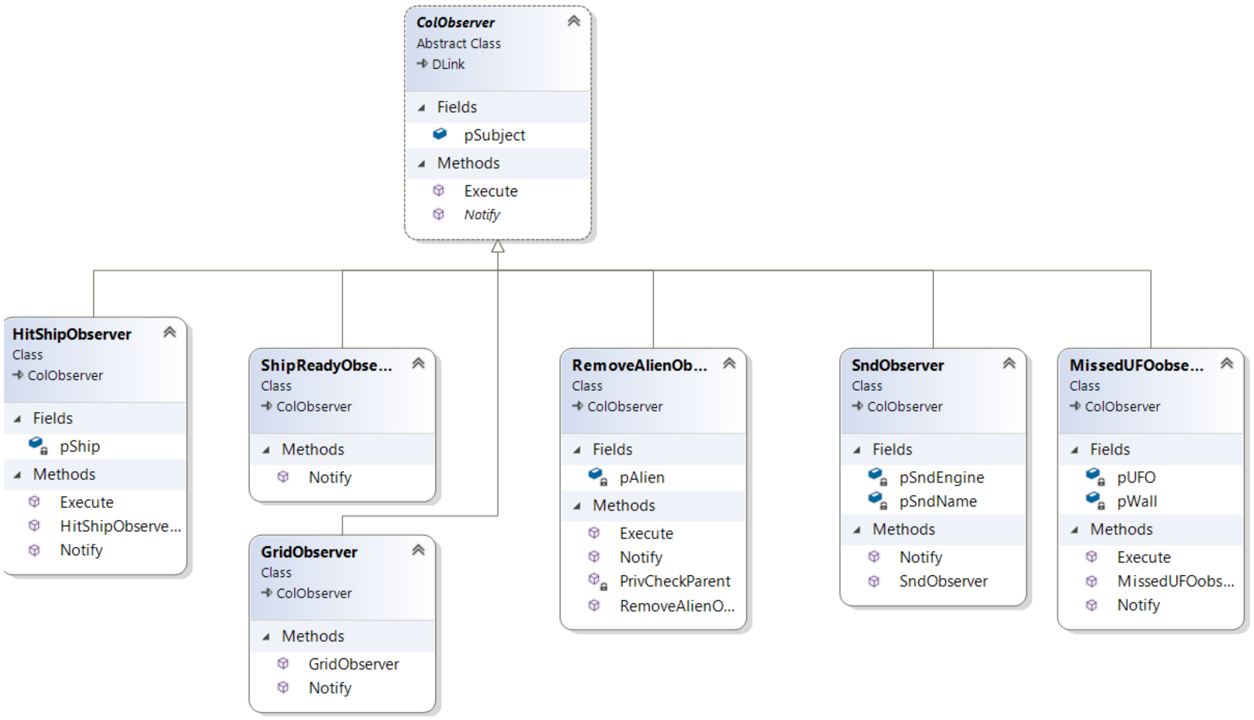
A small example of how these design patterns can work in tandem is when we create a leaf game object like Alien and we attach it to a composite, column, and when we tell our composite to move it enters a loop and calls the move method on all of its attached composites and leaves. Do we have a pattern that can delegate when to do something? We could possibly make a time Event, that has a command, which holds an aggregate reference to the composite. Now we just gave this move command to our timer manager and it does the movements for us!

Visitor Design Pattern w/ Collision System

These various design patterns can start working well together, but when creating our Space Invaders game we need some way for these various Game objects to communicate or interact with each other. This can get very complicated very fast, but with the use of some new and old classes and a design patter or two we can really see some magic happen. Our Azul.Game “hardware” gives us a lot of tools to use and help implement these new goals we have. We already started to encapsulate our primitive data like the Sprite, but we can also wrap the Azul.Rect in a new class that helps us track if say two or more boxes edges overlap or “collide”. We can use these new Collision Rectangles in conjunction with a box Sprite, drawn on the screen, to visually register if objects collide. These can be given to a Collision object that is now owned by the Game Object. These Collision objects exist when the game object is created so it’s a composition, and its gets its dimensions when we use the proxy Sprite. We see a long line of delegating information all the way to the hardware system that gives us some basic information like the dimension of a box that surrounds a Game Sprite. Now that we have these Game Objects that have a proxy and its dimensions, collision object, we can start colliding them. It’s important that we have some kind pattern that checks to see if 2 or more Game Objects are drawn on top of one another. The only issue with this is we need a way for these random objects “know” who they are interacting with. This can be difficult because all the objects know is that there is some form of overlap with another thing, more specifically an object. We can try to assume it’s a game Object, but we have so many different concrete game objects we need to pass information between them. The information we pass is the objects themselves in a new pattern called the Visitor Pattern. A visitor pattern is self-explanatory in nature, but it can get confusing when we pass between classes simply by a function call. This function call is Accept(), which occurs after checking for an overlap in a function called Collide(). We don’t know what objects collided, so we have one accept the other, and that helps us visit the right function call after this swap has occurred. We verify the first subject’s type when we call the accept function which, all of our Game Objects inherit from their hierarchy via a new base class called a Collision Visitor (that has the Component derive from it). So now in this accept function who ever called the function we go to that class. This identifies the first object, and passes in the other object to it. Now we have the other object call the visit Missile and pass in the missile, because that's the only object that can be passed into it. This brings us to the other class, which tells us what we need to do when these two specific classes interact. At this point we used the Game Objects dimensions to see if they overlap, if they do we tell them to "Visit" one another because all we know is their numerical data interacted we use the visitor pattern to identify the two objects and now we……. What do we do? We need to make them do something like maybe change directions away from each other or maybe we remove one. What we need is something to be waiting on standby for these interactions to happen between the pair of collisions.

Observer Design Pattern

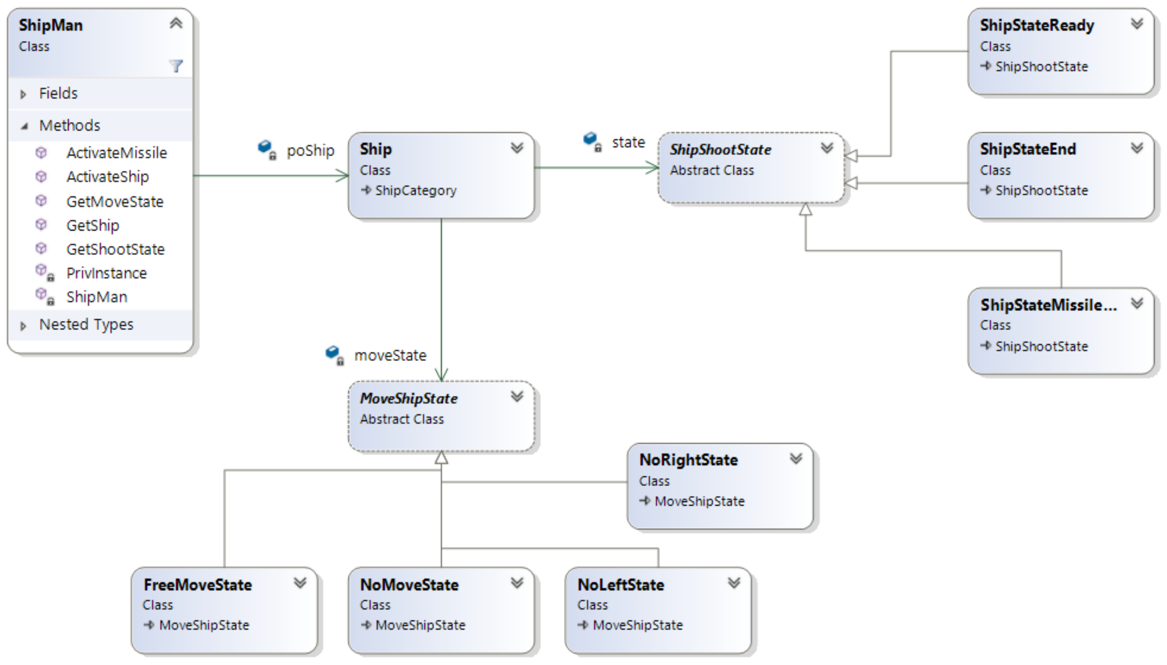
Next on our docket is one of my more favored design patterns, the observer. I think of these as little omnipotent little objects that just wait for things to happen. We attach them to certain objects and when a condition, case, or statement occurs we call these observers to do whatever we task them to do. With the Time Event that told our Alien Group to move, when the group collides with the wall we can tell it to do certain things, like change direction or move down. Or we can remove a game object if it collides with a missile. You can see how important the interaction between our visitor and observer can really make our program more efficient, flexible and ultimately do some nifty things. The diagram below shows the flexibility because we have a slew of different collisions occurring and we can make sound effects happen, delete aliens, change states and so on. All we need to worry about is implementing the base class abstract Execute and Notify method contracts. Inside those methods is where the magic happens. It's weird to think that most of these design patterns basically just give us a hierarchy, or a contract, which essentially gives us a way to communicate from one part of our system through another by passing along data.



We find out which objects interact and then notify any observers that are given tasks to perform when these interactions occur.

State and Strategy Design Patterns

State and Strategy, closely resemble one another. We Have an object and maybe it has a determined set of functions it does, well if it’s in a certain environment maybe we want it to slightly change how it functions. This idea becomes especially important with our collision system and observers. We can tell an object what it should do in a certain scenario if a condition is met, and how do we know if this condition occurs? The answer is an observer. The state of an object almost serves as a set of rules, what it should do and what not to do. We can start thinking about two higher- arching goals of our Space Invaders game. We need the player to control the ship to defend from aliens and we need to control what the game is doing. We want to give the player basic control of the ship move left, move right and shoot. This sounds simple enough, but after understanding some of the rules of the game we see the need to extract these controls into different states. We need a movement state which controls how the ship moves and we need a shooting state to see if the ship has a missile to shoot or not. It wouldn't be much of a game if we could just rapid-fire shoot missiles non-stop forever. We also run into a collision system problem when we try to implement all of the functions in one state. If our ship collides with a wall we want to disable it from moving past that wall off the screen, so we change the movement state via the observer and disable the control that makes us move in that direction. The moment we move in the opposite direction from the wall, we change states again to a free movement state again. If we had kept the single state we would run into situations where if a missile is flying and then we collide with a wall we might gain the ability to shoot another missile because the state we changed to lets us shoot another missile, but that shouldn't be the case because we only can have one missile flying at a time.



I noticed these edge cases and almost immediately, and changed my ship to implement two different states and I haven't needed to worry about the various edge cases. The Game State is a bit different from the ship states, but the basic concepts remain. The Game needs to cycle between a set of visuals states an Intro Screen, a Game Screen, and Game Over Screen(in essence). My Space Invaders game only knows about load Content, update, Draw basically. I do other stuff without it knowing and it thinks it going on doing what it's supposed to do, but really, I'm loading data at different times changing what is on the screen and giving it a much better flow of control. I used a game manager to hold these different properties and when certain conditions were met I had it go to the next level or had it go to game over. It took a lot of work, but I finally got it to cycle properly from beginning middle to end and then go back to the intro screen to start again.

There is so much that I need to do with my project, but I have a lot more confidence and a bunch of different tools now to better construct software architectures because I vastly improved my knowledge of Object Oriented Principles (always, always room for improvement though). I now know how objects can derive from base classes, by implementing certain methods. They can inherit certain properties that allows me to group different objects together as long as they share something unique between all of them. I can even extract similar methods in derived classes and place them in the base class so I don't have a bunch of redundant code in each derived class. I can even override the base class methods (abstract or virtual) by using the override in the derived class to change the method to better fit the derived class. This all helps with making clean, concise, and overall better software by making small iterations overtime and never being satisfied when something is working. I always need to make my code suck less, and

I will do that by refactoring and getting my hands dirty to make something more efficient after I get it working in the first place.