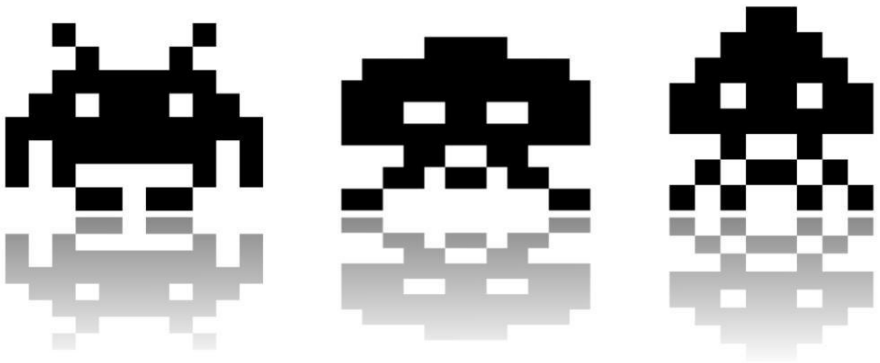
SPACE



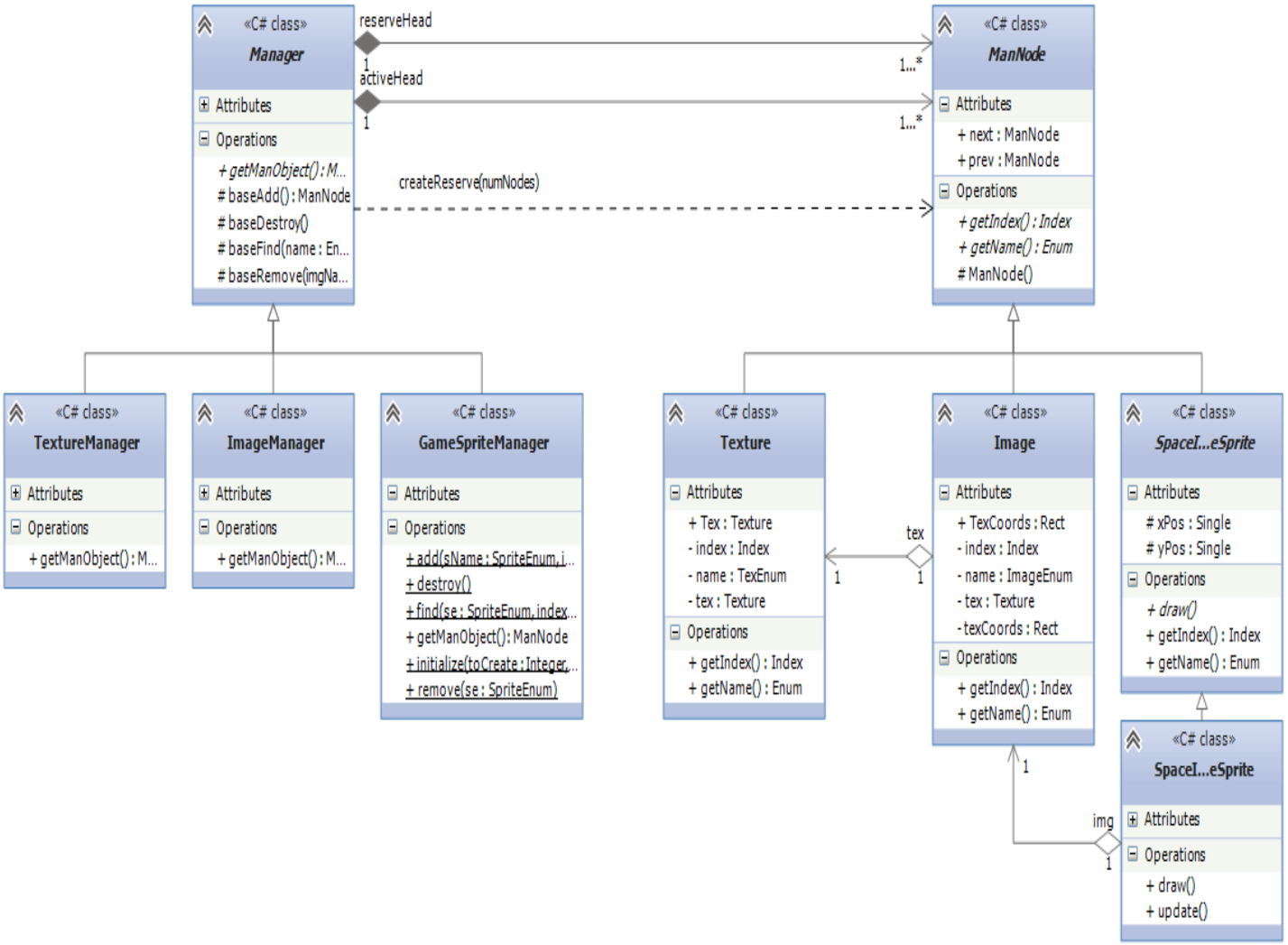
INVADERS

Table of Contents

Link to video:

https://www.youtube.com/watch?v=wesah5x14cg

Manager System



The main, underlying system of Space Invaders would be the manager system. The role of each of the managers is to manage each of the resources so that each resource is in its own centralized location. The managers are derived from a general *Manager* class. This abstract class contains two references to lists. One list is

called the reserve list and the other is called the active list. The reserve list holds all of the objects that have been already allocated, and the active list holds all of the objects that are currently active and in use. If the user wants to use a specific kind of Image, they would ask the ImageManager to add it. The ImageManager would then pop an object off of the reserve list, set it up correctly (what image is it?), and put it on the active list, where the user will be able to use it. The reserve list is created upon creation of the specific managers and restocked once it is empty.[1]

Every manager that is derived from this manager is a specific manager. There is an ImageManager, TextureManager, and GameSpriteManager, just to name a few. There is one, and only one of each of these specific managers.[2] Because of this property, each of them can be accessed whenever and wherever the user is in the system.

Each list in the *Manager* class is made up of multiple *ManNodes*. Each of these *ManNodes* is specific to each manager, because the *ManNode* class has many subclasses, where each subclass has a corresponding manager. This allows the *Manager* class to be as general as it can be. By deriving from *Manager*, the derived accept a contract that requires the derived classes to create the correct *ManNode* for that derived class. It's through the getManObject() method that this occurs. This method is used in the reserve list creation process.

---

1 Factory Pattern, pg. 5

2 Singleton Pattern, pg. 4

Singleton

This design pattern allows each of the managers to exist as only one manager. There is only one TextureManager and one ImageManager. It's the same for the rest of the managers as well. Because of this, it uses the **DRY principle** or Don't Repeat Yourself. There is only one copy of each manager and it is unique.

A singleton is created only when it is first used. If I had a manager that I decided not to every use, it wouldn't ever be created. Because of this **lazy initialization**, it doesn't take up space when it is unneeded. You could emulate a singleton by using global variables, but a singleton is preferred to the use of global variables. Lazy initialization is a very good reason why a singleton is preferred. Global variables would take up space, even if they weren't ever used in the program.

A singleton, by making a system more generalized, can make it easier to understand and efficient. If there were multiple TextureManagers that held different textures, that wouldn't be very easy to comprehend. And it would make it harder to access the texture you really want. By making the TextureManager a singleton, you ensure that there is only one TextureManager, and you make it easy to get access to the texture you need.

Factory

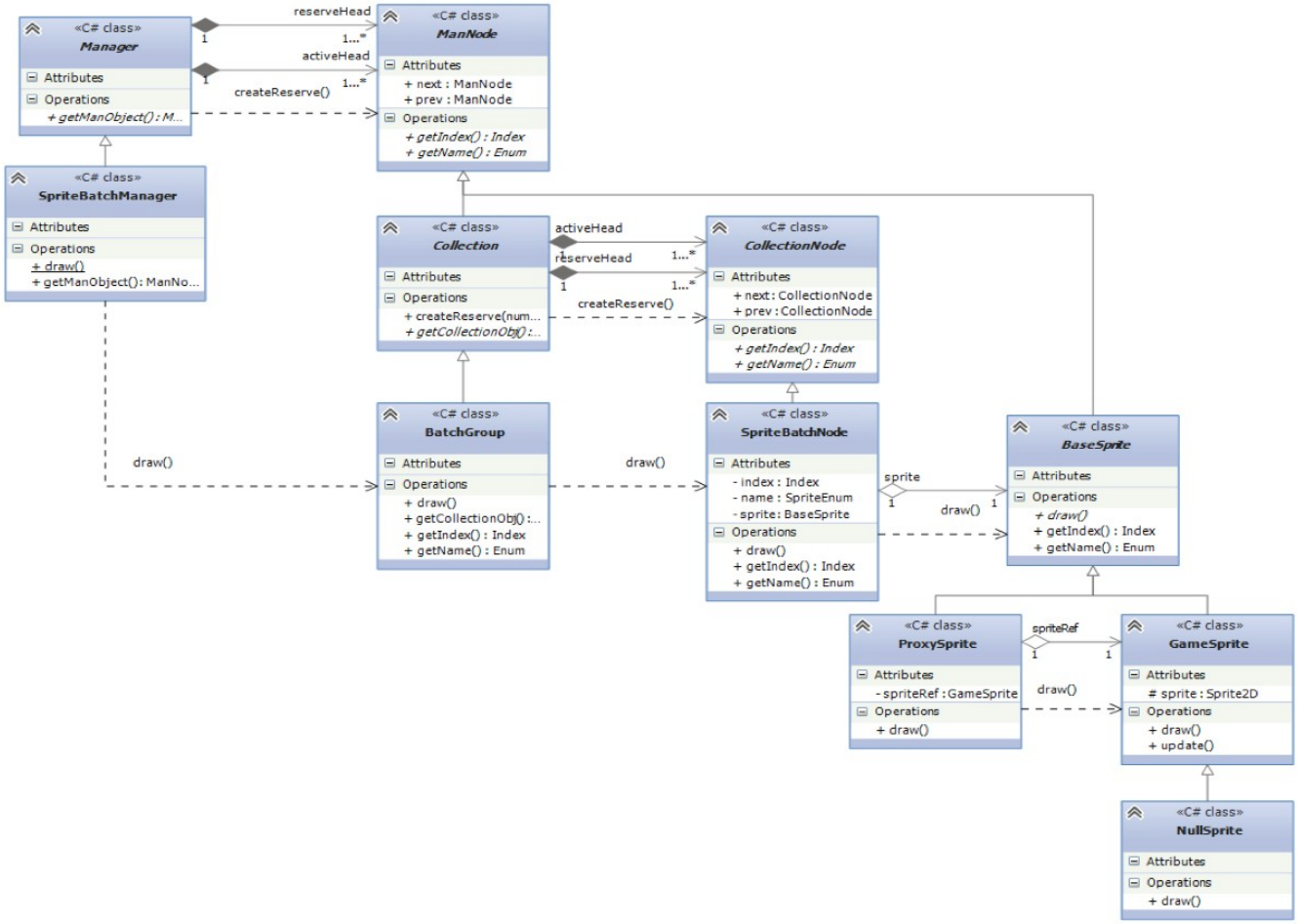
This design pattern focuses on the creation aspect of objects. A factory's main function is object creation. But it doesn't just create an object, it centralizes creation of an object. If someone needs an Alien created, they don't create it themselves. They

ask the factory to create it for them. This takes the worry of figuring out how to create a correct Alien away from the person needing the Alien.

So if someone wants something, they ask the factory to create it. This creates an environment that can be easily debugged and tested. If something went wrong with a specific object, there is no need to go and track down the initialization that caused the issue because all creation is centralized in the factory. Debugging becomes trivial. Testing it is as easy as asking for the factory to create something and if it is correctly created, then testing is basically done. If something isn't created correctly, the factory is to blame. So go in and fix the issue and test it again.

If there are multiple situations that need a slight variation on the same object, a factory can handle that as well. Normally, the user of the object would have to worry about giving the object the right data in each time they want the object, but with a factory, the method for creation is written once and the object that's created will always be the correct one you asked for.

Sprite Batch System



The sprite system is one of the essential systems that allows sprites to be drawn. The sprite batch system holds references to all of the sprites that are going to be drawn each frame. If a sprite is not going to be drawn during a certain frame, or the rest of the game, it is searched for and taken out of the sprite batch system.

The sprite batch system is made up of a singleton SpriteBatchManager that manages BatchGroups. BatchGroups are themselves managers. However, they are not

singleton.[3] There can be many of them in the SpriteBatchManager. The role of the BatchGroup class is to hold similar sprites. If, in Space Invaders, there are Aliens, the player, and Missiles. You can have three BatchGroups each containing the 3 different types of sprites and then mix around the ordering of each group. This ability allows you to control draw priority.

The types of sprites that can be contained in the BatchGroup class are BaseSprites. The BaseSprite class has an abstract method draw() that allows the Sprite Batch System to draw any kind of BaseSprite subclass. The different subclasses are the GameSprite, ProxySprite, and NullSprite, which is derived from GameSprite.

The ProxySprite contains a reference to a GameSprite. The benefit of a ProxySprite is that many different ProxySprites can refer to the same sprite and draw it in different positions.[4] The NullSprite takes the place of a GameSprite. Instead of leaving an unneeded GameSprite that has its draw() method "turned off", the NullSprite has an empty draw() method. So it can replace any GameSprite.[5]

Flyweight

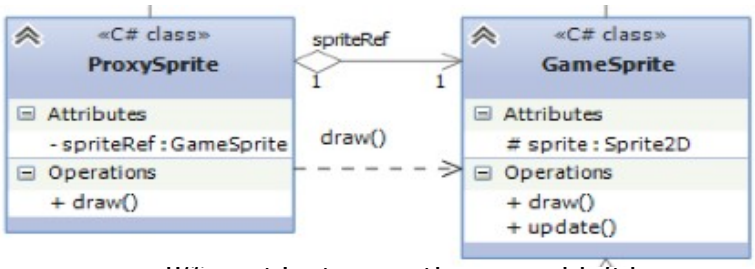
The flyweight pattern is mainly used to **minimize the amount of memory used by resources**. In this example, the ProxySprite has a reference to a GameSprite, which is in a pool of GameSprites. If there was a GameSprite for all of the different sprites on the screen, that amount of GameSprites would be pretty

3 Singleton, pg. 4

4 Flyweight, pg. 8

5 Null Object, pg. 9

memory intensive. This way, the



different places on the screen.

A flyweight is only used w

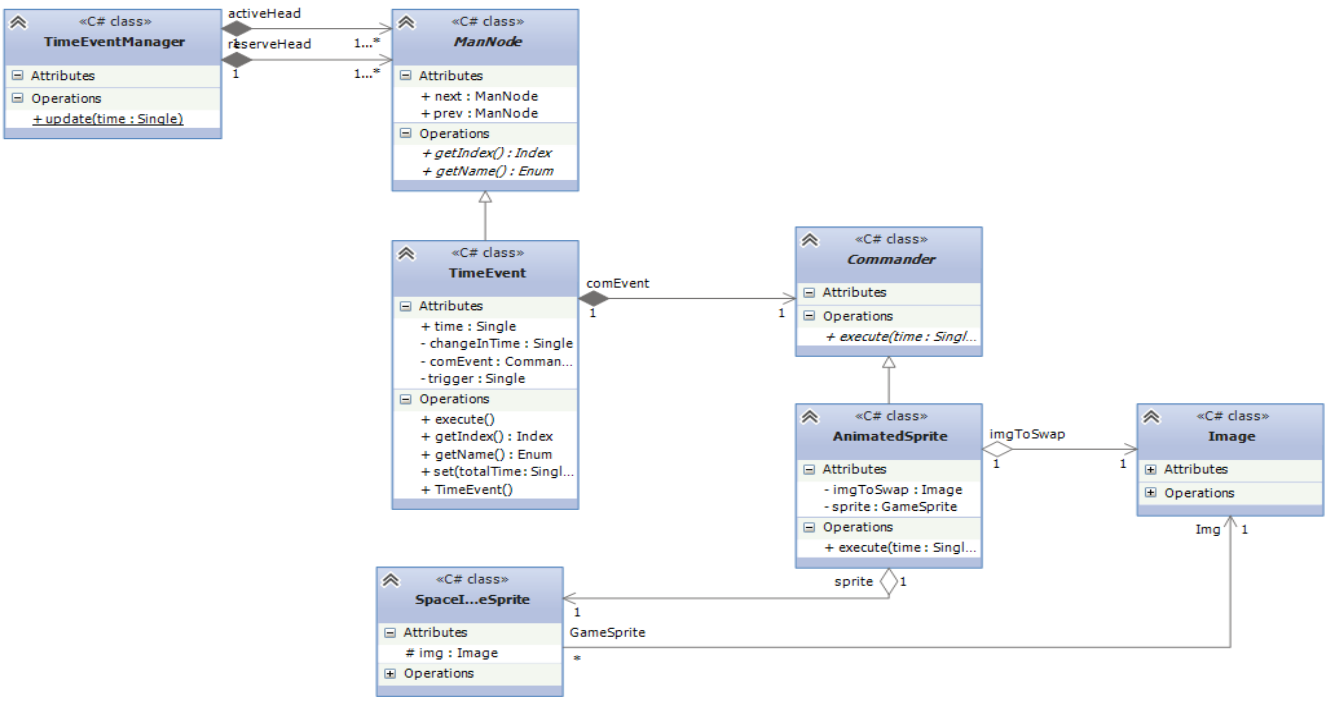
that doesn't have much duplication or many different instances, there wouldn't be much use in using the flyweight pattern. It is an optimization for minimizing the amount of memory instances use. Also, if you implement the flyweight pattern, it is hard to allow single instances to behave differently from the other instances.

Null Object

The null object pattern is the object representation of NULL. The reason you would use a null object instead of making the object NULL is if you want the behavior to stay the same, without writing new code to account for there being a NULL object.

The null object will always do the same thing without screwing up your program. If the object it derives from has a method that it needs to implement, such as the GameSprite draw() method, the null object can just derive from that object, make the method just do nothing, and now it can be inserted wherever the base class is without disrupting the system.

Time Event System



In Space Invaders, we need a way to process events with time because the animation and movement happen at the same time at time intervals. To solve this, Space Invaders has a time event system. The time event system takes in specific events that need to be triggered based on time and checks if the first event in the list is supposed to be processed every frame. When the time comes for an event to be processed, the TimeEventManager calls its execute() method[6] and removes it from the list.

The TimeEventManager class is a singleton. It also has the user enter the amount of time the event will be triggered in. It uses this time and the total game run time to calculate when the event will process. The manager also sorts the events

6 Command, pg. 12

in order of time. So that means the event with the smallest amount of time till it gets processed is first on the list.[7]

The TimeEventManager will accept a couple different events. Animation and movement events will be the main events. Animation and movement happen both at the same time, so they need to both be timed. When they are both processed, the events automatically add themselves back into the manager for later execution.

The TimeEventManager is one of the only managers that doesn't derive from the Manager abstract class. The reason it doesn't is because the Manager doesn't use a list that is sorted. Normally, the list a Manager has doesn't need to be sorted. We don't have a way to sort Images and we don't even need to sort them. For the TimeEventManager, we need to sort the list by time, so it needs a different kind of structure.

Priority Queue

A priority queue is a list of elements sorted so that the element with the highest priority is the very first element. This data structure is important when time is involved because it acts as a timeline for future timed events. The reason we use a priority queue for this system is because the closest time to the current total time is the very first element on the priority queue. The only event you need to check is the very first one.

---

7 Priority Queue, pg. 11

Whenever you insert an element into the priority queue, it needs to be sorted by its priority into the list. You do this because, in a priority queue, you don't want to search through the list to find the element with the highest priority. This works very well for timed events because the priority you sort by is just time.

Command

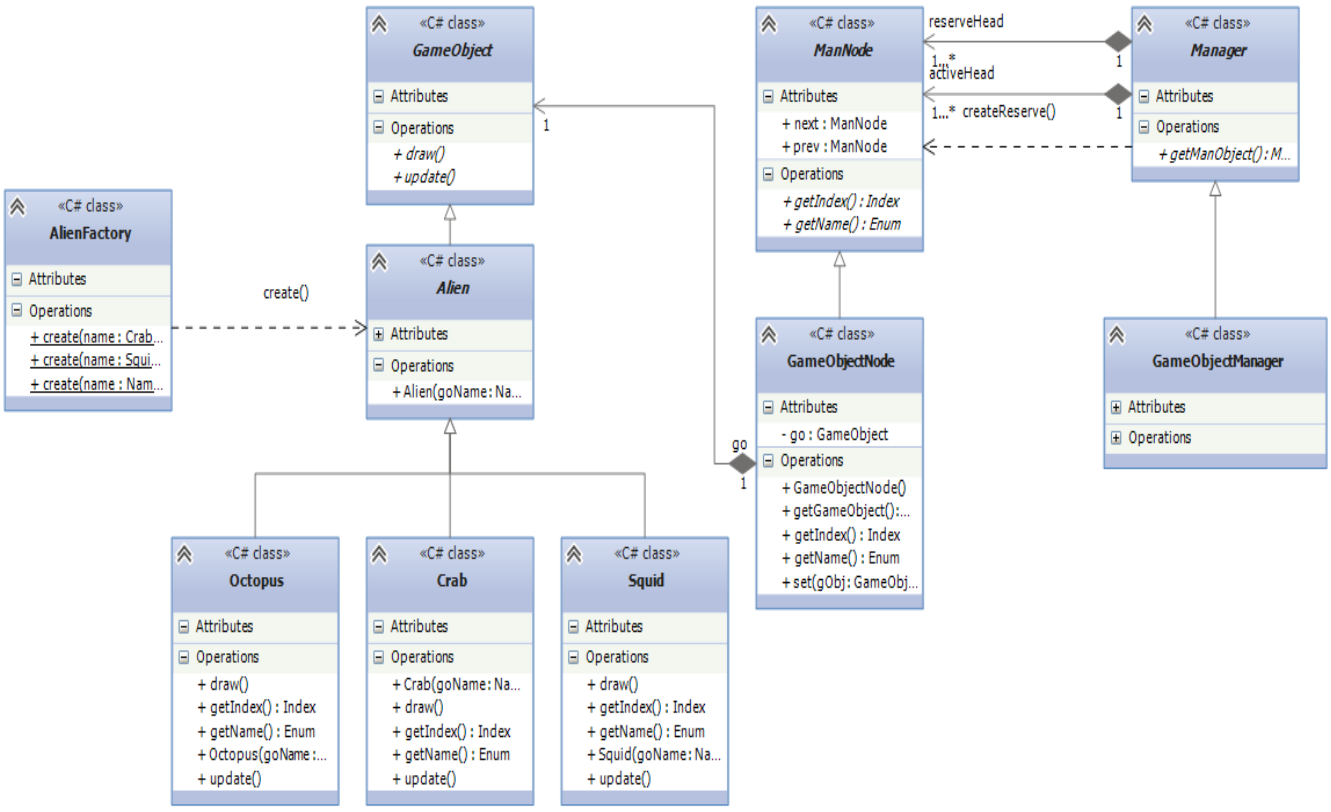
The command pattern main use is that it encapsulates a request as an object. You can pass this object around, store it and pop it, and move it to different areas of a program without changing the request. The command object is called by an invoker that then tells the command object to execute. The command object has a receiver object that performs whatever tasks it needs to when the command object invokes its method.

The command pattern makes it easy to delegate tasks. The command pattern just tells some other object to do something at a specific time. That's all it does. It isn't very smart. The receiver object does all of the work whenever its method is called. Having a general command object allows you to have many diverse objects that do different things without having to check for specific objects. All it does is delegates the work to other objects whilst still being general enough to tell multiple objects to do whatever they need to do.

The invoker is basically a container of command objects that tells each object when to execute. You can now do many different things with the command objects, depending on how you implement the invoker. You can have them be time based executions or you can have them be invoked based on different behaviors of the

overall program. The way the invoker is implemented is important for however you want the command objects to execute.

Game Object Creation



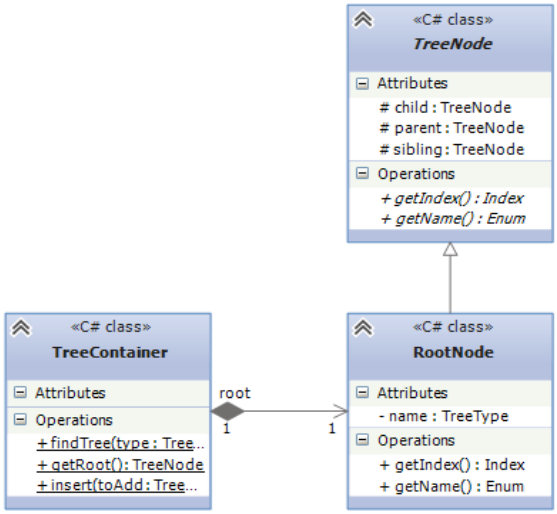
The game object system is pretty straightforward. There is a factory object that creates specific GameObjects.[8] In this UML diagram, you have an AlienFactory creating Aliens. Each GameObject that is created is then stored in a GameObjectManager. This manager derives from Manager[9] and is a Singleton.[10] This manager does another thing, however. It holds all of the GameObjects and it also needs to update them. Every update

8 Factory, pg. 5

cycle, it cycles through all of the GameObjects it holds and calls their specific update. That is specific to each of the GameObjects, but it updates all of them. The system is only used for holding GameObjects and updating them.

PCS Tree System

The tree system is the main part of the alien grid and how we store objects for the collision system as well. It allows the grid to be treated as a whole grid or as individual Aliens. This allows the grid to move as one entity, but also allows individual Aliens to have their own behaviors based on the areas around them. The system contains a TreeContainer manager that holds many different trees. All of the trees are children of the RootNode that the TreeContainer holds a reference to. This replaces the need for some sort of list data structure. The TreeContainer contains one tree, but all we care about are the subtrees of the main tree. Whoever needs a reference to any specific tree will need to go through the TreeContainer.



9 Manager System, pg. 3

10 Singleton, pg. 4

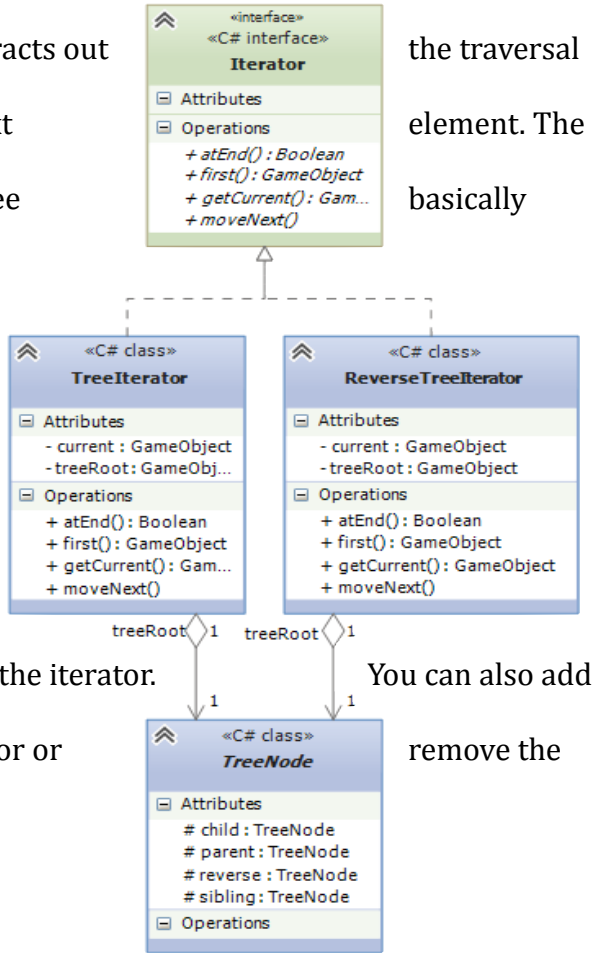
PCS Tree or Composite

For Space Invaders, I am using the PCS (**P**arent, **C**hild, **S**ibling) tree, however, it does function as a replacement for the Composite design pattern. The PCS tree allows one to put objects into a tree structure and treat them as a whole or treat them as individual objects. This becomes important because you can make the entire tree do a specific task, while at the same time, one or a couple of the objects in the tree will do something different or add onto the behavior the whole tree is doing.

With PCS trees, you can easily treat the objects as one object, as a whole, or treat the objects individually, as a part. You use this kind of pattern in a number of systems, like a complex animation system or a collision system, where you can either abort testing because there isn't a collision or go deeper in the tree to actually test if there is a deeper collision.

Iterator

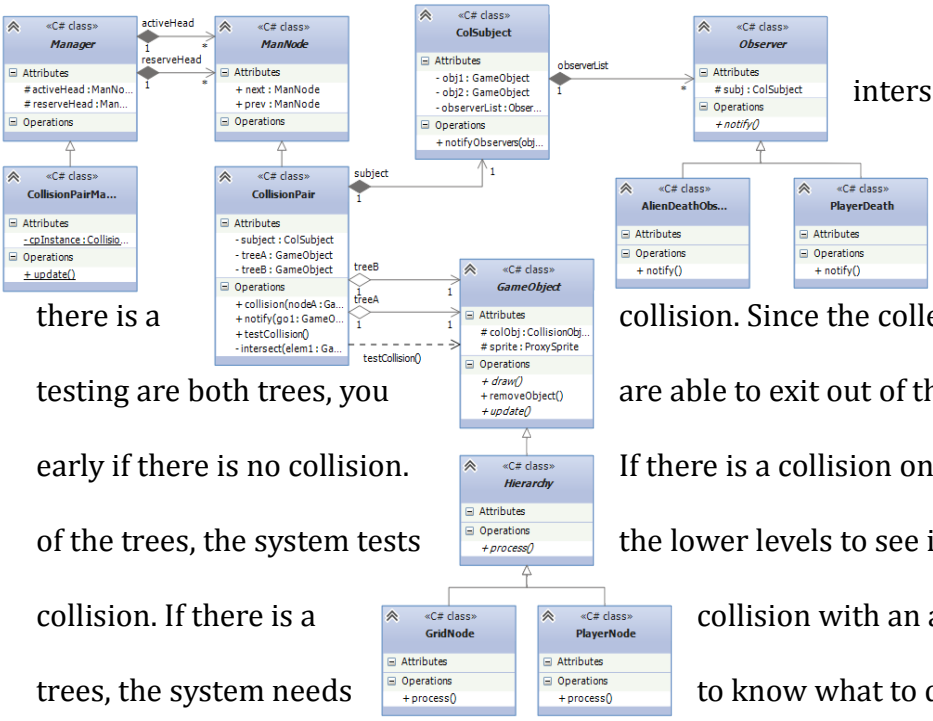
When there is a tree involved in a structure, having an iterator is an incredible thing to have. An iterator abstracts out the traversal of a data structure into moving to the next element. The user of the iterator can go through the tree basically effortlessly.



The iterator traverses the tree without exposing the underlying implementation. This allows for a forward iterator and a reverse iterator without exposing anything to the user of the iterator. You can also add the ability to add an element at the iterator or remove the element the iterator points to.

Collision System

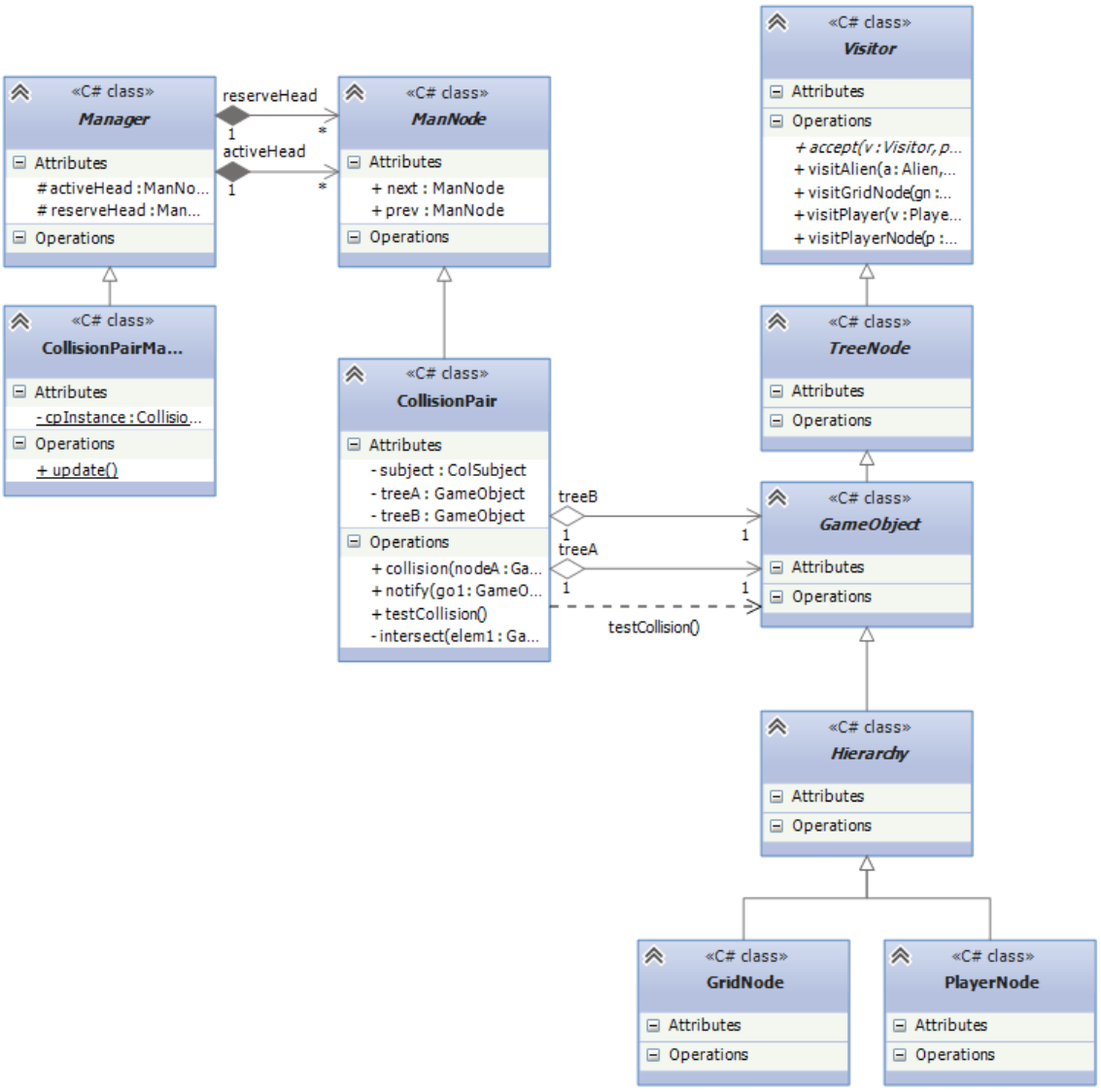
The collision system contains most of the behaviors in Space Invaders. The system works by having CollisionPair objects in their own manager. Each CollisionPair points to two GameObject trees. The trees are all of the objects that could collide with each other in the CollisionPair. One CollisionPair could have the trees of the Aliens and the Missiles. Another pair could have the Bomb tree and the PlayerShip tree. Any two trees that you want to collide with each other can be put in its own CollisionPair object.



Through the intersect and the union methods, you determine if there is a collision. Since the collections you're testing are both trees, you are able to exit out of the collision test early if there is no collision. If there is a collision on the upper levels of the trees, the system tests the lower levels to see if there is a collision. If there is a collision with an actual object in the trees, the system needs to know what to do with those specific objects, but at this point, it doesn't actually know what kind of objects collided with one another. They are just GameObjects at this point. To do this, we use the Visitor pattern to determine the specifics of each GameObject.

Once the Visitor pattern concludes what objects are colliding, the system needs to do specific behaviors for the collision event. The collision system then calls objects the CollisionPair has on a list that dictate what behaviors are going to happen when objects in each tree collide. This is the Observer pattern at work. Once created, the CollisionPairs can have specific Observers attached to them. These Observers are activated whenever there is a collision between the object trees in the CollisionPair. This gives flexibility to the collision system because you can have any number of Observers attached to a CollisionPair. If you want another behavior that you didn't have before, create a new Observer and attach it to the CollisionPair.

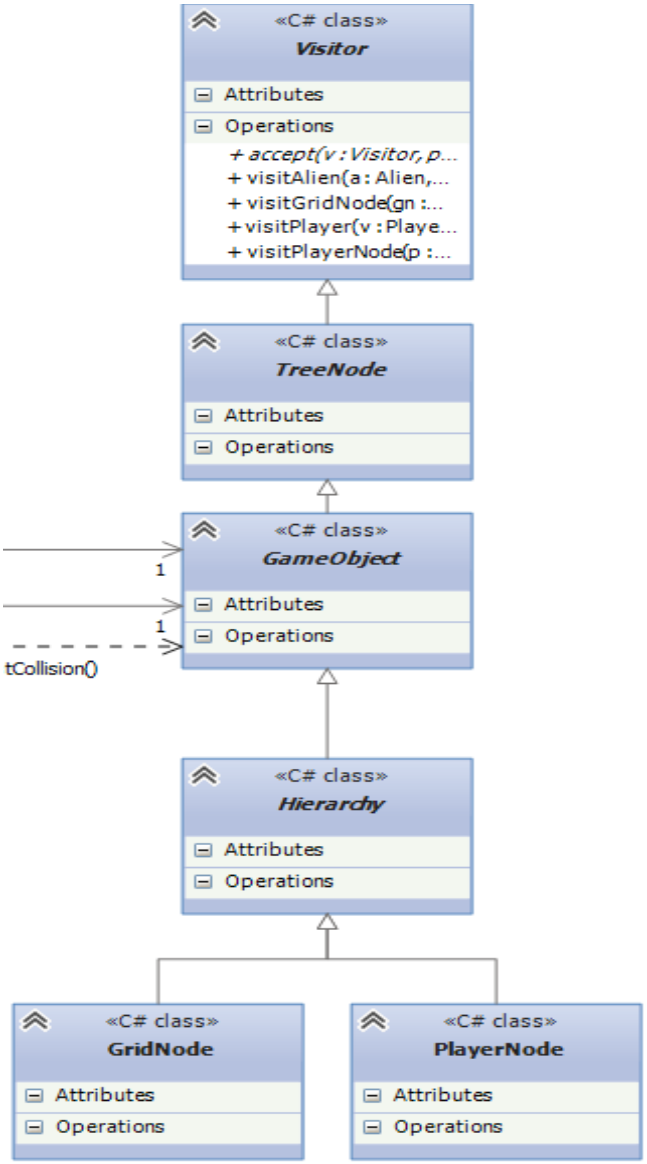
Visitor



The Visitor pattern is an incredibly powerful design pattern. It allows you distinguish between two different objects even though they are derived from the same base class. If you have two GameObjects, the Visitor pattern is able to distinguish between the GameObjects being an Alien and a Missile. It does this through double dispatch. It takes an instance reference as input and implements the

end method. The double dispatch routine goes through this cycle to determine what objects to operate on. The ability to do this is incredibly important.

One benefit from using the Visitor is that you don't have to change the structure you're working with to have the Visitor work. If you're working with a composite structure, you just have to add the behaviors you want, for the Visitor, without changing the hierarchy or structure of the composite.



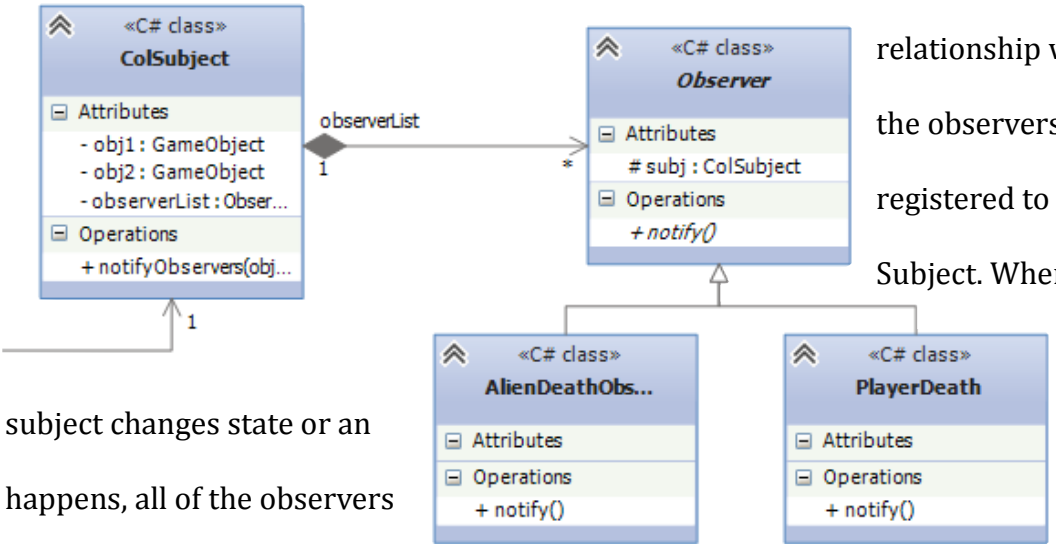
Another benefit to using the Visitor pattern is having a centralized location where all of the operations for the Visitor will live. With the Visitor class, you can also have a default behavior in each method. If there are methods that you never will use, you could put an assert in the method as a default behavior so that if it gets called, you need to take a better look at your system. Adding these new behaviors is relatively easy because of this centralized location.

The Visitor breaks encapsulation, however. That is a drawback of this pattern. The Visitor needs to know what to do with the objects it gets. If you determine that

the two objects are objects that actually collide with each other, like an Alien and a Missile, there isn't an issue really. However, if the objects are hierarchy objects, the Visitor needs to go further down into the tree, so it needs to know about the children of the objects. This traversal of the tree also makes it harder to make changes to the overall structure of the composite tree difficult.

Observer

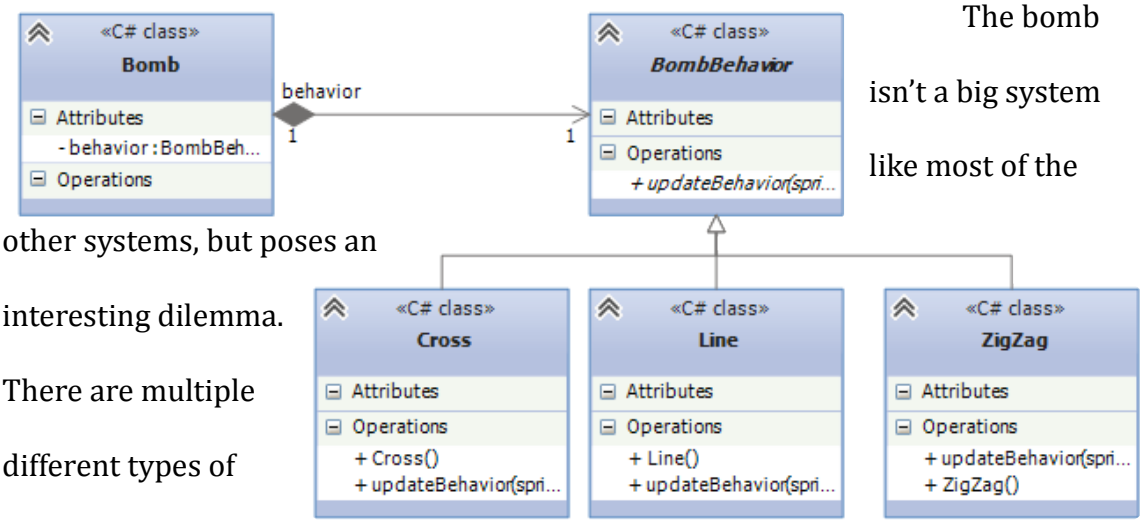
The Observer pattern describes the behaviors of a system. It is a one-to-many relationship where the observers are all registered to one Subject. When the subject changes state or an event happens, all of the observers associated with that subject are notified that the state has changed. They then update themselves automatically.



The Observer pattern is very useful in many scenarios. In the collision system, it allows the user to attach many different observers to one collision pair to be fired off when there is a collision. This is the easiest and most customizable solution. Another solution would be hard coding the behaviors in each collision pair, which would add a whole bunch of problems with the entire system. Observers provides decoupling to a system and encapsulates specific behaviors in a small class

that can be reused in multiple places that would need that behavior.

Bomb Behaviors



The bomb isn't a big system like most of the other systems, but poses an interesting dilemma. There are multiple different types of bombs: line, zigzag, cross, and corkscrew. What I could have done was make each bomb a separate object. That would make it easy to code and separate out the behaviors, but the behaviors are very simple and could be encapsulated in a different class.

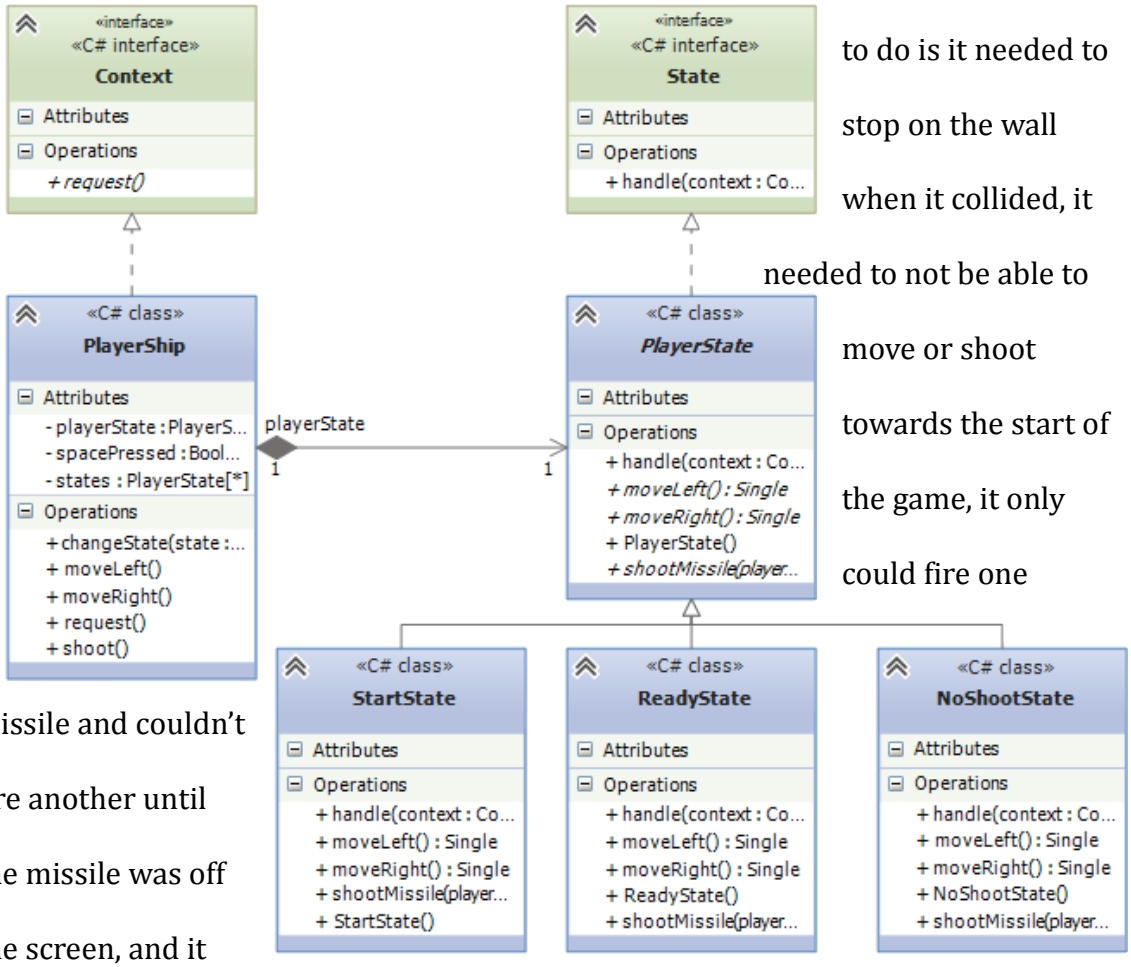
Strategy

The strategy pattern is used when you want to encapsulate multiple behaviors and use them interchangeably. The strategy allows you to use the same object and the only thing that's different about it is an underlying behavior.

The drawback of this solution is it makes the object more complicated to instantiate. You would need get the correct behavior for each specific object. This can be simplified by creating a factory that creates the object with the correct strategy behavior, but if you were to do it by yourself, it gets complicated.

Player

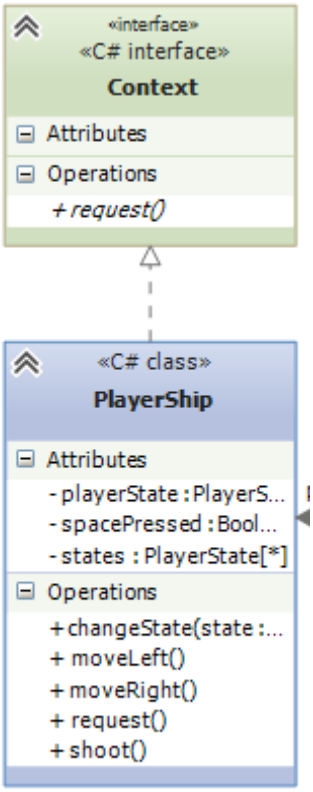
The player's ship that it controlled posed an interesting issue. What it needed to do is it needed to stop on the wall when it collided, it needed to not be able to move or shoot towards the start of the game, it only could fire one missile and couldn't fire another until the missile was off the screen, and it couldn't move or shoot once it died.



The solution to this issue I came up with is using the state pattern. It has a state for when it starts and can't move, a start for when it's ready to do anything, a state for when it can't shoot, and a state for when it dies. This way it just changes state when it something happens. I didn't make a state for the hitting of the walls because I was already testing the inputs for right and left, so I just put a flag testing in both of those checks.

The state also uses the Strategy pattern to encapsulate the behaviors of the player in each state. The player would be able to move left and right and shoot. Depending on what state the ship is in, it will be able to do some, all, or none of these behaviors.

State



The state design pattern allows an object to change its internal state, and this change, in turn, changes the behavior of the object. The context in the State pattern changes its state. The state is kept in the context as well. It gets the behavior through the current state it contains. This pattern is very similar to how a state machine behaves, but the State pattern encapsulates the state machine's behavior into classes. This allows changes to be localized into each specific class.

The state pattern is very similar to the strategy pattern, but the difference between them is why you would want to use them. The strategy creates a class with a specific behavior. The state pattern allows the class to change its behavior and treat the behavior as the object's state. The use of the state pattern generally has more classes in the design of the system. States can be shared between multiple contexts, whereas the strategy pattern typically dictates behaviors for individual classes.

Post Mortem

Space Invaders was, to me, an invaluable project to code. I used more design patterns in this project than I ever have in any other project. It's definitely the biggest project in terms of the number of classes and the amount of lines I've written. It really worked well for me because I love hands on learning, and this project was all hands on.

Looking back on my design, it could definitely be better. I would've liked to implement an actual input system. The way I have it now, only for the PlayerShip, is it just tests if something has happened every frame in the PlayerShip class. The input system may or may not do the same thing, but it would have the inputs in a centralized location that can easily be expanded upon rather than going into a class and adding code there.

Since I have good knowledge about these design patterns, I would like to see how I would think of these kinds of systems. There may be a way to use design patterns I didn't know about when the beginning systems were in development. Combining design patterns with other patterns would possibly be a way of redesign.

All in all, this was sure a fantastic experience. This project taught me an incredible amount and I am the better for putting the amount of time I did into this. It may not be perfect, but I worked my butt off doing this, and the important thing is that I can do this a ton better the next go around. I can't wait for what's next to learn and what big project I'll take up next.