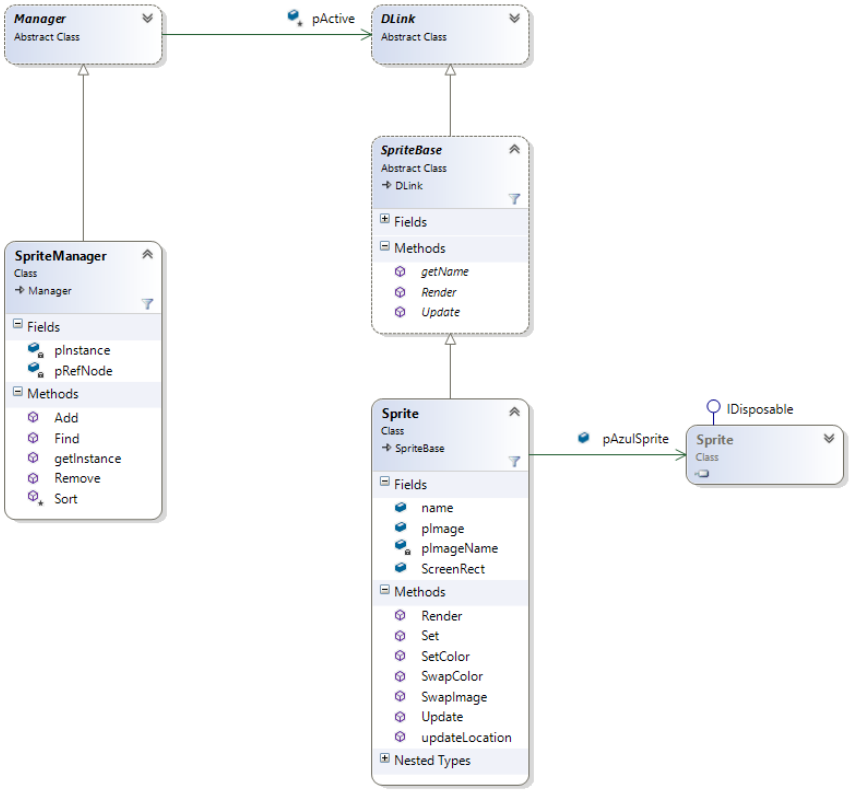
“Space Invaders”

Design Document

Joe Richard

SE 456

Adapter Pattern



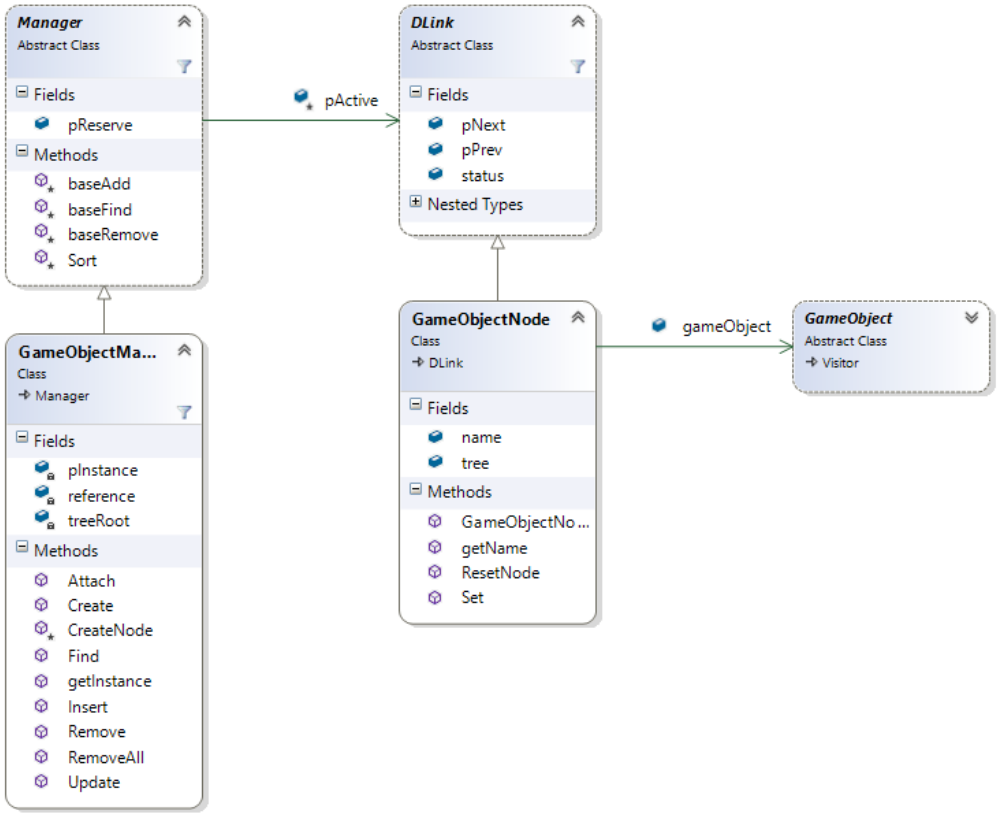
Throughout the Space Invaders project, I have needed the ability to deal with different objects outside of their usual locations. An issue that I ran into several times was that it was difficult to just take an object and plug it in somewhere else and have it work without issue. Usually this was because the whole object class or some of the methods I needed were incompatible with how they were being accessed in the pertaining class. To solve this, I implemented the Adapter pattern.

What the adapter pattern does is quite simple in theory. It takes an object that would be incompatible with how we are using it in the reference class, and wrap it inside another class that is compatible with the referencing class. Inside the Adapter class,

there is a reference to the incompatible object, at which point we can set the methods up inside that adapter class to create a bridge or connection between the reference class and the object class.

The easiest example of the Adapter pattern inside the Space Invaders project is in locations where I needed to access one of the many Azul.Sprite objects. All the data corresponding to each alien sprite on the screen is saved inside of its corresponding Azul.Sprite object. The issue was that the Azul.Sprite methods were difficult to connect with other parts of the project in such a way that made sense. What I ended up doing was wrapping each Azul.Sprite object in a Sprite class, that contained an Azul.Sprite object. I then made all of the Sprite class methods easy to call, and all the methods were doing was invoking a single Azul.Sprite method, or a collection of them. This made it easy to update a sprites size or location on the screen, without having to deal with updating the different fields in the Azul.Sprite object. All I need to do now is call Sprite.updateLocation and pass the necessary parameters, and it takes care of the rest.

Object Pool Pattern



What I learned really quickly during the duration of this project was that implementing any functionality has a cost. In most cases that cost is time, and over time, each little cost adds up. With creating a video game, you must constantly be watching your time cost, and find ways to cut cost where ever possible. If you let the time cost get too high, then the game will run slow or maybe not even work at all.

One of the biggest time costs in any project is the creation and allocation in memory of new objects. In many small cases, it may not seem like a huge issue, but when you have fifty five alien objects, wrapped in 11 column objects, which is all wrapped in a grid object, AND you still have each individual brick in the shield that is created, the

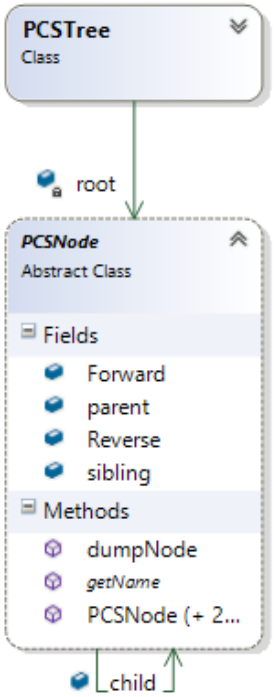
ship, the mother ship, each bomb, each missile, etc.… The point is that all of these objects start to take their toll on the speed of the game, which is bad. So I needed to find a way to keep all of these objects, but find an efficient way to do it.

What the Object Pool pattern means is that instead of creating and deleting and creating and deleting objects, create the objects you need, and then when you don't need them anymore instead of deleting them move them to a list of inactive or reserve objects. Then when the system needs another specific type of object, it can just pull one off of the reserve list and update the data within it to whatever it needs currently. If the reserve is empty, then the pattern creates a few more objects to put on the reserve list for later use. The goal is to call the "new object()" constructor as few times as possible. As a side note, the Object pool pattern is usually constructed inside a singleton object, so that no matter where you are in the project you are able to access the same active and reserve lists of objects.

The object pool design pattern is used in just about every manager class in the space invaders project. Each manager keeps track of a specific object type and how many are active and how many are in reserve. Lets say for example, that I need to get ahold of a new GameObject object which could be a bomb, missile, alien, ship, mothership, shield brick, etc.… these are all subclasses of the GameObject class. The GameObjectManager looks in its reserve list and sees if there are any GameObject objects there, and if there are not, it creates a few new ones to buffer out the list. It then takes one of these GameObject objects, moves it to the active list, and returns a

reference to it that can be updated and filled with the specific subclass information that is needed.

Composite Pattern



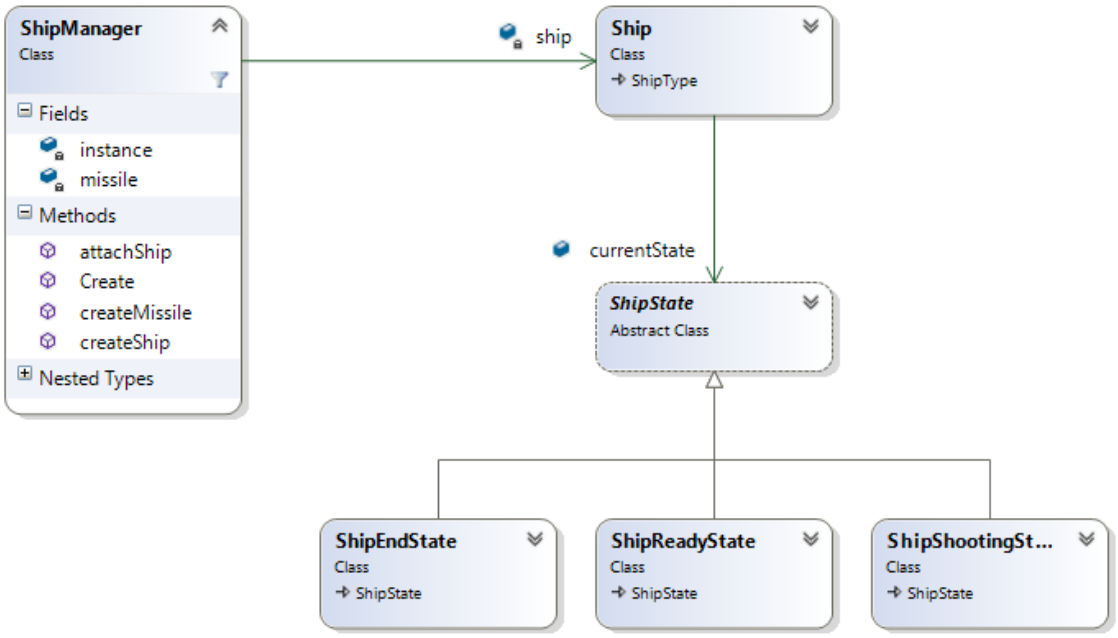
When working on the space invaders project, a problem that I ran into early on was how to get a group of aliens to move in a synchronized fashion. The issue was that it would take quite a bit of time to set up a list of all the current aliens that are active and their update functionality, every frame. I needed to find some way of connecting all of the different pieces of the Alien grouping, so that no matter how many aliens were on the screen, I was able to access them in a simplistic way. The answer to this problem was the composite pattern.

The composite pattern deals with creating a tree of sorts, that holds hierarchical information about a given situation. At the root of the tree is the main object that every other object is associated with in a smaller way. Think of it in terms of an actual tree. At the very root of the tree would be the trunk, and the trunk can have many branches, and on each of those branches you can have many different leaves. At the end of the day, we want to see the Composite tree as just a tree, and not a combination of trunk, branches, and leaves.

Going back to the problem I had above, the composite pattern made it possible to connect all of the aliens in a hierarchical manor. This allowed me to just keep track of one GameObject instead of several. I broke down the Alien grid into just that, a combination of one grid that had eleven columns as children, and each column had five aliens as children. All the children had knowledge of their order in the list, so once the first child was found, it becomes simple to iterate through each of the siblings of that child. For example, when I wanted to move the whole grid at once, I would call move grid on the grid object, at which point that move method would traverse the tree. Once it reached each alien, it would call their move function. Overall, this saves time cost and memory cost since we only need to have a reference to the grid object to access the whole tree.

I ended up taking this composite pattern and implemented it in every type of Game Object that I would be using in the project, just so that it was easy to access any one piece that I needed at any given point.

State Pattern



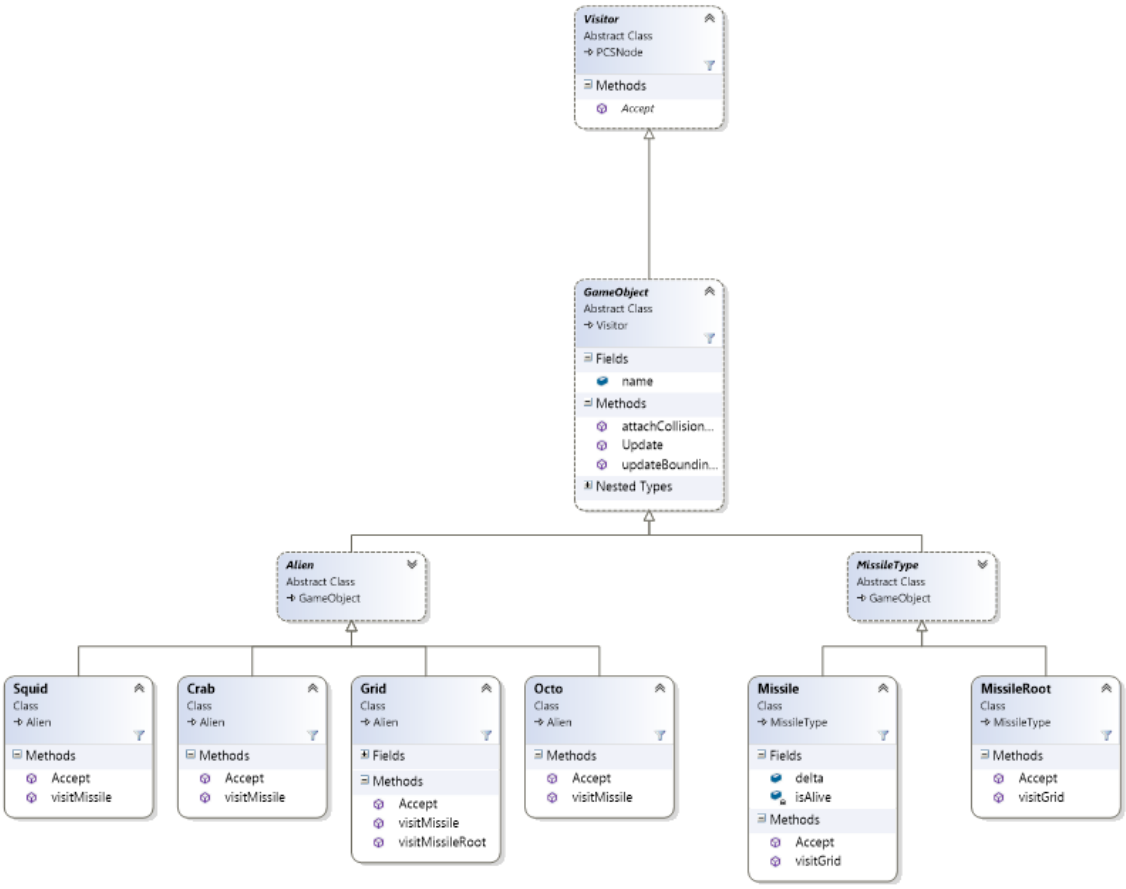
At different points in any game, you have events that occur, and you want certain sub events to take place because they are affected by that event. Whether it is a player losing a life, or a player attacking their enemy, the events can be very broad or very specific. The easiest way to deal with these constant events is with the state pattern.

What the state pattern does is give a way for an object class to change its behavior based on events outside the class. An object can have as many states as you want, with each one dealing with a very specific situation. Each concrete class of the abstract state class carries the same basic functions that allow the Object class to do its jobs. The object class has a reference to the abstract state class, which can be updated to any of the different concrete states that have been created. Whenever that object class needs to do a specific event sensitive functionality, it calls the state reference and then

the corresponding method. In one case that method may do its job, in other states in may purposely do nothing because that is what is expected for the corresponding state.

In the case of space invaders, one of the big event driven classes was the user ship. I wanted it to be able to move and shoot, but not be able to shoot again until the first missile had hit something and been destroyed. In another case, I would want the user ship to do nothing because the player has run out of lives, and therefore the game has ended. The best way to accomplish all of this is with the state pattern. The basic state that the user ship is in is the ready state. In this state it can shoot, and move left and right without issue. When player fires a missile, the ships state reference is switched to a ship shooting state. In this state, the shoot method is left purposefully blank, so that even if the player hits the shoot button again, another missile will not be fired. When the missile that the player did fire is finally destroyed, the user ship's state is switched back to the ready state and the player can fire again. If the player manages to lose all of their lives during the course of a game, then this causes the user ship's state reference to switch to that of the ship end state, where all movement and shooting methods have been left intentionally blank, releasing the player from all their abilities.

Visitor Pattern



In many video games, there are lots of factors to be considered at any point. Using a first person shooter as an example, at every frame of the game the game engine needs to answer among others, these questions:

- Has the user been shot?
- Has the player shot a shootable entity, like an enemy, red barrel, stack of explosives, etc.?
- Which specific object did the player shoot?
- Where on the specific object did the player hit?

This can be a lot to handle, and even more so if these questions need to be answered every frame of the game. Luckily the visitor pattern takes a lot of the guess work out of situations like these, and helps us answer the above questions much faster.

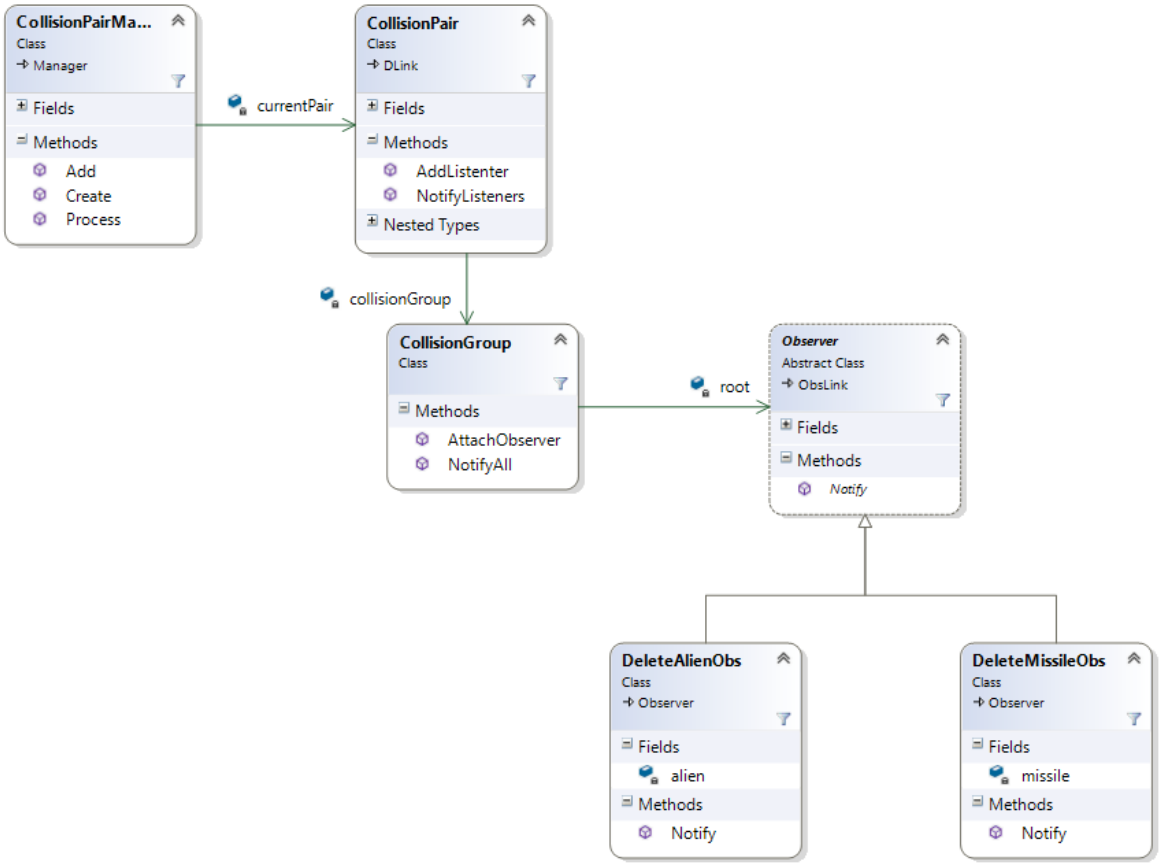
The visitor pattern is exactly what it sounds like. In its most basic form its job is to visit every member of a specific list and do something with each member. Getting more specific, the visitor object can visit every member of a tree, and have them all update a value or field. It is a good way of using automation to solve the issue of updating a lot of objects in the same way.

In the case of space invaders, I used the visitor pattern to figure out collisions. What I mean is that at any given point during the duration of the game, the alien grid can collide with a wall meaning it needs to drop and change direction, a missile can hit an alien, which means that alien needs to be removed and replaced by an explosion sprite, and an alien bomb can collide with the user's ship, which means that the user ship needs to lose a life and do its death animation. These are just a few examples, but they are the most basic that the game needs to work. To check to see if any of these collisions has occurred, I implemented the visitor pattern. Every frame, the visitor checks to see if the roots of any two GameObjects has occurred. If they have, the visitor jumps back and forth between the two game object trees, narrowing its search for the two game objects that were involved in the collision. To use the case of a user missile hitting an alien, the visitor first checks to see if the missile root object hit the alien grid object. If this is true, then the visitor dives into the alien tree, and checks to see which column the missile hit, if any. If the missile root did hit a specific column, the visitor goes

into that sub tree and checks to see which alien was hit, if any. If it does find that specific alien, it then checks to see if the missile inside the missile root actually hit the specified alien object, or if it

was a really close miss. Once it is determined that the player's missile did in fact hit a specific alien, the correct observers can be notified and the correct actions taken to delete the missile sprite and delete the alien object.

Observer Pattern



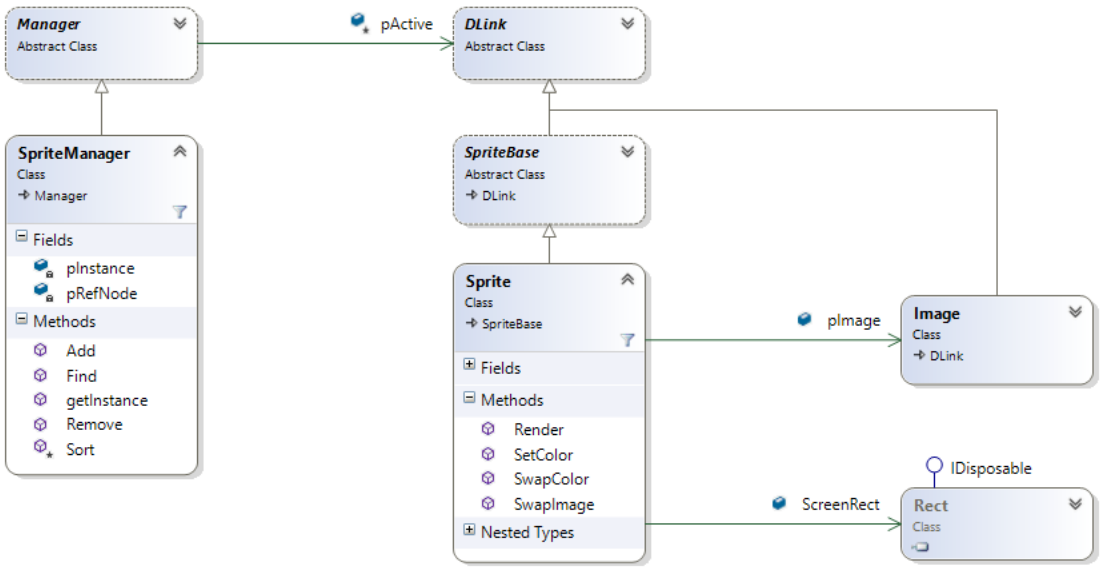
In the above visitor pattern we talked about finding out specific information about specific events or situations that occur during a games lifetime, but we still run into the issue of how do we respond to those specific events when they occur? Going back to the first person shooter example, we have figured out that the player shot a red barrel, and we need the barrel to respond to that situation by exploding into a beautiful fireball. The observer pattern allows us accomplish this task.

What the observer pattern, just like the visitor pattern, does exactly what it sounds like it does. When a specific event occurs during the lifetime of a system that is running,

the observers that are attached to that given event are notified that this event has occurred and that they need to respond accordingly. The observers can then cause whatever sub-event occur that they were set out to do, and then they go back to “observing” for the next event the next occurrence of the event they are attached to.

In the Visitor pattern, our goal was to figure out which two game objects had collided during the space invader game’s lifetime. Once we have figured out which two game objects were involved, we can notify all of the corresponding observers that are attached to the collision group that involves the user missile colliding with an alien. In this case the observers that are notified are the deleteAlienObserver, the deleteMissileObserver, and the shipReadyObserver. The observers delete the alien object, the missile object, and switch the user ship’s state to shipReadyState respectively. This type of notification and response occurs in any collision event that should be expected to give a response.

Flyweight Pattern



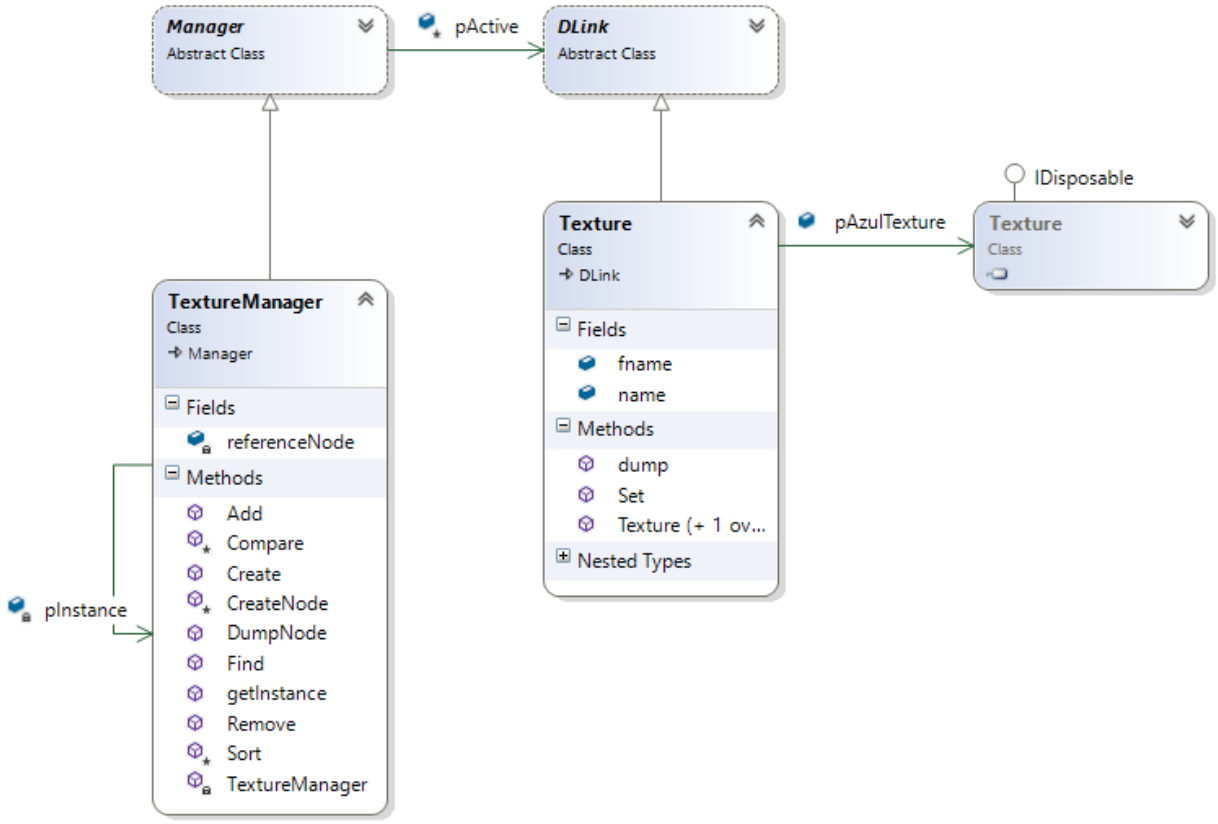
In big software projects, you are usually going to run into the problem that you are constantly needing a class object that have the same state as ten other instances of the same class object, but in a different location in the system than the other class objects. Over time , this can bog down your system, and its impractical since all of these objects have the same state. The solution is the flyweight pattern.

What the flyweight pattern deals with is the redundancy of creating class objects that have the same state as other class objects of its type, just because you are in a different class than the other class object is located so it cannot be reached. What the flyweight pattern does is take that redundancy out of the equation. A flyweight class will hold a reference to a specific class object, and the rest of the class will deal with anything that needs to change else ware around this class object. Every instance of the flyweight class will have a reference to the same single class object that was created,

but they can all update the rest of their information in their own way. This reduces redundancy and the complication of the code.

In space invaders I used the flyweight pattern when dealing with images or textures that I needed to have multiple copies of or needed to use in many different places. Instead of creating eleven identical squid alien image objects and then passing them around to eleven sprite objects that would have to be squid alien sprite objects, it became easier to just use the flyweight pattern. Now every alien sprite that needs to display a squid image is referencing the same image of the squid alien. I only needed to create one squid image object, and every place else it is just being referenced. If I need to change the squid image in any way, I only have to do so in one place, as apposed to several. Overall this cuts down on the overall time cost of creating multiple identical image objects, and there are not multiple identical image objects hogging useful memory on the system.

Singleton Pattern

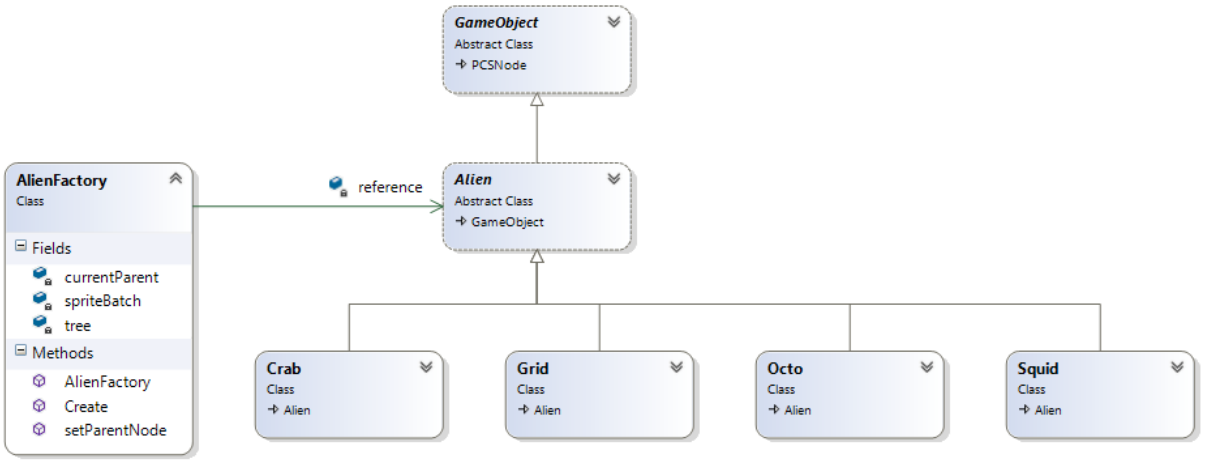


In programming, it is tough to have an object that just has one set of data. Usually each class has its own instance of the object, and deals with that data internally. This can become an issue when we want to share an object between several different classes, and have each class have the most up to date version of the data, even if the data was updated somewhere else. The solution to this problem is the singleton pattern.

The Singleton Pattern is a way of making sure that only one instance of a specific object is created during the lifetime of program. No matter where it is needed, or how many times it is called, the singleton object being referenced is always the same instance that was created at the beginning of the programs lifetime.

The biggest Example of the Singleton design pattern that I used for this project was with the manager classes. I was able to accomplish this by adding an instance variable to each of the manager classes, and only referencing that variable if I wanted to work with that manager. I would use a static method to get the private instance of the class that I needed, and I would get that instance. From the outside it worked just like creating a new object instance, but I was never creating a new instance. I was just being given a reference to the only instance of the manager class that was available. This allowed me to access different sets of data at any point from any location, with minimal issues.

Factory Pattern

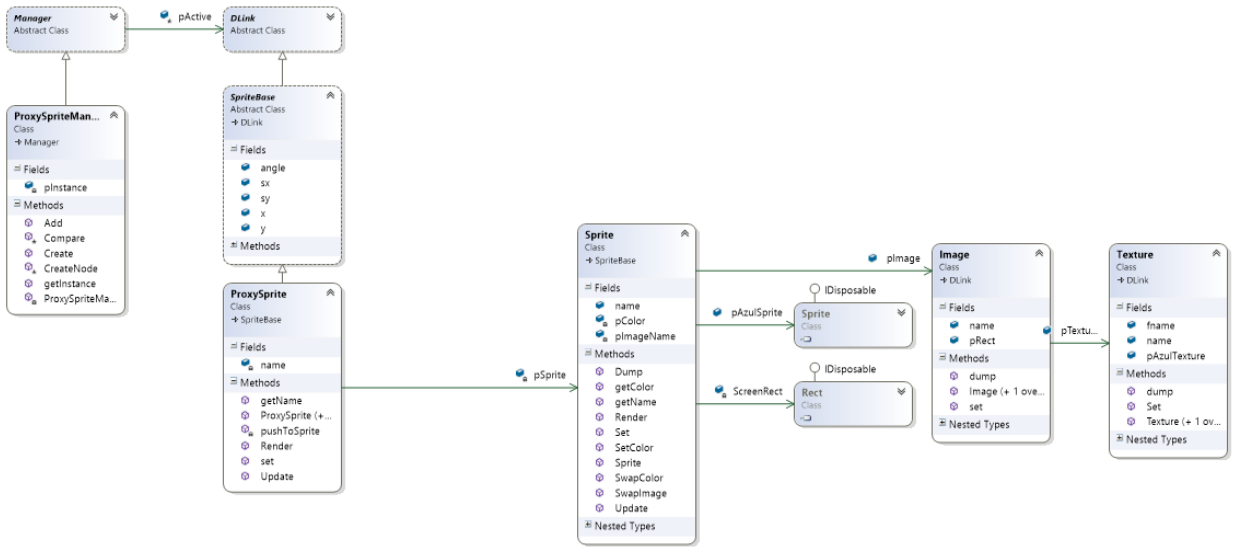


One issue that a lot of programmers have is separation. What I mean is that the front end of the program has a lot of knowledge about what is going on in the back end of the program. The goal in software engineering is to have the front end know as little as possible about the back end and can still be able to set up everything it needs to do to make the program work correctly. One way that we can facilitate the separation is with the Factory Pattern.

The Factory Pattern is a way of simplifying the creation and initialization of specific Objects, while hiding a lot of the backend work that goes on in the setting up of those Objects. It allows the front end to create these objects with as little knowledge of how the Objects are built as is possible. That way, minimal changes need to be made to the front end if the back end is refactored or changed in any way. This also keeps the front end from messing up the back end set up, since it can only access the factory to create objects.

In the case of the space invaders game, I used the factory pattern to facilitate the creation and initialization of all the different alien Game Objects that appear in the game. I set up an alien factory that did a lot of the work to where all I need is to provide the type of alien that I want to create, as well as the location I want it to appear on the screen, as an x and y value. The factory then does the work of creating an instance of the alien that I want, and initializes it. Instead of remembering the constructors and subtleties for Squid, Crab, Octo, Column, and Grid, I just tell the factory which one I want, and it makes an object for me. It also adds that separation between the front and backend that is necessary to keep things simple and organized.

Proxy Pattern



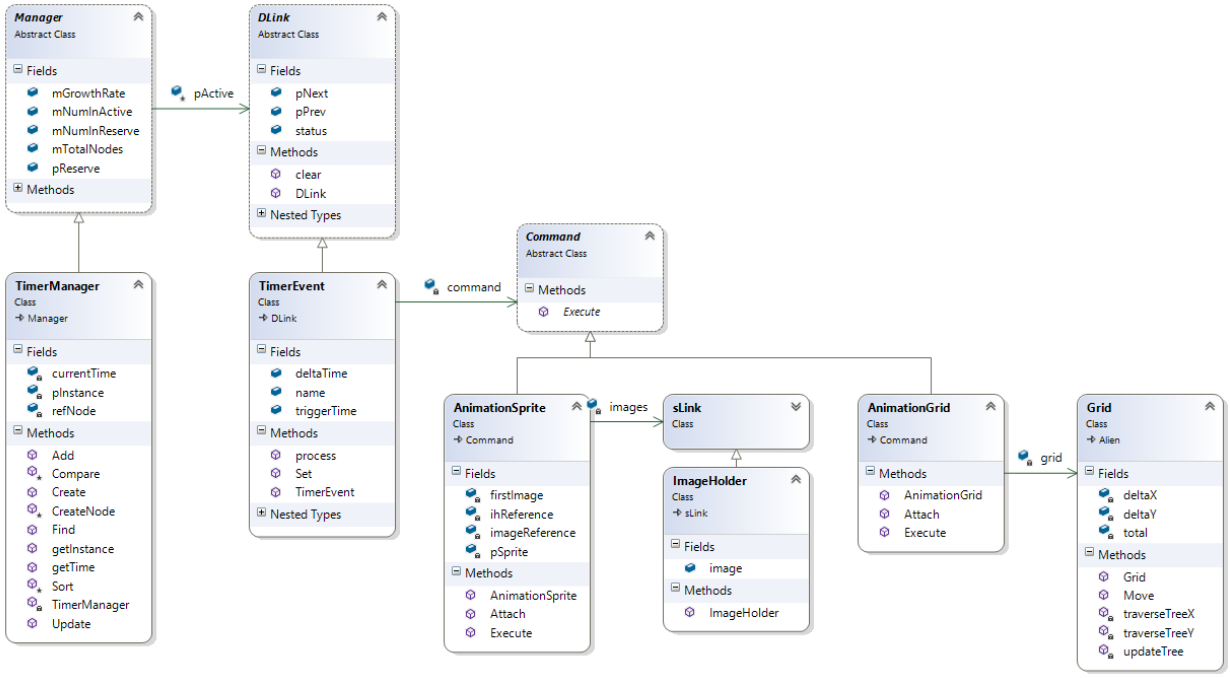
There are certain times where you may want to connect two different class objects, but there is no easy way to have them both interact. This can create difficult situations with the code, which can result in complicated entanglements that are too tightly coupled. A solution to this problem is the Proxy Pattern.

The Proxy Pattern is a way of supplying an interface for one class so that it can interact with a second class. The first class can pass all of the necessary data to the proxy object that can then sort through the data, and pass it on to the second class in the format that it desires. It is pretty much a bridge between classes that would otherwise not be able to connect with each other.

In the case of the Space Invaders project, I used a Proxy Sprite class to make updates to a Sprite Class without having to directly call on that Sprite class. By doing this I was able to have it so that I could change the values of the Proxy Sprite object, and then call an update on the Proxy Sprite that would then pass the values over to the

Sprite class. This makes it easier to connect to that Sprite class, than trying to work with it directly. This also creates some degrees of separation between the front and backend of the program.

Command Pattern



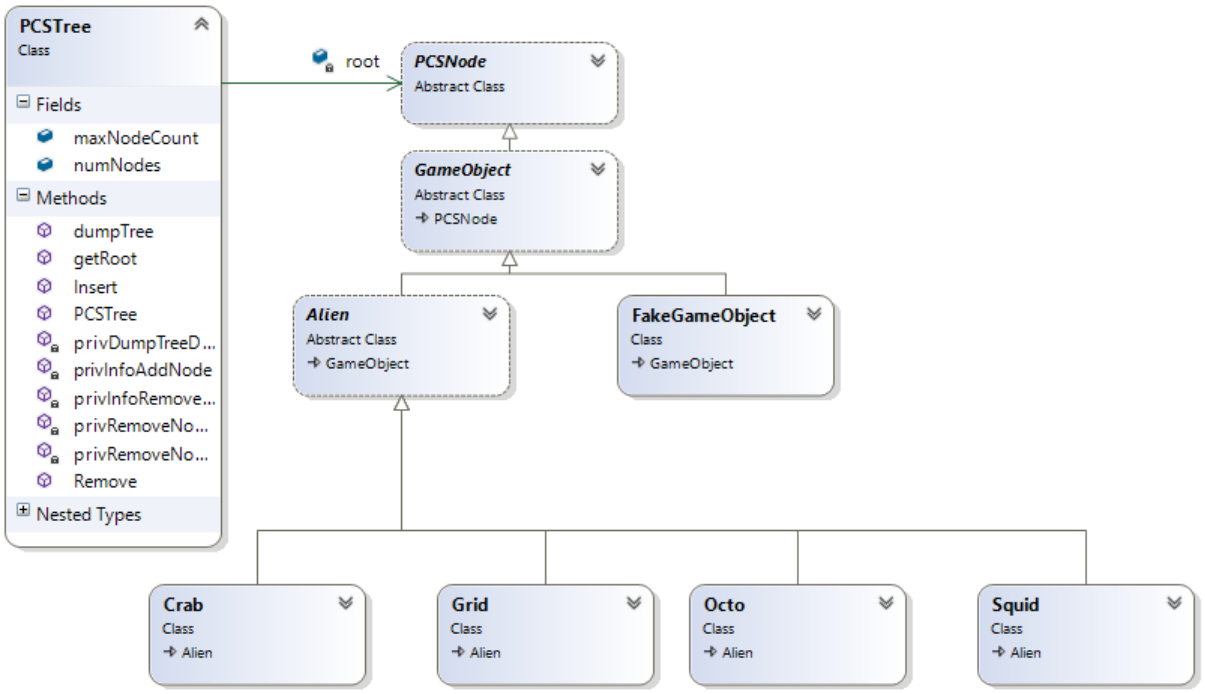
A lot of times during the lifetime of a program, you may wish that things were running like clockwork. Maybe you want something to happen consistently every few seconds, instead of whenever it comes up in the for or while loop. One solution to this dilemma is the Command pattern.

The Command Pattern is a way of setting up several classes that can be called to execute a function whenever a situation is triggered. The Command is usually an abstract class with one method that is used to execute a specific task when overwritten

in the derived classes. Then when a specified situation arises, a command calls its execute method, and the derived class executes the method for the task.

In the context of the Space invaders game, I used a Command Pattern to deal with any tasks that were going to be called at a specific time interval. This includes the movement of the Alien Sprite Grid, as well as the animation of the alien sprites themselves. The AnimationSprite and AnimationGrid classes are derived classes from the Command abstract class, which means that they have an inherited execute method. This method is called within the TimerManager class that checks during every update to see if any of the time intervals have been reached, and if they have to call the necessary Execute methods to make the grid move and sprites animate.

PCS Tree Pattern

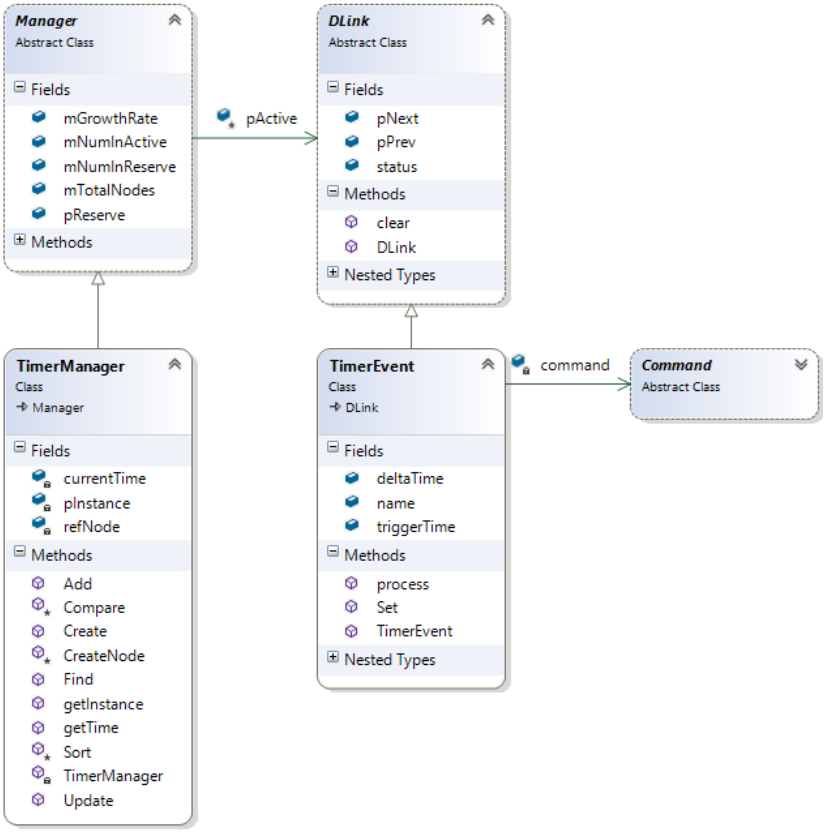


Chances are that if you have been programming long enough you have come across different types of containers called trees. They come in all different shapes and sizes, and each style has its own benefits. For what I wanted to accomplish with the project, the style that I needed was the PCS, or Parent/Child/Sibling, tree Pattern.

The PCS Tree Pattern is a way of structuring objects in a hierarchical manner. Like other styles of tree, there are nodes that can be a parent and/or a child. This pattern is different because you can also have siblings. This means that a parent node can have three children, and those children know about each other because they are siblings.

In the context of the Space Invaders game, I used the PCS Tree Pattern when dealing with the movement of all the alien sprites as a single group or “grid” of alien sprites. What I ended up doing was using the PCS Tree to make the grid a parent, and then have all the alien sprites that needed to me manipulated places as child nodes to the grid parent. This also made all the alien sprites siblings, which made them easier to go through when updating their x and y coordinate location values. This way, when I wanted to move the grid as a whole, I would find that grid node, and recursively call all of its children to update their coordinate values. Then when all the values had been updated, I redrew the sprites, which showed them having moved to the new location I had set for them, thus giving the illusion that they are moving as one unit across the screen.

Sorted Priority Queue Design Pattern



Jumping off of the issue of needing to call a command at a certain interval, once we have the command that we want to call at a certain time, we still need to put it in a container to save it and execute it at the correct time. This can be an issue because most containers are not executed based the priority of what needs to called. What I had to do then was use the Sorted Priority Queue Pattern to create just that.

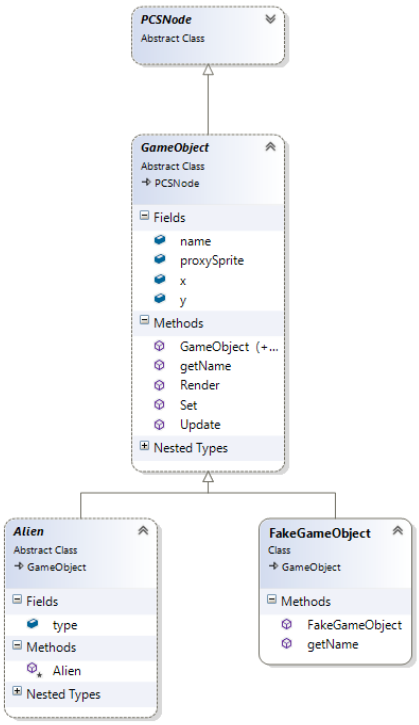
The Sorted Priority Queue Design Pattern is a way of keeping things in a certain order, while constantly adding and removing things from the Queue constantly. The queue needs to keep its correct order and update its order whenever a new object is

added to the queue. The goal of this design pattern is to lower the amount of time it takes to traverse the queue looking for a certain object.

In the context of the space invaders game, the Sorted Priority Queue is implemented as a double linked list of TimerEvents. Each TimerEvent has a command associated with it, so that when a certain time is reached during the game, if that TimerEvent is within that range of time, its command is executed. Right now the commands being executed are either grid movements or the animations of the alien sprites.

For the TimerEvents to always be in order, any time a new TimerEvent is added to the list, it is added at a specific location. To do this, we compare the new TimerEvent's delta time to the delta time of every TimerEvent in the list until we find a TimerEvent that is either equal to or greater than the delta time of the new TimerEvent. We then add the new TimerEvent to the list just behind the last TimerEvent checked.

Null Object Pattern

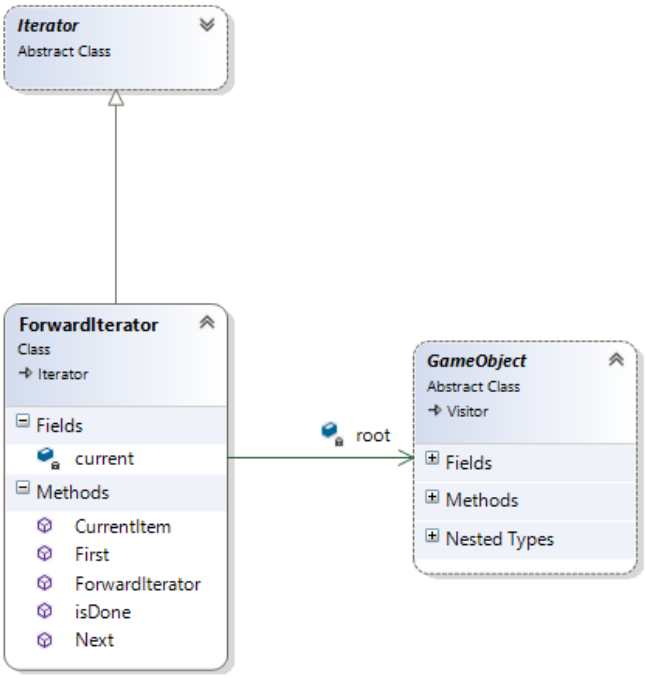


There were quite a few times in my project where I had an object that I wanted to add to a certain container, but I didn't want it to do anything if that container was ever called on. I just wanted it to sit there in the container so that I could access the data later. The issue I was running into was that I still needed to come up with a solution for the situation that the Object was called upon to do something I didn't want it to do. The answer I found was the Null Object Pattern.

The Null Object Pattern is a way of adding an object to a list that does nothing for the most part. It is mainly a placeholder. In some situations it is necessary to pass an Object as a parameter to do something else that is necessary, so you can pass a Null Object that is a real object except for the fact that none of its methods return anything and it has not fields.

In the case of the space invader project, I used the Null Object Pattern to create a FakeGameObject. When using the Alien Factory, all of the game objects are created the same way. Well we don't want to have certain Game Objects update and render such as the Grid that is being used more as a container than a Game Object. So instead of creating a GameObject associated with the Grid, we just plug in a FakeGameObject as a placeholder. It can be updated and drawn, but nothing ever actually happens.

Iterator Pattern



Having created all of these different types of containers for this project, there was no clear cut way to go through these custom containers using a for loop that accessed a different node or link in the list with each iteration. Since I created the containers, I had

to come up with a way to iterate through them as well. This is where the Iterator pattern came in.

The Iterator Pattern is a way of traversing a container of items in a way so that each object in that container is visited and can be tested/executed/method called in some way.

In the space invaders project, I used the iterator pattern quite a bit, since we had to create our own containers instead using any of the C# containers like arrays and/or lists. Once we had a double linked list that we wanted to traverse through, we set a variable to variable to the head of the container, and began the traversal. In the case of the code above, while the variable we created was not null, we would check to see if the current nodes name was equal to that of a specific name that we were looking for. If it was, would break the while loop and return the node that had the name we were looking for. If the node we had did not have the name we were looking for, we set the variable to the next node in the list, and repeat the process until the node we are looking for is found or we reach the end of the list.